CS232 Operating Systems
Assignment 2

Name: Muzammil Tariq
ID: mt04305

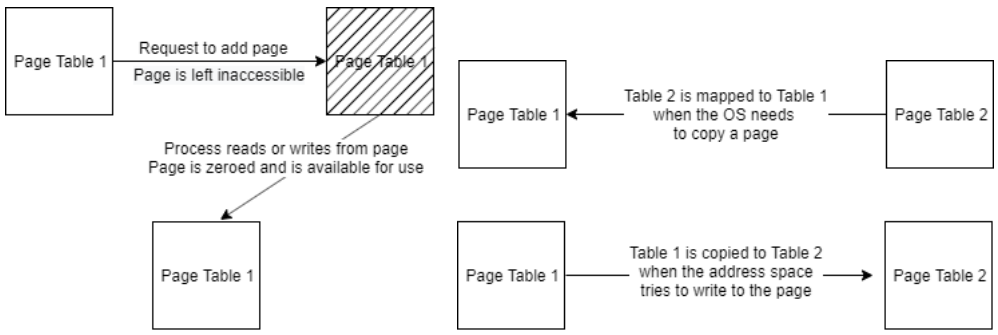
November 1, 2020

1 Virtual Memory Systems

1.1 Introduction

We have already studied the important elements of virtual memory systems where we covered the topics of how a page table is designed, how it interacts with the TLB, and how strategies are made to keep pages in memory or not. In reality there are more features that are part of a virtual memory system. In order to understand them we will study two such systems. The first one is VAX/VMS which was widely used in the 70s and 80s. The reason to study this is because parts of it are still used. The second one is the VMS of Linux which is used on many different types of devices. We will talk about each system in detail to bring forth how many types of techniques and logic build up a complete virtual memory system.

1.2 VAX/VMS



(a) Demand Zeroing (b) Copy-on-Write

Figure 1: Lazy Optimizations in VAX/VMS

Digital Equipment Corporation introduced the VAX-11 architecture in the 70s when they were at their peak. The OS used in this system was VAX/VMS. It was used in a lot of computers so it had to be made in a generalized form, hence it had a lot of flaws which were hidden by several techniques. The hardware of VAX-11 was limited and had a 32 bit virtual address space which was divided into 512 byte pages, which means it had a 23 bit VPN and 9 bit offset. Furthermore, the upper 2 bits of the VPN told what segment the page was in. This meant that the hardware was a mixture of paging and segmentation.

The system firstly had to reduce the burden page tables put on memory due to lack of hardware. It did so by dividing the user address into two and by putting user page tables into kernel virtual memory. This meant that the address space was getting more complex and so a real address space would not be just the code, data, and heap but also kernel data. Every

page table entry had several sets of bits all used for several purposes. In order to solve the issue of processes hogging up memory the developers came up with a new policy called segmented FIFO. What it did was that it let every process have a set number of pages in memory, and when the number exceeds the first in-page got removed. This was pretty simple itself so VMS had two second-chance lists, called clean-page free list and dirty-page list. A process was added into clean list if it was not modified and if it was then it was added in dirty list. VMS also used clustering to group up pages from the dirty list and wrote them in one go. This improved the performance of the VMS.

VMS also has two very interesting lazy optimizations under the hood which have now become industry standard. First is demand zeroing, which renders a page inaccessible until it has to be used. Second is copy-on-write, which maps a page to another page rather than copy it until a write operation is required. Both of these reduce the workload on the OS by delaying the required operation and are explained in Figure 1.

1.3 Linux

We will talk about Linux for Intel x86 as that is the most dominant one. The address space of Linux contains the usual heap, code, stack, and a kernel portion. The kernel portion remains same across all processes. It has two types of kernel addresses, kernel logical address which is the normal address space, and kernel virtual address which is normally not contagious so it is useful when a large space is needed. Since 32 bits were not enough for memory systems anymore they were upgraded to 64 bits. Some bits of this 64 bits are not used currently since there is no need for them as of yet. The first 12 bits are used for offset and the next 36 bits are divided into 4 as this is a 4 level page table.

Linux developers have incrementally added support for larger page sizes ranging up to 1 GB. The advantage this gave is quick action on a TLP miss but the biggest let down in this is the increasing chance of internal fragmentation because of the large size which means much of the space inside a table could go to waste. Swapping also does not work greatly in large pages.

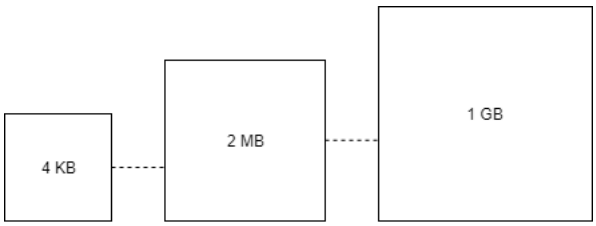


Figure 2: The incremental increase in page sizes

Linux uses its page cache to store popular items such as memory-mapped files, file and meta data, and heap and stack storage. These are stored in a hash table for quick lookup. The table keeps a lookout for dirty data so that it can be written back. Since Linux will have to clean up space in memory if it becomes full it uses a replacement policy called 2Q replacement. What this does in comparison to LRU is that it keeps track of pages in two lists, an active list and an inactive list. When a page is accessed for the first time it is added in the inactive list and if it is accessed the second time it is added int the active list. Replacement is done in the inactive list and periodically pages from the bottom of the active list are kicked back into inactive list too.

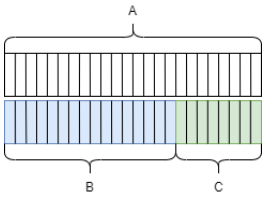


Figure 3: Buffer overflow happens when the computer tries to copy A into B and overrides C

Security has become an important thing in modern systems so Linux had to employ clever tactics to deal with malicious attacks. In order to deal with buffer overflow where extra memory is overwritten a bit called NX-bit is added onto the page table entry of that page which tells whether the code should be executed or not. The way Linux deals with return-oriented programming, where the return address is modified in such a way that it refers to malicious code, is by randomizing the location where the code,heap, and stack

are placed. This technique is called address space layout randomization(ASLR). ASLR has been applied to kernel too but kernel has security issues of its own such as meltdown and spectre. They exploit the tricks used to increase performance. A page table called kernel page-table isolation(KPTI) was created to set apart the data that the computer executes when guessing what instruction will run next, an idea called speculative execution.

1.4 Conclusion

This was a short summary of how both of these virtual memory systems work. This is by no way a complete description of them as there is a lot of nitty gritty detail to these. Linux in itself is a very complex and detailed system and gets ideas from techniques used in the past.

2 Securing the Buffer

Since computer systems keep changing and upgrading every year their security also needs to improve. One such is buffer overflow. There can be several ways to tackle this but one excellent way is to run a trace on the code beforehand which recognizes any such anomalies. The compiler picks these up usually but sometimes they can get unnoticed when pointers are used. At this point artificial intelligence can be involved to guess which part of the code could produce a buffer overflow. It would use a simple algorithm such as linear regression to predict if that part of the code would produce issues or not, if it does then it would warn the user of the said code and not execute it. This AI will be trained by a rather large dataset of malicious and not malicious code, and even more the AI will be a simple AI because it will only classify the code as malicious or not malicious. This is demonstrated in Figure 4.

Code	Malicious
Code 1	Yes
Code 2	No
Code 3	No
Code 4	Yes
Code 5	No
Code 6	No
Code 7	No

Figure 4: The malicious and non-malicious code will be classified as such

References

[1] X. Jia, C. Zhang, P. Su, Y. Yang, H. Huang and D. Feng "Towards Efficient Heap Overflow Discovery," 26th USENIX Security Symposium, pp. 989-1006, 2017.

[2] Emery D. Berger and Gene Novark, "DieHarder: Securing the Heap," University of Massachusetts Amherst, 2010.

[3] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, "Complete Virtual Memory Systems," in Operating Systems: Three Easy Pieces, Wisconsin, University of Wisconsin, 2019, pp. 261-278.