Virtual Memory In Nachos

Michael Jantz

Prasad Kulkarni

Introduction

- This lab is intended to introduce the final Nachos assignment.
- Please grab the starter code for this assignment from the class website
- Untar, make it, tag it:

```
-bash-3.2$ tar xvzf eecs678_pa_vm.tar.gz
-bash-3.2$ cd nachos
-bash-3.2$ make; ctags -eR
```

- **NOTE:** This starter code reverts to the static priority scheduler. None of the experiments you will run with this assignment use threads that change their default static priority. Therefore, for this assignment, the scheduling policy may be thought of as Round Robin.

A Simulated Physical Disk

- Nachos simulates a physical disk via operations on a Unix file.
- This “disk” has a single surface, which is split up into “tracks” and each track is split up into “sectors”
    - See machine/disk.h
- There are 32 tracks, each containing 32 sectors. Each sector corresponds to 128 bytes of addressable space. Therefore, the disk in Nachos contains 128KB (128B * 32 * 32) of addressable space.

Main Memory in Nachos

- Main memory is implemented as an array of bytes.
- Memory is byte-addressable and organized into 128-byte frames, the same size as disk sectors.
- By default, Nachos has 128 frames of main memory. Thus, Nachos has 128 frames * 128 bytes = 16KB of main memory.
- See machine/machine.h
  - Lines 39 – 44
  - For the purposes of this assignment, you may ignore the use of the TLB in Nachos and in userprog/memmgr.h

Memory Management in Nachos

- Nachos simulates paging to give its threads a contiguous logical address space.
  - The address space for each process is 128KB
- In order to help support this, a virtual memory scheme is used to minimize the number of “physical” Nachos memory pages used at any given time
- In this scheme, pages of memory that are not in use are swapped out to the “disk”, from which they are read back into physical memory as needed to support parts of the logical address space being used by the program

The Memory Management Unit

- In most modern operating systems, paging is achieved through the use of a memory management unit (MMU).
- In Nachos, the *MemoryManager* and *Machine* objects together serve as the MMU of the system.
- The *MemoryManager* keeps track of which physical pages are free, which are used, and the owners of each page.
  - *MemoryManager* is defined in userprog/memmgr.h. Notice the array of Frames. This is where the “physical memory” in Nachos resides.
- Machine holds the page table of the currently running process.
  - *Machine* is defined in machine/machine.h. Notice the TranslationEntry pointer named pageTable. This is the aforementioned page table. More on this in the following slides.

Address Space

- Since each Nachos thread has its own contiguous logical address space, each thread needs a way of mapping its logical (i.e. virtual) address space to the physical address space maintained by Nachos.
- Therefore, each Nachos thread maintains its own *AddrSpace* object. The *AddrSpace* object is the Nachos abstraction of the thread's logical address space.
- The *AddrSpace* object maintains a table mapping the thread's logical addresses to actual physical addresses in Nachos “physical memory.”
- This is known as the thread's *page table*, and the page table used by the *Machine* object is loaded using this pointer
  - Specifically, by use of *currentThread->space->pageTable*
- The figure on the following slide shows how Nachos uses page tables to give each thread a contiguous logical address space

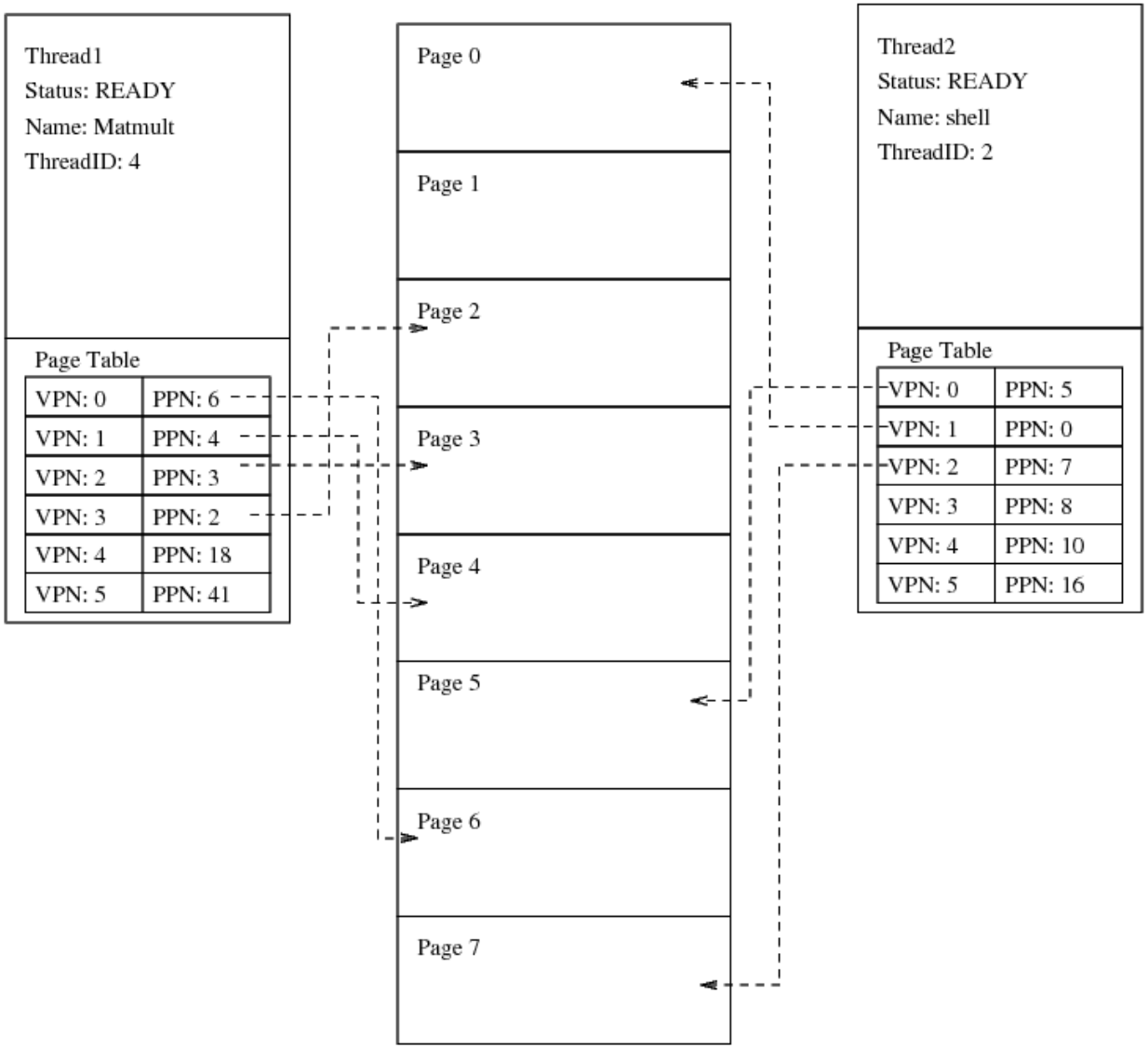

Figure 7: Non-contiguous Memory Allocation

Translation Entries

- In Nachos, each thread's page table is implemented as an array of *TranslationEntry* objects
  - See machine/translate.h
- For each virtual page in each thread's address space, there is exactly one *TranslationEntry* object.
- Each *TranslationEntry* maintains the following information:
  - **virtualPage** and **physicalPage** – the virtual to physical mapping of the page this object represents.
  - **valid** – whether or not the page resides in memory
  - **readOnly** – read only status. Program text pages are usually considered read only.
  - **use** – whether or not the page has been referenced by hardware.
  - **dirty** – whether or not the page has been modified
  - **zero** – whether or not page is initialized with zeros.
  - **cow** – whether or not page should be duplicated when written to.
  - **file** and **offset** – Refer to the file (and offset within) this page corresponds to. If the page was modified before it was swapped out, these refer to the swapfile. If it was not modified, this refers to the original executable file. If the page contained only uninitialized data, these are NULL.

Doing the Translation

- When a process makes a reference to a logical memory address, Nachos invokes *Translate()* to get the corresponding physical address.
- If the page is “valid” (i.e. in physical memory), then *Translate* simply computes the corresponding physical address, and this is then used to complete the memory reference.
- This process is illustrated on the following slide.

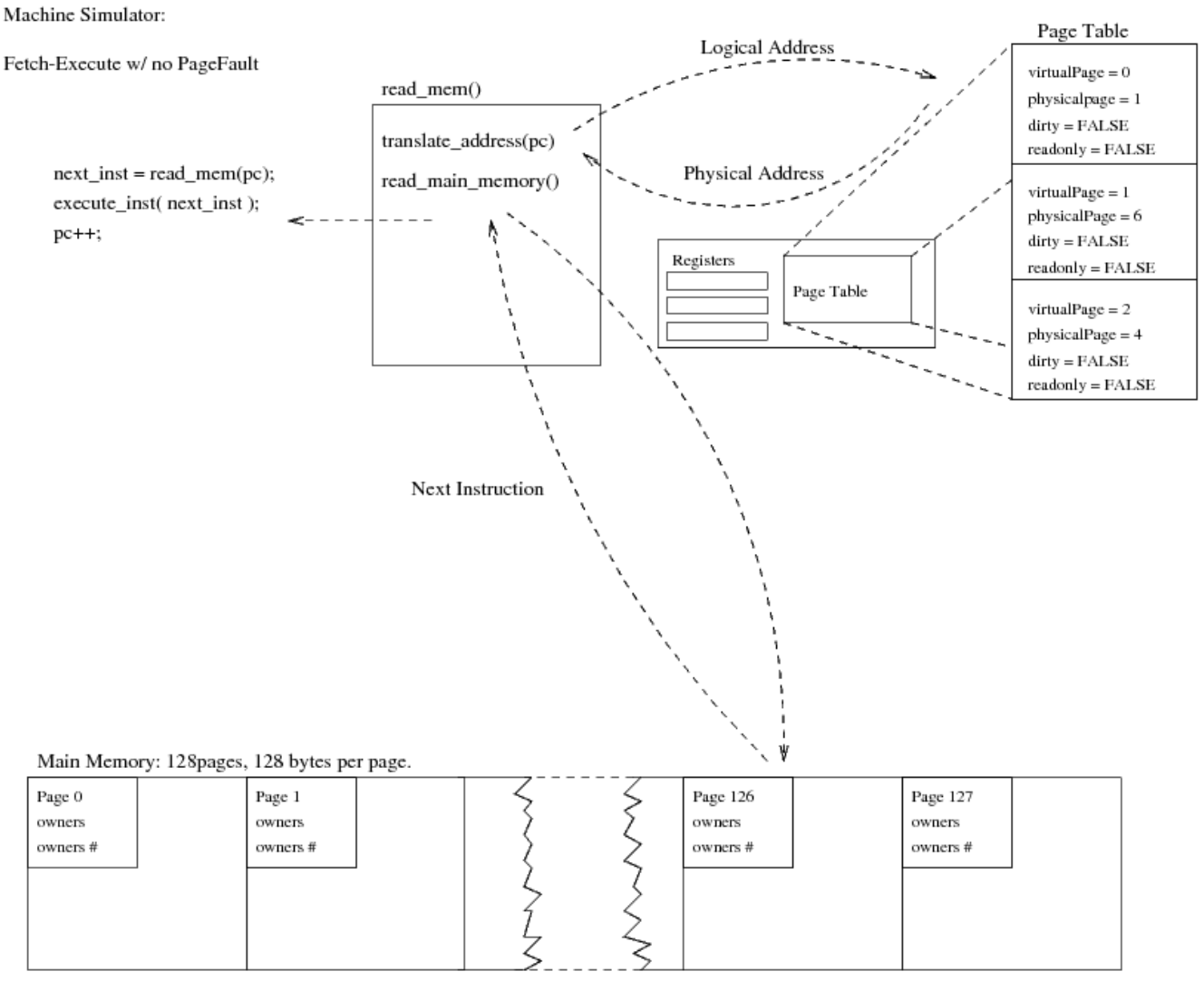

Figure 3: Address Translation

Page Faults

- If the page is not “valid” (i.e. not in physical memory), a *page fault* exception is raised, and the exception handler swaps the page into memory. The instruction that originally performed the memory reference is then restarted.
- This process is illustrated on the following slide.

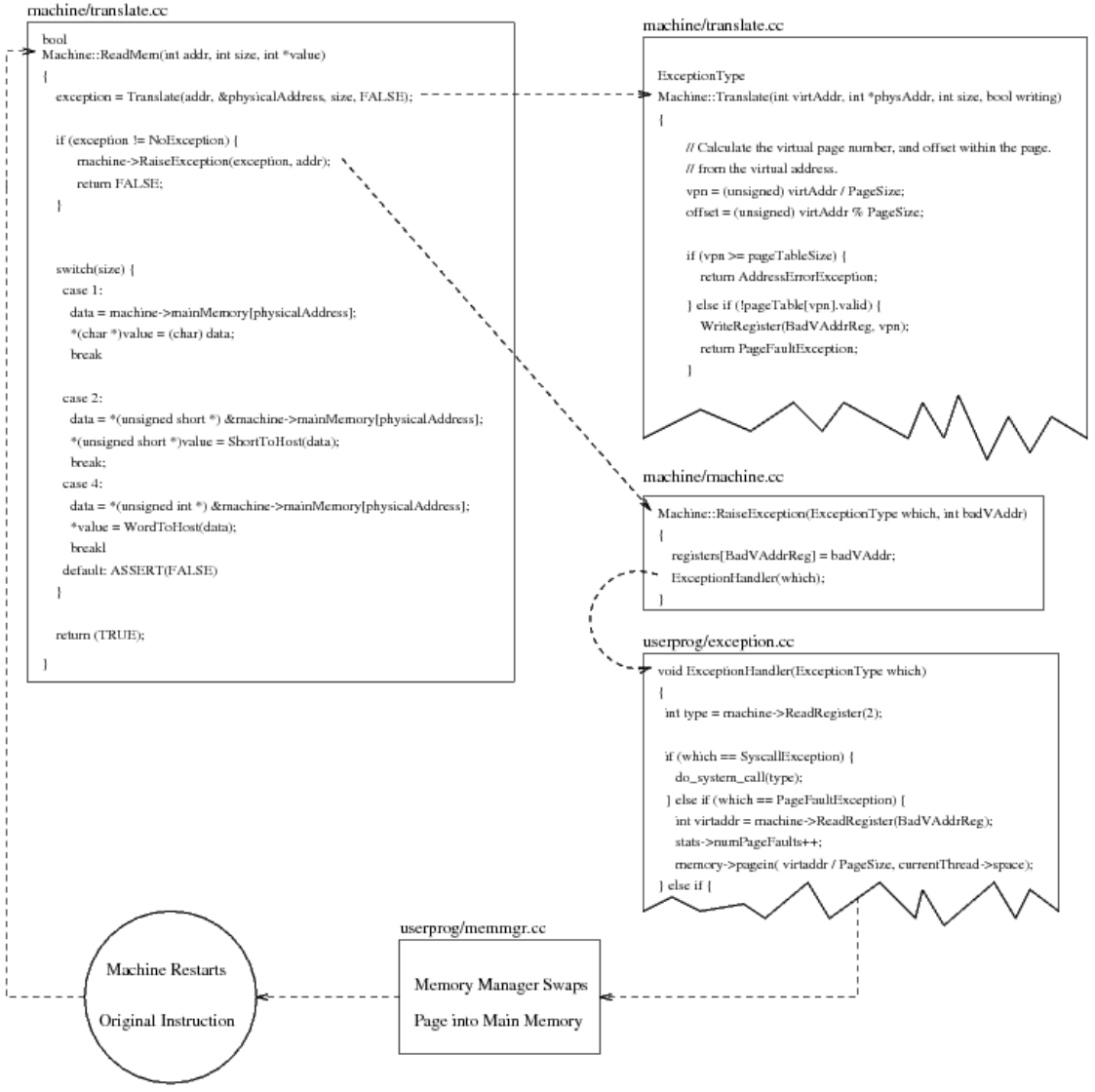

Figure 8: A Page Fault

pagein and *pageout*

- When a page fault occurs, the *MemmoryManager::pagein* (userprog/memmgr.cc) function is called to page the correct physical page into main memory.
- If there are no free pages in the physical memory, which is entirely possible because each process may address the entire physical memory, Nachos must remove a page from physical memory to create space for the page that caused the page fault.
    - This page is thus *swapped out*
- If this is indeed the case, Nachos uses *MemoryManager::Choose_Victim* to select a victim page, and *MemoryManager::pageout* to swap this page out of main memory, see userprog/memmgr.cc:211-218
- *MemoryManager::pageout* uses the page's OpenFile * to determine which file to write the page out to.

Your Assignment

- Your final programming project is to modify the virtual memory system in Nachos by introducing:
  - A working set for each process.
  - A scheme for adjusting the configured working set size for each process running on the system by remembering how many pages each process has required in the past *delta* time units.
  - Three page replacement algorithms discussed in class and in your book.
- You will then analyze how the system behaves differently with different *delta* values and different page replacement algorithms.

The Working Set Scheme

- This assignment changes the memory management policy in Nachos by introducing a *working set* for each thread.
- The working set is the set of physical pages a thread has mapped into its logical address space.
- Initially, each process' working set size should be set to the minimum allowable size.
- Periodically, the system should prescribe new working set sizes for each of its threads by “remembering” how many pages each thread has used in the last accounting period, as this is an indicator of how many pages it might need in the future, and updating each thread's working set size accordingly.
- Now, the system should page out one of its pages in memory if a process attempts to page in more memory when it already owns more pages than allowed by its prescribed working set size.

Refreshing the Working Set Size

- For this assignment, we've chosen to update a thread's working set size after it has run for 10 quanta.
- When the scheduler's timer interrupt occurs, a counter associated with the current thread is incremented. When this counter reaches a certain value, in this case 10, the current thread's working set size is refreshed for its next run.
- To refresh the working set size, the system should count the number of pages in this thread's page table that have been referenced in the last (*delta* * quanta) time.
    - This is the new working set size.
- This operation can be performed by maintaining a history field on each *TranslationEntry* and updating this field when that *TranslationEntry* is referenced.

Page Replacement Policies

- In addition to the working set scheme, you will also implement three new page replacement policies:
  - FIFO – Victims are chosen based on the order they were paged in.
  - LRU – Victims are chosen based on the last time they were referenced.
  - Second Chance – A variation of FIFO, but pages which have not been referenced (best) or were referenced but not modified (second best) are chosen before the FIFO page is chosen

Stub Functions

- There are a number of stub functions to get you started. First, the functions local to each process' address space.
- **NOTE:** The Nachos distribution we've provided calls these functions when needed for your assignment. You simply need to implement each function.
  - int AddrSpace::**TooManyFrames( )** {return 0;}
    - Checks if this process owns too many pages. Return 1 if it does.
  - int AddrSpace::**FIFO_Choose_Victim(int notMe)** {return 0;}
  - int AddrSpace::**LRU_Choose_Victim(int notMe)** {return 0;}
  - int AddrSpace::**SC_Choose_Victim(int notMe)** {return 0;}
    - These stubs are for implementing each of the page replacement policies. The value **notMe**, if not equal to -1, indicates a physical page that you must not use. You should return the number of the physical page you have chosen to release.

Stub Functions (cont.)

- There are a number of stubs for the *TranslationEntry* object as well. Again, the Nachos system we've provided calls these when appropriate for your assignment. You simply need to implement each function:
  - void TranslationEntry::**clearSC( )** { }
    - Clears the second chance information for this page.
  - void TranslationEntry::**clearRefHistory( )** { }
    - Clears the reference history for this page.
  - void TranslationEntry::**setTime(unsigned int time)** { }
    - Sets the load time for this page.
  - unsigned int TranslationEntry::**getTime( )** { }
    - Gets the load time for this page.

Fields You Should Add

- This will become clearer after you've read the assignment description, but, for reference, you should add some fields to the *TranslationEntry* class:
  - You will need a field to track the *reference history* of this *TranslationEntry*. You can reuse this field to implement the **LRU** page replacement policy.
  - You will need a way of tracking the *arrival time* for each page for the **FIFO** and **Second Chance** page replacement algorithms.

Good Luck

- Good luck, and, please start early. This assignment is more difficult than the others.
- Let us know if you have any questions.