Project 3

Andrew Quinn

Due: Monday May 24th, 2024 at 11:59 PM.
Learning Objectives:
1. Deepen your understanding of virtual memory.
2. Practice managing complexity in a large system.
3. Practice explaining and justifying computer system designs.

OVERVIEW

The goal of this assignment is to add virtual memory support to PintOS. In the last project, the number and size of processes that you could execute was limited by the physical memory on your emulator. You will remove this restriction in this assignment. This project builds on Project 2. So, bugs from project 2 are likely to crop up again—you should work to fix any such issues ASAP.

REQUIREMENTS

Your submission should include a design document, paging support, and support for a growable stack.

This project will be completed in partnerships. However, you will each submit individually on canvas and will each need to have your group's source code in your individual CSE 134 repository. We suggest that you and your partner choose one of your repositories as "the working repository". You and your partner would do your project development using this repo, pushing at regular intervals as you make progress. Once you are finished, you would push your local repository to the other person's CSE 134 repository. The easiest way to do this is to setup multiple `remotes` in `git`.

One note: we expect to see code commits produced by each partner. It is probably a good idea for you and your partner to "pair program" the assignments. Nonetheless, we will ask questions if the code commits indicate that one person did all of the work.

DESIGN DOCUMENT

Your submission should include a design document that describes key design decisions that you made in your system. The document should be located at `docs/p03.md`. Your design document should include a separate section for each of the other tasks (paging and a growable stack). For each section, you should outline (1) any data structures that you created or extended in your design; (2) any algorithms that you created for your design; (3) any synchronization used in your design; (4) [*new*] any changes that you had to make to your solution from project 2; and (5) a justification of your design (Why is it correct? Why is it fast? etc.). You should aim to have enough detail in your design that a fellow 134 student

would be capable of re-implementing your system by following it. Note: specify "N/A" if your design does not require any of these features.

PAGING

Implement paging for user processes. Your solution should allow a process's pages to be located in memory or in swap (i.e., on a block device). It should promote and evict memory pages as processes on the system use memory. Your solution should provide the following features:

LAZY LOADING. All pages should be loaded lazily. That is, a process should not use memory or block resources until they are required by the process. This means that you should not allocate memory or swap space for the process's executable segments (i.e., the memory allocated in `load_segment (\ldots)` from `pintos/userprog/process.c`) until the process tries to use them.

EFFICIENT EVICTION. Your solution should implement efficient eviction. In particular, your solution should only write an evicted page to swap if that page cannot be created from existing information in the system. For example, your solution should not write any read-only pages to swap.

PARALLELISM. Your solution should allow for parallelism across processes. Namely, if one process requires I/O to handle a page fault, another process that does not require I/O should be able to make progress.

GROWABLE STACK

Implement a stack capable of growing. If a user process tries to access an address that "appears" to be on the stack (e.g., if it is close to the end of the stack), you should allocate a new page for the stack. You should design a heuristic for this task, but keep in mind that you may need to refine the heuristic to pass all of the tests. Your solution should create stack pages as needed, except for the first stack page, which your solution can create when it loads a process. Finally, you can create a maximum stack size, but you should plan on making it relatively large (e.g., the default on Linux is 8 MiB). Stack pages should be evictable.

RUBRIC

We will use the following rubric for this assignment:

Category	Percentage
Testing	60%
Design	40%

TESTING. We will use the provided tests for testing you assignment. You should have 91 tests. If you see more than that, then you should update your repository from the course upstream (by clicking "update fork" on gitUCSC). This will account for 60% of your grade. You can run these tests by executing the following from the `src/vm/build` directory (after you have run `make` from the `src/vm` directory):

```
make check
```

To see the tests as they will be weighted for the final score, execute:

```
make grade
```

DESIGN. We will evaluate your design document based upon the following criteria:

1. **Sufficient:** (30%) Does your design document describe the system with sufficient detail as to be re-creatable by an engineer?
2. **Accurate:** (30%) Does your design document accurately describe the design that you implemented?
3. **Correctness:** (30%) Would your proposed design, assuming it were implemented correctly, satisfy the requirements of the assignment?
4. **Simplicity:** (10%) Is your design simple, rather than overly complex?

HINTS

This section provides design guidance, some tips, and suggestions for getting started.

DESIGN GUIDANCE

This project will require you to manage three resources: virtual memory pages for each process, physical memory frames in the system, and blocks of swap space. We suggest that you create new data structures to manage each of these resources:

SUPPLEMENTAL PAGE TABLE. Pintos provides a page directory that implements the structure expected by the 80x86 architecture. However, this page directory only includes the mapping for pages that are located in physical frames. Your solution will probably need to store more information about the pages that each process uses. We suggest a supplemental page table as a new data structure to store this additional information, with one supplemental page table per process. This structure will mostly useful for the page fault handler: it will include information about the status of all user pages so that your solution can page-in process data. It will also be useful during process termination: it will include information about all allocated resources so that your solution can cleanup data.

FRAME TABLE. In addition to tracking how each process' pages map to each frame, your solution will likely also need the opposite: a global table mapping each frame to the page currently occupying it. This table will be necessary to implement page eviction when your solution processes a page fault.

SWAP TABLE. Your solution will need to store pages that are not in memory in a swap partition. You will likely want a global table to track all of the memory located in swap, which we call the Swap Table. The command line allows you to create a swap partition for PintOS by passing in the parameter `--swap-size=n`. Your code can access the partition through the `struct block*` structure; you can get the structure for the swap partition by calling `block_get_role(BLOCK_SWAP)` from `pintos/src/devices/block.h`. This file also includes functions for getting the size of the device, reading from a block, writing to a block, etc.

TIPS

There are a number of tips for this assignment:

1. Allocate pages for user processes using `palloc_get_page(PAL_USER)`. PintOS provides different memory pools in its allocator; odd things will happen if you allocate user memory from a different pool.
2. You do not need to implement a sophisticated eviction policy. But doing something very poor, such as always evicting the same frame, will not work.
3. You will need to update your solution to validating user memory when you implement paging.
4. Don't forget to modify the hardware page directory, located in `pintos/src/userprog/pagedir.c`, when you process a page fault.
5. You will likely want to create a `status` for each page to track where it is located. Each page can be in one `frame`, indicating that it is in a frame, `swap`, indicating that it is in swap, or `file`, indicating that it is located in the original executable. Why do you need this third status?
6. Think extra carefully about the interface that each of your components will provide. Design *then implement*. Plan to re-design and re-implement iteratively.
7. You will need to use synchronization to ensure correct concurrent access to frame tables, supplemental page tables, and the swap table. It is quite difficult to ensure that your solution is both safe and deadlock free. It is possible to implement safety with carefully considered lock ordering. But, you can also consider "hacks" such as using `lock_try_acquire()`. Use such hacks sparingly.
8. You do not need to page kernel virtual memory—we suggest that you do not try. PintOS currently heavily relies its direct mapping between kernel virtual pages and physical pages, which you would have to change.
9. You can store all new structures in kernel virtual memory, e.g., using PintOS's `malloc`/`free`.
10. Your solution can panic (`assert(0);`) if it runs out of swap space but needs to evict a frame.
11. The 80x86 CPU maintains state that you will find helpful for efficient eviction. When a process reads or writes to a page, the CPU sets an *access bit* for that page in the hardware pagedir to 1. When process writes to a page, the CPU sets the *dirty bit* for that page in the hardware pagedir to 1. The CPU never resets these bits to 0, although your solution can. You can interact with this data by using functions from `pintos/src/userprog/pagedir.h`. See the PintOS site for details.

12. Your solution will need to treat all file system operations as a critical section and use synchronization accordingly. Your solution will not need to treat block operations critical sections.
13. The pintos assignment page includes more suggestions around what to do on each page fault, etc.

GETTING STARTED

There is a fair amount to implement for this project. We suggest the following order:

1. Implement lazy loading. Don't bother with eviction yet. You can terminate your kernel if a process page faults but the system runs out of frames. This will require a fair amount of work: you will need to build a supplemental page table *and* a frame table.
2. Implement stack growth.
3. Implement swapping/eviction.

Check in with what tests you pass as you go. You should currently have a solution that passes a number of the tests that we'll run (e.g., at least all of the Project 2 tests). As you add new features, make sure that you don't introduce "regression" bugs—i.e., bugs where you fail tests that you once passed.