Filesystem Upgrades, Swap Memory, and a Page Daemon in Weenix

Luke Primis

Department of Computer Science
Brown University

A thesis submitted in partial fulfillment of the requirements for the degree of Applied Math-Computer Science, Bachelor of Science with Honors

Advisor: Professor Thomas Doeppner
Second Reader: Professor Malte Schwarzkopf

April 2024
Abstract

For my honors thesis, I made several substantial changes to Weenix’s memory management and filesystem. These upgrades include completing Professor Doeppner’s recent s5fs caching changes, a B-tree implementation in kernel with various applications, swap space at the vfs and s5fs level, and finally a page daemon kernel process. I provide some metrics to demonstrate the performance of these changes. The repository containing the full Weenix operating system along with my final changes can be found here.

Contents

1 Background 1
1.1 Weenix 1
1.2 S5FS 1
1.3 Mobj and Pframe 2
1.4 Swap Memory 2
1.5 Page Daemon 2

2 Design 2
2.1 Caching 2
2.1.1 S5FS 2
2.1.2 Memory Object 3
2.2 Page Daemon 3
2.2.1 Detecting Page Access 3
2.2.2 Reverse Shadow Tree 3
2.2.3 Two-Handed Clock Algorithm 5
2.3 Swap Memory 5

3 Implementation 6
3.1 Filesystem Caching Changes 6
3.2 B-Tree 8
3.3 Swap Memory 8
3.3.1 VFS Layer 8
3.3.2 S5FS Layer 11
3.3.3 Anonymous and Shadow Objects 11
3.4 Page Daemon 11
3.4.1 Building the Reverse Shadow Tree 11
3.4.2 Checking Access 12
3.4.3 Unmapping 12

4 Results 19
4.1 Time in Weenix 19
4.2 B-Tree Performance 19
4.3 Page Daemon and Swap Memory 21

5 Conclusion 22
5.1 Reflection 22
5.2 Future Work 22
5.3 Acknowledgements 23

1 Background

1.1 Weenix

Weenix is an x86, 64-bit Unix-like operating system used by Brown’s CSCI 1670 and CSCI 1690 courses to teach operating systems concepts [1]. Students are tasked with implementing processes, basic drivers such as the TTY, the Virtual Filesystem (VFS) layer, S5FS to act as the backing filesystem, and finally virtual memory to allow for userspace processes.

Weenix has seen many changes, iterations, and upgrades over time. For example, it originally was written as a 32-bit OS. There was a major effort to upgrade Weenix to a 64-bit address space, which is its current state. Most recently, Professor Doeppner began overhauling functionality in S5FS, caching blocks differently in memory after they were read from the disk. My work began here, finishing Professor Doeppner’s changes and adding further optimizations to the filesystem. Following this work, I began designing a swap memory system for Weenix, along with a page daemon to utilize the new swap space.

1.2 S5FS

For students developing Weenix, at first it uses a in-memory filesystem called RAMFS that actually doesn’t interact with the disk. Thus, changes are not saved after halting Weenix. Only after implementing S5FS, Weenix’s true filesystem, are changes made persistent. S5FS is Weenix’s implementation of the
System V Filesystem, which uses inodes as representation of files and directories [2]. The inode structure stores the necessary information mapping file blocks to disk blocks which is the core of the filesystem.

1.3 Mobj and Pframe

The mobj (memory object) and pframe (page frame) data structures compose the core of Weenix’s memory management in the kernel. mobj’s effectively represent collections of pages. The pages in question are represented by pframes, which store kernel virtual addresses allocated from the PHYSMAP region, thus having a 1:1 mapping to physical memory.

The memory objects are used throughout the kernel to represent vnodes, anonymous objects, shadow objects, and some other less common types. Memory objects have a linked list containing pointers to all of the page frames that it holds.

1.4 Swap Memory

Swap memory, also referred to as swap space, is a feature common in modern operating systems where the kernel, when experiencing significant memory load, moves pages of memory onto the disk. When processes need to use the swapped-out pages, the operating system loads them back in from disk. Effectively, the disk space is utilized to extend the amount of available memory, albeit at the performance cost of the disk’s being significantly slower than RAM. Swap space is only used for privately mapped or anonymous mobj’s; any files that are share-mapped are already backed to disk by their vnode and its according underlying filesystem.

There are different methods for determining which pages should be paged out to disk. One approach, first-in-first-out (FIFO), frees pages which have been in memory the longest. However, this ignores the usage of the page; the oldest page may be the most frequently used one. One can consider many different examples, such as executable or text sections of long-running programs, to convince themselves this is the case. Other, more nuanced approaches generally follow some variation of the least-recently-used (LRU) principle, paging out based on usage rather than time in memory alone [3, p. 301-315].

Prior to this project, Weenix lacked any swap memory functionality. Thus, it would have been unable to page-out anything that was not backed to disk: pages belonging memory objects other than vnodes. In order to page-out pages from anonymous and shadow objects, Weenix needs swap memory.

1.5 Page Daemon

The page daemon is the kernel thread responsible for determining which pages to swap out to disk. It also determines when to do so, usually by tracking memory usage and taking action if it is above some critical threshold.

The page daemon alleviates memory pressure on the system, allowing for processes to continue operation even under high RAM usage. Weenix also lacked any page daemon implementation prior to this project, meaning all page frames, even ones backed to disk that could be flushed safely, remain in memory when usage is very high.

2 Design

2.1 Caching

2.1.1 S5FS

The caching changes to S5FS were designed and largely implemented by Professor Doeppner; I simply ironed out some edge cases and completed the implementation. However, since I did spend a significant amount of time working in the completion of this effort, I outline here the design and thought process behind the changes. The implementation section details further what I specifically changed to complete this upgrade.

Prior to Professor Doeppner’s changes, caching of disk blocks in memory was occurring at the block device level. The new upgrades now change the caching scheme. Each inode now caches its pages that have been read into memory in its own memory object, which is pointed to by its vnode. Blocks that are
used for the operation of the filesystem, such as indirect blocks, are now cached inside the memory object of the filesystem itself, as they are considered "meta" disk blocks.

The change is largely a matter of design more than performance. Prior, the caching of meta disk blocks was somewhat of a black-box, particularly to students, since they wouldn’t have had much reason to investigate and understand the caching at the block device level. Now, students really have to understand how the caching works at the S5FS level and what distinguishes "meta" disk blocks from normal file disk blocks.

### 2.1.2 Memory Object

If a vnode’s underlying mobj doesn’t contain a specific pframe, we read it from disk then store it in the mobj’s cache of pframes. If our cache is a linked list, checking the cache necessitates iterating over the entire list of pframes. Clearly, this is an $O(n)$ operation with respect to the number of pframes we currently have stored in our mobj, which is inefficient in cases of large files, for example, where many of the blocks have been read from the disk into pframes.

The initial goal of optimizing this search operation was for the purpose of large files; however there are actually numerous other places throughout Weenix where the mobj’s cache of pframes is checked, such as in the pagefault handler and traversing shadow chains.

![pframe diagram](image)

**Figure 1:** pframes are stored in a linked list in the mobj

We implement a B-tree data structure in the Weenix kernel in order to optimize the search to be in sub-linear time. It’s keys are page numbers, and the values are pointers to pframes.

Additionally, the B-tree is useful in other parts of the kernel, as it can serve as an efficient search tree to be used as a cache in some other scenarios. Weenix lacks data structures such as hashsets or hashmaps, and the B-tree provides a simpler alternative that is also capable of sublinear-time search operations. The implementation section will go into more detail.

### 2.2 Page Daemon

#### 2.2.1 Detecting Page Access

The format for x86, 64-bit pagetable entries (PTE’s) includes an ACCESS bit as the 5th bit. Whenever a PTE is accessed by the memory management unit (MMU) either through translation or TLB lookup, the ACCESS bit is set to 1. The hardware never sets this bit to 0; it defers to the software to use and clear the bit as it chooses, if at all. In the case of the existing Weenix code, it actually never uses the ACCESS bit. Thus, it is perfectly safe for the page daemon to check and clear this bit, making it perfect for our use case and incredibly convenient as the MMU already handles the setting of this bit. Furthermore, the ACCESS bit is in the same index for PML4, PDP, and PD entries; thus, we can use the same method to check access regardless of whether the virtual address in question resides in a 4KB, 2MB, or 1GB page.

![PTE format](image)

**Figure 2:** Format for an x86, 64-bit PTE

#### 2.2.2 Reverse Shadow Tree

A critical design issue for the page daemon is the ability to find every single virtual address that maps to a given a pframe’s physical page in memory. We can see this issue come to the forefront of both of the page daemon’s primary tasks: checking access of pages and flushing those which haven’t been accessed.

To illustrate the issue, and arrive at the solution, we begin with important context for Weenix’s memory management system. In Weenix, copy-on-write is implemented using shadow objects. As processes fork, a shadow chain grows. Upon a pagefault, if it is for a write, the pagefault handler traverses the shadow chain downward in search of the correct page...
The problem here is, as established above, there is a one-to-one mapping between vmarea to pframe, but there is a one-to-arbitrarily-many mapping of a given pframe to pagetable entries (PTE’s) that map to it. Thus, we need to invalidate every single mapping that maps to the page we are freeing; otherwise, userspace processes will use the PML4 and TLB which translate to an old physical address of a page which has been freed. In other words, we must ensure that another pagefault is triggered for any page we are freeing the next time any process tries to access it.

Figure 4: New reverse shadow tree provides bottom up access from root of a shadow chain

Since we only have shadow chains that point downward, we would have to iterate over every process' vmmap, for every vmarea, and traverse its shadow chain to both determine if it should be unmapped as well as eventually unmap it. This algorithm would not only be very complicated to execute but also quite inefficient. The page daemon already has to iterate over every pframe of every mobj; now, we would be iterating over every vmarea, of every process’ vmmap, for every pframe of every mobj. To add onto this monstrosity of nested iteration, we also lack access to many common data structures that we would likely need to make this algorithm somewhat efficient, such
as vectors, hashmaps, and hashsets, due to being in the Weenix kernel.

To solve this, we add a new data structure: the reverse shadow tree. Where, before, shadow objects only maintained a pointer to the object that it shadows, now every memory object (whether it be vnode, shadow, or anonymous) maintains a linked list of every object that shadows it. We now essentially create an n-ary tree, where a node is a mobj, and each branch is the reverse of a shadow chain which leads to that said node. It is easy to maintain, since we simply append or remove from a linked list whenever we modify the shadow chain. The final missing piece is to give mobjs knowledge of all of vmareas that they back. Now, iteration over mobjs allows us to check which, if any vmareas they back directly. Crucially, by executing a tree search algorithm on all shadow chains terminating at this node, we can now find any vmareas backed by upstream mobjs, which the current node indirectly backs by nature of being downstream of that shadow chain.

### 2.2.3 Two-Handed Clock Algorithm

The two-handed clock algorithm allows the page daemon to relatively easily implement an LRU page out policy. The front hand simply iterates over all of the pframes by iterating over all mobjs, and clears the access bit of all PTEs mapping to this specific page-frame using the previously described reverse shadow tree.

The back hand checks if the ACCESS bit is set to 1 for any PTE mapping to this page. If any of the bits are set, then the page is left alone. If every single PTE has its ACCESS bit clear, that indicates that no process has accessed this page since the front hand cleared all of the ACCESS bits of PTEs that map to it. In that case, we free the pframe, writing it out if it’s dirty either using swap space or normal filesystem functionality if the pframe is from a vnode.

The specifics of checking and clearing the ACCESS bits are left to be discussed later in the implementation section.

### 2.3 Swap Memory

The final component to complete the page daemon is to give Weenix swap space functionality. The swap functionality is be implemented at the VFS level. By making swap a VFS-level abstraction, there is significant flexibility to iterate and experiment with different swap systems moving forward. While I also include a swap implementation using S5FS, one could easily move swap functionality away from being managed by the filesystem and create a wholly different system for swap space. Additionally, it allows for flexibility if one were to implement a new filesystem entirely for Weenix. If they choose not to implement swap space at all, the page daemon will see that the VFS layer doesn’t have this ability and opt not to page out anonymous or shadow objects.

As for the S5FS specific swap space I implemented, it follows a lazy allocation scheme. At the time when a page from a shadow or anonymous object is set to be paged out, we will allocate an inode and write the page to the file.

When a pagefault occurs, now, each mobj simply
checks if a page is stored in swap prior to following its normal procedure for getting a pframe. If it's in swap, then the page will be read from the disk and the block it was stored on will be freed so as to preserve filesystem resources.

3 Implementation

3.1 Filesystem Caching Changes

The completion of the changes to the filesystem was largely a matter of identifying edge cases and ironing out the implementation to be more robust and fully functional. The first group of these changes were locking fixes to avoid deadlocks in certain edge cases. While somewhat time-consuming to debug, these changes were all fairly straightforward conceptually and were more a task of identifying which threads were causing the locking issues and where they were being caused.

The next wave of changes had to do with saving filesystem changes properly upon halting. Weenix only synchronizes its filesystem, writing out the super block to disk, when it halts. First, I re-implemented the s5fs_sync function to properly use the new system of meta disk blocks. Prior, while the rest of the filesystem was using the new caching method, s5fs_sync would still fetch the super block from disk, as it was not hitting the meta block cache in the filesystem’s memory object. Thus, it would simply get the original super block then write it back out, and none of the new files would be present after restarting Weenix.

Next, I identified an issue with pframes’ not being properly dirtied if they were fetched from the cache. When a pframe was originally created, it was marked as dirty if it was being created for a write. Then, it would be cached in the memory object as previously described. The issue here is that if the operating system initially created a pframe for reading, it would be cached with its dirty bit cleared. Then, a subsequent write would hit the cache, but the dirty bit would still be clear. Thus, the changes would not be written out to disk. The solution was simply to set the pframe’s dirty bit upon a cache hit if the forwrite argument of the function hitting the cache was set to 1.

The final change had to do with an edge case only experienced when using most of the system’s available inodes. Basically, if the filesystem was using a disk block as a meta disk block, it would cache it under the filesystem’s memory object if it was fetched from disk. However, if that block was later freed, it wasn’t being properly removed from the cache. If the block wasn’t reused, then it wouldn’t cause any problems upon halting and restarting Weenix. However, if it was being reused, then the filesystem would write some data to the block, then upon halting, s5fs_sync would overwrite that block number on disk with whatever data was currently being stored at the address that pframe pointed to, which would be garbage since it had been freed. This would corrupt the data on that block, thus making the state of the filesystem invalid after halting and restarting Weenix. This was most easily observable after a stress test: for example, running the S5FS unit test that uses all of the inodes in the system then subsequently creating a file or directory. Using so many inodes caused many blocks to be stored as meta disk blocks. The failure to remove these from the cache overwrote and corrupted the files and directories created after the stress test.

To solve this issue, I created a new helper function to cleanup and destroy a pframe while ignoring the dirty bit; simply calling the existing functions to free the pframe from the S5FS mobj’s cache would have unnecessarily written some data out to disk when really we just want to destroy and ignore the cached data for that block. Using the new function easily removed pframes representing freed disk blocks from the cache, ensuring s5fs_sync only wrote out correct data to disk and preserving the sanity and state of the filesystem.

After implementing all of these changes, the transition to the new caching system was complete. S5FS was then completely functional both for CSCI 1670 and CSCI 1690 as well as for the rest of my thesis.
static void s5fs_sync(fs_t *fs)
{
    s5fs_t *s5fs = FS_TO_S5FS(fs);
    mobj_t *mobj = &s5fs->s5f_mobj;

    pframe_t *pf;
    s5_get_meta_disk_block(s5fs, S5_SUPER_BLOCK, 1, &pf);
    memcpy(pf->pf_addr, &s5fs->s5f_super, sizeof(s5_super_t));
    s5_release_disk_block(&pf);

    mobj_lock(&s5fs->s5f_mobj);
    mobj_flush(mobj);
    mobj_unlock(&s5fs->s5f_mobj);
}

Listing 1: s5fs_sync now properly writes the super block back out to disk

static long s5fs_get_pframe(vnode_t *vnode, uint64_t pagenum, long forwrite,
                               pframe_t **pfp)
{
    // ...
    mobj_find_pframe(&vnode->vn_mobj, pagenum, pfp);
    if (*pfp)
    {
        // block is cached
        (*pfp)->pf_dirty |= forwrite;
        return 0;
    }
    // ...
}

Listing 2: We ensure cached pframes are properly dirtied
3.2 B-Tree

The B-tree implementation is fairly standard, except more verbose than common versions in languages like Python or C++ due to the lack of classes in C. Furthermore, as a kernel data structure, it does need to work within Weenix’s memory management system. Namely, we need to use a slab object allocator in order to dynamically allocate and free nodes as the B-tree grows and shrinks. As can be seen from the header file for the B-tree, it provides a standard API for insertion, deletion, and search. I omit the entire implementation since, as previously mentioned, it is standard and quite dense [4] [5] [6, pp. 497-513]. It stores void *’s as the values, so it is useful for storing any 8-byte datatypes.

One important detail is the BRANCHING FACTOR parameter. The branching factor of the B-tree dictates several important properties of the data structure. On the node level, it determines the maximum number of keys and children that a node can have. Let \( t \) be the branching factor of the tree: the maximum number of keys is equal to \( 2^t - 1 \), thus making the maximum total number of children equal to \( 2^t \). Furthermore, we can place a bound on the height of the B-tree: \( h_{\text{max}} = \lceil \log_t \left( \frac{n+1}{2} \right) \rceil \) [7, 179]. This bound gives us important properties. Since the B-tree is balanced, all leaf nodes are at the same height. Thus, when combined with the fact that the maximum number of keys is constant, we are guaranteed that the average runtime complexity of insertions, deletions, and searches are asymptotically logarithmic.

The results section elaborates on the implications of the time complexities of the B-tree operations. One note I would like to emphasize here, however, is that the branching factor can be tuned for performance of the system depending on the needs. Larger branching factors result in shorter trees, since more data can be stored in each node. However, this makes each node take up more memory, as well as increasing the time to search each node.

The main usage of the B-tree is as the cache of pframes in each mobj. Once the B-tree was implemented, it was simply a matter of maintaining the B-tree as pframes were created or removed as well as taking advantage of the B-tree for search operations, particularly in mobj_find_pframe which is used extensively throughout the filesystem as well as virtual memory implementation.

3.3 Swap Memory

3.3.1 VFS Layer

At the VFS level, I implemented swap such that it is fully abstract and independent of the filesystem’s underlying operations. The functions relevant to swap have been added as filesystem operations to the fs_t’s fs_ops field. Weenix makes use of this “Object Oriented C” design paradigm extensively. Where a C++ class could simply have methods and polymorphism, instead Weenix data structures, such as the fs_t or mobj_t types, have function pointers to operations.

Specifically, we implement write_swap, read_swap, and cleanup_swap. The write and read functions simply take in a memory object as well as the number of the page the caller wishes to swap out or read back in. Currently, the writing functionality is only used in the page daemon and the read function is only called upon a pagefault, although further development on Weenix could easily use these functions elsewhere since there is a globally accessible vfs_root_fs which can be used to call the VFS level swap functions. The cleanup function is called on mobj destruction to ensure filesystem resources that were dedicated to pages stored in swap are properly freed.

![Figure 6: The B-Tree stores addresses of the pframes’ indexed by their page numbers, allowing us to jump to the pframe we found by searching: in this example the pframe for page number 3](image-url)
#pragma once

#include "kernel.h"

/*
 * Standard btree implementation based on the following and Python implementation:
 * Introduction to Algorithms (Cormen, Leiserson, Rivest, Stein)
 * https://github.com/msambol/dsa/blob/master/trees/b_tree.py
 */

/*
 * Branching factor determines certain bounds on the size of our tree as follows:
 * Let t be our branching factor.
 * Lower bound: each node has at least t-1 keys, at least t children
 * Upper bound: each node has at most 2t-1 keys, at most 2t children
 * Height of the tree is bound by \log_t(n + 1) / 2 where n is the number of nodes
 * Change it as needed for performance
 */

#define BRANCHING_FACTOR 2
#define MAX_KEYS (2*BRANCHING_FACTOR) - 1
#define MAX_CHILDREN 2*BRANCHING_FACTOR

/*
 * btree_node is the core of our rough btree implementation here
 * - n_keys keeps track of the number of keys (out of the max 2t-1) we have on this node
 * - n_children keep track of the number of children
 * - keys is the array of keys
 * - data is the array of addresses corresponding to keys
 * - children is the array of children for this node
 * - data is an array of void*, but you can use this to store any 8 bit data (pointer to anything)
 * and just cast
 * Note that keys and data are indexed the same. For the pageframe example, keys[i] is
 * the pagenum of the i’th pageframe, and data[i] is the address of the i’th pageframe
 * where i is the index of the pageframe stored in this node. In this example, i has
 * absolutely nothing to do with the pageframe’s index in the mo_pframes linkedlist
 */

typedef struct btree_node
{
    unsigned int n_keys;
Listing 3: `btree.h` provides a standard B-tree API as well as some useful definitions. The full implementation is omitted for brevity.

```c
void mobj_find_pframe(mobj_t *o, uint64_t pagenum, pframe_t **pfp)
{
    *pfp = NULL;
    KASSERT(kmutex_owns_mutex(&o->mo_mutex));
    pframe_t *pf = (pframe_t *)btree_search(o->mo_btree, pagenum);
    if (pf != NULL)
    {
        kmutex_lock(&pf->pf_mutex);
        pf->pf_used = 1;
        *pfp = pf;
        return;
    }
}
```

Listing 4: `mobj_find_pframe` now leverages the B-tree, providing quicker search to many functions that call it.
3.3.2 S5FS Layer

For the swap implementation at the S5FS level, I kept it fairly simple. For each mobj, we simply allocate an inode if we need to write a page out to swap. One small detail here is that we do not link the file being used as swap space anywhere else in the filesystem. Thus, it is unavailable to userspace programs and safe to be used by the kernel for this purpose. However, since it is not linked by the filesystem, we make sure to store the inode number, and manually set its linkcount to 1 so that S5FS does not delete the file when we put it away. When writing a pframe to swap space, it is always written to the file at position PAGE_SIZE * pf->pf_pageno. Also, we always write all PAGE_SIZE bytes of the page out. Thus, when reading back in, we know what position in the file it will be stored at based on the page number, so we read it back in using the existing S5FS functionality. Afterwards, we make sure to free the utilized blocks and shorten the file length if necessary so as to not leak filesystem resources.

By using mostly existing S5FS functionality for the underlying swap space, I kept the swap implementation simple enough to be easily built upon, while sufficiently robust for the page daemon to effectively utilize.

3.3.3 Anonymous and Shadow Objects

Anonymous and shadow objects are the only memory objects that interact with swap, since vnode's are already backed to disk. To facilitate swap space usage, we reuse the B-tree from earlier. Now, whenever we page out to swap, we store the page number in a B-tree pointed to by the mobj. This way, we can use it as a cache to quickly check if a page is stored in swap space. Upon paging back in, we simply remove it from the B-tree. Additionally, we store the inode number of the file created for swap in the B-tree under the key S5_SWAP_INO so that the S5FS layer functions can access the file for reading and writing. In our implementation, we define this key as −1, since we know the rest of the tree only stores page numbers which are guaranteed to be non-negative.

For anonymous objects, integrating swap is straightforward. Now, in addition to checking the existing cache whenever we call anon_get_pframe, we also must check the swap B-tree and read from swap if it is stored there.

For shadow objects, there are substantially more cases we must consider. When calling shadow_get_pframe, shadow_fill_pframe, or shadow Collapse, we must now check the swap B-tree for the page in question as we traverse the shadow chain. At any given time when traversing the shadow chain, it’s possible that the page we are searching for has actually been paged out. This detail is absolutely crucial as processes fork and the shadow chain becomes increasingly complex, as missing a page that has been brought up to a shadow object due to copy-on-write can result in entirely corrupted stack spaces for forked processes.

3.4 Page Daemon

3.4.1 Building the Reverse Shadow Tree

There are several key components to maintain in order for the page daemon to have access to all necessary and relevant data structures. First, we create a global mobj list and a corresponding mutex to handle concurrency. Whenever a mobj is created or destroyed, the mutex is locked and it is appended to or removed from the list. When the page daemon gets processor time, it locks the mutex and iterates over all of the mobj’s in the system. However, there is one complication that necessitates a slightly unusual locking scheme to ensure thread safety. Since vnode’s are a type of mobj, we must lock the mutex when creating or destroying them. Also, there is a global list of vnodes as well as another corresponding mutex for that list. Since the page daemon creates and destroys vnodes when writing out to swap space, we establish a locking order of first locking the mobj_list_mutex and then locking the vnode_list_mutex. However, there is one specific case to consider; since the page daemon thread will only ever call vget and vput with the mobj_list already locked, we implement helper functions to lock and unlock the mobj_list_mutex if and only if the current thread is not the page daemon. Using these functions and the previously mentioned
locking order, the page daemon as well as any other threads that are creating vnode and other types of mobjs can ensure the thread-safety of the global list as well as avoid deadlock.

Expanding on the topic of concurrency, the page daemon also relies on the shadow chains and the global list of processes, which previously were never used in a multi-threaded context. Other threads’ modifying their own shadow chains or these global lists can disrupt kernel state that the page daemon relies on. Thus, I also implemented global locks for any modification of a shadow chain or the global list of processes. Since the modification of any shadow chain now impacts the reverse shadow tree, which is a global representation of all virtual memory, the state of it must remain static while the page daemon does one full iteration over all mobjs.

After establishing the global mobj list, each mobj needs to maintain a list of all vmareas which it directly backs. The vmarea also provides a pointer to the vmmmap and eventually the process to which it belongs. With access to the process, the page daemon can access the relevant PML4. Thus, with the one-to-many relationship of mobj to vmarea now maintained, the page daemon can access all relevant information in order to check access, calculate the virtual address, and unmap it in the PML4.

Finally, to actually traverse up all of the shadow chains, we ensure that wherever the shadow chain is modified, we append or remove the shadow object from the shadowed mobj’s list of mobjs that shadow it. In concept, this doesn’t sound that complicated, but in practice it necessitates extreme caution. The sanity of the reverse shadow tree is essential to ensure that it is a concrete and exhaustive source of truth for the page daemon to locate all possible locations that are mapping a pframe. For checking access, this is important so as to not page out pframes that are actually being actively used. If the page daemon were to miss one location that indicated an access, it wouldn’t be absolutely detrimental; a fault would simply trigger the page in question to be paged back in. In the case of unmapping, however, being able to unmap every single mapping is absolutely crucial. Failure to do so has fatal consequences for the system. For example, failing to unmapped mappings to executable code or stack space will make it impossible for userspace processes to run.

3.4.2 Checking Access

As previously established, we can utilize the accessed bit on the x86, 64-bit PTE format to check if an individual entry has been accessed. In order to check every possible mapping to a specific page, the page daemon executes a recursive depth-first-search of the shadow tree starting from the mobj that owns the pframe we are trying to free. The base case for this search is finding a shadow object which has a pframe with the same number as the one we are trying to check; this indicates that copy-on-write has occurred, and any shadow objects upstream from the current one in recursion are not mapping to the pframe being paged out. As the page daemon traverses the tree, it clears every ACCESS bit: the front-hand functionality of the two-handed clock algorithm.

There is one other location that needs to be checked for accesses. In addition to the userspace processes represented by the reverse shadow tree, it’s also possible that a process has accessed the page in kernel space. To account for this, the page daemon iterates over the list of all processes and checks and clears their bits after executing the recursive algorithm for the tree.

One important detail to consider is that even after one PTE is found to have an been accessed, the page daemon would consider this page to have been recently used and thus not page it out. Therefore, it may seem convenient to simply end the recursion or skip over checking the kernel PTE’s once the first accessed PTE is found. However, we must recall the overall goal of the front-hand of the two-handed clock algorithm. It is responsible for clearing every ACCESS bit. Thus, the recursion must continue to completion and the kernel mappings must be cleared even if an ACCESS is found to be set early on in the recursion.

3.4.3 Unmapping

The recursive algorithm for unmapping is identical to checking access, but instead of clearing bits on the PTEs it calculates virtual addresses using the vmarea
fields and the page number being paged out then un-maps them in the processes’ PML4s. Unlike checking access, kernel virtual address mappings are not considered, as these should be maintained in a 1-to-1 mapping with the PHYSMAP region and thus are never unmapped. Kernel pagefaults are not supported in Weenix, so the case of a page having been paged out should already be handled by checking the mobj’s pframe cache anywhere relevant in kernel.

There is one very important reason that unmapping must occur before writing to disk. Threads performing disk operations yield processor control and are woken up by an interrupt upon completion of the operation. This reveals the potential for a nasty edge case: Let Thread A be the page daemon thread, and let Thread B be some userpace process. Thread A tries to page out page n at time t₀, causing it to yield to Thread B due to the initiation of a write. Thread B now modifies some data in page n at time t₁, then yields back. Thread A wakes up from the write operation and frees the pframe from memory. Some time passes, and Thread B faults on the virtual address backed by page n. However, it reads from disk and retrieves the data from time t₀, since that is when Thread A initiated the write operation, instead of the correct state of the page at t₁.

After seeing this complicated race condition, it may seem naive to assume that simply unmapping prior to initiating a write operation would fix it. However, due to the fact that the page daemon holds the mobj’s mutex at the time of initiating the write, it follows that the pagefault handler would simply be forced to wait on that same mutex, thus avoiding this race condition and preserving the sanity of the state of memory.

3.4.4 Checking Memory Usage

Checking memory usage is trivial since Weenix’s page allocator keeps track of the total number of available pages as well as the number of pages remaining. The page daemon simply calculates

\[
1 - \frac{\text{page\_free\_count()} \times 100}{\text{page\_total\_count()}}
\]

to get the percentage of pages being used. Note that the numerator is multiplied by 100 to get the percentage as an integer, since floating point operations are not possible in the Weenix kernel.

To ensure that the page daemon is run more frequently when memory pressure is high, the pagefault handler now checks memory usage and yields to the page daemon if it is passed the defined critical threshold.

3.4.5 Prioritization

Weenix lacks any notion of thread priority, but the page daemon needs a way to be prioritized if memory pressure is high. To accomplish this, I implemented functions that simply move a thread to the front of the run queue. Now, if memory pressure is past the set threshold, the pagefault handler will prioritize the page daemon thread then immediately yield the processor, ensuring that the page daemon is next to run.

3.4.6 Main Page Daemon Loop

The main logic of the page daemon, after implementing all of the previously outlined functionality, is straightforward. It loops until it is cancelled, iterating over all of the anonymous, shadow, or vnode mobjs in the system. For each mobj, it checks if each pframe has been accessed by any thread, clearing the ACCESS bits as it checks. If it hasn’t been accessed, it pages it out either using the downstream filesystem functionality for vnodes or swap space for shadow and anonymous objects.
static void *pagedaemon_run(long arg1, void *arg2)
{
    while (!paged_thread->kt_cancelled)
    {
        running = 1;
        uint64_t cur_count = swapo_count;

        /*
         * even though pagefault handler checks, we shouldn't really run the
         * pagedaemon
         * if it naturally gets processor time but we aren't in need of its
         * services
         */

        if (mem_usage_check(CRITICAL_THRESHOLD))
        {
            kmutex_lock(&mobj_list_mutex);
            list_iterate(&mobj_list, mo, mobj_t, mo_link)
            {
                if (curthr->kt_cancelled)
                {
                    kmutex_unlock(&mobj_list_mutex);
                    goto ret;
                }
                // trylock so we just move onto freeing other mobj's pages. Not
                // worth hanging this thread
                // on a mobj thats currently being used
                if (mo->mo_type != MOBJ_FS && mobj_trylock(mo))
                {
                    list_iterate(&mo->mo_pframes, pf, pframe_t, pf_link)
                    {
                        if (curthr->kt_cancelled)
                        {
                            mobj_unlock(mo);
                            kmutex_unlock(&mobj_list_mutex);
                            goto ret;
                        }
                        KASSERT(mo->mo_type == MOBJ_VNODE || mo->mo_type ==
                                MOBJ_SHADOW || mo->mo_type == MOBJ_ANON);
                        kmutex_lock(&pf->pf_mutex);

                        // check if anything above this in shadow chain(s)
                        // accessed this page
                        int accessed = shadow_tree_dfs_accessed(mo, pf->pf_pagenum);

                        // check if any process accessed it in kernel space
                        list_iterate(&proc_list, p, proc_t, p_list_link)
                        {

```
if (p->p_pid != PID_PAGEOUTD) {
    pte_t *pte = pt_get_pte(p->p_pml4, (uintptr_t)pf->pf_addr);
    accessed |= ((*pte) & PT_ACCESSSED);
    (*pte) &= ~PT_OBJECT;
}

    // if it's been unused by any possible process in either
    // user or kernel space
    // we page it out
    if (!accessed) {
        /*
        * unmap first - we yield the processor when writing out to disk
        * so we need to handle the edge case of another process trying to
        * access what we're currently paging out while the pagedaemon yields
        * during the disk operation. In order to handle this another pagefault
        * has to occur, which will wait on this mobj's lock while our daemon
        * thread finishes up. See pagefault.c to understand how its locking
        * scheme combines with this file's locking scheme to ensure this.
        */
        shadow_tree_dfs_unmap(mo, pf->pf_pagenum);
        size_t pf_pagenum = pf->pf_pagenum;
        if (mo->mo_type == MOBJ_VNODE) {
            // already backed by a file; downstream logic
            handles this already
            list_iterate(&proc_list, p, proc_t, p_list_link)
            
            { pt_unmap_range(p->p_pml4, USER_MEM_LOW,
                USER_MEM_HIGH);
            }
            mobj_free_pframe(mo, &pf);
            paged_write_count++;
        } else if (mo->mo_type == MOBJ_ANON || mo->mo_type == MOBJ_SHADOW) {
            KASSERT(mo->mo_type == MOBJ_SHADOW || mo->mo_type == MOBJ_ANON);
            // swap/anon objects aren't backed to disk - use
            swap if provided by VFS
            list_iterate(&proc_list, p, proc_t, p_list_link)
            {
        }
pt_unmap_range(p->p_pml4, USER_MEM_LOW, USER_MEM_HIGH);
}

tlb_flush_all();
swapo_count++;

if (vfs_root_fs.fs_ops->write_swap)
    vfs_root_fs.fs_ops->write_swap(&vfs_root_fs, mo, pf);
else
    kmutex_unlock(&pf->pf_mutex);
}
mobj_unlock(mo);
}
kmutex_unlock(&mobj_list_mutex);

if (cur_count != swapo_count)
    paged_log();
    running = 0;
sched_yield();
}
ret:
/*
 * setting running to 1 here ensures the pagefault handler no longer
 * tries to prioritize and yield to the page daemon
 */
    running = 1;
    return;
}
static void bruteforce_unmap(mobj_t *mo, pframe_t *pf)
{
    list_iterate(&proc_list, p, proc_t, p_list_link)
    {
        KASSERT(p->p_vmmmap);
        list_iterate(&p->p_vmmmap->vmm_list, vma, vmarea_t, vma_plink)
        {
            // this pagenum could be mapped to by this vma
            if (pf->pf_pagenum - vma->vma_off < vma->vma_end - vma->vma_start)
            {
                mobj_t *mobj = vma->vma_obj;
                int found = 0;

                // descend the shadow chain of its vma_obj
                while (mobj != mo && mobj->mo_type == MOBJ_SHADOW)
                {
                    mobj_shadow_t *so = MOBJ_TO_SO(mobj);
                    int found = mobj->mo_btree && btree_search(mobj->mo_btree, pf->pf_pagenum);
                    found |= mobj->mo_swap && btree_search(mobj->mo_swap, pf->pf_pagenum);
                    if (found)
                    {
                        break;
                    }
                    mobj = so->shadowed;
                }

                if (mobj == mo && !found)
                {
                    uintptr_t vaddr = (uintptr_t)PN_TO_ADDR(pf->pf_pagenum - vma->vma_off + vma->vma_start);
                    pt_unmap(p->p_pml4, vaddr);
                }
            }
        }
    }
}

Listing 6: The top-down bruteforce method of unmapping ends up examining every shadow chain in the system
static void shadow_tree_dfs_unmap(mobj_t *o, size_t pagenum)
{
    // iterate over all vmareas that object backs
    list_iterate(&o->mo_vmareas, vma, vmarea_t, vma_mlink)
    {
        if (vma->vma_vmmap->vmm_proc && (pagenum - vma->vma_off < vma->vma_end -
            vma->vma_start))
        {
            KASSERT(pagenum >= vma->vma_off);
            uintptr_t vaddr = (uintptr_t) PN_TO_ADDR(pagenum - vma->vma_off + vma->
                vma_start);
            pt_unmap(vma->vma_vmmap->vmm_proc->p_pml4, vaddr);
        }
    }

    // iterate over all mobjs that shadow o
    list_iterate(&o->mo_shadows, shadow, mobj_t, mo_slink)
    {
        if (!shadow->mo_btree || btree_search(shadow->mo_btree, pagenum) == NULL)
            && (!shadow->mo_swap || btree_search(shadow->mo_swap, pagenum) == NULL)
        {
            // this upstream shadow obj hasn’t done copy on write, so we recurse
            shadow_tree_dfs_unmap(shadow, pagenum);
        }
    }
}

Listing 7: Leverage the new reverse shadow tree, the page daemon is able to only traverse relevant shadow chains for a far simpler unmapping algorithm
4 Results

4.1 Time in Weenix

Prior to discussing performance and results, it's important to know how Weenix keeps track of time so as to understand how all of the time measurements were obtained. Weenix initializes a global tickcount to 0 then registers a handler for the APIC Timer interrupt which is fired by hardware. Letting \( f \) be the frequency of the processor, \( f/16 \) interrupts are fired every millisecond. Every time the interrupt fires, the global tickcount is incremented. This allows us to calculate latencies by storing \( t_1 \leftarrow \text{timer\_tickcount} \), executing an operation, storing \( t_2 \leftarrow \text{time\_tickcount} \), then calculating the latency as \( t_2 - t_1 \). Since all the time considerations for these results are relative, the results are all left in terms of ticks, since the conversion to microseconds or milliseconds is not perfect and varies depending on frequency changes in the emulated hardware that Weenix runs on.

4.2 B-Tree Performance

In order to evaluate the B-tree, I first did some benchmarking to check time to find a specific pframe in the B-tree compared to the time required to complete an entire linear scan of the linked list of pframes. The results convincingly show that the B-tree was successfully able to optimize the search operation, particularly with extremely high numbers of pframes (>30000). The entire linear scan would represent a cache miss.

To examine the same experiment for a cache hit, with \( n \) as the number of pframes in the mobj, we search for the pframe numbered \( \frac{n}{2} \). While in practice, pframes are not necessarily stored in order by number, this test ensures that they are. Thus, we are measuring the latency to search directly halfway through the list in the linked list case. As we can see in the results, the search operation is still faster using the B-tree.

While these very high quantities of pframes certainly aren’t realistic, they still show that the B-tree’s search is highly optimized. Even at relatively small amounts of pframes, it was consistently able to complete the search before a tick occurred. While linear scans could sometimes accomplish this, more often...
they took at least one tick to complete.

To examine a more realistic case, we also measure the average latency across 10 trials of vfstest, the Weenix unit-test suite for VFS. As seen in the resulting graph, the average latency is very similar due most of the testing using quantities of pframes significantly smaller than the tests where we saw the B-tree significantly outperforming the linked list.

![Average latency of 10 vfstest runs](image)

Figure 9: Average latency of 10 vfstest runs

However, compared to the $O(1)$ time complexity of insertion to a linked list, we see the tradeoff of the B-tree’s increased time complexity for insertion.

Going back to the earlier discussion of the branching factor parameter, the very high latency for insertion is caused by the need to significantly rebalance the tree when nodes fill up. An average insertion, if a node has free space, is not that expensive. When inserting very large amounts of keys, however, the very expensive rebalance brings up the average latency significantly. As seen in the results when comparing different branching factors, it is possible to reduce the latency of this operation by increasing the branching factor, compared to all previous testing where the branching factor was set to 2.

Once again, I would like to call attention to the fact that these very high numbers of pframes are somewhat unrealistic to see frequently in Weenix. Thus, we are unlikely to experience the super high latency.

![Insertion times using B-tree and linked list for different numbers of pframes](image)

Figure 10: Insertion times using B-tree and linked list for different numbers of pframes. Each data point is the total average insertion time

![Insertion times using different branching factors for the B-tree](image)

Figure 11: Insertion times using different branching factors for the B-tree. Each data point is the total average insertion time
when inserting into our B-tree. Furthermore, it is important to realize that due to the caching system in Weenix, we are using the search operation to check the cache far more often than we are creating or deleting. The function which is primarily responsible for checking the mobj’s cache, mobj_find_pframe is called extensively throughout Weenix. Thus, it is justifiable to choose to optimize for search, which the B-tree certainly accomplishes.

4.3 Page Daemon and Swap Memory

To evaluate the page daemon, we lack have a prior performance benchmark to compare with like we had with the B-tree evaluation. However, we can still examine its operation by keeping track of how many pages it has flushed out over time.

When Weenix starts up sbin/init, the userland program responsible for creating the TTY’s, it forks 3 times and thus triggers copy-on-write scenarios. For this reason, properly booting up into sbin/init is one of the major landmarks in the students’ Weenix implementations in CSCI 1690. Since it forks numerous times, maps in executable code, and executes copy-on-write for stack space, which is mapped as an anonymous object, it is a great test case to observe the page daemon in. Also, it serves to test swap memory, since the page outs for anonymous and shadow objects will need to store data in and read data from swap space. In order to examine it, I first leaked 200,000 pages prior to following the normal boot process, eating roughly 80% of Weenix’s total pages. This somewhat ridiculous and unrealistic scenario serves to put Weenix under memory pressure while booting so as to force the page daemon to run.

As seen in the resulting graph, the page daemon actually pages out over 100 pages to swap and over 10 pages to files during this startup process. Under this memory pressure with the page daemon actively paging data out, it is still able to properly pagefault and read the necessary pages back in from both the normal filesystem as well as the swap space.

Seeing a significant number of page outs gives us confidence in both swap space and the page daemon’s complex functionality. For swap space, since it’s paging out pages containing stack space, it means that it is properly storing and fetching the state of the stack to and from disk; otherwise, a corrupted stack would mean basically guaranteed faulting and crashing for the running userland threads trying to fork and start the shell. As for the page daemon, it means that it is properly traversing the reverse shadow tree and locating the proper locations to unmap; failure to unmap would also result in important mappings, such as the stack or the executable code, mapping to physical addresses of pages that have been freed, which also would result in the user processes crashing. Finally, as for the quantities of pageouts, this instills confidence that it is checking page access effectively. Since the only file-backed pages that are being freed in this scenario are pages of executable files, it makes sense that they are being accessed more often. They are being accessed by the initial process’ thread and the three threads from the processes it forks, so it follows that there is more often an ACCESS bit set for a mapping to these pages, causing them to be paged out less frequently.
5 Conclusion

5.1 Reflection

After completing Weenix in CSCI 1690, I always knew I wanted to somehow continue working on it; however, I didn’t know it would be to this degree of involvement. I’ve learned so much about operating systems, having to understand everything on a fundamental level in order to complete this thesis. I’ve had to rely on almost every aspect of my computer science education, from data structures and algorithms, to multiprocessor synchronization, to, of course, operating systems in order to solve the problems presented by this thesis. The original scope of the page daemon was, admittedly, much smaller. However, I thoroughly enjoyed taking each new challenge as they arose, from adding swap memory to allow it to page out non-file-backed pages all the way to interfacing with virtual memory to unmap pages.

Kernel programming is hard, but I enjoyed the attention to detail required to solve these problems. Implementing and debugging these features required a comprehensive low-level understanding of all the different moving parts of Weenix. It pushed me beyond my normal coursework and forced me to learn and understand operating systems on an even deeper level.

5.2 Future Work

I’m privileged to have had written a small chapter in the story of Weenix, and I anticipate that its story will continue to be written by Brown students long after myself. Specific to my work in this thesis, I can name some areas that would be worth exploring further.

Firstly, the DFS used to traverse the reverse shadow tree could definitely stand to be optimized. When the shadow chain is very long, the tree may be too deep to traverse recursively. I’d be interested in seeing an iterative approach, or potentially some dynamic programming or memoization approach to optimize it. There is certainly the possibility for some redundant recursive cases being hit in the purely DFS approach. Due to time constraints and the already substantial complexity of my approach I was unable to explore these possibilities, but it would certainly be a valid effort to make the page daemon even more robust.

Another area that could be optimized is the B-tree. Since the keys in each node are already sorted, a somewhat quick optimization could be to use binary search to locate indices within nodes quicker. Deletion could also be changed so that it makes only one pass instead of the implementation I opted for which uses a search and then recursively deletes in the subtree. It’s possible that similar optimizations could be use to mitigate latencies for insertion and deletion.

Swap space could also be changed to adopt a different approach. Since I abstracted it at the VFS level, there could be numerous different ways to explore optimizing swap. If the filesystem in Weenix were to be changed, it could be a good opportunity to implement a new system for swap space.

Although this is somewhat unrelated to the bulk of my work, another improvement could be to implement a full tiered queue system for Weenix’s thread queue to have proper notions of priority. This model could look very similar to the approach in uthreads, another CSCI 1670 assignment. The workaround I used to prioritize the page daemon works in this scenario since it’s the only thread that might possibly need to be prioritized, but a more robust queue system like the one from uthreads could be a nice upgrade long-term.

The new locking I added to remedy the concurrency issues and race conditions in the page daemon is extremely coarse-grained. The new locking is very globalized, and this locking scheme could have performance impacts. Thus, another area of future work would be to first investigate the possible performance impacts and, if it is found to be detrimental, devise a more fine-grained locking scheme.

The final area of improvement that could follow my work is more extensive use of the PTEs in the kernel. Currently, S5FS relies on a pf_dirty field to determine if the page should be written out to disk when the pframe is freed. However, the PTE actually has a DIRTY bit that is set if the physical page it maps is modified. Thus, an area of improvement
may be to use this bit to determine if a page should be written out instead. This would be particularly useful in the swap implementation, since currently an unnecessary write occurs if a previously paged out anonymous page is read back in on a pagefault, but it isn’t modified after being read back from swap. If the \texttt{mobj} is anonymous, it is likely mapped for writing, which would cause the \texttt{pf\_dirty} field to be set, causing a write in the event that it is paged out again even if it wasn’t modified since it was paged back in.

The main issue and consideration here, however, is that some of the pages that Weenix allocates in \texttt{page\_alloc} are actually 4 KB subsections that are mapped in 2MB pages. This is due to how the fill functions that are used to create the \texttt{PHYSMAP} region optimize for very large mappings. Thus, the granularity of the \texttt{DIRTY} bit would be at the 2MB level for these pages. This could make leveraging the \texttt{DIRTY} bit on the PTE more complicated or potentially impossible in its current state, but it could definitely be researched as a potential optimization to avoid unnecessary disk operations in the page daemon and elsewhere in Weenix.

5.3 Acknowledgements

I would like to thank Professor Tom Doeppner for being an excellent advisor and originally inspiring me to take on this thesis topic. Weenix has been an instrumental learning tool in my time at Brown, starting in CSCI 1690, then as a teaching assistant for the course, and finally with my thesis. I’m honored to have had a small part in such a fixture of Brown CS, and I owe that opportunity to Professor Doeppner.

I also want to thank Professor Malte Schwarzkopf for originally introducing me to systems programming in CSCI 0300 the first semester he taught it. I’ve loved systems ever since, and I am incredibly grateful to have him as the second reader of this thesis.

Finally, to all of my friends and professors at Brown: thank you. These past 4 years have taught me so much, both in the classroom and outside of it, and I owe that to all of you.
References


