Operating Systems – Project 2

- Virtual Memory (Paging) -

ChangYoon Lee, ChanHo Park

Dankook University, Korea Republic
32183641@gmail.com, chanho.park@dankook.ac.kr

Abstract. Virtual memory is a huge notion that enables us to develop program easily and protect our program from the other malicious programs. With virtual memory, each process can access memory as it owns the entire memory. In this paper, we will discuss the concept of main memory, virtual memory, and the additional concepts that are used for these concepts. Also, we will implement a virtual memory (paging) program that operates in a similar way to the actual virtual memory operation in the operating system. At the end of this project, we will present the result of the execution of a virtual memory (paging) program and evaluate it according to the different replacement algorithms and size of the table look-aside buffer (TLB).

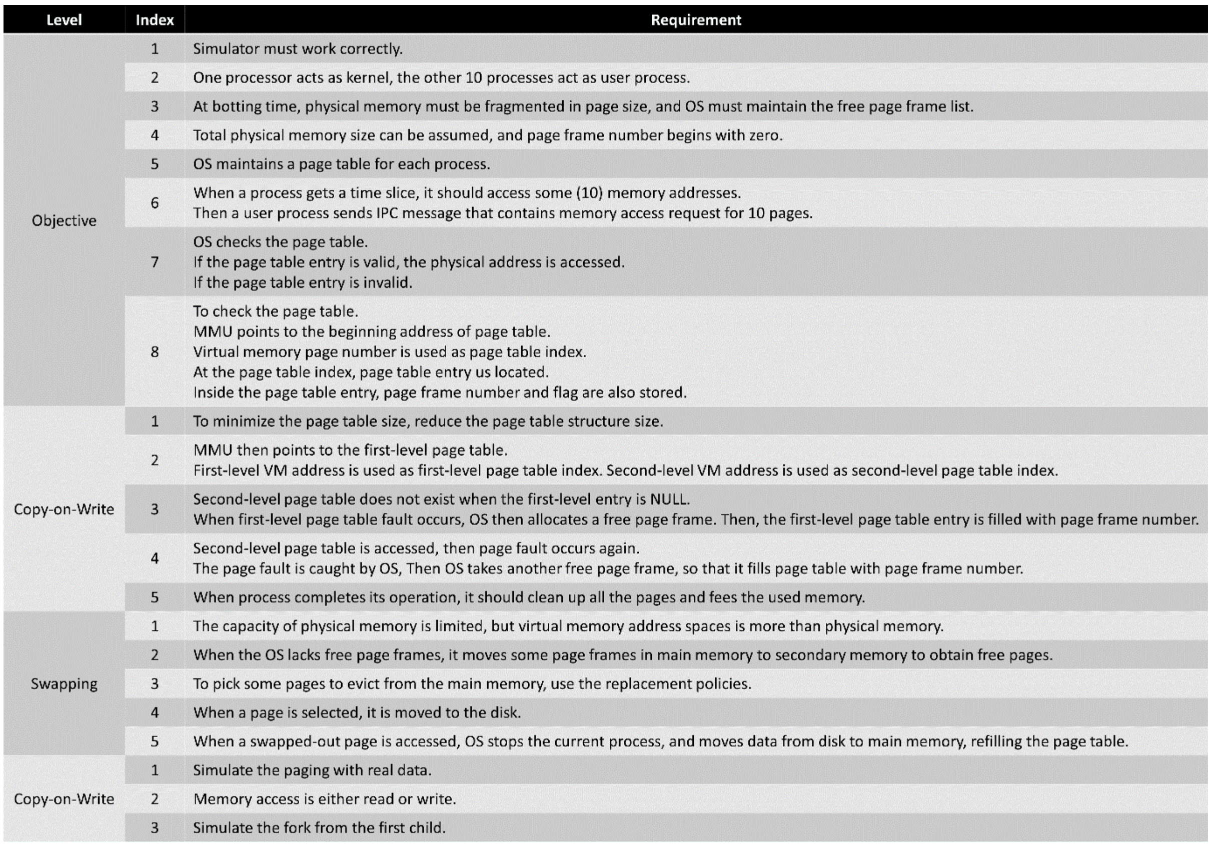
Keywords: Virtual Memory, Paging, Segmentation, Replacement Algorithms.

1 Introduction

Virtual memory is a method that enables the running of processes that are not entirely in memory. One significant advantage of virtual memory is that programs can be larger than physical memory. Additionally, virtual memory separates logical memory as seen by the programmer from physical memory by abstracting primary memory into an enormous, homogeneous array of storage. The following concept frees programmers from the concerns of limitations of memory storage. However, virtual memory is hard to implement and may substantially decrease performance if used carelessly.

In this paper, we will first explain the concepts of main memory, which contains the concepts of the difference between physical and logical memory address, paging, table look-aside buffer (TLB), protection, shared page, hierarchical paging, and swapping. Also, we will explain the concept of virtual memory, which contains the concepts of demand paging, free frame list, performance of demand paging, copy-on-write, and page replacement policies. By applying these concepts, we will explain how we implemented the virtual memory (paging) program and its results for different replacement algorithms, such as random, first-in-first-out (FIFO), least recently used (LRU), least frequently used (LFU), most frequently used (MFU), second chance (SCA), and enhanced second chance (ESCA) algorithms, and sizes of TLB. Also, we will show the performance of each replacement algorithm and the size of the TLB based on the features discussed in a later section. At the end of the paper, we will present the result of the execution of the different replacement algorithms and the size of the TLB and evaluate them.

2 Requirements



Level	Index	Requirement
Objective	1	Simulator must work correctly.
	2	One processor acts as kernel, the other 10 processes act as user process.
	3	At botting time, physical memory must be fragmented in page size, and OS must maintain the free page frame list.
	4	Total physical memory size can be assumed, and page frame number begins with zero.
	5	OS maintains a page table for each process.
	6	When a process gets a time slice, it should access some (10) memory addresses. Then a user process sends IPC message that contains memory access request for 10 pages.
	7	OS checks the page table. If the page table entry is valid, the physical address is accessed. If the page table entry is invalid.
	8	To check the page table. MMU points to the beginning address of page table. Virtual memory page number is used as page table index. At the page table index, page table entry us located. Inside the page table entry, page frame number and flag are also stored.
Copy-on-Write	1	To minimize the page table size, reduce the page table structure size.
	2	MMU then points to the first-level page table. First-level VM address is used as first-level page table index. Second-level VM address is used as second-level page table index.
	3	Second-level page table does not exist when the first-level entry is NULL. When first-level page table fault occurs, OS then allocates a free page frame. Then, the first-level page table entry is filled with page frame number.
	4	Second-level page table is accessed, then page fault occurs again. The page fault is caught by OS, Then OS takes another free page frame, so that it fills page table with page frame number.
	5	When process completes its operation, it should clean up all the pages and fees the used memory.
Swapping	1	The capacity of physical memory is limited, but virtual memory address spaces is more than physical memory.
	2	When the OS lacks free page frames, it moves some page frames in main memory to secondary memory to obtain free pages.
	3	To pick some pages to evict from the main memory, use the replacement policies.
	4	When a page is selected, it is moved to the disk.
	5	When a swapped-out page is accessed, OS stops the current process, and moves data from disk to main memory, refilling the page table.
Copy-on-Write	1	Simulate the paging with real data.
	2	Memory access is either read or write.
	3	Simulate the fork from the first child.

Figure 1 - Requirement Specification

Figure 1 shows the requirements for a virtual memory (paging) program of each implementation levels. The implementations for these requirements will be described in detail afterwards.

3 Concepts

3.1 Main Memory

A modern computer system's main memory is essential to its functionality. Memory is made up of a sizable array of bytes, each of which has a unique address. In line with the program counter's value, the CPU fetches instructions from memory. These instructions may result in more loading into and storing of memory addresses.

Only a stream of memory addresses is visible to the memory unit. They are unaware of how they are produced or their intended use. This indicates that memory is only concerned with the list of memory addresses produced by the running program.

3.1.1 Logical Versus Physical Address Space

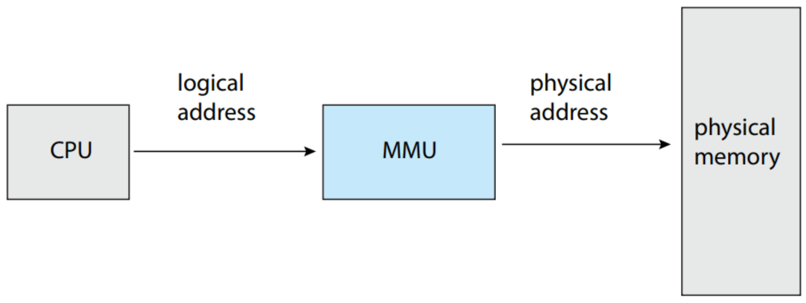


Figure 2 - Memory Management Unit (MMU) (Silberschatz et al., 2014)

The term "logical address" refers to an address created by the CPU, but the term "physical address" refers to an address seen by the memory unit, which is to state, the address that is loaded into the memory-address register of the memory.

The logical and physical addresses produced by binding an address at either compile time or load time are the same. However, different logical and physical addresses come from the execution-time address-binding concept. In this instance, the logical address is referred to as a virtual address. Logical address space is the collection of all logical addresses produced by a program. A physical address space is the collection of all the physical addresses that map to these logical addresses. The logical and physical address spaces differ in the execution time to address binding concepts.

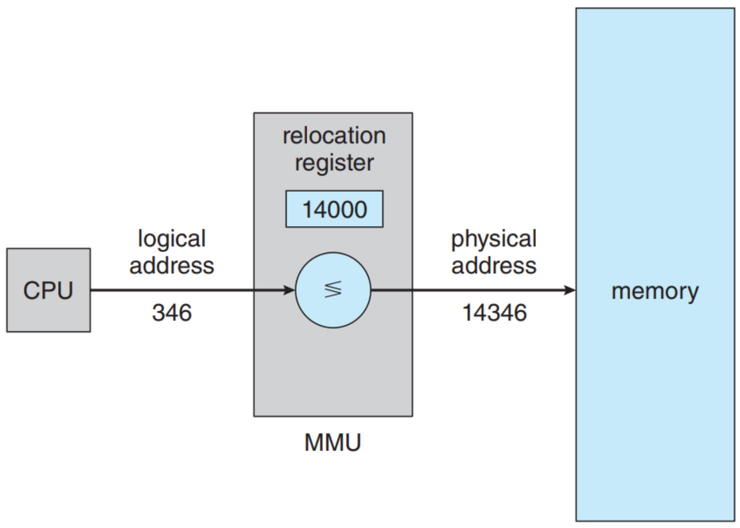


Figure 3 - Dynamic Reallocation using a Reallocation Register (Silberschatz et al., 2014)

The memory management unit (MMU), a hardware component seen in Figure 2, performs the run-time translation from virtual to physical addresses. Every address generated by a user process is added to the reallocation register at the time the address is sent to memory, which is where the smallest legal physical memory address is kept. The following process is presented in Figure 3.

The actual physical addresses are never accessed by the user software. A pointer to a location can be created by the program, stored in memory, and then used in various ways, including comparisons with other addresses. Logical addresses are dealt with by the user software. Logical addresses are transformed into physical addresses by the memory mapping circuitry. A memory address that is referenced cannot be identified until the reference is made.

The two types of addresses we currently have been logical and physical addresses. Only logical addresses are generated by the user software, and it assumes that the process occupies memory spaces ranging from zero to maximum. Before being used, these logical addresses must be mapped to physical addresses. Proper memory management is based on the concept of a logical address space that is connected to a separate physical address space.

3.1.2 Paging

he idea of paging allows physical address space to be non-contiguous in memory management. Paging avoids contiguous memory allocation concerns with external fragmentation and the associated necessity for compaction. Paging, in all its incarnations, is utilized in most operating systems because it has several benefits. The operating system and the computer hardware work together to implement paging.

The fundamental approach to implementing paging involves dividing physical memory into fixed-sized blocks known as frames and logical memory into identical-sized blocks known as pages. The process pages are loaded from their source into any open memory frames or clusters before execution. A cluster of numerous frames or fixed-sized blocks the same size as the memory frames make up the backing store.

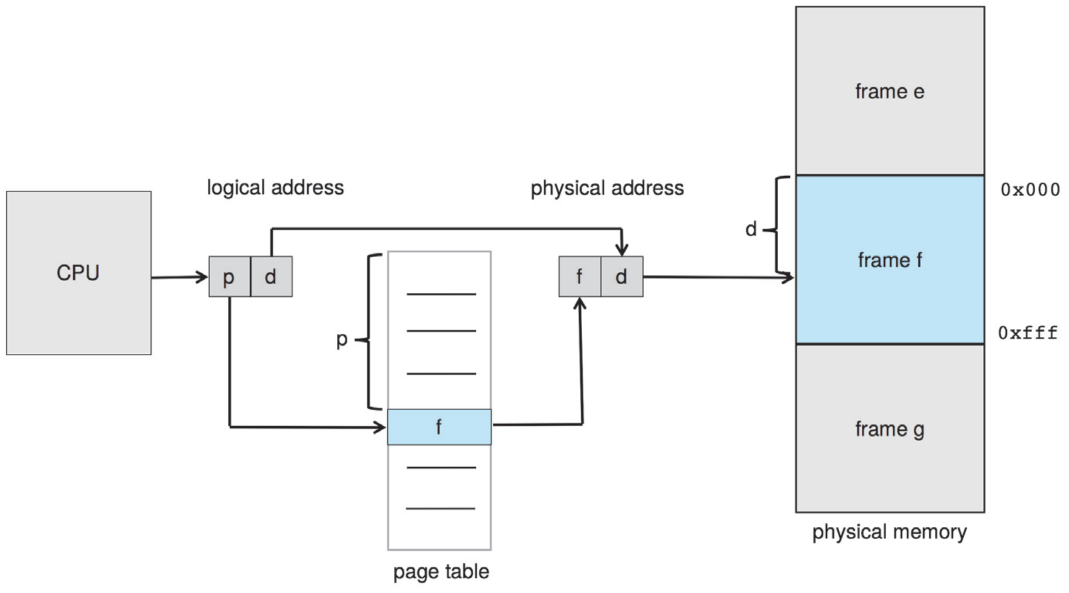


Figure 4 - Paging Hardware (Silberschatz et al., 2014)

Every address produced by the CPU has two components: a page number (p) and a page offset (d). The pre-process page table can be accessed using the page number as an index. Figure 4 shows the following structure. The offset represents the location in the frame being referenced, while the page table provides the actual address of each frame in physical memory. The physical memory address is also determined by combining the frame's base address and page offset.

The hardware determines the page size. Depending on the computer architecture, a page might range in size from 4 KB to 1 GB. The logical address space is translated into 2^m bytes when a power of 2 is chosen for the page size 2^n. The high-order m-n bits of a logical address then identify the page number, and the n low-order bits designate the page offset.

When we apply the concept of paging, there is no external fragmentation. Any available frame can be given to a process in need. We might, however, be internally fractured. You'll see that units called frames are assigned. The last frame allocated cannot be filled if a process's memory needs do not align with page boundaries. The distinct division between the programmer's perception of memory and the actual physical memory is a key feature of paging. Memory is seen by the programmer as a single area that only contains this one program. The user program, which also contains other applications, is dispersed throughout physical memory. Addressing translation hardware bridges the gap between physical memory and how a programmer perceives it. It converts logical addresses into physical addresses. The operating system oversees this mapping, which is concealed from the programmer.

The operating system must be aware of the details of physical memory allocation, including which frames are allocated, which frames are available, how many total frames there are, and other information, since it is responsible for managing physical memory. Typically, a frame table, a type of data structure, is used to store this data. Each physical page frame includes a single record in the frame table that indicates whether it is free or assigned, and if allocated, to which page of which process.

Additionally, all logical addresses must be mapped to physical addresses by the operating system so that it is aware that user processes run in user space. An address that a user passes into a system call as a parameter must be mapped to produce the right physical address. The instruction counter and register contents are copies that the operating system keeps on hand. When the operating system needs manually map a logical address to a physical address, this copy is utilized to convert logical addresses to physical addresses. It is also employed. Paging thereby increases the time needed to switch contexts.

3.1.3 Hardware Support

Since page tables are per-process data structures, a pointer to the page table is stored in the process control block of each process along with the other register values. The user registers and corresponding hardware page table values must be reloaded from the stored user page table when the CPU scheduler chooses to execute a process.

There are various ways to implement the page table on hardware. The page table can simply be implemented as a group of specific high-speed hardware registers. The following method, however, extends the time it takes to switch contexts because each of these registers must be exchanged.

Since most modern CPUs accommodate considerably larger page tables, the page table is kept in the main memory, and a page table base register (PTBR) links to it. Changing page tables requires only one register to be changed, significantly reducing context-switch time.

3.1.4 Translation Look-Aside Buffer (TLB)

Faster context switches can be attained by keeping the page table in the main memory, although longer memory access times may also follow. Let's say we want to locate something. Using the values in the PTBR offset by the location page number, we must first index them into the page table. This operation requires one memory access. The frame number is given to us, and when combined with the page offset, it yields the real address. Then, we got to the memory location we wanted. This operation requires two memory accesses to access the data. In other words, memory access causes total operating delays, which are typically intolerable in most circumstances.

The use of a translation look-aside buffer, a unique, compact, rapid lookup hardware cache, is the conventional solution to the following issue (TLB). TLB is a quick, associative memory. The key and the value are the two components that make up each entry in the TLB. An item is compared with all keys at once when associative memory is presented with it. The corresponding value field is returned if the item is found. The search operation is fast.

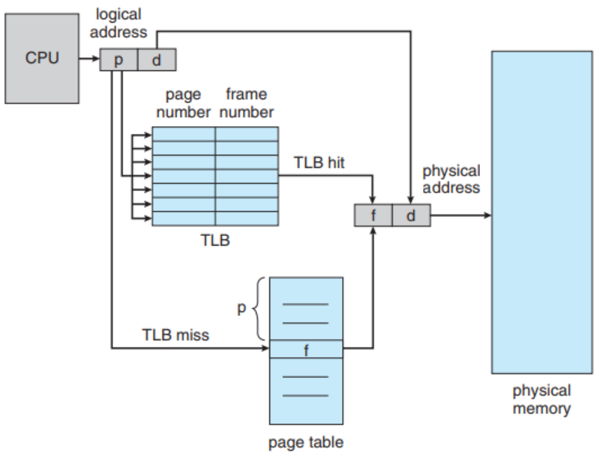


Figure 5 - Paging Hardware with TLB (Silberschatz et al., 2014)

If the page number is not in the TLB, which is known as a TLB miss, it extracts the page number and uses it as an index into the page table, extracts the appropriate fame number from the page table, replaces the page number in the logical address with the frame number, and adds the page number and frame number to the TLB so that the frame number can be quickly accessed on the next reference. The following operation is presented in Figure 5.

If the TLB is already full of entries, an existing entry must be selected for replacement. Replacement policies range from least recently used (LRU) through round-robin to random. However, TLB is usually organized as small-sized, which is 128 to 512 entries, and fully associative cache, due to the fast access.

TLBs are a hardware feature, thus an operating system wouldn't appear to be too concerned with them. The purpose and characteristics of TLBs, which vary depending on the hardware architecture, should be understood. An operating system must implement paging following the TLB design of a specific architecture for optimal operation. A change in the TLB design may also need a change in how the operating systems that use it implement paging.

3.1.5 Protection

Memory protection in a paged environment is accomplished by protection bits associated with each frame. Generally, these bits are kept in the page table.

One bit can determine whether a page is read-only or read-write. All memory access uses the page table to determine the appropriate frame number. The protection bits can be verified to confirm that no writing is being done on read-only pages while the physical address is being calculated. The operating system encounters a hardware trap when trying to write to a read-only page.

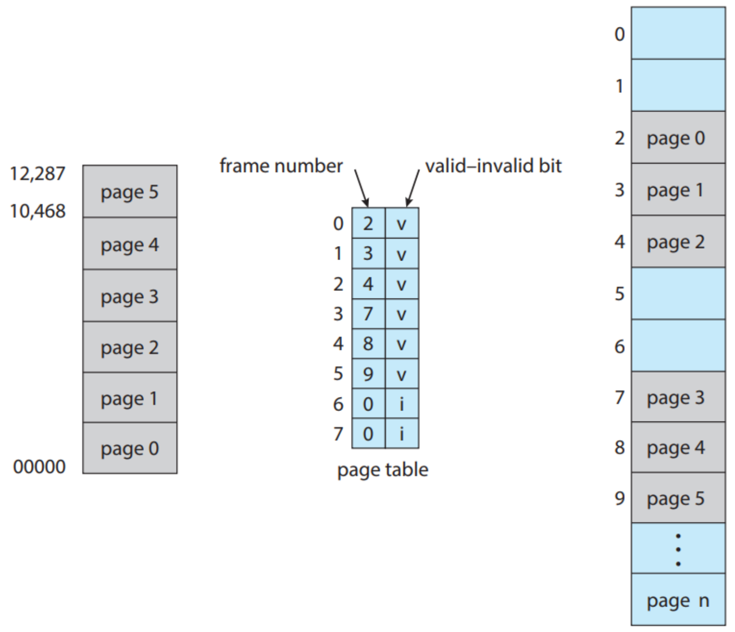


Figure 6 - Valid-Invalid bit in a Page Table (Silberschatz et al., 2014)

Each entry in the page table often has a valid-invalid bit as an additional bit. The associated page is legal and is in the process of logical address space when this bit is set to valid. The page is not in the process of logical address space when the bit is set to invalid. The valid-invalid bit is used to detect illegal addresses. To permit or deny access to a page, the operating system sets this bit for each page. The following bits are presented in Figure 6.

The process rarely makes use of its entire address range. Many programs only utilize a small portion of the available address space. In these situations, generating a page table with entries for each page would be a waste of precious RAM. To show the size of the page table, some systems offer hardware in the form of a page table length register (PTLR). Every logical address is compared to this number to ensure that it falls within the acceptable range for the process. If this test is unsuccessful, the operating system will fall into an error trap.

3.1.6 Shared Pages

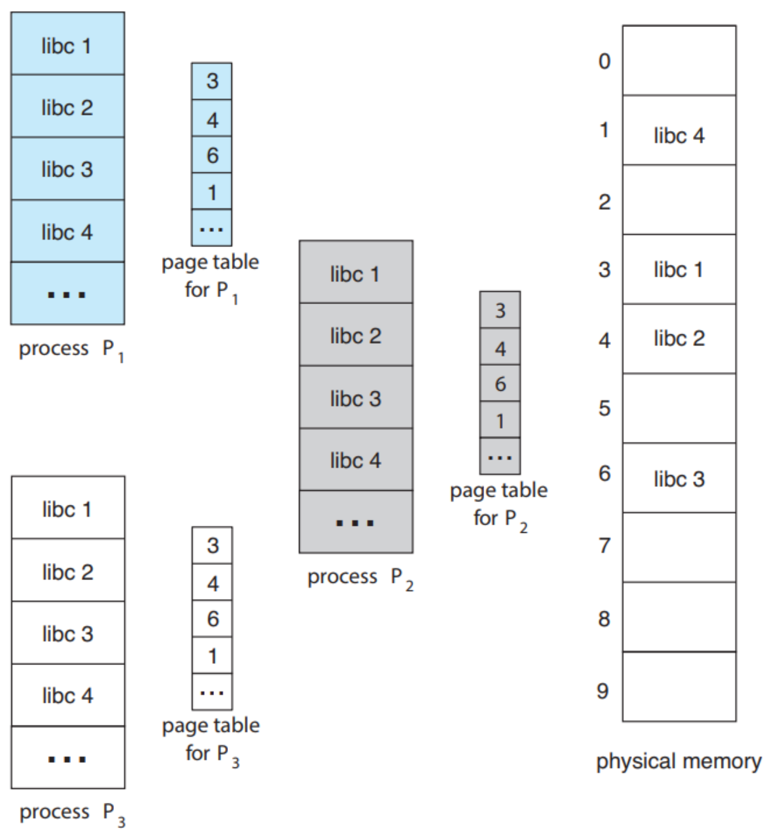


Figure 7 - Sharing of Library in a Paging Environment (Silberschatz et al., 2014)

Paging has the benefit of allowing for the sharing of common code; an environment with numerous processes makes this factor especially crucial. The code can be shared though if it is reentrant. Figure 7 shows the case in question. Reentrant code is a core that is not self-modifying. Throughout the course of the show, it never changes. Additionally, the same code may be executed concurrently by two or more processes. To hold the data needed for the execution of the process, each process has its own set of registers and data storage. There will be differences in the data for two separate processes. Since each user process's page table corresponds to the same physical copy of the library, only one copy of the library needs to be stored in memory. We can cut down on duplicate pages and preserve the memory spaces in this situation.

Along with enabling many processes to share the same physical pages, organizing memory according to pages has many other advantages.

3.1.7 Hierarchical Paging

Most modern computer systems support a large logical address space (2^32 to 2^64). In such an environment, the page table itself becomes excessively large. We do not want to allocate the page table contiguously in the main memory. One simple solution to the following problem is to divide the page table into smaller pieces.

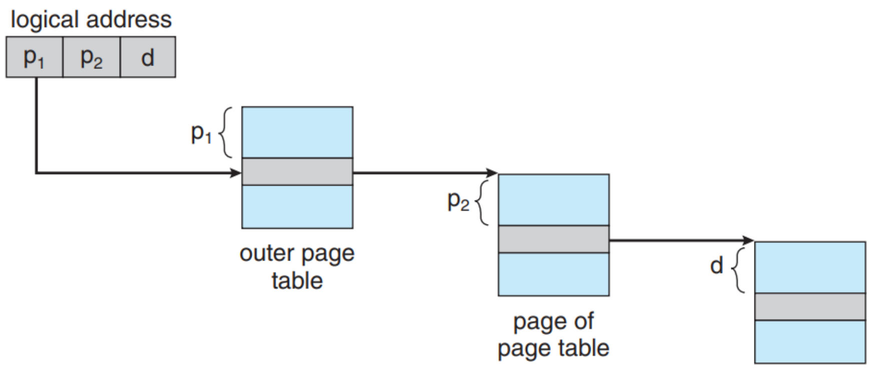


Figure 8 - Address Translation for a Two-Level 32-Bit Paging Architecture (Silberschatz et al., 2014)

Utilizing a two-level paging technique in which the page table is also a page is one option. Think about a system with a 32-bit logical address space and a 4 KB page size, for instance. The page number, which is made up of 20 bits, plus the page offset, which is made up of 12 bits, make up a logical address. The page number is further separated into a 10-bit page number and a 10-bit page offset since we page the page table. The first 10 bits of the address will become the outer page table index, and the next 10 bits will become the inner page table index. Figure 8 shows the address translation technique for this design. The following design is sometimes known as a forward-mapped page table since address translation occurs from the outer page table to the inner page table.

3.1.8 Swapping

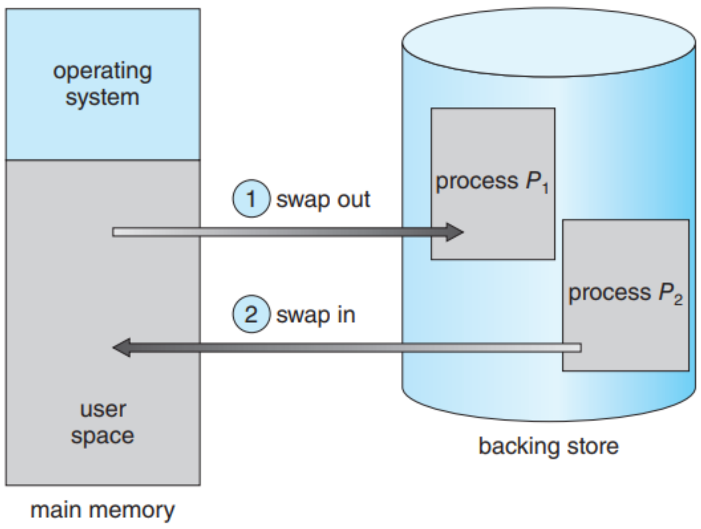


Figure 9 - Swapping of Two Processes using a Disk as a Backing Store (Silberschatz et al., 2014)

To be carried out, process instructions need to be loaded into memory together with the data they operate on. To continue execution, a process or a section of a process can be temporarily swapped to a backing store and then brought back into the memory, as shown in Figure 9. Swapping increases the amount of multiprogramming in a system by enabling the entire physical address space of all processes to exceed the system's actual physical memory.

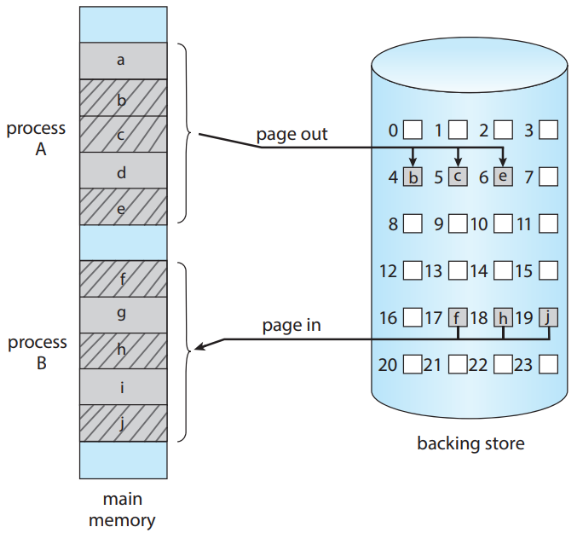


Figure 10 - Swapping with Paging (Silberschatz et al., 2014)

Most systems, including the Linux and Windows, use a variation of swapping in which pages of a process, not an entire process, can be swapped. The following strategy still allows physical memory to be oversubscribed but does not incur the cost of swapping the entire processes, as presumably only a small number of pages will be involved in swapping. A page out operation moves a page from the memory to the backing store. The reverse process is known as a page in. Swapping with paging is presented on Figure 10.

3.2 Virtual Memory

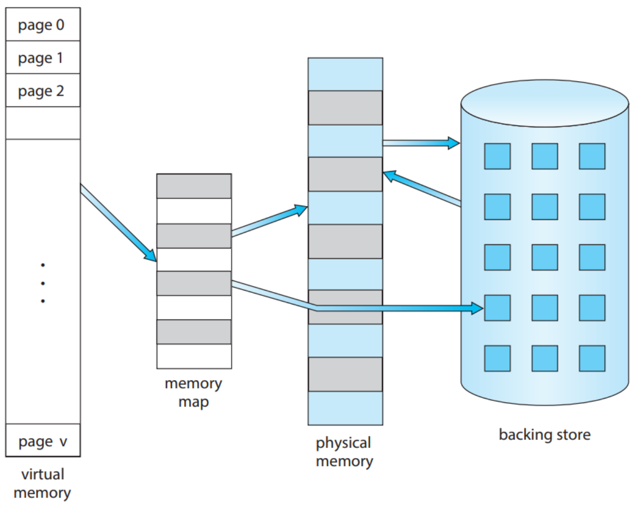


Figure 11 - Virtual Memory that is Larger than Physical Memory (Silberschatz et al., 2014)

Virtual memory is the separation of physical memory from what programmers consider to be logical memory. When just a limited physical memory is available, this separation enables an enormously large virtual memory to be supplied for programmers. Figure 11 shows the separation that occurs. Since the programmer is no longer concerned with the amount of physical memory available, programming becomes significantly simpler.

The logical view of how a process is stored in memory is referred to as its virtual address space. This viewpoint often holds that a process starts at a specific logical location. The physical page frames that are assigned to a process may not all be contiguous since physical memory is arranged in this method. The task of mapping logical pages to physical page frames in the memory belongs to the memory management unit (MMU).

3.2.1 Demand Paging

At the time of program execution, we can load the full executable program into physical memory from the secondary storage. The following method has a drawback in that we might not initially require the complete program to be in memory. Whether or whether an option is finally chosen by the user, loading the complete program into memory causes executable code for all options to be loaded.

A different approach is to only load pages as they are required. Demand paging, sometimes known as this approach, is frequently applied in virtual memory systems. Pages are only loaded into virtual memory with demand pages when they are required for program execution. Therefore, no physical memory is ever loaded with pages that are never accessible. Like a paging system with swapping, a demand-paging system places processes in secondary memory. Demand paging describes one of the main advantages of virtual memory: memory is used more effectively since only the necessary bits of applications are loaded.

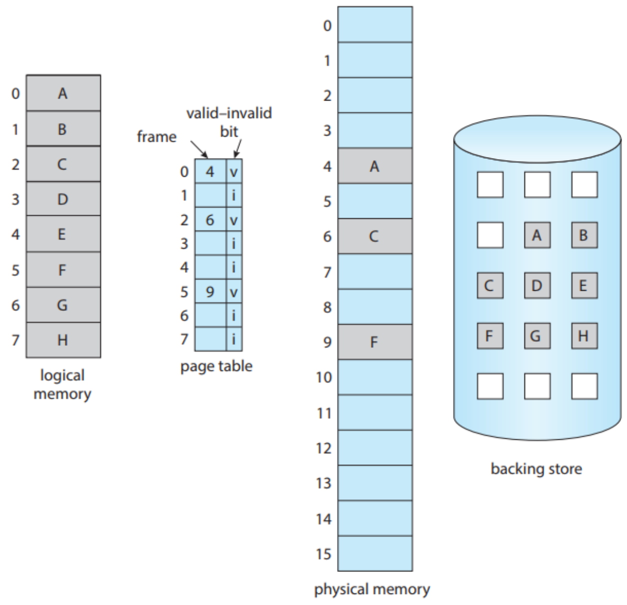


Figure 12 - Page Table when Some Pages are not in Main Memory (Silberschatz et al., 2014)

Demand paging's central thesis is to only load a page into memory when it is required. As a result, some pages will be in memory, and some will be in secondary storage while a process is running. Additionally, we require hardware support to distinguish between the two. The following uses are possible for the valid-invalid bit concept. However, the corresponding page is valid and present in the memory when the bit is set to valid. The page is either invalid or valid but is currently stored in secondary storage if the bit is set to invalid. A page that is currently in memory merely has its page table entry flagged as invalid. The situation shown in Figure 12 is detailed.

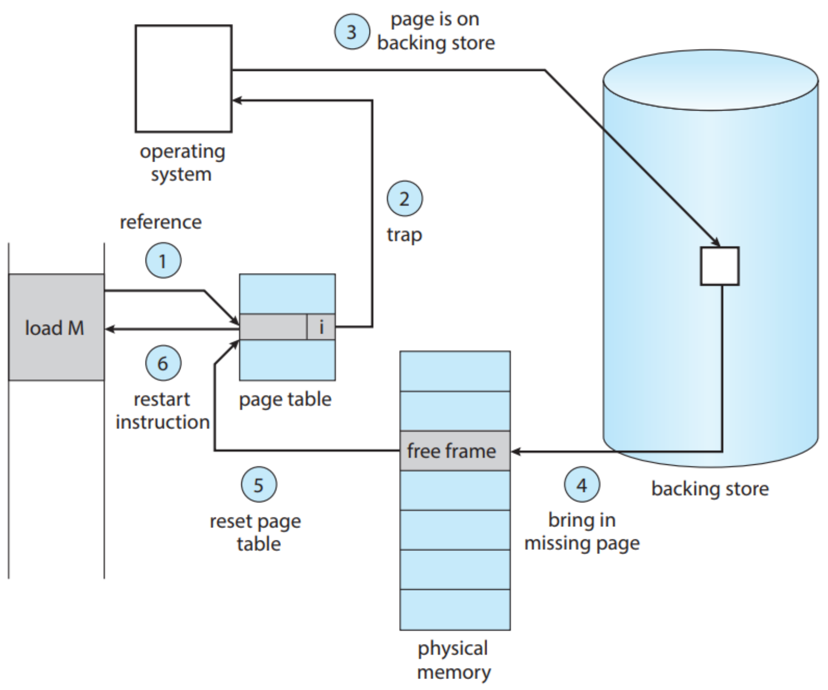


Figure 13 - Handling a Page Fault (Silberschatz et al., 2014)

However, access to a page that has been marked as invalid will result in a page fault if the process tries to access one that has not been brought into memory. When translating the address across the page table, the paging hardware will detect that the incorrect bit is set, trapping the operating system. The operating system's failure to load the desired page into memory led to the creation of this trap. Figure 13 shows the process for handling this page fault.

1. We checked the internal table for this process to determine whether the reference was valid or invalid memory access.
2. If the reference is invalid, we terminate the process. If it is valid but we have not yet brought it to that page, we now page it in.
3. We found a free frame.
4. We have scheduled a secondary storage operation to read the desired page into the newly allocated frame.
5. Once the storage read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

The ability to continue any instruction after a page fault is a critical requirement for demand paging. When a page fault happens, we save the state of the interrupted process, thus we must be able to restart the process at the same location and state, with the exception that the needed page is now in memory and is reachable. This criterion is often met by everyone. Any memory reference can experience a page fault. We can restart by fetching the instructions once more if the page fault happens during the fetch of the instructions. When fetching an operand when a page fault occurs, we must first fetch and decode the instruction before retrieving the operand.

In a computer system, paging is added between the CPU and the memory. An entire process should be transparent to it. Therefore, it's a common misconception that paging may be implemented in any system. This assumption is accurate for non-demand paging environments where a page fault is a fatal mistake, but it is incorrect for environments where a page fault just necessitates bringing a new page into memory and restarting the process.

3.2.2 Free Frame List

The operating system must transfer the desired page from secondary storage into main memory when a page fault occurs. Most operating systems keep a free-frame list, which is a pool of available frames for handling such requests, to resolve page faults. Operating systems often use a method called zero-fill-on-demand to distribute free frames. Before being allocated, zero-fill-on-demand frames are zeroed out, deleting their prior contents.

A system's free-frame list is initially filled with all the memory that is accessible. The size of the free-frames list decreases as more frames are requested. The list must be repopulated when it eventually reaches zero or drops below a predetermined threshold.

3.2.3 Copy on Write (CoW)

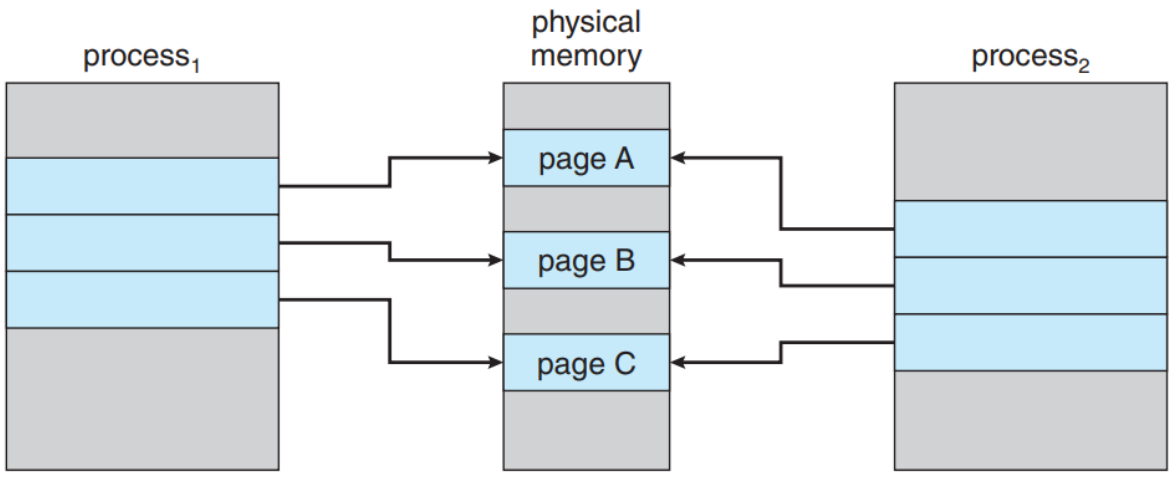


Figure 14 - Initial Phase of Copy-on-Write Operation (Silberschatz et al., 2014)

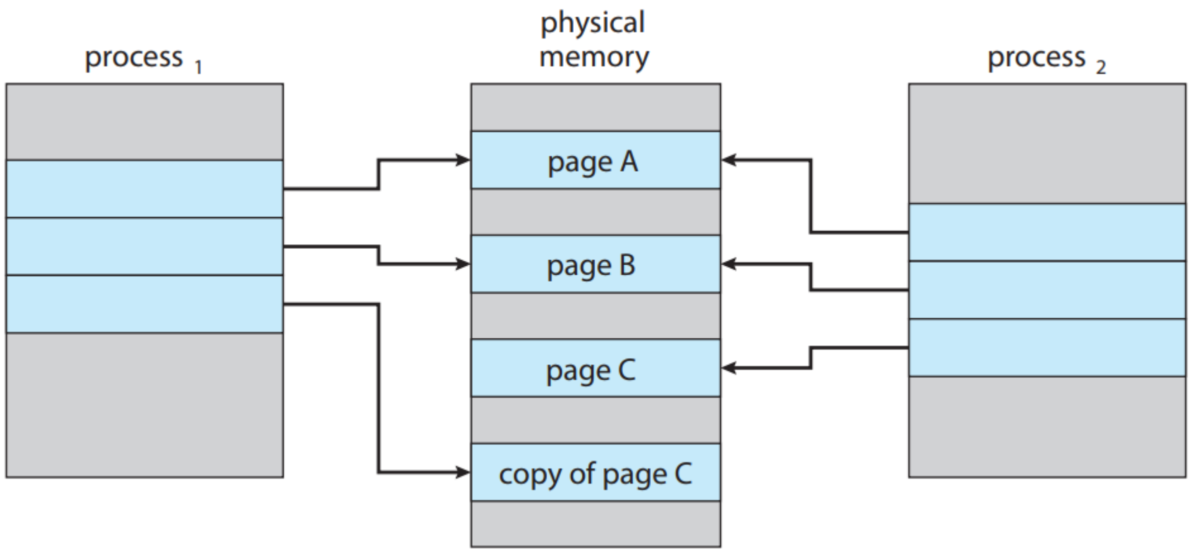


Figure 15 - After Modifying a Page on Copy-on-Write Operation (Silberschatz et al., 2014)

A copy of its parent process is created by the fork system call in the child process. A copy of the parent's address space was traditionally made for the child for the fork system call to the function. duplicate copies of the parent page. The copying of the parent's address space might not be required, though, given that many child processes immediately invoke the exec system call after being created. Instead, we can make use of a method called copy-on-write, which functions by first letting the parent and child processes share the same pages. These shared pages are designated as copy-on-write pages, which means that a copy of the shared page is made whenever either process is written to one. Figures 14 and 15 show how the copy-on-write works.

Only the pages that are modified by either procedure are copied when the copy-on-write method is used. The parent and child processes can share any unmodified pages. Only modifiable pages need to be tagged as copy-on-write. The parent and child can share pages that cannot be changed. Many operating systems, including Windows, Linux, and macOS, use the copy-on-write mechanism.

3.2.4 Page Replacement

Increased multiprogramming results in excessive memory allocation. The following circumstances are a result of excessive memory allocation. A page fault happens when a process is running. The operating system identified the location of the needed page on secondary storage but discovered that the free-frame list was empty.

Currently, the operating system provides several options. It might stop the procedure. Demand paging, on the other hand, is an attempt by the operating system to increase throughput and system usage. Users shouldn't be aware that they are using a paged system to run their operations. Paging should be clear to the user. As a result, the procedure that follows is not the best option.

To free up all its frames and lower the level of multiprogramming, the operating system could use ordinary swapping and switch-out procedures. Operating systems no longer employ the following technique, nevertheless, because of the overhead involved in transferring a complete process between memory and swap space. Currently, most operating systems combine page replacement and page swapping.

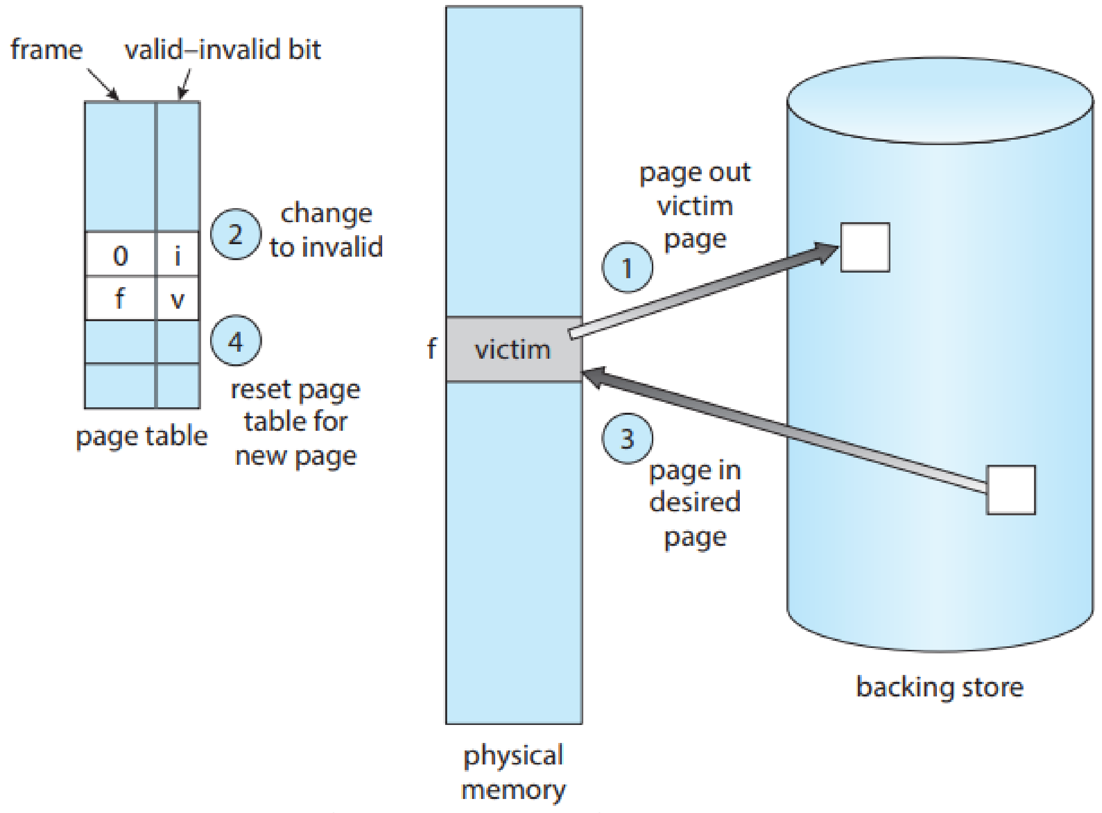


Figure 16 - Page Replacement (Silberschatz et al., 2014)

The following strategy is used for page replacement. If no frames are available, we can locate one that isn't currently being used and release it. By swapping space with its contents and updating the page table to indicate the page's absence from memory, we can release a frame. The following situation is presented in Figure 16. The page for which the process failed can now be held in the free frame. We have added a page replacement to the page-fault service routine:

1. Locate the appropriate page on the secondary storage system.
2. Find a free frame
   A. If there is a free frame, use it.
   B. Use a page-replacement policy to pick a victim frame if there isn't a free frame.
   C. Save the victim frame to secondary memory. Modify the frame and page tables.
3. Read the desired page into the empty frame. Modify the frame and page tables.
4. Continue the process from the point at where the page fault happened.

Two-page transfers are necessary if there are no available frames. The subsequent circumstance effectively doubles the page fault service time and raises the effective access time in proportion.

Using a modified bit, commonly referred to as a dirty bit, we can lower the following overhead. Each page or frame has a changed bit attached to it in the hardware when this concept is employed. Every time a byte is written into a page, the hardware sets the page's modified bit to indicate that the page has been modified. We look at the modified bit of the page before choosing it for replacement. If the bit is set, we know that since it was read from secondary storage, the page has been modified. We must write the page to storage in this situation. The page has not been modified since it was read into memory, though, if the modified bit is not set. We don't need to write the memory page to the storage in this situation. There it is already. The next method can also be used with read-only pages. Such pages may be deleted at any time and cannot be changed. The following idea, which cuts I/O time in half if the page hasn't been modified, can significantly save the amount of time needed to handle a page fault.

Demand paging requires page replacement as a basic. It completes the division of physical memory from logical memory. With the help of this technology, a programmer can access a large virtual memory on a small physical memory. Logical addresses are translated into physical addresses without demand paging; thus, the two sets of addresses may differ. All the process' pages must still be physically stored in memory, though. Demand paging eliminates the physical memory restriction on the logical address space.

To implement demand paging, we must find a solution to two significant issues: creating a frame-allocation mechanism and a page replacement technique. We must choose how many frames to allocate to each process if numerous processes are running in memory, and we must choose which frames to replace when a page replacement is necessary. Since secondary storage I/O is expensive, it is crucial to design the right algorithms to address these issues. Demand-paging techniques that are only slightly improved result in significant system performance increases.

Page replacement algorithms come in a variety of forms. Almost every operating system has a unique replacement plan. The lowest page fault rate is the objective of these algorithms. In the following sections, several page replacement algorithms will be covered.

4 Virtual Memory (Paging)

4.1 Memory Management Unit (MMU)

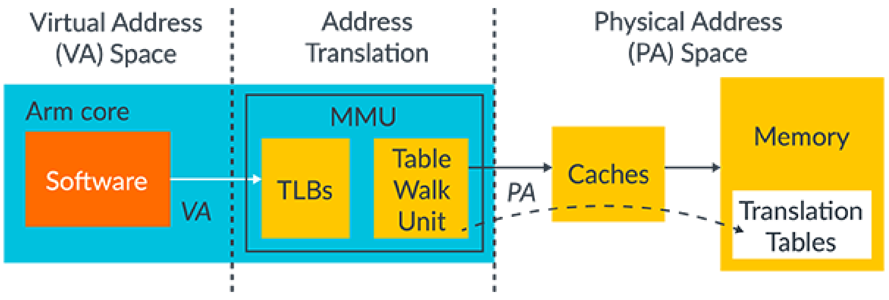


Figure 17 - Memory Management Unit (MMU) (*Documentation – arm developer*)

A hardware component called a memory management unit (MMU) converts a virtual address into a physical address. Every memory reference passes through the MMU. The virtual address is converted into a physical address. Virtual addresses used by the software must be converted into physical addresses used in the memory system by the MMU (Documentation – arm developer). MMU has the following characteristics:

- The table walk unit, which has logic that takes translation tables out of memory.
- Translation Lookaside Buffers (TLBs), which store translation that has recently been used.

Virtual memory addresses are used exclusively by software. MMU receives these memory addresses and queries TLBs for any recently cached translations. The table walk unit reads the required table entry, or entries, from memory if the MMU is unable to locate a recently cached translation, as shown in Figure 17.

Before enabling memory access, a virtual address must be converted to a physical address. Data that has been cached also requires translation since, in later processes, physical addresses are used to access the data. Therefore, before a cache lookup is finished, the address must be translated.

In this project, a memory management unit (MMU) is used for the translation between the virtual addresses and physical addresses, corresponding with two-level paging and the table lookaside buffer. The following features will be described in the later sections.

4.2 Two-Level Paging

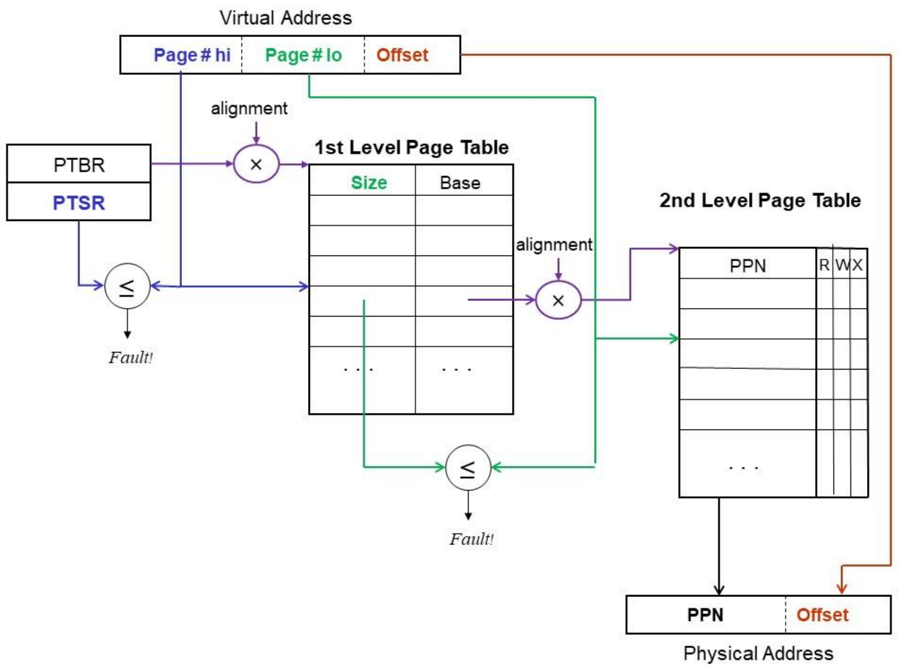


Figure 18 - Two-Level Paging (*CS 537 notes, section #16: Paging*)

Figure 18 presents the two-level paging model that associates with the memory management unit (MMU). In two-level paging model, it uses two levels of mapping to make tables manageable (*CS 537 notes, section #16: Paging*). In this feature, first and second level page tables are stored in main memory, which is RAM. All the page tables can start only on a page boundary, and the first level page table can have a maximum of 512 entries. Each page has permission bits for read, write, and execute operations. The following model is applied in our project, with less bits for page table level 1 and level 2 indexes and slightly modified size of the page tables.

4.3 Page Swapping

Generally, the resources that processes use are placed in the main memory, which is also called RAM. However, the capacity of the main memory is not enough for all processes to use. Therefore, the operating system loads some process resources on the main memory and stores others in the secondary storage. If it needs the resources in the secondary storage, the operating system swaps those resources with the resource in the main memory, which is useless. This operation is called swapping, and the resource that is not loaded in the main memory but stored in secondary storage is called swap space.

The concept of swapping is implemented in this project. Swapping occurs in the following cases: lack of the main memory space and page fault occurred in the page table level 2. In the first case, which is the situation that lacks space in the main memory, it selects the page that should be replaced according to the replacement policy and performs swapping between the main memory and disk storage. In the second case, it selects the replacement page from the page table level 2 and performs swapping between the tuple and the disk storage.

4.4 Table Look-Aside Buffer (TLB)

Translation lookaside buffer (TLB) is a memory cache that stores recent translations of virtual memory to physical addresses for faster retrieval (What is translation lookaside buffer (TLB): Definition from TechTarget 2014). The search begins in the CPU when a program references a virtual memory address. Instruction caches are examined first. The system must seek the physical address of the requested memory if it is not already in these quick caches. Locations in physical memory are now quickly referenced using TLB. When a search for an address in the TLB is unsuccessful, a memory page crawl operation must be used to search the physical memory. Values referenced are added to TLB as virtual memory locations are translated. Speed is increased when value can be recovered from the TLB because the memory address is kept in the TLB in the processor. Most processors have TLBs because of their built-in proximity to reduce latency and the high operating frequency of modern CPUs, which together speed up virtual memory operations.

Table lookaside buffer (TLB) is implemented in this project and is associated with the memory management unit (MMU). TLB operates on the following cases: TLB hit and TLB missing. In the first case, which TLB hit, it just returns the stored physical address from the TLB. In the second case, which is TLB missing, it performs searching the physical address through page table level 1 and level 2. After searching the physical address from the page table level 2, it stores the following address and updates the TLB.

4.5 First-In First-Out (FIFO) Page Replacement

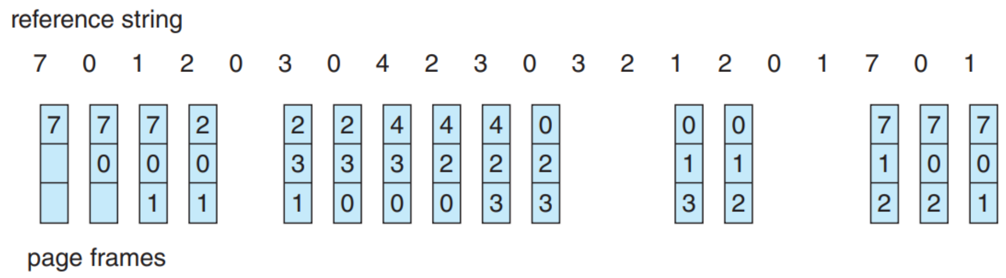


Figure 19 - FIFO Page Replacement Algorithm (Silberschatz et al., 2014)

The first-in, first-out (FIFO) algorithm is the most basic page replacement algorithm. Each page's time of entry into memory is associated with it via a FIFO replacement algorithm. The oldest page is used when a page must be changed. It should be noted that keeping track of the time a page is brought in is not necessarily necessary. We have established a FIFO queue to store all the memory pages. The page at the front of the list has been changed. We insert a page at the end of the queue when it is brought into memory. The following operation is presented in Figure 19.

It is simple to comprehend and implement the FIFO page replacement method. Its performance isn't always excellent, though. On the one hand, the page that was replaced might have been an initialization module that was in use previously but is no longer required. However, it can also have a heavily used variable that was initialized early and is continuously in use.

4.6 Least Recently Used (LRU) Page Replacement

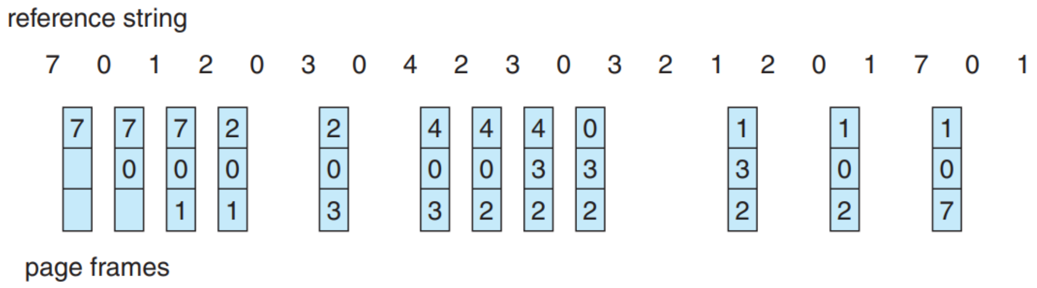


Figure 20 - LRU Page Replacement Algorithm (Silberschatz et al., 2014)

The LRU replacement algorithm allocates each page to the date and time of its most recent use. LRU selects a page that hasn't been used for the longest amount of time when a page needs to be replaced. This approach can be compared to the best page replacement method that looks backward in time rather than ahead. The operation shown in Figure 20 is the one after.

The LRU policy is a reliable page replacement mechanism that is frequently utilized. How to implement LRU replacement is the main issue. An LRU page replacement technique might require a lot of hardware support. The issue is figuring out a priority for the frames based on when they were last used. In this project, the LRU method is implemented using counters. In the simplest scenario, we add a logical clock or counter to the CPU and attach a time of use field with each entry in the page table when using counters. Every time a memory reference is made, the clock is increased. The contents of the clock register are copied to the time of use filed in the page table entry for each time a reference to a page is made. Replace the page with the shortest time value is replaced. For each memory access in this scheme, a write memory must be performed in addition to a page table search to locate the LRU page. When page tables are modified, the times must likewise be preserved. The clock's overflow must be considered.

Note that without hardware support other than the typical TLB registers, neither LRU solution would be possible. For each memory reference, the clock fields or stack must be updated. Every memory reference would be at least ten times slower if we used an interruption for each one to allow the software to update these data structures, which would also slow down every process by ten times. Few systems could withstand that much memory management overhead.

4.7 Least Frequently Used (LFU) and Most Frequently Used (MFU) Page Replacement

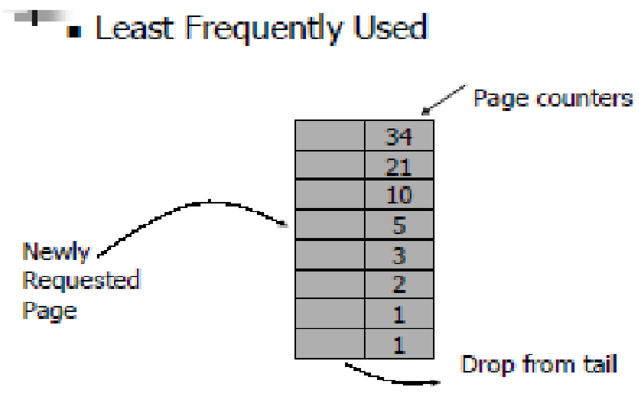


Figure 21 - LFU Page Replacement Algorithm

The least frequently used (LFU) algorithm is the page replacement algorithm where the less frequently used pages are evicted first among the other pages. This algorithm can be used for scenarios that depend on a few paged objects that will be used repeatedly for every request.

To implement the following page replacement algorithm, we need to add a counter that stores the number of references to the pages. With the following counter, the LFU algorithm works the same as the LRU algorithm, by having only a difference in the benchmark that is used to select the page that should be evicted or updated.

Most frequently used (MFU) algorithms operate in the opposite way of the operation of the LFU algorithm. It evicts the page that is most frequently used first among the other pages. This algorithm can be used for scenarios that depend on the most paged object that will be used repeatedly for every request.

4.8 Second Chance (SCA) Page Replacement

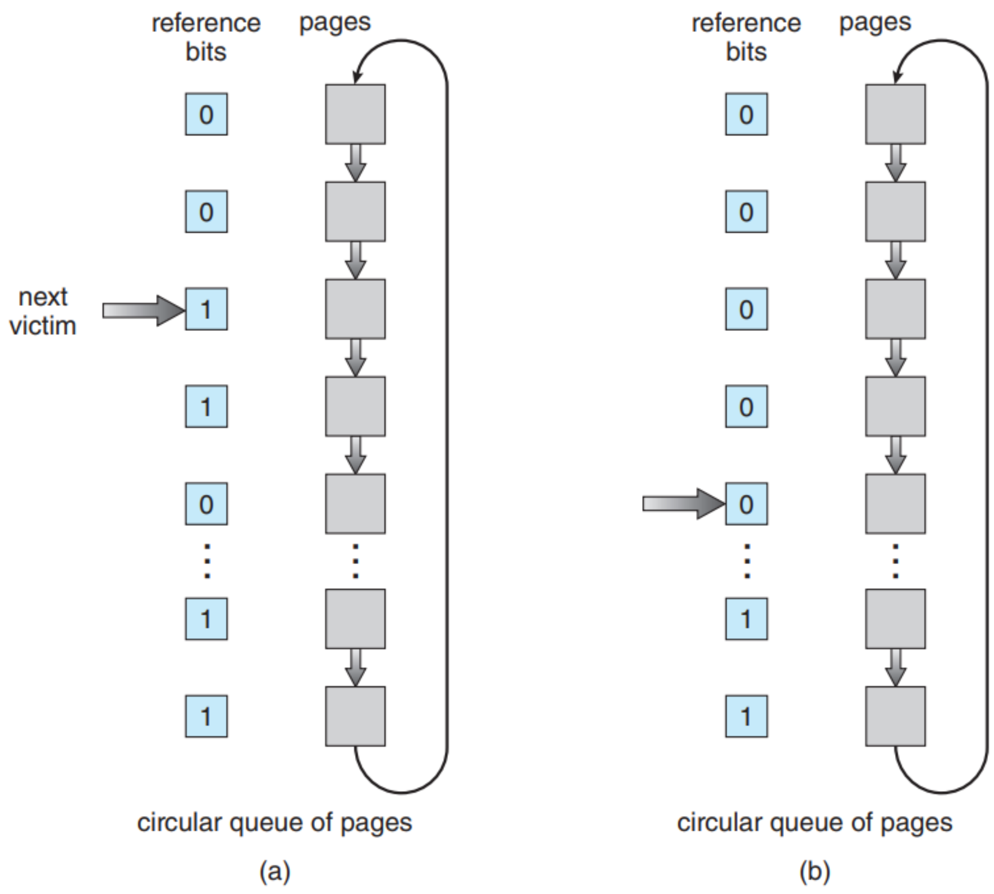


Figure 22 - Second Chance Page Replacement Algorithm (Silberschatz et al., 2014)

The second chance replacement is based on a FIFO replacement algorithm. After a page has been chosen, we look at its reference bit. We will replace this page if the value is 0. If the reference bit is set to 1, we will give the page another chance and choose the following FIFO page. When a page is given a second chance, its arrival time is reset to the current time and its reference bit is cleared. A page that receives a second chance won't be replaced until every other page has been changed. A page won't ever be changed if it is used frequently enough to keep its reference bit set.

A circular queue is one approach to putting the second chance algorithm into practice. The next page that must be replaced is indicated by a pointer. The pointer advances when a frame is required until it comes across a page with a 0-reference bit. It clears the reference bit as it advances. The following operation is presented in Figure 22. When the victim page is identified the page is swapped out and the replacement page is added to that spot in the circular queue. The pointer cycles over the entire queue in the worst scenario, giving each page a second chance, when all bits are set. Before choosing the next page to replace the current one, it clears all the reference information. If all bits are set, a second chance replacement becomes FIFO replacement.

4.9 Enhanced Second Chance (ESCA) Page Replacement

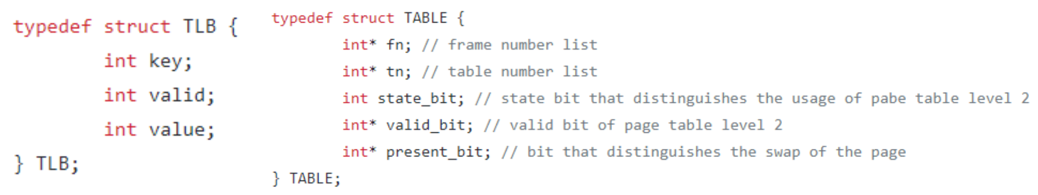
By treating the reference bit and modify bit as an ordered pair, we can improve the second chance method. We have the following four classes with these two bits:

1. (0, 0) - Neither recently used nor modified: best page to replace.
2. (0, 1) - Not recently used but modified: not quite as good, because the page will need to be written out before replacement.
3. (1, 0) - Recently used but clean: probably will be used again soon.
4. (1, 1) - Recently used and modified: probably will be used again soon, and the page will need to be written out to secondary storage before it can be replaced.

Every page belongs to one of these four categories. When a page replacement is required, we apply the same strategy as in the clock algorithm, but instead of determining if the page we are pointing to has the reference bit set to 1, we look at the class to which the page is a member. For the lowest non-empty class, we swap out the first page we come across. Keep in mind that it can take many scans of the circular queue to locate a page that has to be changed. The algorithm's main distinction from the more straightforward clock algorithm is that it prioritizes pages that have been altered to minimize the number of required I/O operations.

5 Implementation

5.1 Memory



```
typedef struct TLB {
        int key;
        int valid;
        int value;
} TLB;
```

```
typedef struct TABLE {
        int* fn; // frame number list
        int* tn; // table number list
        int state_bit; // state bit that distinguishes the usage of pabe table level 2
        int* valid_bit; // valid bit of page table level 2
        int* present_bit; // bit that distinguishes the swap of the page
} TABLE;
```

Figure 23 - Structures of TLB and Page Table

Figure 23 shows the translation look-aside buffer (TLB) and page table structures that are used in the implementation of the virtual memory. As the TLB stores the information to perform the caching operation, we set the TLB to contain the key, which is the bits that are used to identify the page table level 1 and level 2, valid bit, and value. In the page table, since we are implementing two-level paging, we set the page table to contain the frame number, table number, and state bit, which is used to distinguish the usage of page table level 2, valid bit, and present bit, which distinguishes the swapping operation of the page.

```
void memory_init();
void virtual_memory_free();
void virtual_memory_alloc();
```

Figure 24 - Memory Initialization, Allocation, and Remove Functions

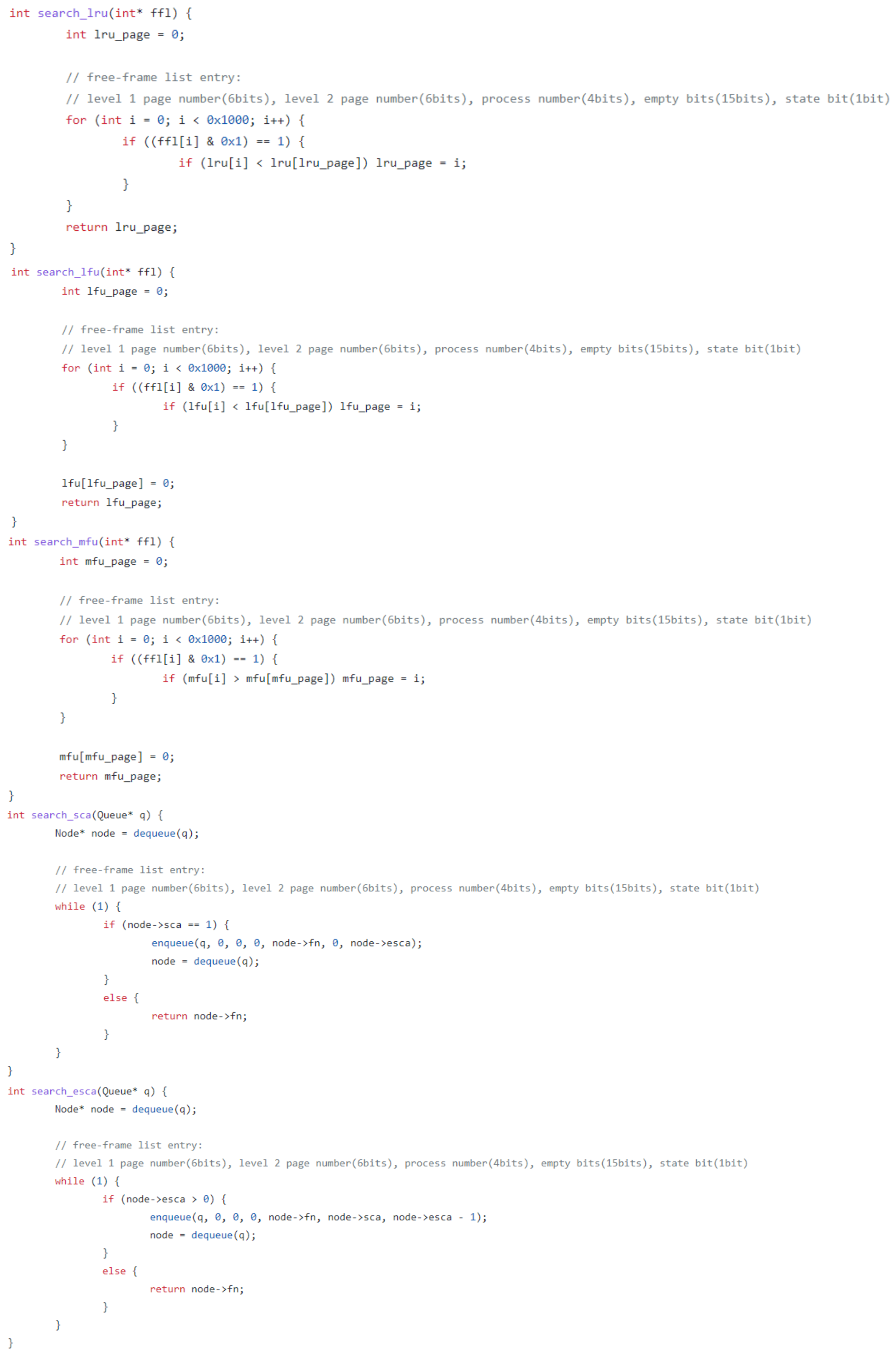
Figure 24 shows the functions that are used to initialize, allocate, and remove the memory that is used in the implementation of the virtual memory. If the memory initialization function is executed, it loads the binary program into the main memory. If the memory allocation function is executed, hardware features with free frame lists and counters that are used in the page replacement algorithms, and page tables are allocated. If the memory remove function is executed, all the memory-related variables are freed and removed.

```
void copy_page(int* src, int src_idx, int* src_list, int* dest, int dest_idx, int* dest_list) {
        for (int i = 0; i < 0x100; i++) {
                dest[(dest_idx * 0x100) + i] = src[(src_idx * 0x100) + i];
                src[(src_idx * 0x100) + i] = 0; // initialize the swaped memory to zero
        }
        dest_list[dest_idx] = 1;
        src_list[src_idx] = 0;
}
```

Figure 25 - Copy Page Function

Figure 25 shows the functions that copy the page from the source memory (storage) into the destination memory (storage). Since the page contains 256 entries, it iterates 256 times and copies every single entry from the source memory (storage) into the destination memory (storage). The following function mainly operates between the main memory and the secondary storage.

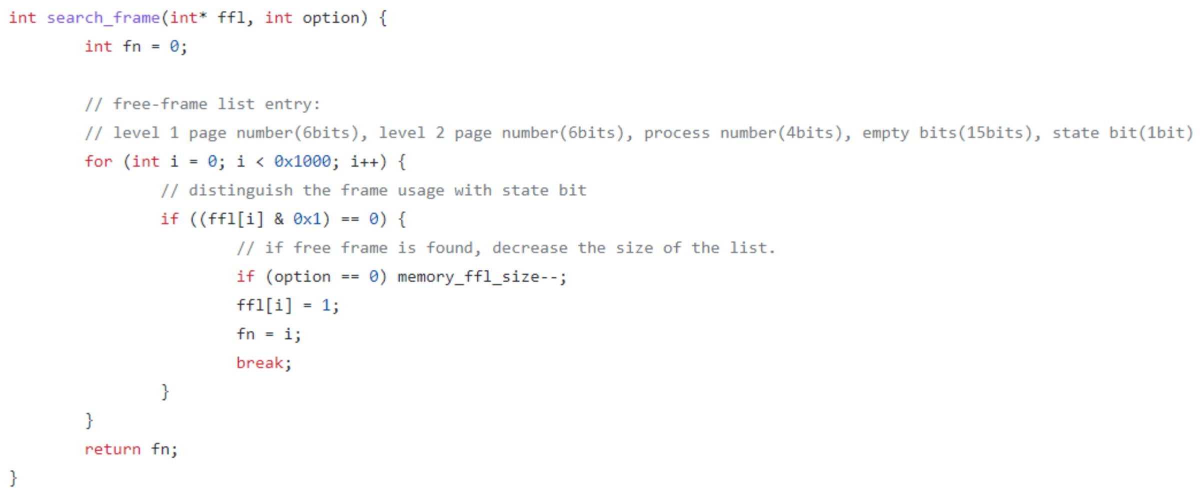
```
int search_random() {
        // free-frame list entry:
        // level 1 page number(6bits), level 2 page number(6bits), process number(4bits), empty bits(15bits), state bit(1bit)
        srand((unsigned int)time(NULL));
        return rand() % 0x1000;
}
int search_fifo(Queue* q) {
        // free-frame list entry:
        // level 1 page number(6bits), level 2 page number(6bits), process number(4bits), empty bits(15bits), state bit(1bit)
        Node* node = dequeue(q);
        return node->fn;
}
```



```c
int search_lru(int* ffl) {
        int lru_page = 0;

        // free-frame list entry:
        // level 1 page number(6bits), level 2 page number(6bits), process number(4bits), empty bits(15bits), state bit(1bit)
        for (int i = 0; i < 0x1000; i++) {
                if ((ffl[i] & 0x1) == 1) {
                        if (lru[i] < lru[lru_page]) lru_page = i;
                }
        }
        return lru_page;
}
int search_lfu(int* ffl) {
        int lfu_page = 0;

        // free-frame list entry:
        // level 1 page number(6bits), level 2 page number(6bits), process number(4bits), empty bits(15bits), state bit(1bit)
        for (int i = 0; i < 0x1000; i++) {
                if ((ffl[i] & 0x1) == 1) {
                        if (lfu[i] < lfu[lfu_page]) lfu_page = i;
                }
        }

        lfu[lfu_page] = 0;
        return lfu_page;
}
int search_mfu(int* ffl) {
        int mfu_page = 0;

        // free-frame list entry:
        // level 1 page number(6bits), level 2 page number(6bits), process number(4bits), empty bits(15bits), state bit(1bit)
        for (int i = 0; i < 0x1000; i++) {
                if ((ffl[i] & 0x1) == 1) {
                        if (mfu[i] > mfu[mfu_page]) mfu_page = i;
                }
        }

        mfu[mfu_page] = 0;
        return mfu_page;
}
int search_sca(Queue* q) {
        Node* node = dequeue(q);

        // free-frame list entry:
        // level 1 page number(6bits), level 2 page number(6bits), process number(4bits), empty bits(15bits), state bit(1bit)
        while (1) {
                if (node->sca == 1) {
                        enqueue(q, 0, 0, 0, node->fn, 0, node->esca);
                        node = dequeue(q);
                }
                else {
                        return node->fn;
                }
        }
}
int search_esca(Queue* q) {
        Node* node = dequeue(q);

        // free-frame list entry:
        // level 1 page number(6bits), level 2 page number(6bits), process number(4bits), empty bits(15bits), state bit(1bit)
        while (1) {
                if (node->esca > 0) {
                        enqueue(q, 0, 0, 0, node->fn, node->sca, node->esca - 1);
                        node = dequeue(q);
                }
                else {
                        return node->fn;
                }
        }
}
```

Figure 26 - Multiple Page Replacement Algorithms

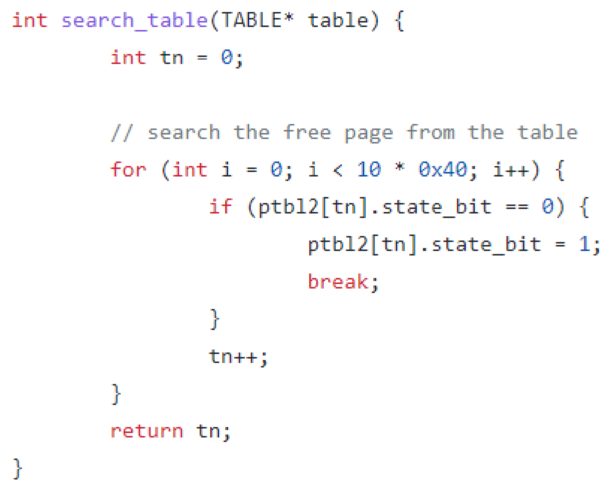
Functions presented in Figure 26 show the various implemented functions of page replacement algorithms. Each function follows the exact operations that we have mentioned in the previous sections. For first-in, first-out (FIFO), second chance (SCA), and enhanced second chance (ESCA) page replacement algorithms, we used circular queue operations. For the least recently used (LRU), least frequently used (LFU), and most frequently used (MFU) page replacement algorithms, we used a simple list consisting of counters. By using these page replacement algorithms, we successfully implemented the operation of the virtual memory program.



```
int search_frame(int* ffl, int option) {
        int fn = 0;

        // free-frame list entry:
        // level 1 page number(6bits), level 2 page number(6bits), process number(4bits), empty bits(15bits), state bit(1bit)
        for (int i = 0; i < 0x1000; i++) {
                // distinguish the frame usage with state bit
                if ((ffl[i] & 0x1) == 0) {
                        // if free frame is found, decrease the size of the list.
                        if (option == 0) memory_ffl_size--;
                        ffl[i] = 1;
                        fn = i;
                        break;
                }
        }
        return fn;
}
```

Figure 27 - Free Frame Search Function

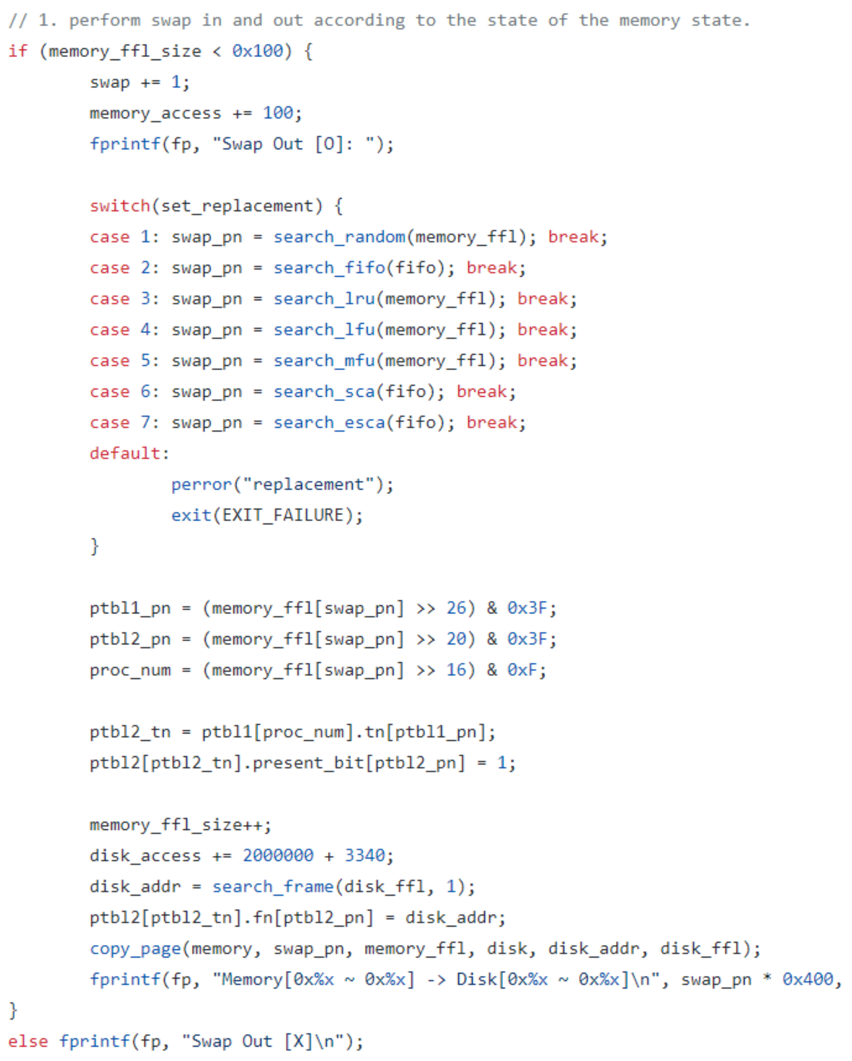
Figure 27 shows the function that searches the free frames in the given free frame list. From the first frame, the function starts the operation by checking whether the state bit is 1 or not. If the state bit is 1, it now checks the options parameter. If the optional parameter is 0, which means that the free frame is found from the list, it decreases the size of the list and returns the following frame number of the free frame. By using the following function, we can search for a free frame and use it in the swapping operation of the virtual memory system.



```
int search_table(TABLE* table) {
        int tn = 0;

        // search the free page from the table
        for (int i = 0; i < 10 * 0x40; i++) {
                if (ptbl2[tn].state_bit == 0) {
                        ptbl2[tn].state_bit = 1;
                        break;
                }
                tn++;
        }
        return tn;
}
```

Figure 28 - Free Page Table Entry Search Function

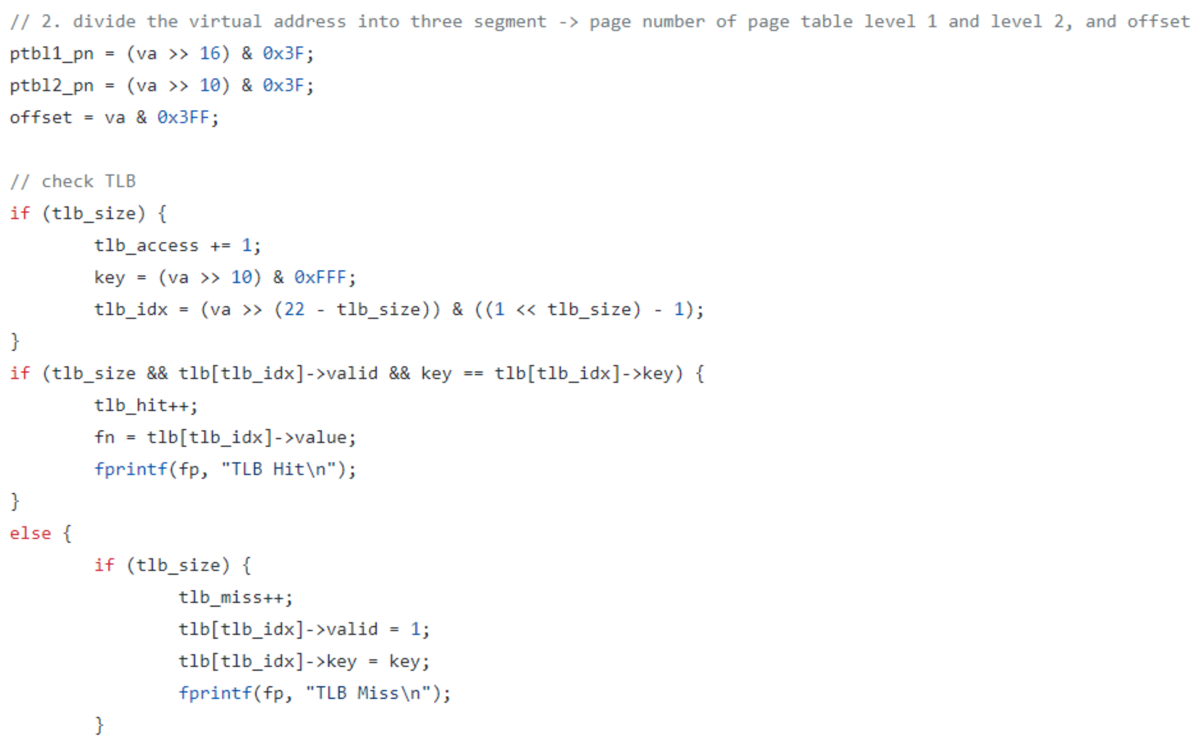
Figure 28 shows the function that searches the free page table in the given page table list. From the first-page table, the function starts the operation by checking whether the state bit of page table level 2 is 1 or not. If the state bit is 1, it moves on to the next available page table. However, if the state bit is 0, then it changes the state bit of the following page table into 1 and returns the following table number of the page table entry. By using the following function, we can search for a free page table and use it in the allocating the page table level 2 to page table level 1 of the virtual memory system.



```
// 1. perform swap in and out according to the state of the memory state.
if (memory_ffl_size < 0x100) {
        swap += 1;
        memory_access += 100;
        fprintf(fp, "Swap Out [O]: ");

        switch(set_replacement) {
        case 1: swap_pn = search_random(memory_ffl); break;
        case 2: swap_pn = search_fifo(fifo); break;
        case 3: swap_pn = search_lru(memory_ffl); break;
        case 4: swap_pn = search_lfu(memory_ffl); break;
        case 5: swap_pn = search_mfu(memory_ffl); break;
        case 6: swap_pn = search_sca(fifo); break;
        case 7: swap_pn = search_esca(fifo); break;
        default:
                perror("replacement");
                exit(EXIT_FAILURE);
        }

        ptbl1_pn = (memory_ffl[swap_pn] >> 26) & 0x3F;
        ptbl2_pn = (memory_ffl[swap_pn] >> 20) & 0x3F;
        proc_num = (memory_ffl[swap_pn] >> 16) & 0xF;

        ptbl2_tn = ptbl1[proc_num].tn[ptbl1_pn];
        ptbl2[ptbl2_tn].present_bit[ptbl2_pn] = 1;

        memory_ffl_size++;
        disk_access += 2000000 + 3340;
        disk_addr = search_frame(disk_ffl, 1);
        ptbl2[ptbl2_tn].fn[ptbl2_pn] = disk_addr;
        copy_page(memory, swap_pn, memory_ffl, disk, disk_addr, disk_ffl);
        fprintf(fp, "Memory[0x%x ~ 0x%x] -> Disk[0x%x ~ 0x%x]\n", swap_pn * 0x400,
}
else fprintf(fp, "Swap Out [X]\n");
```

Figure 29 - Memory Management Unit (MMU) Operation 1

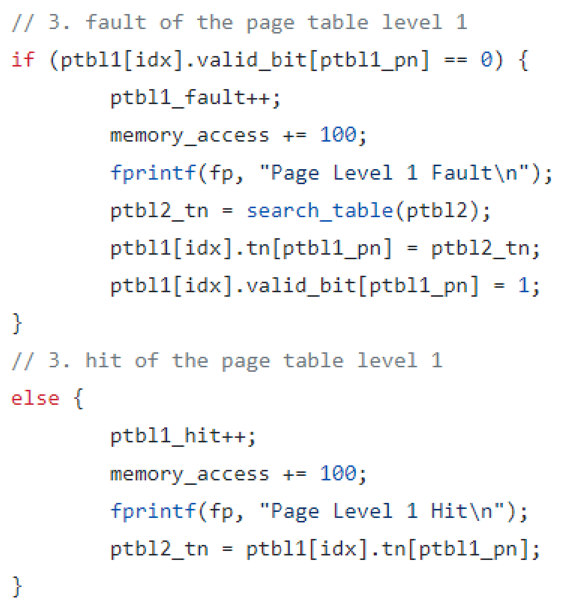
Figure 29 show the first substantial implemented code of the memory management unit (MMU) function. It presents the operation of swapping which is occurred due to the lack of free space for the free frame list of the main memory. If the following situation is occurred, it first selects the proper page number that should be replaced according to the page replacement algorithm. Then, it fetches the information of page table index for page table level 1 and level2 with the offset. After this progress, it performs swapping between main memory and the secondary storage. The operation is held by storing the page from the main memory into the secondary storage.



```
// 2. divide the virtual address into three segment -> page number of page table level 1 and level 2, and offset
ptbl1_pn = (va >> 16) & 0x3F;
ptbl2_pn = (va >> 10) & 0x3F;
offset = va & 0x3FF;

// check TLB
if (tlb_size) {
        tlb_access += 1;
        key = (va >> 10) & 0xFFF;
        tlb_idx = (va >> (22 - tlb_size)) & ((1 << tlb_size) - 1);
}
if (tlb_size && tlb[tlb_idx]->valid && key == tlb[tlb_idx]->key) {
        tlb_hit++;
        fn = tlb[tlb_idx]->value;
        fprintf(fp, "TLB Hit\n");
}
else {
        if (tlb_size) {
                tlb_miss++;
                tlb[tlb_idx]->valid = 1;
                tlb[tlb_idx]->key = key;
                fprintf(fp, "TLB Miss\n");
        }
```

Figure 30 - Memory Management Unit (MMU) Operation 2

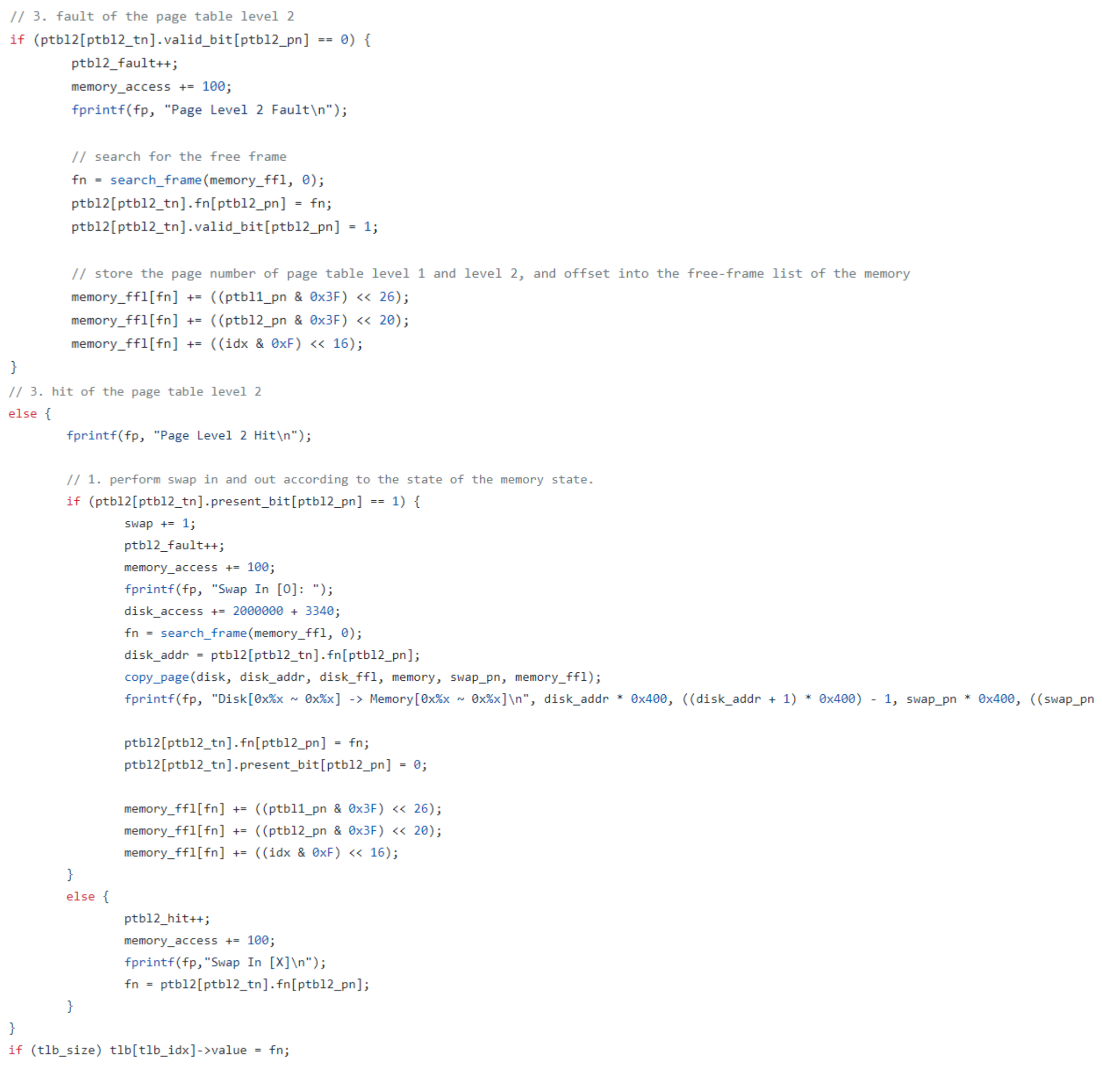
Figure 30 shows the second substantial implemented code of the memory management unit (MMU) function. It presents the operation of dividing the virtual address into three segments, which are page numbers of page table level 1 (6 bit) and level 2 (6 bit) and offset (10 bit). Since the size of the memory allocation has a limitation, we limited our accessible memory address to 22 bits. After partitioning the virtual address, it checks whether the following virtual address is in the table lookaside buffer (TLB). If there is a matching entry, it just calls the physical memory address from the TLB. If not, it will perform later operations.



```
// 3. fault of the page table level 1
if (ptbl1[idx].valid_bit[ptbl1_pn] == 0) {
        ptbl1_fault++;
        memory_access += 100;
        fprintf(fp, "Page Level 1 Fault\n");
        ptbl2_tn = search_table(ptbl2);
        ptbl1[idx].tn[ptbl1_pn] = ptbl2_tn;
        ptbl1[idx].valid_bit[ptbl1_pn] = 1;
}
// 3. hit of the page table level 1
else {
        ptbl1_hit++;
        memory_access += 100;
        fprintf(fp, "Page Level 1 Hit\n");
        ptbl2_tn = ptbl1[idx].tn[ptbl1_pn];
}
```

Figure 31 - Memory Management Unit (MMU) Operation 3

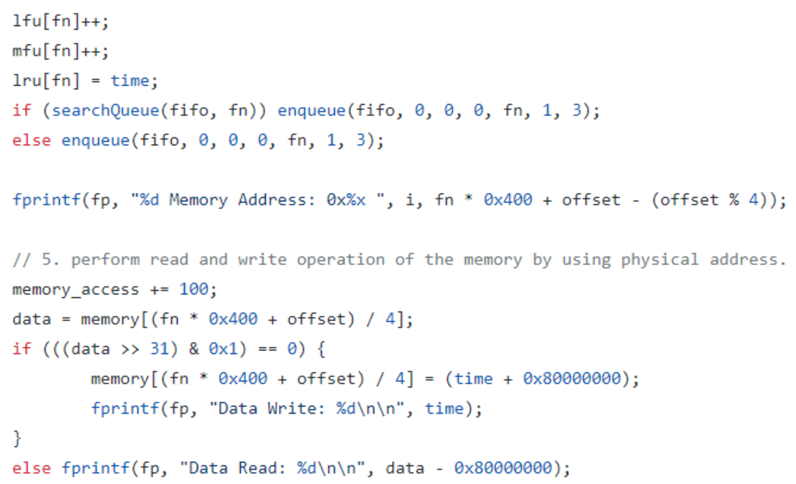
Figure 31 shows the third substantial implemented code of the memory management unit (MMU) function. It presents the operation of scanning the page table level 1. If it fails to find the page table from page table level 1, search for the free page table and allocate its page table number of page table level 2 to page table level 1. However, if it finds the proper page table from page table level 1, it returns the page table number to page table level 2.



```
// 3. fault of the page table level 2
if (ptbl2[ptbl2_tn].valid_bit[ptbl2_pn] == 0) {
        ptbl2_fault++;
        memory_access += 100;
        fprintf(fp, "Page Level 2 Fault\n");

        // search for the free frame
        fn = search_frame(memory_ffl, 0);
        ptbl2[ptbl2_tn].fn[ptbl2_pn] = fn;
        ptbl2[ptbl2_tn].valid_bit[ptbl2_pn] = 1;

        // store the page number of page table level 1 and level 2, and offset into the free-frame list of the memory
        memory_ffl[fn] += ((ptbl1_pn & 0x3F) << 26);
        memory_ffl[fn] += ((ptbl2_pn & 0x3F) << 20);
        memory_ffl[fn] += ((idx & 0xF) << 16);
}
// 3. hit of the page table level 2
else {
        fprintf(fp, "Page Level 2 Hit\n");

        // 1. perform swap in and out according to the state of the memory state.
        if (ptbl2[ptbl2_tn].present_bit[ptbl2_pn] == 1) {
                swap += 1;
                ptbl2_fault++;
                memory_access += 100;
                fprintf(fp, "Swap In [O]: ");
                disk_access += 2000000 + 3340;
                fn = search_frame(memory_ffl, 0);
                disk_addr = ptbl2[ptbl2_tn].fn[ptbl2_pn];
                copy_page(disk, disk_addr, disk_ffl, memory, swap_pn, memory_ffl);
                fprintf(fp, "Disk[0x%x ~ 0x%x] -> Memory[0x%x ~ 0x%x]\n", disk_addr * 0x400, ((disk_addr + 1) * 0x400) - 1, swap_pn * 0x400, ((swap_pn

                ptbl2[ptbl2_tn].fn[ptbl2_pn] = fn;
                ptbl2[ptbl2_tn].present_bit[ptbl2_pn] = 0;

                memory_ffl[fn] += ((ptbl1_pn & 0x3F) << 26);
                memory_ffl[fn] += ((ptbl2_pn & 0x3F) << 20);
                memory_ffl[fn] += ((idx & 0xF) << 16);
        }
        else {
                ptbl2_hit++;
                memory_access += 100;
                fprintf(fp,"Swap In [X]\n");
                fn = ptbl2[ptbl2_tn].fn[ptbl2_pn];
        }
}
if (tlb_size) tlb[tlb_idx]->value = fn;
```

Figure 32 - Memory Management Unit (MMU) Operation 4

Figure 32 shows the fourth substantial implemented code of the memory management unit (MMU) function. It presents the operation of scanning the page table level 2. If it fails to find the proper frame of the main memory from page table level 2, it finds the free frame from the free frame list of the main memory. Then, it stores its information in the free frame. However, if it finds the proper frame of the main memory from page table level 2, it can perform two operations. First, if the present bit of the current page is 1, which represents the situation that the following page is not on the main memory, the function performs swapping between secondary storage and main memory and loads the page on the main memory. Second, if the present bit of the current page is 0, which means that the page is on the main memory, it takes the frame number from the page table level 2.

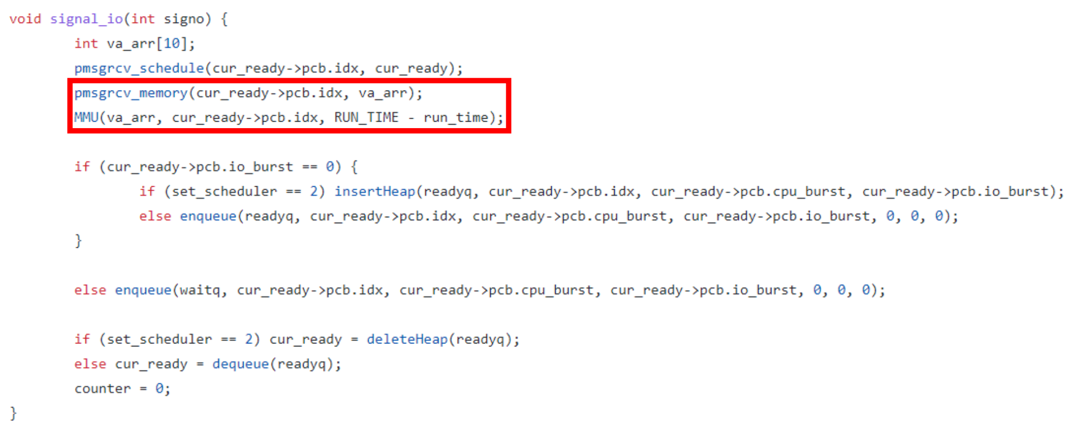


```
lfu[fn]++;
mfu[fn]++;
lru[fn] = time;
if (searchQueue(fifo, fn)) enqueue(fifo, 0, 0, 0, fn, 1, 3);
else enqueue(fifo, 0, 0, 0, fn, 1, 3);

fprintf(fp, "%d Memory Address: 0x%x ", i, fn * 0x400 + offset - (offset % 4));

// 5. perform read and write operation of the memory by using physical address.
memory_access += 100;
data = memory[(fn * 0x400 + offset) / 4];
if (((data >> 31) & 0x1) == 0) {
        memory[(fn * 0x400 + offset) / 4] = (time + 0x80000000);
        fprintf(fp, "Data Write: %d\n\n", time);
}
else fprintf(fp, "Data Read: %d\n\n", data - 0x80000000);
```

Figure 33 - Memory Management Unit (MMU) Operation 5

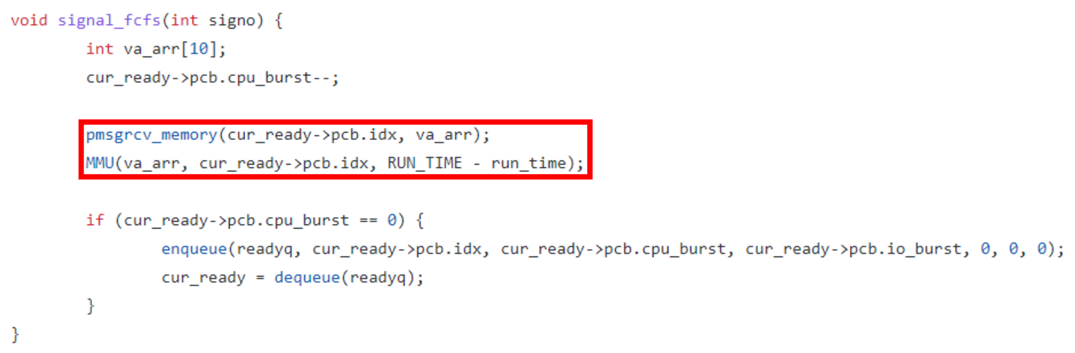
Figure 33 shows the fifth substantial implemented code of the memory management unit (MMU) function. It presents the operation of updating the counters and queue that is used in the operation of page replacement algorithms, and the read and write operation on the given physical memory address. As we mentioned previously, for first-in, first-out (FIFO), second chance (SCA), and enhanced second chance (ESCA) page replacement algorithms, we use circular queue operations. Therefore, in this part, we enqueue the referenced frame number into the queue. If the frame number is already in the queue, then we pop that node from the queue and enqueue in the head of the queue. However, if it is not in the queue, then we just enqueue it into the head of the queue. Also, for the least recently used (LRU), least frequently used (LFU), and most frequently used (MFU) page replacement algorithms, we just update the counters for each algorithm. As part of the read and write operation on the given frame number, if the frame is empty, then we perform a write operation in the frame with the data of the current time tick with a sign bit. However, if the frame is already full of the data, then we just performed the reading operation, by substituting the sign bit in front of the data.

5.2 Signal



```
void signal_io(int signo) {
        int va_arr[10];
        pmsgrcv_schedule(cur_ready->pcb.idx, cur_ready);
        pmsgrcv_memory(cur_ready->pcb.idx, va_arr);
        MMU(va_arr, cur_ready->pcb.idx, RUN_TIME - run_time);

        if (cur_ready->pcb.io_burst == 0) {
                if (set_scheduler == 2) insertHeap(readyq, cur_ready->pcb.idx, cur_ready->pcb.cpu_burst, cur_ready->pcb.io_burst);
                else enqueue(readyq, cur_ready->pcb.idx, cur_ready->pcb.cpu_burst, cur_ready->pcb.io_burst, 0, 0, 0);
        }

        else enqueue(waitq, cur_ready->pcb.idx, cur_ready->pcb.cpu_burst, cur_ready->pcb.io_burst, 0, 0, 0);

        if (set_scheduler == 2) cur_ready = deleteHeap(readyq);
        else cur_ready = dequeue(readyq);
        counter = 0;
}
```

Figure 34 - Memory Management Unit (MMU) Operation in I/O Signal Function

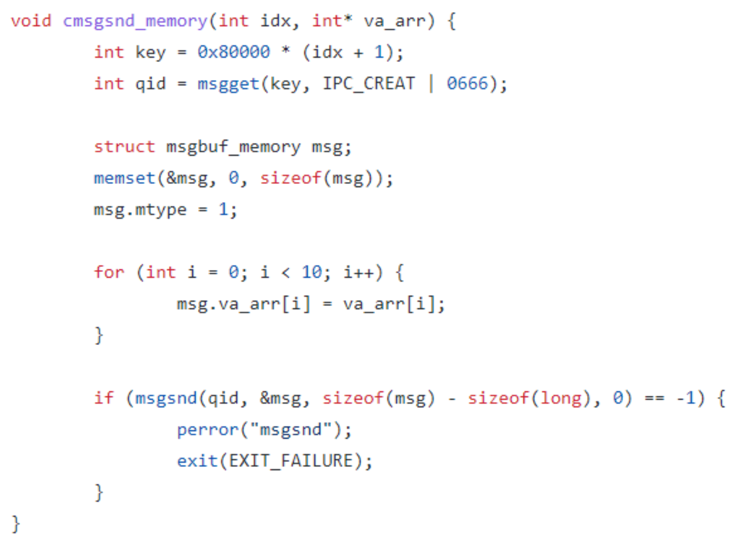


```c
void signal_fcfs(int signo) {
        int va_arr[10];
        cur_ready->pcb.cpu_burst--;

        pmsgrcv_memory(cur_ready->pcb.idx, va_arr);
        MMU(va_arr, cur_ready->pcb.idx, RUN_TIME - run_time);

        if (cur_ready->pcb.cpu_burst == 0) {
                enqueue(readyq, cur_ready->pcb.idx, cur_ready->pcb.cpu_burst, cur_ready->pcb.io_burst, 0, 0, 0);
                cur_ready = dequeue(readyq);
        }
}
```

Figure 35 - Memory Management Unit (MMU) Operation in Scheduling Signal

Figures 34 and 35 show how the memory management unit (MMU) function works in the signaling functions. According to Figure 34, we can see that for every I/O signal operation, it sends the data of virtual memory addresses to the parent process and performs the MMU operation for the currently running process. Also, according to Figure 35, we can see that for every scheduling event, it sends the data of the virtual memory address to the parent process and performs the MMU operation for the currently running process. The following operation is not only implemented in Figure 35, which is the signal function of first-come, first-served (FCFS) scheduling operation but also implemented in the other scheduling operations, such as shortest job first (SJF) and round-robin (RR).

5.3 Message



```c
void cmsgsnd_memory(int idx, int* va_arr) {
        int key = 0x80000 * (idx + 1);
        int qid = msgget(key, IPC_CREAT | 0666);

        struct msgbuf_memory msg;
        memset(&msg, 0, sizeof(msg));
        msg.mtype = 1;

        for (int i = 0; i < 10; i++) {
                msg.va_arr[i] = va_arr[i];
        }

        if (msgsnd(qid, &msg, sizeof(msg) - sizeof(long), 0) == -1) {
                perror("msgsnd");
                exit(EXIT_FAILURE);
        }
}
```

Figure 36 - Message Send Function of the Child Process that Sends Virtual Addresses

Figure 36 shows the message send function of the child process that sends the virtual memory addresses to the parent process. Before operating the following function, the child process first decides which virtual addresses it will use. After it decides on accessing addresses, it sends a set of accessing addresses to the parent process, so that the parent process can perform the additional operations that are needed to implement the virtual memory system. This operation will be described in a later section.

```
void pmsgrcv_memory(int idx, int* va_buffer) {
        int key = 0x80000 * (idx + 1);
        int qid = msgget(key, IPC_CREAT | 0666);

        struct msgbuf_memory msg;
        memset(&msg, 0, sizeof(msg));

        if (msgrcv(qid, &msg, sizeof(msg) - sizeof(long), 0, 0) == -1) {
                perror("msgrcv");
                exit(EXIT_FAILURE);
        }

        for (int i = 0; i < 10; i++) {
                va_buffer[i] = msg.va_arr[i];
        }
}
```

Figure 37 - Message Receive Function of Parent Process that Receives Addresses from Child Process

Figure 37 shows the message-receiving function of the parent process that receives virtual memory addresses from the child process. First, the parent process receives the virtual memory address that the child process is referencing by using this function. Then, the parent process performs the memory management unit (MMU) operation with these addresses.

5.4 Main

```
// create 10 random virtual addresses
for (int j = 0; j < 10; j++) {
        c_idx = (c_idx + 1) % 10000;
        c_va_arr[j] = map[c_idx];
}
// send the virtual adddresses that child process accesses to the parent process.
cmsgsnd_memory(idx, c_va_arr);
```

Figure 38 - Operation of Child Processes dealing with the Virtual Addresses

Figure 38 shows the operations of the child process dealing with the virtual memory addresses. First, it loads the referencing virtual memory address from the given memory access pattern. Then, by loading 10 referencing virtual memory addresses, it sends the set of virtual memory addresses to the parent process by using the message-sending function which is presented in the previous section. One of the considerations in the following code is the memory access pattern of the child process. In this project, we generated two access patterns, which are the randomly generated pattern (uniform) and the access pattern generated by considering the general memory access pattern of the program with the sections of reserved (0x0 ~ 0x400), text (0x400 ~ 0x10000), static (0x10000 ~ 0x10800), and dynamic (0x10800 ~ 0x80000). This consideration will be discussed in a later section.

6 Build Environment

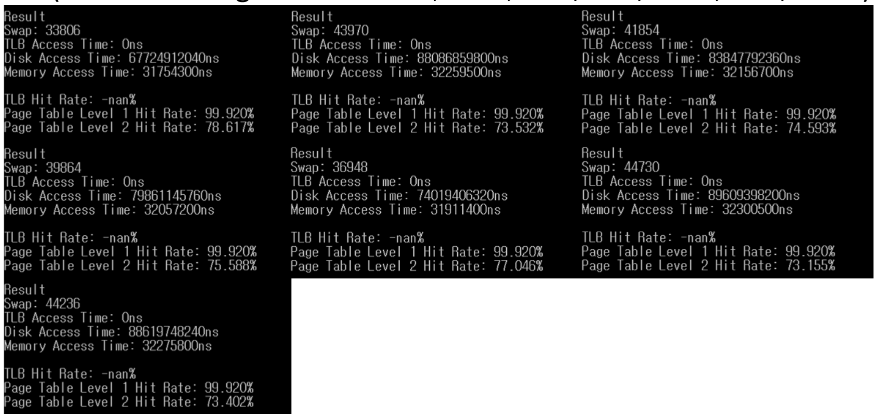
- Build Environment:
  1. Linux Environment -> Vi editor, GCC Complier
  2. Program is built by using the Makefile.

- Build Command:
  1. $make main -> build the execution program for virtual memory (paging).
  2. $make clean -> clean all the object files of the virtual memory (paging) program.

- Execution Command: ./main {$scheduler} {$replacement policy} {TLB size} {$max limit}
  1. scheduler: (1. FCFS, 2. SJF, 3. RR)
  2. replacement policy: (1. RANDOM, 2. FIFO, 3. LRU, 4. LFU, 5. MFU, 6. SCA, 7. ESCA)
  3. TLB size: (128, 256, 512)
  4. max limit: integer

7 Results

- Result of the virtual memory (paging) program with memory access pattern - uniform (from left to right → Random, FIFO, LRU, LFU, MFU, SCA, ESCA):



```
Result
Swap: 33806
TLB Access Time: 0ns
Disk Access Time: 67724912040ns
Memory Access Time: 31754300ns

TLB Hit Rate: -nan%
Page Table Level 1 Hit Rate: 99.920%
Page Table Level 2 Hit Rate: 78.617%
```

```
Result
Swap: 43970
TLB Access Time: 0ns
Disk Access Time: 88086859800ns
Memory Access Time: 32259500ns

TLB Hit Rate: -nan%
Page Table Level 1 Hit Rate: 99.920%
Page Table Level 2 Hit Rate: 73.532%
```

```
Result
Swap: 41854
TLB Access Time: 0ns
Disk Access Time: 83847792360ns
Memory Access Time: 32156700ns

TLB Hit Rate: -nan%
Page Table Level 1 Hit Rate: 99.920%
Page Table Level 2 Hit Rate: 74.593%
```

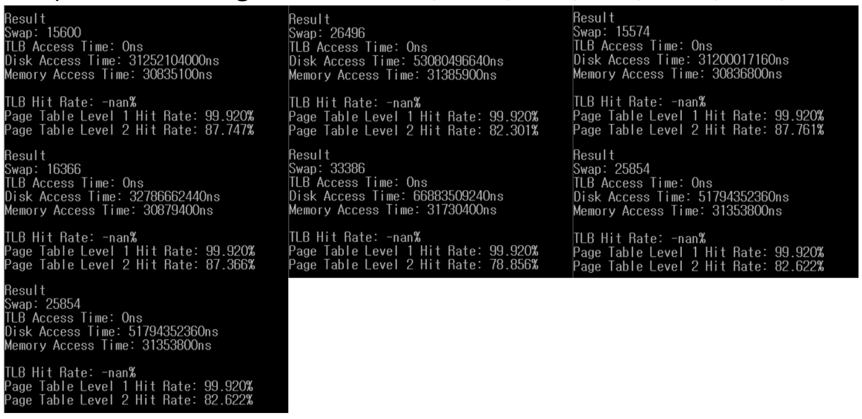
```
Result
Swap: 39864
TLB Access Time: 0ns
Disk Access Time: 79861145760ns
Memory Access Time: 32057200ns

TLB Hit Rate: -nan%
Page Table Level 1 Hit Rate: 99.920%
Page Table Level 2 Hit Rate: 75.588%
```

```
Result
Swap: 36948
TLB Access Time: 0ns
Disk Access Time: 74019406320ns
Memory Access Time: 31911400ns

TLB Hit Rate: -nan%
Page Table Level 1 Hit Rate: 99.920%
Page Table Level 2 Hit Rate: 77.046%
```

```
Result
Swap: 44730
TLB Access Time: 0ns
Disk Access Time: 89609398200ns
Memory Access Time: 32300500ns

TLB Hit Rate: -nan%
Page Table Level 1 Hit Rate: 99.920%
Page Table Level 2 Hit Rate: 73.155%
```

```
Result
Swap: 44236
TLB Access Time: 0ns
Disk Access Time: 88619748240ns
Memory Access Time: 32275800ns

TLB Hit Rate: -nan%
Page Table Level 1 Hit Rate: 99.920%
Page Table Level 2 Hit Rate: 73.402%
```

- Result of the virtual memory (paging) program with memory access pattern (from left to right → Random, FIFO, LRU, LFU, MFU, SCA, ESCA):



```
Result
Swap: 15600
TLB Access Time: 0ns
Disk Access Time: 31252104000ns
Memory Access Time: 30835100ns

TLB Hit Rate: -nan%
Page Table Level 1 Hit Rate: 99.920%
Page Table Level 2 Hit Rate: 87.747%
```

```
Result
Swap: 26496
TLB Access Time: 0ns
Disk Access Time: 53080496640ns
Memory Access Time: 31385900ns

TLB Hit Rate: -nan%
Page Table Level 1 Hit Rate: 99.920%
Page Table Level 2 Hit Rate: 82.301%
```

```
Result
Swap: 15574
TLB Access Time: 0ns
Disk Access Time: 31200017160ns
Memory Access Time: 30836800ns

TLB Hit Rate: -nan%
Page Table Level 1 Hit Rate: 99.920%
Page Table Level 2 Hit Rate: 87.761%
```

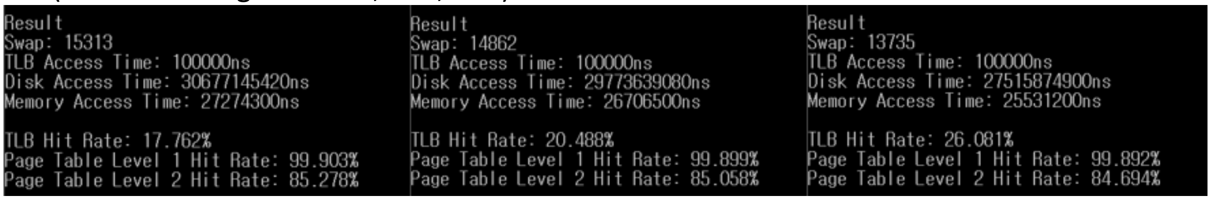
```
Result
Swap: 16366
TLB Access Time: 0ns
Disk Access Time: 32786662440ns
Memory Access Time: 30879400ns

TLB Hit Rate: -nan%
Page Table Level 1 Hit Rate: 99.920%
Page Table Level 2 Hit Rate: 87.366%
```

```
Result
Swap: 33386
TLB Access Time: 0ns
Disk Access Time: 66883509240ns
Memory Access Time: 31730400ns

TLB Hit Rate: -nan%
Page Table Level 1 Hit Rate: 99.920%
Page Table Level 2 Hit Rate: 78.856%
```

```
Result
Swap: 25854
TLB Access Time: 0ns
Disk Access Time: 51794352360ns
Memory Access Time: 31353800ns

TLB Hit Rate: -nan%
Page Table Level 1 Hit Rate: 99.920%
Page Table Level 2 Hit Rate: 82.622%
```

```
Result
Swap: 25854
TLB Access Time: 0ns
Disk Access Time: 51794352360ns
Memory Access Time: 31353800ns

TLB Hit Rate: -nan%
Page Table Level 1 Hit Rate: 99.920%
Page Table Level 2 Hit Rate: 82.622%
```

- Result of the virtual memory (paging) program with (from left to right → 128, 256, 512):



```
Result
Swap: 15313
TLB Access Time: 100000ns
Disk Access Time: 30677145420ns
Memory Access Time: 27274300ns

TLB Hit Rate: 17.762%
Page Table Level 1 Hit Rate: 99.903%
Page Table Level 2 Hit Rate: 85.278%
```

```
Result
Swap: 14862
TLB Access Time: 100000ns
Disk Access Time: 29773639080ns
Memory Access Time: 26706500ns

TLB Hit Rate: 20.488%
Page Table Level 1 Hit Rate: 99.899%
Page Table Level 2 Hit Rate: 85.058%
```

```
Result
Swap: 13735
TLB Access Time: 100000ns
Disk Access Time: 27515874900ns
Memory Access Time: 25531200ns

TLB Hit Rate: 26.081%
Page Table Level 1 Hit Rate: 99.892%
Page Table Level 2 Hit Rate: 84.694%
```

8 Evaluation

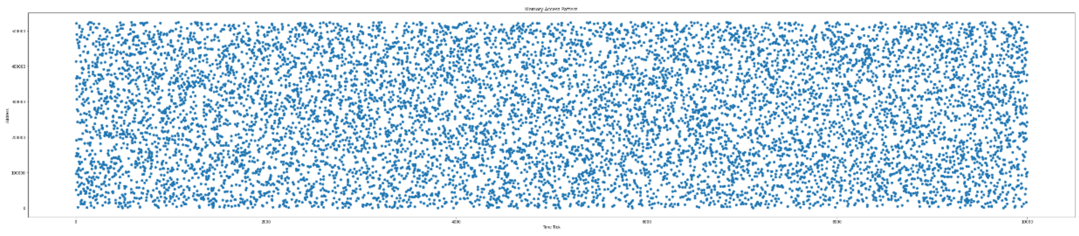


Figure 39 - Memory Access Pattern – Uniform

The first approach that we performed was running the virtual memory (paging) program on the randomly generated memory access pattern. This pattern is also called the uniformed pattern since it randomly references all the memory address ranges in the same proportion. Figure 39 presents the visualization of the following access pattern, where the x-axis is the time tick of the CPU, and the y-axis is the accessed memory address.

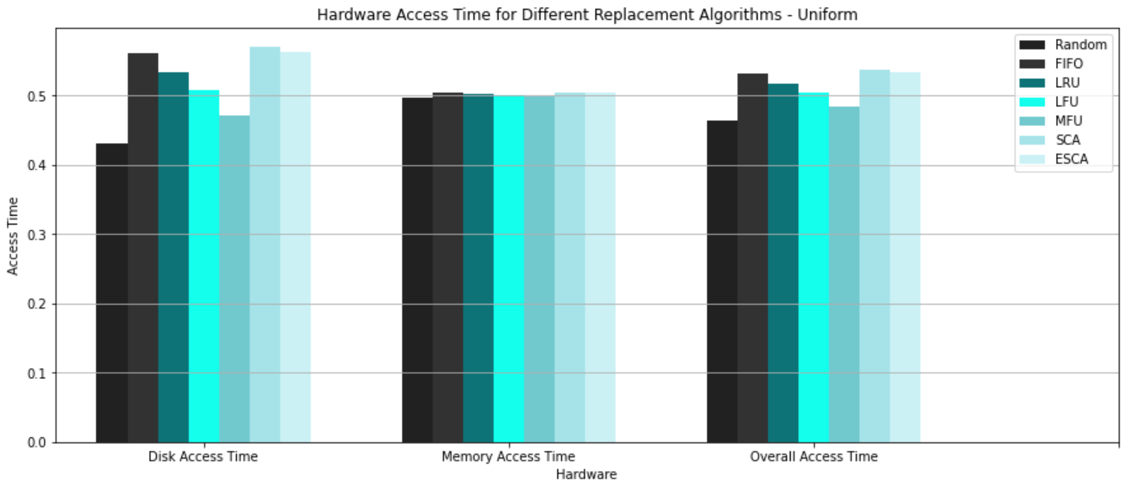


Figure 40 - Representation of the Normalized Hardware Access Time on Memory Access Pattern – Uniform

The following approach seems to be reasonable, but it occurs some problems and cannot indicate the actual operation of the memory access. First, the process does not access the memory with the same proportion of references. Its access pattern is weighted to the data section, such as static or dynamic data sections. Second, as we can see from Figure 40, sine the access pattern is uniformed, there are no big differences in the performance between different page replacement algorithms. To resolve these problems, we need a different approach.

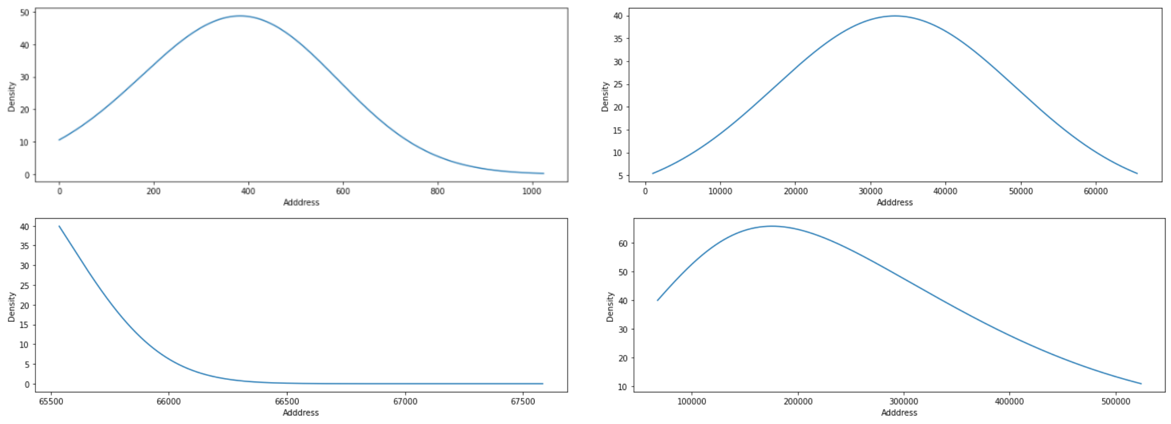


Figure 41 - Access Proportion of Addresses

To resolve the situation that is presented above, we set the different memory access patterns to each section of the process memory, which are the reserved (0x0 ~ 0x400), text (0x400 ~ 0x10000), static (0x10000 ~ 0x10800), and dynamic (0x10800 ~ 0x80000). We assume that the process is running the program in a recursive loop. Therefore, the reserved and text area can have a high proportion of memory access in the middle of its area, which can be presented in the tilted Gaussian distribution function. However, in the static and dynamic memory area, we assumed that it follows the ZIPFs law. The ZIPFs law describes a probability distribution where each frequency is the reciprocal of its rank multiplied by the highest frequency. In short, for static and dynamic memory areas, the proportion of memory access is high in the left-sided addresses and low in the right-sided addresses. Figure 41 shows visualizations of the memory access proportions of each memory area.

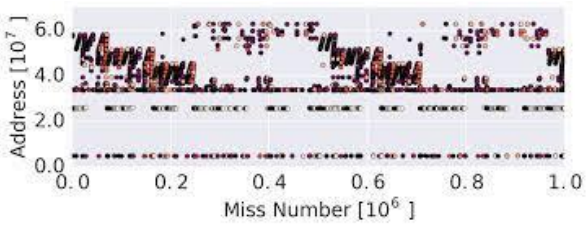


Figure 42 - Actual Memory Access Pattern (Hashemi et al., 2018)

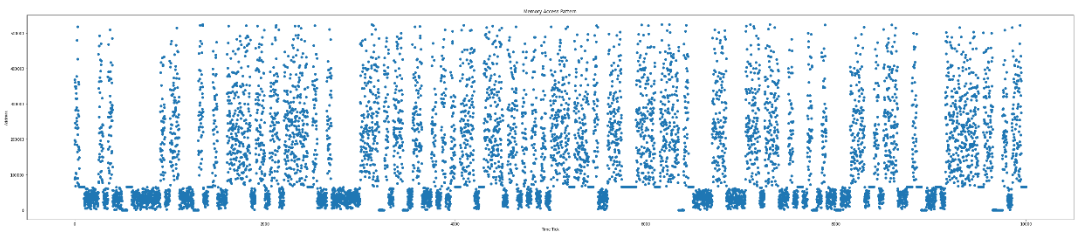


Figure 43 - Generated Memory Access Pattern

Using the strategy above, we generated the memory access pattern which is presented in Figure 43. Since it is hard to implement a real memory access pattern that is generated by the process, which is presented in Figure 42, we just employed the memory access proportion into the set of the array and divided it into the set of blocks and shuffled it. As a result, even though it is not a real memory access pattern, we generated a similar memory access pattern to the actual memory access pattern.

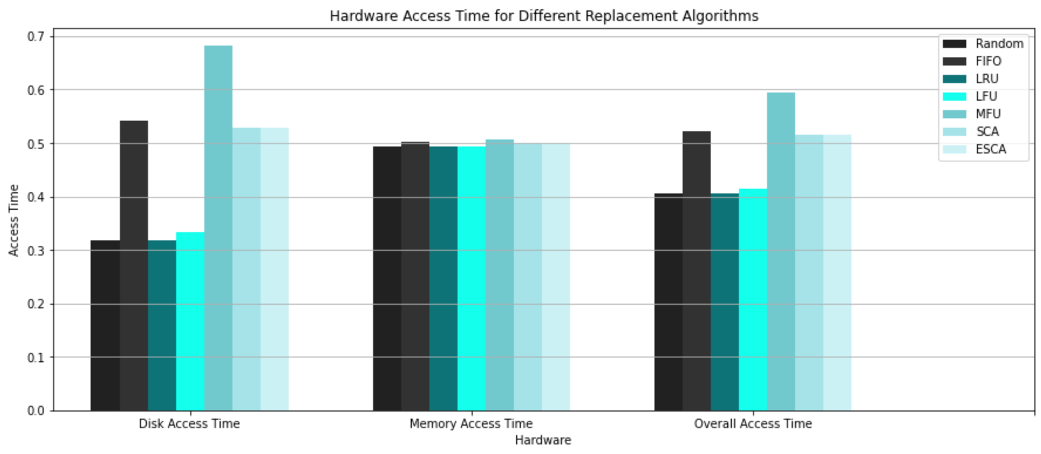


Figure 43 - Representation of the Normalized Hardware Access Time on Memory Access Pattern

In short, we will get the results presented in Figure 43. The following result shows the significant difference in the performance among the different page replacement algorithms. As we can see, the FIFO and MFU algorithms show the worst performance among other algorithms. SCA and ESCA algorithms show slightly better performance than the FIFO algorithm, and the LRU algorithm shows the best performance among others. Therefore, in the given memory access pattern, we can conclude that the LRU algorithm shows the best performance.

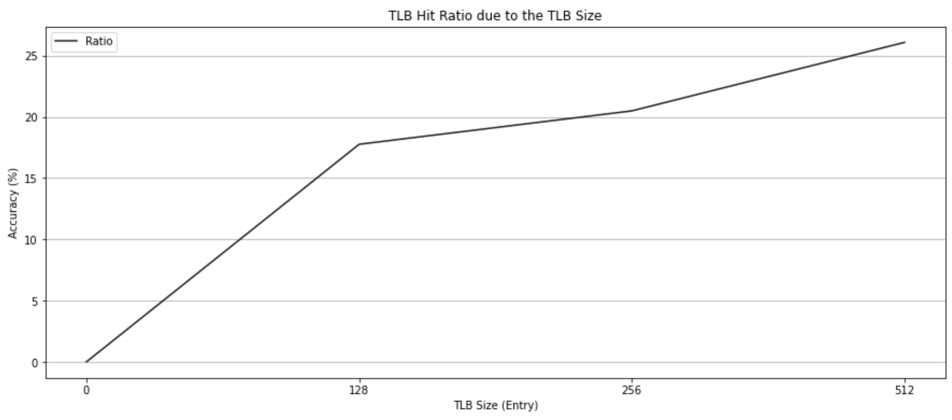


Figure 44 - TLB Hit Ratio

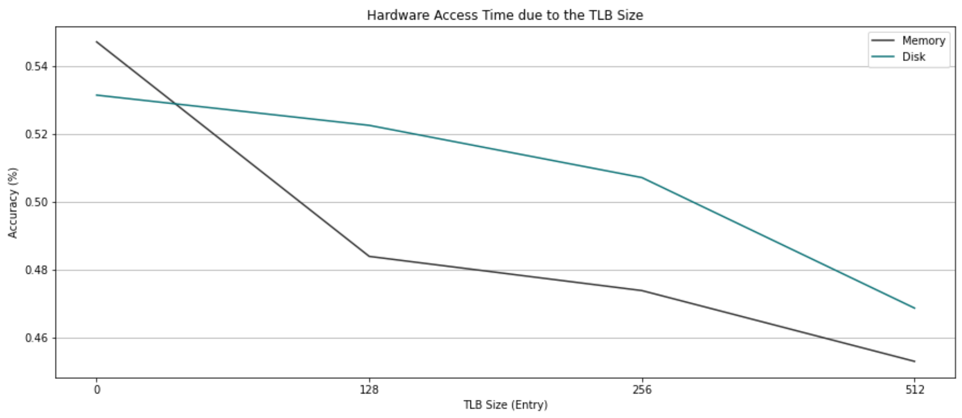


Figure 45 - Hardware (Disk, Main Memory) Access Time

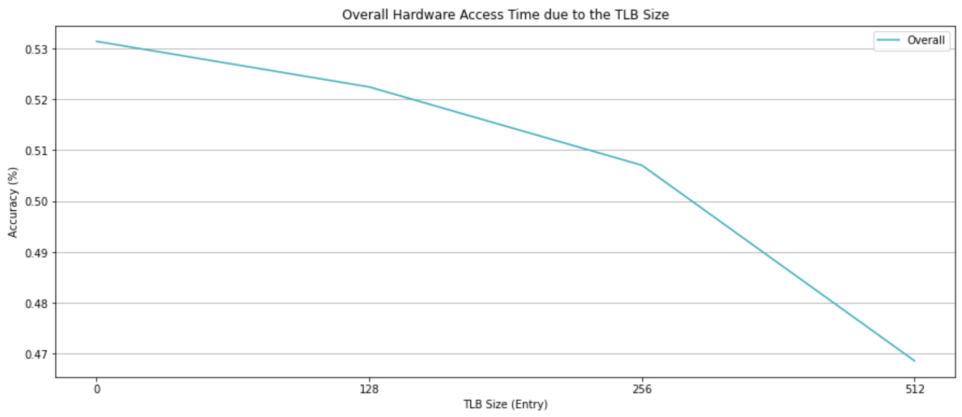
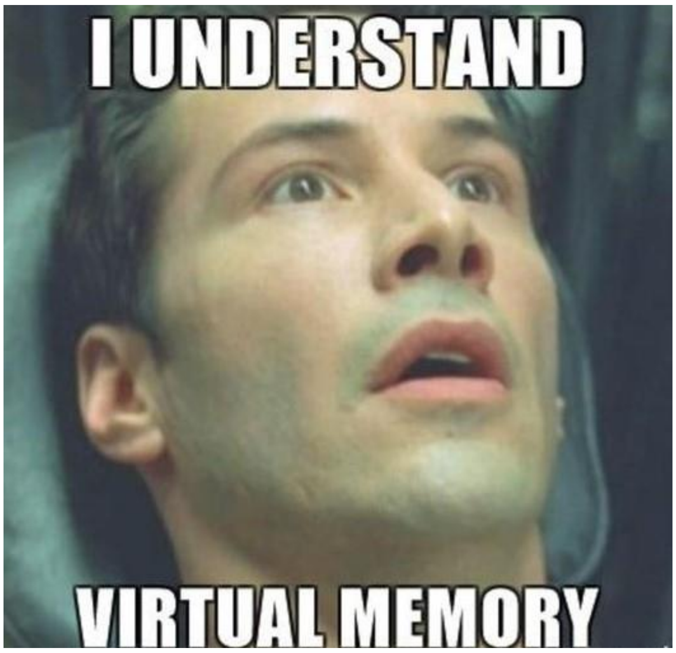


Figure 46 - Overall Hardware Access Time

Also, the size of the table look-aside buffer (TLB) matters to the performance of the virtual memory (paging) program. Figures 44, 45, and 46 present the performance of the virtual memory system according to the different TLB sizes. For faster access to TLB, we assumed that TLB is a fully associative cache with different sizes of bits for the keys. Therefore, since the TLB is formed in a fully associative cache, its accuracy is not that high, which has a maximum accuracy rate of 26.081% when the TLB has 512 entries. However, even though its accuracy is not that high enough, it still shows a significant increment in performance with decreased hardware access time.

From these results, we can conclude that the virtual memory (paging) system can perform at high performance with the proper page replacement algorithm according to the program's memory access pattern, and enough TLB size.

9 Conclusion



Processes that are not entirely in memory can nevertheless run due to the use of virtual memory. This system has the significant benefit that programs can be larger than available physical memory. Additionally, virtual memory separates logical memory as seen by the programmer from physical memory by abstracting main memory into an enormous, homogeneous array of storage. This method removes programmers from the worries associated with memory storage constraints. However, virtual memory is difficult to implement and, if utilized improperly, may significantly reduce performance.

To implement the virtual memory (paging) program that performs the following operation, we first explained the concepts that are used in it. We explained how main memory works, the difference between logical and physical address space, the paging scheme, hardware support for the virtual memory system, translation look-aside buffer (TLB), protection through a valid bit, shared pages, hierarchical paging scheme, swapping, demand paging, free frame list, copy on write, and page replacement policies. Next, we introduced the memory management unit (MMU), two-level paging system, page swapping operation, TLB operation, and various page replacement algorithms, such as first-in, first-out (FIFO), least recently used (LRU), least frequently used (LFU), most frequently used (MFU), second chance (SCA), and enhanced second chance (ESCA) algorithms. Then, we discussed our implementation code for a virtual memory (paging), which contains the code of the TLB and page table structure, memory initialization, allocation and remove operations, copy page, various page replacement algorithms, a function that finds free frame or page table, a function that performs the operations done by the MMU, and additional operations to implement the virtual memory system. At the end of the report, we presented results for the execution of virtual memory (paging) programs with different page replacement policies and different sizes of TLB. We presented the memory access patterns, which are uniformly generated and generated with some strategies. By using these patterns, we showed the different hardware access times, which can be presented as a performance of the system, among different page replacement algorithms. We also showed the TLB hit rate and hardware access time according to the size of the TLB. From those results, we found out that a proper page replacement algorithm that fits in the memory access pattern of the program and enough TLB size increases the performance of the virtual memory (paging) system.

By understanding this paper, we can understand the basic concepts of a virtual memory system and the paging scheme. Also, we can understand the multiple-page replacement algorithms with different memory access patterns. In short, we get the knowledge of concepts and functions that are needed to implement the virtual memory (paging) system.

Citations

[1] Contributor, T. T. (2014, September 9). What is translation lookaside buffer (TLB)?: Definition from TechTarget. WhatIs.com. Retrieved December 16, 2022, from https://www.techtarget.com/whatis/definition/translation-look-aside-buffer-TLB

[2] CS 537 notes, section #16: Paging. (n.d.). Retrieved December 16, 2022, from https://pages.cs.wisc.edu/~bart/537/lecturenotes/s16.html

[3] Documentation – arm developer. (n.d.). Retrieved December 16, 2022, from https://developer.arm.com/documentation/101811/0102/The-Memory-Management-Unit--MMU-

[4] Hashemi, M., Swersky, K., Smith, J. A., Ayers, G., Litz, H., Chang, J., Kozyrakis, C., &amp; Ranganathan, P. (2018, March 6). Learning memory access patterns. arXiv.org. Retrieved December 17, 2022, from https://arxiv.org/abs/1803.02329

[5] Silberschatz, A., Galvin, P. B., &amp; Gagne, G. (2014). 2.2.1 Command Interpreters. In Operating Systems Concepts. essay, Wiley.