Project 3: Demand Paging

Spring 2019

Due: Friday, June 7 at 11:59pm (No extensions) (Really)

In project 2, each process had a page table that was initialized with physical pages and their contents when the process was created. In project 3, you will be implementing a more sophisticated memory management system where physical pages are allocated on demand and pages that cannot fit in physical memory will be stored on disk.

Background

You will implement and debug virtual memory in two steps. First, you will implement demand paging using page faults to dynamically initialize process virtual pages on demand, rather than initializing page frames for each process in a [illegible] you will implement page replacement, enabling [illegible] free up a physical page frame to satisfy a page fault [illegible] ow your kernel to "overbook" memory by exec[illegible] mory at any one time, using page faults to multiple [illegible] er number of process virtual pages. When implem[illegible] ser programs unless they monitor their own perf[illegible]

You project will implement t[illegible]

1. **Demand Paging**. Page[illegible] o physical pages are free, it is necessary to [illegible]
2. **Lazy Loading**. To fulfill [illegible] no pages when started, and depend on deman[illegible] xecute. When you are done, `loadSection`[illegible]
3. **Page Pinning**. At times [illegible] ng it temporarily impossible to evict.

The changes you make to Na[illegible]

- `VMKernel.java` — an extension of UserKernel
- `VMProcess.java` — an extension of UserProcess

You will notice that these classes inherit from `UserKernel` and `UserProcess`. Try to depend on the implementation of your base classes as much as possible. Note that, with `readVirtualMemory` and `writeVirtualMemory`, very little code needs to know the details of virtual addressing. For example, you should not have to change any of the primary code which serviced the `read` syscall. That said, you should not change the base classes in any way that makes them dependent on project 3. It should still be possible to run `nachos` from the `proj2` subdirectory to run user-level programs.

You will compile and run the project in the `proj3` directory. Unlike the first two projects, you will not need to learn any new Nachos modules and will continue to use functionality that you became familiar with in project 2. Before starting your implementation, also see the Tips section below.

Design Aspects

Central to this project are the following design aspects:

1. **TranslationEntry bits.** You will extend your kernel's handling of the page tables to use three special bits in each TranslationEntry (TE):

    - Valid bit: Nachos will set or clear the valid bit in each TE to tell the CPU which virtual pages are resident in memory (a valid translation) and which are not resident (an invalid translation). If a user process references an address for which the TE is marked invalid, then the CPU raises a page fault exception and transfers control to the Nachos exception handler.
    - Used bit: The CPU sets the used bit (aka reference bit) in the TE to pass information to Nachos about page access patterns. If a virtual page is referenced by a process, the machine sets the corresponding TE reference bit to inform the kernel that the page is active. Once set, the reference bit remains set until the kernel clears it.
    - Dirty bit: The CPU sets the dirty bit in the TE whenever a process executes a store (write) to the corresponding virtual page. This step informs the kernel that the page is dirty; if the kernel evicts the page from memory, then it must first "clean" the page by writing its contents to disk. Once set, the dirty bit remains set until the kernel clears it.

2. **Swap File.** To manage [illegible] `tem` (via `ThreadedKernel.f` [illegible] choices, but we suggest using a single, global s [illegible] lifetime of your kernel, and be properly delete [illegible] asonably unique file name.

    When designing the s [illegible] es. Thus you should try to conserve disk space [illegible] ry: if you end up having gaps in your swap spa [illegible] e upon program termination), try to fill [illegible] free list works well. You can assume that the s [illegible] ot be any read/write errors. Assert if there [illegible]

3. **Global Memory Accou** [illegible] e managed as in project 2), there are n [illegible] elevance to all processes: which page [illegible] The former is necessary to prevent the eviction [illegible] to managing eviction of pages. There are man [illegible] t using a global inverted page table (see the tip [illegible]

4. **Page Pinning.** When your code is using a physical page for system calls (e.g., in `readVirtualMemory` or `writeVirtualMemory`) or I/O (e.g., reading from the `COFF` file or the swap file), you will need to "pin" the physical page while you are using it. Consider the following actions:

    1. Process A is executing the program at user-level and invokes the `read` system call.
    2. Process A enters the kernel, and is part way through writing to user memory.
    3. A timer interrupt triggers a context switch, entering process B.
    4. Process B immediately generates numerous page faults, which in turn cause pages to be evicted from other processes, including some used by process A.
    5. Eventually, process A is scheduled to run again, and continues handling the `read` syscall as before.

In this example, the page to which A is writing should be pinned in memory so that it is not chosen for page eviction. Otherwise, if process B evicted the page, then when process A was rescheduled it would accidentally write over the page B loaded. Just use another data structure to keep track of which pages are pinned.

Tasks

1. (30%) Implement demand paging. In this first part, you will continue to preallocate a physical page frame for each virtual page of each newly created process at `exec` time, just as in project 2. And as before, for now continue to return an error from the `exec` system call if there are not enough free page frames to hold the process' new address space. You will not yet need to implement the swap file, page replacement, page pinning, an inverted page table, etc. Instead, you just need to make the following changes:

1. In `VMProcess.loadSections`, initialize all of the TranslationEntries as **invalid**. This will cause the machine to trigger a page fault exception when the process accesses a page. Also do not initialize the page by, e.g., loading from the `COFF` file. Instead, you will do this on demand when the process causes a page fault. As a result, `loadSections` will continue to allocate physical page frames in the page table for each virtual page, but delay loading the frames with content until they are actually refer [illegible]

2. Handle page fault exc [illegible] the process references an invalid page, the m [illegible] arked valid, no fault is generated). Modify yo [illegible] dle it by preparing the requested page on de [illegible]

3. Add a method to prep [illegible] on different pages are handled in different w [illegible] nding code page from the `COFF` file, a fault [illegible] page from the `COFF` file, and a fault on a st [illegible] ne.

   For this step, for refer [illegible] `UserProcess.load` [illegible] age 0, for example, then load the first page of c [illegible] when you handle a page fault you will use the v [illegible] ialize that page: if the faulting address is in t [illegible] ge; if the address is in the data segment, the [illegible] ge, zero-fill it. It is fine to loop through the se [illegible] te section and page to use (assuming it is in the `COFF` file).

   Once you have paged in the faulted page, mark the TranslationEntry as **valid**. Then let the machine restart execution of the user program at the faulting instruction: return from the exception, but do not increment the PC (as is done when handling a system call) so that the machine will re-execute the faulting instruction. If you set up the page (by initializing it) and page table (by setting the valid bit) correctly, then the instruction will execute correctly and the process will continue on its way, none the wiser.

4. Update `readVirtualMemory` and `writeVirtualMemory` to handle invalid pages and page faults. Both methods directly access physical memory to read/write data between user-level virtual address spaces and the Nachos kernel. These methods will now need to check to see if the virtual page is valid. If it is valid, it can use the physical page as before. If the page is not valid, then it will need to fault the page in as with any other page fault.

Testing: As long as there is enough physical memory to fully load a program, then you should be able to use test programs from project 2 to test this part of project 3. See the tips in the Testing section below for how you can control (increase or decrease) the number of physical pages (e.g., `write10` is going to need more than the default of 16 pages). If you give Nachos enough physical pages, you can even run the `swap4` and `swap5` tests (and these tests do not use any system calls other than `exit`).

2. (70%) Now implement demand paged virtual memory with page replacement. In this second part, not only do you delay initializing pages, but now you delay the allocation of physical page frames until a process actually references a virtual page that is not already loaded in memory.

1. In part one for `VMProcess.loadSections`, you allocated physical pages for each virtual page, but you marked them as invalid so that they would be initialized on a page fault. Now change `VMProcess.loadSections` so that it does not even allocate a physical page. Instead, merely mark all the TranslationEntries as invalid.
2. Extend your page fault exception handler to allocate a page frame on-the-fly when a page fault occurs. In part one, you just initialized the contents of the virtual page when a page fault occurred. In this part, now allocate a physical page for the virtual page and use your code from part 1 above to initialize it, mark the TranslationEntry as valid, and return from the exception.

You can get the above two changes working without having page replacement implemented for the case where you run a single prog [illegible] Before moving on, be sure that the two changes a [illegible] y.

Now implement page replac [illegible] ge faults:

1. Extend your page faul [illegible] mory becomes full. First, you will need to select [illegible] ion strategy should be the clock algorithm. Th [illegible] d.
2. Evict the victim page. I [illegible] be used immediately; you can always recove [illegible] irty, though, the kernel must save the page co [illegible]
3. Read in the contents o [illegible] from swap (see below).
4. Implement the swap f [illegible] You will want to implement methods t [illegible] ap (for page out), read from swap to memory [illegible]

As you implement the above [illegible]

- As in the first part of the project, the first time a page is touched it needs to be initialized (e.g., from the executable file for code and data). If this page is subsequently evicted to swap, it will be read from there on further page faults. To be concrete, consider a page used for data. On the first access (fault) on that page, you will read that page in from the executable file. Assume the process dirties the page. When this page gets evicted, you should write it to the swap. If the process faults on the page again, you should read the page in from the swap, not from the executable file (the executable file contains the initial version of the page, the swap contains the most recent version of the page).
- When running multiple processes, the page replacement algorithm may select a victim page to evict from another process. As a result, you will need to update the TranslationEntry in the page table for *that* process, not the faulting one.

- `readVirtualMemory` and `writeVirtualMemory` will need to pin memory. It is possible that a process needs a page, but not only are all pages in use (meaning an eviction must occur), but all pages are pinned (meaning an eviction must not occur *now*). Handle this situation using synchronization. If process A needs to evict a page, but all pages are pinned, block A. When another process unpins a page, it can unblock A. In terms of prioritizing, implement this functionality towards the end after you have general page replacement working.

Finally, you should only do as many page reads and writes as necessary to execute the program, and as dictated by the page replacement algorithm. You will soon discover that the first page fault is different than subsequent ones on a particular page. As described above, on the first fault on a page you need to read from the executable file, and on the second you may need to read from swap. Your implementation needs to be able to handle this situation. In short:

- Read-only `COFF` sections originate in the `COFF` file, and should never appear in swap.
- Read/write `COFF` sections originate in the `COFF` file, but may need to enter swap if modified.
- Other pages should be zero-initialized, and are never read from the `COFF` file.

Tips

- Read Ryan Huang's tip

- There are many poten processes are accessing shared data structure the data structures. This will only be necessary do not worry about it until after you have pa virtual address space exceeds physical mem

- Occasionally check to `proj2` subdirectory.

Testing

- Here are a couple of e testing your implementation:

  - swap4.c: Initializ
  - swap5.c: Initializ

- Begin testing with a ru start implementing swapping, you can control the number of pages (and thus, indirectly, the necessity to swap) with the `-m` paramater to `nachos`:

```
% nachos -m 8 -x swap4.coff
```

  or by modifying `nachos.conf` in the `proj3` subdirectory. Your implementation should not depend upon any particular number of physical pages, but we will always test with at least four physical pages.

- Implement a method to print memory state, and use it liberally when hunting for errors. `Lib.debug()` and `Lib.assertTrue()` will also continue to be helpful.

Code Submission

As with the earlier projects, you do not have to do anything special to submit your project. We will use a snapshot of your Nachos implementation in your github repository as it exists at the deadline, and grade that version. (Even if you have made changes to your repo after the deadline, that's ok, we will use a snapshot of your code at the deadline.)

As a final step, create a file named README in the proj3 directory. The README file should list the members of your group and provide a short description of what code you wrote, how well it worked, how you tested your code, and how each group member contributed to the project. The goal is to make it easier for us to understand what you did as we grade your project in case there is a problem with your code, not to burden you with a lot more work. Do not agonize over wording. It does not have to be poetic, but it should be informative.

Cheating

You can discuss concepts with students in other groups, but do not cheat when implementing your project. Cheating includes copying code from someone else's implementation, or copying code from an implementation found on th mation.

We will manually check and nultiple Internet distributions (if you can find