Lab Virtual Memory in NachOS

26th of October, 2021

In the previous lab, you wrote system calls to enable "multiprogramming." In this assignment, you will build a virtual memory system to allow for more processes to be run than availabe system memory. Your code will be implemented in the `userprog`, `test`, and `threads` directories. You will need to compile in the `vm` directory to have the correct TLB flags set.

Group Task 1 – Repairs

Your first task is to complete/correct the `Exec` system call from the userprog project. You will need a functioning `Exec` call to launch multiple processes and test your new virtual memory system. Fix any other bugs as you debug `Exec`.

Group Task 2 – Shell

The second task is to build a working shell program. Your shell program should be able to run multiple processes concurrently. Seperate the different processes with the `;` character. The shell should not read another line until all of the previous processes have finished.

Group Task 3 – Page Replacement and Demand Paging

Right now, your system loads the entire program into physical memory in the `LoadExecutable` function. This means that the number of processes we can concurrently run is limited by the size of Physical Memory. To fix this, we are going to implement swapping. The basic idea is that your code and data will be stored on disk in a special "swap file" rather than directly in physical memory. Another way to think about this is we will store virtual memory in the file and use Physical Memory as a cache.

We are also going to implement "demand paging". Instead of loading all of the code and data immediately when we launch a program (either the main thread or through `Exec`), we are going to load only the pages we need when they are first accessed.

The `NachOS` "hardware" will generate a page fault exception for each invalid page. These execeptions can be caught and handled in `exeception.cc`. You will need to add an `else if` to your code.

1. Create a **swapfile** data structure that can hold at least four times the amount of physical memory and manages a file on the disk. This file can be created and manipulated with the FileSystem calls provided by `NachOS` .
2. Create a data structure (InvertedPageTable) to map a TranslationEntry from a process to a physical memory frame. Your InvertedPageTable will replace the current synchronous bitmap. Unlike the `pageTable`, it will be shared across all `AddrSpace` objects.
3. Utilize the TLB provided by `NachOS` . You will need to remove your page table code through macro guards. `NachOS` will utilize the TLB when compiled with the `-DUSE_TLB` flag (set when compiling in the `vm` directory).
4. Implement "demand paging" by loading pages on page faults. You should first check if the entry is in physical memory and then check swap memory. Non-writable pages should not be placed in swap.
5. Implement page replacement where the kernal can evict any virtual page from physical memory to satisfy a page fault.

SwapFile

Implement a new structure call `SwapFile`. The "swap file" will be shared across all processes, and will manage a file on the disk. You will need to implement functions for adding a page to the file, removing a page, and fetching a page.

1. `bool Fetch(Page* p)` returns `TRUE` and updates the page's buffer if page's entry is located in the swap file.
2. `bool Place(Page p)` returns `TRUE` and writes the page's buffer if page's entry is located in the swap file, or can be placed into the swap file.
3. `void Remove(TranslationEntry e)` removes the page associated with the translation entry if it is contained in the swap file.

InvertedPageTable

Implement a new structure call `InvertedPageTable`. The inverted page table tracks the translation entry for every frame. To start, the inverted page table will mimic the bitmap you are currently utilizing. Once there are more pages than total memory, you will need to evict pages and make sure the appropriate translation entries are updated on a context switch (`SaveState` and `RestoreState`).

Translation Entry Buffer

`NachOS` implements a transition lookaside buffer (TLB). The TLB is shared across all address spaces, and the entries of the TLB are updated by the simulator. If a virtual page number is not in the TLB then a page fault will be generated. The TLB will need to be updated and the page loaded into memory if it is not.

Individual Task 4 – Handle other exceptions

`NachOS` has several additional exceptions defined in `machine.h`. Currently, your implementation will crash if encountering one of them. You should "bullet proof" your kernel so that it does not crash. Fatal exceptions should be caught and processes exited. How would you handle a process with multiple threads?

Individual Task 5 – Sleep

Add an additional system call: `Sleep(int howlong)`. Sleep should be implemented with the `AlarmClock` from lab1. You will need to change `syscall.h` as well as `start.s`.

Individual Task 6 – Pre-fetching

Pre-fetching is an optimization that takes advantage of spatial locality to speed up memory access. Spatial locality is the idea that if we access data on one page, we are fairly likely to access data on the subsequent page soon. We can take advantage of this by loading more than one page into memory at a time when a page fault occurs. In `NachOS` , we won't see a huge performance increase from this (in fact, it might actually hurt performance), but on a real system with a physical hard drive, this can significantly reduce the number of page faults and lead to a big performance boost.

Extend your virtual memory system so that instead of loading a single page, it loads pages in pairs. For example, if virtual page 2 gets loaded into memory, virtual page 3 should also get loaded. If virtual page 5 gets loaded, so should virtual page 4. When loading an even page, you should load the next page at the same time. When loading an odd page, you should load the previous page.

Handing in

Create a README.txt file which describes what each group member has done. You should include which files have been changed and why. If you were unable to get something to work, explain the problem. The documentation is an important part of the project.

This lab will be due on 11/8/2019 at 11:59pm.

This assignment is adapted from one by Dr. Robert Marmorstein and this lab, and all the other `NachOS` labs were derived from the original `NachOS` projects by Tom Anderson at UC Berkeley. They have been modified to fit our lab environment and changes in the compilation software since `NachOS` was originally published.