

Workshop 6 - System Calls

Team Cardinal
CSCI 3453 Spring 2017

April 21, 2017

1 Paging

1.1 Overview

Within Pintos, each process has independent pages, which are allocated below virtual address `PHYS_BASE`. The set of kernel pages, on the other hand, are located above `PHYS_BASE` and is global and fixed, remaining the same regardless of what thread or process is active. The kernel has the ability access both user and kernel pages, but a user process is restricted to user pages.

Each virtual page beginning is placed at memory addresses that are multiples of the page size. This strict placement is referred to as page alignment and allows for efficient and fast access to memory.

Paging is a virtual memory management mechanism by which a computer stores and retrieves data from secondary storage (disk, harddrive, etc.) for use in main memory (RAM). Within this mechanism, the operating system retrieves data from secondary storage in continuous blocks of uniform size called pages. This size varies between operating systems, but within the Pintos 80x86 architecture a page is specified as 4096 bytes. Paging is a necessary part of virtual memory since the sum of each active programs required memory exceeds the physical memory available to the system. By using secondary storage, active programs memory requirements can exceed the actual size of available physical memory.

When a program is about to execute, the operating system has to divide its associated memory into pages, and then transfer the pages into RAM to begin execution. To the user, the entire program appears to occupy contiguous space in RAM at all times. In reality, not all pages of a program are necessarily in RAM, and the pages most likely dont occupy contiguous space. When that virtual page becomes inactive or the allocated space is needed for a different page, it is pushed to secondary storage. Poor use of locality and exhaustive use of the physical memory can result in a constant state of paging, which is more accurately called thrashing. This is the process by which the operating system must rapidly exchange pages from secondary storage to continue executing a program. Since the speed of retrieval from disk is orders of magnitude slower than RAM or even cache, the user would notice considerable declined performance, as more time is spent paging than executing.

In P3, Pintos requires that paging only be implemented when a page fault occurs and a program requires a page that isnt currently in virtual memory. Pintos doesnt require that we preempt pages into cache to anticipate need.

Pintos Project 3 contains four subproblems involving paging:

1.2 Segments Loaded from Executables

Pintos is designed to load executable files in a lazy manner. When a process is first loaded from memory there are two ways we can go about loading all the necessary information needed to run the program. First, we could load the process and in addition, load each page that the process will need to access throughout the course of the program execution. An alternate method is load the process and only load associated pages when the process actually needs access to them. The main benefit to lazy loading is that each process experiences less latency when initially starting up since less information is being loaded from the disk. Because of this, more processes are able to be loaded at once. The cost of this is small delays in each program each time a new page is accessed.

The most notable consequence of this design is what signal the operating system has that a new page needs to be loaded into memory. As the program runs, it will encounter a page fault every time it attempts to access a page which has not yet been loaded into main memory. The Pintos kernel is then tasked with intercepting the page fault and loading the appropriate page into memory before the process can continue, evicting a page or pages as necessary to create room in main memory. A major design consideration in Project 3 will be how to handle page faults as they arise during the course of program execution but in general the following actions must be taken by the page fault handler:

1. Look up the page that faulted in the Pintos supplemental page table
 - (a) Verify that the memory reference is in a valid address space
 - (b) Terminate process if access to that page is invalid (read-only, kernel addr. space etc.)
2. Obtain a new frame to store the page in main memory
 - (a) May lead to eviction of page currently loaded in memory
3. Read the data from disk/swap into the newly acquired frame
4. Link the new physical page to the faulty address in the page table to subsequent access can be made

Accomplishing this will require modification of the page fault handler code. The `page_fault()` function is located in `thread/exception.c`. Additionally, functions to handle adding the address to the page table can be found in `userprog/pagedir.c`.

1.3 Page Replacement

1.3.1 Least Recently Used (LRU)

Pintos hardware provides some assistance for implementing page replacement algorithms, through a pair of bits in the page table entry (PTE) for each page. On any read or write to

a page, the CPU sets the accessed bit to 1 in the pages PTE, and on any write, the CPU sets the dirty bit to 1. The CPU never resets these bits to 0, but the OS may do so.

It is important to consider that two (or more) pages can refer to the same frame; this is known as aliasing. When an aliased frame is accessed, the accessed and dirty bits are updated in only one page table entry (the one for the page used for access). The accessed and dirty bits for the other aliases are not updated. In Pintos, every user virtual page is aliased to its kernel virtual page. A design implementation could check and update the accessed and dirty bits for both addresses.

The access and dirty bits are as follows:

PTE_A Bit 5, the accessed bit. PTE_D Bit 6, the dirty bit.

See A.7.4.2 for more details on the composition of the entry.

This algorithm of global page replacement should be designed and implemented as follows:

Maintain a circular list of pages keeping a running count of how often a page is accessed or referenced. Utilize both the accessed and dirty bits to determine replacement policy. A pointer traverses the circular linked list looking for the accessed bit ==0. If the current page accessed bit ==0, then check the dirty bit. If the dirty bit is also equal to 0, replace the page. If the dirty bit is 1, then reset the bit to 0 and then advance the pointer for evaluation at the next page. This policy makes use of the dirty bit as a second chance due to the expensive cost of replacing dirty pages. In `threads/pte.h`, it goes over in greater details convenience functions, defines, and comments on the use of the PTE. It is also important to note that the page replacement algorithm will also be used to manage the frame table.

1.4 Page Fault Synchronization

1.5 Extension of the Program Loader

The core of the program loader must be modified. The program loader is the loop found in the `load_segment()` function.

Each time around the loop:

- `page_read_bytes` receives the number of bytes to read from the executable file
- `page_zero_bytes` receives the number of bytes to initialize to zero, following the bytes read `PGSIZE` is always the sum of `page_read_bytes` and `page_zero_bytes` and is always equal to 4,096

The page handling process depends on the above variables values. Cases and corresponding page handling is as follows:

- If `page_read_bytes` equals `PGSIZE`
 - The page should be demand paged from the underlying file on its first access.
- If `page_zero_bytes` equals `PGSIZE`
 - The page does not need to be read from the disk at all (because it contains all zeroes).

- Create a new page consisting of all zeroes at the first page fault.
- If neither `page_read_bytes` or `page_zero_bytes` equals `PGSIZE`
 - An initial part of the page must be read from the underlying file and the remainder must be zeroed.

Open Questions

1. Should page faults that require I/O be blocked until all non-faulty running processes and non-I/O page faults finish execution, or should their precedence be determined with a round-robin scheduler? In other words, is starvation a concern with page fault synchronization?
2. Why is a "second change" LRU implementation preferential to a FIFO implementation?
3. If a page has already been loaded by one process, how can we ensure that the second process does not page fault and attempt to load that page into memory a second time.

Conclusion

References

- Pintos Source Code
 - `userprog/process.c`
 - * `load_segment()`
- Operating Systems: Principles and Practices Second Edition by Thomas Anderson and Michael Dahlin
- Pintos Manual by Ben Pfaff