Implementing Paging in Xinu

Final Report

Thomas Gagne
University of Puget Sound
1500 N. Warner St.
Tacoma, Washington 98416
tgagne@pugetsound.edu

ABSTRACT

This report discusses on the design and implementation of virtual memory management through paging in the Xinu operating system done as part of the CS-440 Capstone course. It covers the theoretical aspects of paging to provide background, then provides an in-depth discussion on the specific details of various components needed to successfully implement paging and virtual memory management. The completed project's source code can be found at https://github.com/ThomasGagne/Xinu-x86-Paging.

ACM Reference format:
Thomas Gagne. 2017. Implementing Paging in Xinu. In *Proceedings of University of Puget Sound, Tacoma, Washington USA, May 2017 (CS 440),* 9 pages.
https://doi.org/

1 INTRODUCTION

In most modern computer systems, accessing and modifying memory values in RAM commonly encompasses much of work performed by user programs. Hence, in environments where numerous programs are running in parallel and are attempting to simultaneously access memory, the task of arbitrating memory between programs becomes an incredibly important role of the operating system. In particular, the operating system must provide a means to dynamically allocate memory to programs in a way which guarantees safety, efficiency, and is convenient for programs to utilize. To accomplish this, many operating systems designers choose to use a scheme of virtual memory management, which provides all three of these by creating a virtual space for each program to work in and defining a translation between each program's virtual space to the actual physical space in real memory. The most common of these schemes is paging, and it is because of the prevalence of this scheme that I chose to implement it in the Xinu operating system for my Capstone project.

There are two primary reasons why I chose to do this: The first reason is that it afforded me an opportunity to explore the world of operating systems development by allowing me to both design the theoretical structures and mechanisms which the operating system would use to provide support for paging, as well as actually implement these structures and mechanisms and demonstrate their correctness. This gave me valuable experience in operating systems design and development on a deep and fundamental level which would be difficult to match in an operating systems course. The second reason was realized after the project had already begun, and is that I plan to have this paper act as a useful and comprehensive guide to those planning on implementing paging for operating systems running on x86 architecture. Over the course of this project I faced numerous struggles due to the sheer lack of resources on this topic, and consequently set out with the goal that this paper helps fill this void and will act as a significantly useful resource for those taking on similar projects in the future.

Finally, as for why I chose to implement paging in the Xinu operating system for the x86 architecture in 32-bit mode in particular, I chose Xinu since I already had some experience in Xinu development and since I knew that the operating system's lack of complex utilities such as virtual memory management, dynamic loading of user programs, disk access, etc. would simplify my task. Since Xinu runs on a variety of architectures, I additionally chose to implement paging for x86 in particular since this is the most widely-used architecture in modern operating systems development and I knew that choosing this would allow me access to the most resources during this project.

2 BACKGROUND

In this section we shall first cover Xinu's current memory management model, then the notion of virtual memory management and the benefits it provides over Xinu's memory management model, and finally we shall discuss what exactly paging is and how it implements virtual memory.

Because Xinu was designed for use in embedded systems where large amounts of memory were unlikely to be used and where typically only a small collection of pre-defined programs will ever be run, Xinu only provides a very elementary form of memory management which does not scale well to large systems. By default, Xinu runs a form of memory management known as *memory segmentation.* In this scheme, all the free memory in the system is divided into a linked list of free segments, with memory initially beginning as one large segment. When memory is requested by programs, the operating system traverses this linked list to find a segment large enough in size to meet the requirements of the program. When such a segment is found, it is divided into two new segments: one in-use segment with size equal to that needed by the program (which is then allocated to the program), and a free segment comprising of the remaining memory in the original segment. As memory is no longer needed by programs and is freed, those in-use segments will become free segments and will be re-inserted into the linked list of free segments. [1]

This form of memory management may work well when large amounts of memory are not needed or when only a small number of programs will be run simultaneously, but this system unfortunately

CS 440, May 2017, Tacoma, Washington USA
2017. ACM ISBN .
https://doi.org/

will not scale well to larger systems where these assumptions are not true. In particular, one of the most significant problems incurred by memory segmentation is *fragmentation*, where after long periods of various-sized segments being allocated and freed, the remaining free memory in the system will be divided into numerous non-contiguous segments throughout memory which are each unusably small in size. Hence, even though the system as a whole might have a large amount of total free memory, it will be divided into so many different segments that the free memory becomes unusable. Another significant issue with memory segmentation is the lack of memory protection between programs. In particular, Xinu does not implement any sort of check that a program attempting to modify a memory location has actually been allocated a segment containing that memory location, allowing programs to easily overwrite the memory and code of other programs, including that of the kernel itself. Finally, as memory becomes more heavily used by programs, the task of traversing the linked list to identify a free segment of suitable size can become an unreasonably slow process, which can become especially frustrating when combined with fragmented memory due to potentially spending a large amount of time searching for large enough segments which do not exist.

In response to these issues, the notions of virtual memory management and paging were developed. Virtual memory management is a solution to provide memory protection by instead of allowing programs to access memory arbitrarily and hoping that they only access memory which was allocated to them, each program now has its own separate *virtual memory space* which it can freely access and manipulate. To guarantee safety, the operating system then ensures that each programs virtual memory space is only visible to the program it corresponds to, and the operating system then provides a translation between the virtual addresses used by each program and physical addresses in real memory. By maintaining a scheme where no two virtual addresses map to the same physical addresses, the operating system can then guarantee safety by translating each programs virtual addresses to real addresses as it executes. Virtual memory management additionally solves the issue of fragmentation, since adjacent virtual addresses do not need to be mapped to adjacent real addresses, implying that the physical location of free memory does not affect its usability. [2]

Virtual memory management itself is not a replacement for memory segmentation though, since we have not defined the specific mechanism of translating virtual addresses to physical ones. To develop such a mapping, we introduce the concept of paging, which acts as an implementation of virtual memory management. On x86 architecture, paging works by dividing both the virtual memory space of programs and the real memory space into 4KB-sized and aligned chunks known as *pages* and *frames* respectively. The operating system then describes an address mapping for each program by mapping each virtual page to a physical frame, effectively mapping 4KB of contiguous virtual addresses to 4KB of contiguous physical addresses at a time. Address translation is then performed by the memory management unit (MMU) during program execution by identifying the virtual page containing the given virtual address and finding the corresponding real address in the corresponding frame. By doing this, paging implements virtual memory management, which effectively provides memory protection between programs and solves the issue of fragmentation. This latter point is because adjacent pages in virtual memory do not need to be mapped to adjacent frames in real memory, and since we are now using static page sizes rather than arbitrary-sized segments.

To provide a mechanism for address translation, paging on x86 architectures in 32-bit mode keeps a *page directory* and a collection of *page tables* for each program. Each directory and table occupies 4KB of memory and is structured to contain 1024 4-byte entries. The directories and tables are additionally required to be 4KB address aligned, meaning each directory and table will fill exactly one frame in memory. A page directory contains the base addresses of 1024 page tables, as well as some tracking info about the status of those tables, which is stored in the rightmost 12 bits of the addresses, since the addresses are required to be 4KB address aligned. Page tables are structured near identically to this, with the exception that the 1024 stored entries instead point to the address of a frame in memory.

To perform address translation, the MMU uses a program's page directory and collection of page tables to identify the physical address corresponding to a given virtual address. To do this, the CR3 register in the CPU is initially loaded with the base address of the page directory for the currently running program when a context switch to that program is performed. When a virtual address is sent to the CPU as part of an instruction, the MMU then extracts a *page directory index*, *page table index*, and *page offset* from the virtual address. The MMU then takes the page directory index and identifies that particular entry in the page directory to identify which page table the virtual address is referencing. The MMU then does a similar lookup in that page table using the page table index to identify the address of the frame the address is referencing. Finally, the page offset is used to identify the exact physical address in the frame the virtual address is mapped to. See Figure 1 for information about how the page directories and tables are structured, and how the MMU extracts the indices from a virtual address to perform an address translation.[1] [3] [4]

Note that in x86 architecture, the operating system is not required to perform any of the address translation itself. Instead, it is the job of the operating system to provide memory management by ensuring that the page directory and page tables of each program are structured correctly.

3 IMPLEMENTATION

Giving that we now have an understanding of what exactly virtual memory management and paging are and how the latter works in an x86 32-bit environment, we can now cover the implementation details of this project. In particular, we shall describe the necessary tasks which must be performed by an operating system in order to initialize and dynamically modify the paging structures for each program. Note that this itself is an open problem and can be accomplished in a variety of ways. In this paper I am simply describing the solutions I chose to use in implementing paging in Xinu, which were chosen based upon their simplicity, ease of implementation, and intuitiveness compared to other potential solutions. I also wish to point out that in this section I will not order these components

[1]Used with permission from https://en.wikipedia.org/wiki/File:X86_Paging_4K.svg

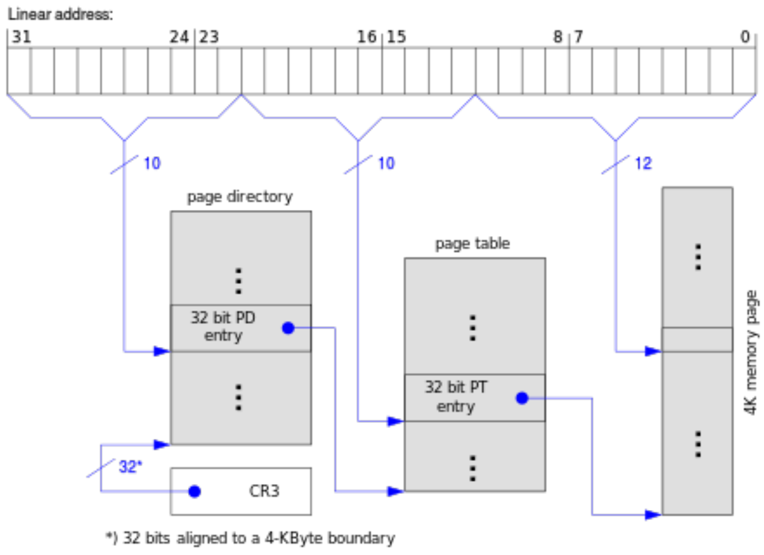

Figure 1: The format of virtual addresses and its relationship to the structure of page directories and page tables.

in terms of importance or significance but rather in terms of topological dependency, so that for each component I discuss I will have already covered the components which it relies upon.

3.1 Frame Allocation

The problem of frame allocation is the first issue which must be tackled when attempting to implement paging in an operating system. Since in a paging scheme all of virtual and physical memory is divided into 4KB-sized pages and frames and since the paging structures work on a 4KB granularity, it is necessary to develop some component which keeps track of the status of each 4KB frame in memory and whether or not we have already allocated it to be used by a program. The purpose of the frame allocator in particular is to use this tracking component to locate free frames when requested and then mark them as in-use. [5] [6]

This tracking can be done in a wide variety of ways, each of which have their own merit. In this project though, I have chosen to use a *frame bitmap* to track the status of each frame, due to its overall simplicity and small size. This latter point is especially important, since in a system where we have all 4GB available (since we have a 32-bit system) we will need to track the status of 1,048,576 frames. Consequently, to minimize the size of the data structure which will track the status of each of these frames, I chose to use the smallest amount of information possible to represent the availability of each frame: a single bit indicating whether or not the frame is available. In particular, upon initialization of the Xinu operating system I take the number of available frames in memory, say n, and designate $n/8$ bytes of memory to act as the bitmap. In this bitmap, the ith bit then corresponds to the availability of the ith frame in memory.

Using this scheme, to allocate a frame we simply need to look over each byte in the bitmap until we find one which is not equal to 0xF, then extract the bit number of the free bit and use that byte number in the bitmap and the index of the bit to calculate the address of the corresponding frame. We then set the bit to 1 to mark the frame as in-use. Under this scheme, this can be performed in $O(n)$ worst-case-scenario time. However, we can often speed up our ability to find a free frame by storing the index of the last byte we allocated from and beginning our search from there. If a large number of frame allocations and reclaims are performed in arbitrary places during the system's uptime the gains from this will eventually become negligible, but in the majority of circumstances this small addition can greatly increase frame allocation speeds, particularly after a system is just started up. Another benefit of this scheme is the $O(1)$ time to mark the status of a particular frame, which occurs when reclaiming frames and marking them as available again.

3.2 Structure of Virtual Memory

Now that we have the ability to allocate frames in memory, we can use our frame allocator to begin allocating and initializing the paging structures of programs. However, we should not jump so suddenly to this task since we first must develop a plan for how both real memory and virtual memory will be structured. The structure of real memory will actually be relatively simple. In Xinu's simple state we can divide memory into two primary logical regions: the kernel image and data, and the remaining free memory which our frame allocator will allocate frames from. As far as organizing structures within these regions, we do not need to change anything with the kernel image, and our frame allocator actually provides an abstraction over the remaining free memory since once our frame allocator has been completed we can simply rely on it to manage the frames in that region.

The structure of virtual memory is not as simple though, and there are many possibilities for how we can structure it to provide the most convenience to running programs. In general, we can assume there are two logical regions of memory which a program must have access to: a region of memory which will contain the program's code, heap, and stack; and it must have the ability to call system calls provided by the operating system. For the latter, this is accomplished by making the kernel image visible to each running program, meaning that we must designate a region of memory in virtual space to contain the kernel image. Hence, the most intuitive way to provide access to these two logical regions of memory is to divide virtual memory into two major regions: *kernel space* and *user space*. Kernel space will contain the kernel image and data and should be unmodifiable by user programs. For this reason, it will have a shared view between all processes. User space, on the other hand, should be specific to each process and will contain the process' code, heap, and stack.

The simplest way to accomplish this is to designate a region of addresses to act as kernel space, then for each process we map the pages corresponding to that virtual region to the kernel image. We now have a choice as to which region of virtual addresses will correspond to kernel space though. One popular approach is to use what is known as a *higher-half kernel*, where the highest virtual addresses are used for kernel space, such as addresses 0xC0000000 - 0xFFFFFFFF. The benefits of doing this are that it is easier to set up VM86 processes, it is simpler for compiled user programs to begin at address 0x0, and if the operating system is 64-bit then 32-bit applications will be able to use the full 32-bit address space. However, one difficulty of this is it requires the kernel to be compiled so that once paging is enabled in the system, all address references in the kernel thereafter must be aware that the kernel is now located in the higher half of memory. The alternative (and simpler) approach is to identity map the kernel in the lower half of memory, where it is normally loaded to in real memory. This is the approach that I chose in my implementation of paging in Xinu, primarily since I was not interested in rewriting Xinu's compile scripts so that a higher-half kernel could be used. For simplicity, I chose to set kernel space to be the region from 0x00000000 - 0x003FFFFF, and I then identity-mapped everything in this region. This is because these are all the addresses accessible by the first page table in a processes' paging structures, meaning that if we create a page table which identity maps this region, then every process can share it by simply having the first page directory entry point to that page table, saving memory and complexity. Additionally, this allowed the kernel more than enough room to grow in the future. All the addresses from 0x00400000 onwards then belong to user space. [7]

3.3 Initialization of Paging Structures

Now that we have our frame allocator and have a plan on how to organize our paging structures, we can begin the initialization of a process' page directory and page tables when it is being created. Since a page directory and table both occupy exactly 4KB in memory, we can easily use our frame allocator to allocate space for these structures. Once paging is enabled though we will no longer be able to directly modify these frames, since there is no page table entry pointing to them. Hence, for the first thread in the system we must initialize these structures manually before paging is enabled while we can still directly modify real memory.

To do this, upon system initialization we will first allocate the first thread's page directory and initialize each of the page directory entries to be empty. We then must allocate our page table to be used for identity mapping the kernel, which shall be known as the *kernel identity table*, initialize each of the entries to facilitate identity mapping of that region, and finally set the first page directory entry to point to that page table. [8] At this point we are done with the first thread's paging structures and can safely enable paging, at least in the case for Xinu. This is because the first thread in Xinu, the *null thread*, does not perform any actual work other than initializing various kernel structures and spawning the first thread in the system, meaning that it does not require any use of the heap. Furthermore, since in Xinu the stack of the null thread is included in the kernel image, we can still use the null thread's stack once paging is enabled since it is in the identity mapped region of kernel space.

This is certainly not the case for threads in general though, and it actually requires a fair amount of work in order to spawn a thread from another thread in general. This is because once paging has been initialized there is no longer any way to directly access memory, meaning that when we are initializing a new thread even if we know the real address of the thread's page directory and page tables we cannot directly access them. This presents a challenge, since our frame allocator is only capable of returning the physical address of its allocated frames. To resolve this issue, I used a scheme which for the purposes of this project I have dubbed *readdressing*. Readdressing is a technique where when spawning a new thread we temporarily redirect some of the page table entries of the currently running thread to point to the frames which will contain the new thread's structures. In particular, we need to initialize the new thread's page directory and the page tables which will point to the new thread's heap and its stack. We additionally will need to modify some memory values in the frames which will hold the beginning of the process' heap (if we plan to use a scheme such as segmentation to provide per-process heap allocation, which we will cover later) and will hold the top of the process' stack so we can initialize them.

To accomplish this I chose to take the first four pages of user space, addresses 0x00400000 - 0x00403FFF, identify the page table entries of the current thread determining which frames these pages map to, then modify the locations of these pages to point to the necessary positions. I chose these addresses since in my implementation of paging in Xinu these addresses correspond to the beginning of the heap for the process, and it would not be necessary to access the heap while spawning a new process. I additionally chose to work with four pages rather than one since I found it simpler to map one page to the new thread's page directory, one page to any of the new thread's page tables as was necessary, and the remaining two pages were used to modify regions of user space, namely initializing the heap and stack. I chose to have two pages to modify regions of user space since during the stack initialization there was the possibility that the initialized stack would contain addresses in two different pages, and it was simpler to use two pages at once rather than remap a single page multiple times. By using this scheme I was able to successfully initialize the paging and memory structures necessary to spawn a new thread properly from a currently running thread.

3.4 Modification of Paging Structures

Through the technique of readdressing we are now able to spawn multiple threads which will each possess an initialized page directory, some page tables, and initialized memory structures for the heap and stack. However, in paging we do not constantly have every page directory and page table entry pointing to a specific location; many pages in virtual space do not yet have a frame which they are mapped to. Consequently, to prevent a process from incurring page faults caused by attempting to access pages which have not yet been mapped to frames, it is necessary to have the ability to dynamically modify page directory and page table entries. In particular, we need a system where when a process needs to access a certain page, we can use our frame allocator to allocate a free frame for it, then modify the appropriate page table entry so that that certain page is mapped to that frame. However, since we have paging enabled we can only modify specific real memory values if we have a page mapped to those values. This means that we must develop some special way to access the page directory and page table entries of a process while that process is currently running.

Like many aspects of implementing paging, there are a variety of solutions to this problem. For example, when a process needs access to a page table we could perform a context switch to some kernel thread which has pages pointing to all the currently allocated page tables and page directories in the system. This is an expensive and memory-intensive solution though, and leads to awkwardness as to how we notify this kernel thread that we need a specific page mapped.

A much better and much more common solution to this—and the solution I chose for this problem—is to map one of the page directories entries to itself, commonly the last entry. This strategy might seem confusing at first, but a bit of deeper insight into how page directories and page tables actually work can help us understand the elegance of this solution. The first important point to note is that there is actually hardly any distinction between page directories and page tables; for all practical purposes, a page directory is itself a page table. Furthermore, we point out that page directory entries point to the base address of page tables in the same way that the page table entries point to the base addresses of frames. Finally, we remind the reader of the fact that page tables and page directories are exactly 4KB in size—the size of a single frame.

Given these three points, it should be apparent that if the last entry of a page directory points to itself, then we can use all the addresses in that region—namely 0xFFC00000 - 0xFFFFFFFF—to modify the entries of the current process' page tables and page directory. To see this, suppose that we pass address 0xFFC00000 to the CPU. The MMU will first extract the ten leftmost bits to act as the page directory index, which in this case will notify the MMU to look in the last entry of the page directory. Since the last entry of the page directory points to its own address though and since the page directory is functionally a page table, the MMU will use the page directory itself as the page table during this address translation. The MMU will then extract the middle ten bits, all zeroes, as the page table index, which in our example will be the first page table entry. Since the current page table is really the page directory though and since the page directory's first entry points to the base address of the first page table, the MMU will then attempt to modify an address inside the frame which contains the first page table. In our case, this is the first address, which is the first page table entry of the first page table. In this manner, it should now be obvious that by mapping the final entry of the page directory to itself we have given ourselves the ability to easily modify page table entries. Furthermore, we can use the fact that the page directory points to itself twice to modify page directory entries by using the addresses in the range 0xFFFFF000 - 0xFFFFFFFF, since under this current scheme these addresses will correspond to the page directory entries. [4]

With this simple trick, we have therefore given ourselves a mechanism to easily modify the page table and page directory entries of the currently running process. This additionally gives us the benefit that these addresses will modify the paging structures of the current process regardless of which process is running, meaning that we do not have to perform any sort of thread-specific lookups when attempting to modify these structures. However, there is a slight disadvantage to this technique in that we have somewhat reduced the amount of virtual memory available to processes. This is a negligible amount though, only 4MB, and the simplification to our implementation of paging is rather significant, so as a whole this is a more than worthwhile investment. Using this technique is also an additional argument for why a higher-half kernel could be preferred, since that would mean that these addresses could be part of the kernel space rather than making an additional page table space. Alternatively, I could also have chosen to place the page table space located just after kernel space and combined them together. For overall ease of implementation and extension though, in my implementation I chose to have the kernel space positioned from addresses 0x00000000 - 0x003FFFFF, the user space positioned from addresses 0x00400000 - 0xFFBFFFFF, and the page table space positioned from addresses 0xFFC00000 - 0xFFFFFFFF.

3.5 Abstracting Memory Management for Processes

Now that we have the ability to initialize paging structures for new processes, the ability to dynamically update them by using the virtual addresses located in page table space, and have a meaningfully structured virtual memory space, we have all the tools necessary to allow programs to dynamically allocate memory as necessary. However, we should certainly not consider it the job of the programmer to have to deal with any of these aspects; after all, it is the responsibility of the operating system to deal with memory management for processes. Consequently we must develop some mechanism of abstracting the process of dynamically mapping pages to frames during the program's lifetime. To do this, we start by asking ourselves when exactly a process will need to access new pages in its virtual memory space. When initializing the process' stack we gave it a set amount of stack space and paged that entire region, so we do not need to deal with that. The only other way a process should be able to access memory in user space is by allocating memory on the heap, which in Xinu is done with the memget() and memfree() system calls. [2] Hence, it is necessary to rewrite these methods so that they take paging into account.

[2]In Xinu, these are the methods which underlie and perform the actual work of malloc() and free().

We must additionally rewrite these utilities in general since Xinu's original code for these methods was for a memory segmentation scheme of memory management.

To rewrite these system utilities, we must first ask ourselves how exactly we want memory to be allocated during a malloc() call. In particular, we recognize that we must have some form of memory management specific to each process which has the ability to find regions of free memory in the process' user space large enough to hold various data. Furthermore, we point out that while in user space we cannot add on another layer of virtual memory management; we must find some scheme of memory management which works without virtual memory. For these reasons, it is chosen in many operating systems to have memory allocated from a process' user space by using memory segmentation, since this is a simple way to provide memory management on a per-process basis. I additionally chose to use memory segmentation for handling user space in my implementation of paging in Xinu since the memget() and memfree() methods provided by Xinu already implement memory segmentation, and it is a surprisingly easy task to modify the code to act on a per-process basis once virtual memory is enabled. The reason for this is because Xinu's code for memory segmentation holds all tracking data about free memory in those regions of memory themselves rather than in kernel space, meaning that we can simply move this tracking data to reside in user space to achieve a similar effect. Then when memget() is called, the utility behaves identically to as if virtual memory were not enabled but only one thread were running, since each process has its own private view of its virtual user space. Hence, it is a trivial task to modify Xinu's memget() and memfree() utilities to implement memory segmentation on a per-process basis to provide dynamic memory allocation.

One observant question to ask at this point is whether or not memory segmentation is necessarily a good scheme to choose for per-process memory management, considering how all this work is being done to implement paging so that we do not have to do memory segmentation in the first place. In response to this, I first point out that one of the primary reasons we chose to leave memory segmentation behind was because it did not provide memory protection between processes. However, since in this circumstance we are using memory segmentation on a per-process basis and in each process' independent user space, this is not an issue. The other issue is whether or not we will incur the same issue of memory fragmentation we encountered with memory segmentation. While this is certainly a possibility, it however is not as much of an issue when we are using segmentation on a per-process basis. The reason for this is because when using memory segmentation in general, memory fragmentation was caused by many processes running over the operating system's lifetime and allocating varying amounts of memory, which is not an unusual scenario. However, to achieve the same effect when using per-process memory segmentation, it is necessary for this single process to allocate large amounts of memory overall in many different allocations, which is a much rarer occurrence. Additionally, when memory segmentation is used in general and fragmentation occurs, the only solutions are to either reorganize all of memory to resolve the issue or to reboot the operating system. When a process incurs fragmentation in its own user space though, it is a much less dramatic solution to simply kill the problematic process rather than be forced to restart the entire operating system.

There still exists another issue with dynamic memory allocation which we have not dealt with yet, however. This is that we are not updating a process' page tables when allocating memory so that when a region of memory is allocated, the pages containing that region are mapped to frames in real memory. To resolve this issue—and to simplify many other aspects of implementing paging—I chose to create a pageregion() utility which takes a region of addresses and uses our frame allocator to map each page which contains some of that region to a frame on disk. With this utility in place, whenever memget() finds a region of memory which it will allocate to the currently running process, we can first pass this region of addresses to our pageregion() utility before returning the memory to the process, thus resolving the issue. Note that this utility will not page the same region multiple times; if a page has already been mapped to a frame then the utility simply won't change that page mapping.

At this point, there remains one final concern of memory management which we have not yet covered: that of reclaiming memory when a process frees memory or terminates. For the first point, that of reclaiming frames when a process calls memfree(), in my implementation I actually chose not to have the operating system reclaim frames while a process was still running. The reason for this was because I considered it both challenging and expensive to have the operating system check when we free memory that all the memory in that page is not currently in use, which would be the only safe way to reclaim frames when memfree() is called. Because of this, the only time memory is reclaimed by the operating system is when a process is terminated. In particular, when a thread terminates it will call the system utility xdone(), which cleans up the thread's data in the kernel. To allow the operating system to reclaim a process' allocated memory when it terminates, I modified this utility so that during termination the utility would walk along the process' page tables and reclaim any page table entries which were mapped to frames. Once this was done, the utility walks along the page directory and reclaims the frames which held the page tables, then finally reclaims the frame holding the page directory itself. This process was greatly simplified by our earlier trick of mapping the last entry of the page directory to itself to introduce a page table space, since it allows us to simply scan over page table space and reclaim any entry which is mapped to a frame.

3.6 Final Additions

With the completion of each of these components, we have finished the major structural components of implementing paging. All that remains to complete our system is to make a few final changes to tie each of these components together. In particular, we need to make sure that paging is initialized during system startup once the null thread's paging structures have been initialized. This is simply done by setting the value of the leftmost bit in the CR0 register in the CPU to 1 to indicate that paging is now enabled. We additionally need a mechanism to notify the CPU of the physical location of the page directory of the currently running thread. This is done by loading the CR3 register with the base address of the page directory, so I created a short utility called loadCR3() which takes in a page directory address and loads it into the CR3 register. Note that since neither of these registers are directly accessible from C code, it is necessary to write these utilities in x86 assembly. Finally, we must modify the structure of a thread struct in the kernel so that it contains the base address for the thread's page directory, and we must update the resched() utility so that when rescheduling occurs, the CR3 register is loaded with the page directory of the process we are switching to. [4]

4 DISCUSSION

Through implementing these components, I was able to near completely implement paging in the Xinu operating system for x86 architecture in 32-bit mode. I say "near completely" because at the time of writing this paper there still remains at least one outstanding bug preventing the project from working properly. One of the most frustrating aspects of implementing paging is that due to the nature of virtual memory management, the entire system will either work perfectly or will face catastrophic failure. While the latter is unfortunately the current state of my implementation, this is however not due to any major mistake made during implementation or structuring of the project but rather—I suspect—is a small issue that is a consequence of a few erroneous lines of code which are causing some particular structure to become initialized wrong. These sorts of issues came up numerous times during the development of this project and presented some of the most significant challenges to overcome. In particular, these issues commonly presented themselves in strange ways since the project required me to first initialize the paging structures and then hope that I had structured them correctly for when they would be used later. If I made an error though, the issue could hide itself for a considerably long time before finally making itself known, and could easily present itself in curious ways that were not indicative of the actual issue.

As an example of one of the most confounding issues I faced during this project, at a point I found myself in a position where after spawning the main thread from the null thread, if I attempted to spawn a second thread from the main thread the operating system would suddenly appear to restart and jump back to running its initialization sequence again. In addition, this would occur immediately after reloading the CR3 register during the second process' initialization, and would sometimes occur at different times depending on when whether or not I had included debugging print statements in the code. Although initially confusing, this problem ended up being rather intuitive once I realized the underlying issue. Understanding exactly what the issue was in the first place though was not obvious at all given the problem's nature. The underlying cause for this was because while initializing the second thread, rather than using exclusively readdressing to modify the new thread's paging structures I had decided it would be simpler to just initialize the thread's paging directory using readdressing, then reload the CR3 register with that directory so that we could use the entries in page table space to initialize the process' memory structures. This was problematic, however, in that when I reloaded the CR3 register I would lose visibility of the execution stack in the thread which was spawning the new process. This meant that when the CPU attempted to exit the subroutine for reloading the CR3 register, it would attempt to jump back to the address marked on the execution stack, but having lost visibility of the stack it was instead causing a page fault to occur. Furthermore, the loss of the stack entirely meant that the subroutine for trap handling could not be correctly called, causing the CPU to have no other option than to simply clear all registers, which caused the operating system to appear as though it were restarting. In addition, this sudden loss of visibility of the stack is why the inclusion or exclusion of debugging print statements would cause the CPU to incur errors in different places. This was also the reason for why the error only came up when spawning the second thread, since the null thread's execution stack was located in kernel space which was visible to all processes.

This is one such example of some of the confusing issues I ran into during this project which made development so difficult at times. Apart from actual implementation and debugging though, another one of the most frustrating challenges I ran into during this project (and part of the reason why I chose to write this paper) was the overall lack of detailed information on the implementation of paging. In particular, I faced a general lack of information on the specific implementation details of Xinu, as well as a lack of information on how paging itself was actually implemented in practice, despite there existing a large amount of material covering the theoretical aspects of paging and virtual memory management. In the end, I had to consequently learn many of these topics myself through either scouring through various resources and trying to piece together information or by simply having to come up with solutions myself. For this reason, I wish to point out that even though I set out to write this paper with the goal of providing a hopefully informational and comprehensive guide on how to implement paging in an operating system, many of the topics I covered in the paper were my own solutions to the problems I encountered and may not necessarily be the most ideal or conventional solutions. In addition, the topics I covered in this paper are specific to paging on x86 architecture in 32-bit mode, and on different architectures the task of implementing paging may require a considerably different approach.

As a whole though, I would consider my implementation of paging in Xinu to be rather successful—despite not necessarily having a completely working product at the time of writing this paper—and I found this project to be an exciting, informative, and deep journey into the world of operating systems development. I found one of the most rewarding things during this project was the ability to essentially have a large workspace—that of real memory—and being given the task of developing various structures in this workspace to accomplish a well-defined goal. There is a certain satisfaction to be obtained from working on an environment where one does not have the ability to call upon an underlying operating system for help and having to construct these structures entirely on my own. For example, rather than being able to ask the operating system to allocate memory to store a particular data structure I needed and not having to worry about its actual location, I had to examine the current state of my project as a whole and identify regions of memory which I had not yet used and could designate to store those data structures. In this regard, I found it very rewarding to be able to construct a large and complex system in terms of data structures in memory, rather than the more common task in computer science of focusing on the computational aspects of systems.

I also found this project to be a rewarding insight in how to take the abstract and theoretical concepts of operating systems components and actually implement those concepts. In particular, I was surprised by the level of detail and complexity that was required to implement such the relatively simple concept of virtual memory management, and doing this project helped me develop a better appreciation for the work that actually goes into operating systems development once a theoretical model has been envisioned.

5 FURTHER EXTENSIONS TO THIS PROJECT

Due to only having so much time and resources available to me over the course of this project, I unfortunately was only able to complete what could probably be considered the simplest implementation of paging possible, despite all the work I put into it. This is because virtual memory management and paging as a whole are rather abstract concepts and there are numerous details and aspects one could implement to further extend and improve memory management in an operating system. In this section I shall cover some of these potential extensions and describe their purpose and utility.

One of the most useful advantages of paging is the ability to swap out currently unused pages to a region of disk known as *swap space* to allow currently running programs to utilize those frames, thus giving the illusion that the computer possesses more memory than it actually does. This is helped by the static page size, since we can easily load pages back in from disk to any frame in memory, which is a nontrivial task in segmentation memory due to non-regular sized segments. Hence, implementing the ability for an operating system to swap out pages to disk is an incredibly useful extension to paging, so much that it was actually my original proposed project before I realized that Xinu did not support paging for x86 architecture and was forced to change my project to implement paging itself. The problem of implementing page swapping itself is no simple task though, and is further made difficult by the fact that it requires disk formatting, something which Xinu does not support.

Another primary aspect of the implementation of paging which I did not touch upon in this project is the role of the *translation lookaside buffer* (TLB). The TLB is a piece of hardware in the MMU which acts as a cache for the most recently accessed frames during address translation, greatly speeding up the process of address translation. The TLB itself must be programmed by the operating system upon startup, however, and must be modified and flushed during events such as rescheduling. However, because Xinu did not have any support for TLB programming and because the emulation software I used to run Xinu (QEMU) did not include a TLB either, I was unable to include TLB programming as part of my project.

Another useful component of paging is the ability to allow multiple threads to share regions of memory. In particular, if a process spawns multiple threads they need to have a shared view of the process' code and data, but their own private heaps and stacks. Paging provides a useful way for us to implement kernel-level threads in this regard by taking a linux-like approach and making threads more similar to processes by giving them their own page directories and page tables, then allowing the threads to share various page tables and frames. In particular, we could potentially divide the process' user space during runtime into multiple regions which could contain each thread's heap and stack while still allowing them to share the process' code and data. To implement this, however, would require a large overhaul of how processes work in Xinu and would require developing some interface to allow processes to spawn kernel-level threads. For this reason, implementing this would be considered a substantial project in itself and I obviously chose not to pursue it during my own project.

Finally, the details of paging which I have discussed in this paper only pertain to 32-bit systems, restricting us to a maximum of 4GB of memory. However, the majority of consumer computers today have significantly more memory available to them and run 64-bit architecture, meaning that we are limiting ourselves by restricting ourselves to 32-bit systems in our implementation of paging. Implementing paging for 64-bit systems simply adds on yet another layer of difficulty in implementing paging though. In a 32-bit system, we simply use a page directory and a page table. In 64-bit systems though, 4 levels of address indirection are used, starting with a PML4 table pointing to a page-directory-pointer table, which in turn points to a page directory, which then points to a page table, then finally pointing to physical frames. Paging on x86-64 systems also requires the operating system to allow both 4KB pages as well as 2MB pages, and in some circumstances 1GB pages. For these reasons, implementing paging on 64-bit machines introduces an additional layer of difficulty and complexity for the operating system, and so obviously I chose to restrict myself to implementing paging in 32-bit mode for this project. [9]

6 CONCLUDING REMARKS

To summarize the main points covered in this paper, I wrote this paper with the intention of both describing the work that I did as part of this project, as well as with the intention that it would act as a potential resource to those who attempt to implement paging and find their efforts frustrated by the general lack of resources on this topic in the same way that I did. To do this, I have first covered the background of how virtual memory management and paging work, then I described the components which I implemented to allow Xinu to support paging. In particular, I discussed the role and details of: a frame allocator; an abstraction of virtual memory into the logical components of kernel, user, and page table space; a kernel identity table to facilitate shared kernel space; mapping an entry of the page directory to itself to facilitate modification of page directory and page table entries; an abstraction of dynamic memory allocation through the memget() and memfree() syscalls; and several other small components necessary to connect each of these larger structures.

Overall I found this project to be exceptionally enjoyable to work on in a variety of ways. I found it rewarding to have the opportunity to work on a project which was low-level and focused on creating specific and complex structures in memory rather than focusing on the computational aspects of a system. I additionally found it rewarding to gain such a deep insight into the world of operating systems development, and this project helped me gain a much better understanding of how to take abstract operating systems concepts and turn them into actually implementable components. I would strongly recommend this sort of project to any student with a high level of self-motivation and good research skills who wishes to learn more about operating systems development, since over the course of this project I have learned much more about all aspects of operating systems than I imagine I ever could in a traditional operating systems course.

The completed project's source code can be found at https://github.com/ThomasGagne/Xinu-x86-Paging.

REFERENCES

[1] Douglas Comer. *Operating System Design: The Xinu Approach*. Second Edition. CRC Press, Taylor & Francis Group, 2015. 156—170.
[2] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. *Operating System Concepts*. Ninth Edition. John Wiley & Sons, Inc., 2013. 397—400.
[3] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. *Operating System Concepts*. Ninth Edition. John Wiley & Sons, Inc., 2013. 386—387.
[4] http://wiki.osdev.org/Paging.
[5] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. *Operating System Concepts*. Ninth Edition. John Wiley & Sons, Inc., 2013. 421—425.
[6] http://wiki.osdev.org/Page_Frame_Allocation.
[7] http://wiki.osdev.org/Higher_Half_Kernel.
[8] http://wiki.osdev.org/Identity_Paging.
[9] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. *Operating System Concepts*. Ninth Edition. John Wiley & Sons, Inc., 2013. 387.