Virtual Memory

Hohyeon Cha [32224560]
Nathaniel Vallen [32221415]
Global SW Convergence

List of Contents

Introduction

This project is an expansion of the previous "Simple Scheduling Simulation" project by introducing another concept, which is virtual memory (paging). Virtual memory can be defined as a memory management technique that gives users the impression of having more memory than the computer actually has. This is mainly done by temporarily transferring data from the memory (RAM) to disk storage. The implementation of virtual memory brings some benefits for the user. Firstly, virtual memory promotes the ease of programming for programmers who develop apps which will run on the computer. Virtual memory also provides better protection for each process, mitigating memory contention by providing each process its own virtual address space, which stands independent from other processes. Lastly, virtual memory also frees the user from the computer's physical memory limitation. Through concepts such as address space abstraction, paging, and swapping, the user can run larger and even multiple programs simultaneously.

As mentioned previously, virtual memory provides per process address space, an address space in which a process can work. This is also often referred to as logical/virtual address space. The physical address and virtual address space is connected with a translator inside the CPU. The translator, also called MMU (Memory Management Unit), adopts the concept of paging. Conceptually, the CPU operates using the virtual address space, the hardware accesses the physical address space, and the OS/kernel manages the translation between the two.

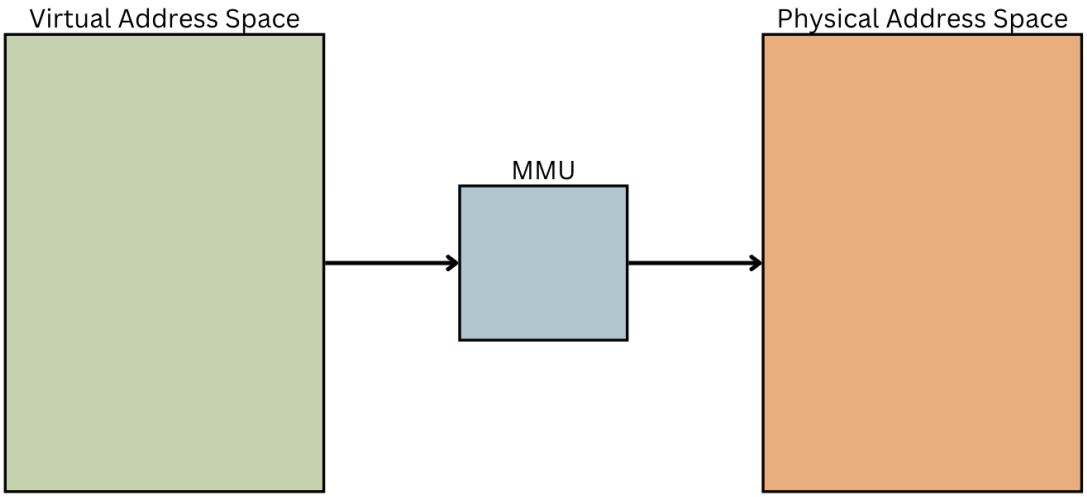

Figure 1. Big picture of memory virtualization

A virtual address space is divided into small units called pages, while a physical address space is divided into page frames. Each of these pages are mapped by the OS to a specific page frame. Pages are representations of the virtual address space which is identified by virtual addresses, and page frames are expected to contain data corresponding to a specific virtual address once it is mapped. Every unit has its own index to differentiate it with the others. A page has a

page number (or a virtual page number (VPN)), while a page frame has a page frame number (PFN) to identify them. A specific page address is usually denoted by a page offset. In a virtual address space, the format of a specific address can be written as "*VA(page_number:page_offset)*", and "*PA(page_frame_number:page_offset)*" in a physical address space.

The MMU translates a virtual address into a physical address by referring to the VA-to-PA mapping in the page table. Specifically, the MMU looks up the virtual page number (VPN) in the page table to obtain the page frame number (PFN) in the physical address space. Since a page table is stored in the memory (RAM), it's crucial for the MMU to know the location of the page table in the memory. Therefore, there exists a register that keeps the beginning address of the page table. This register is called the TTBR (Translation Table Base Register). Each process has its own page table which is loaded when a context switch to the said process occurs. As a consequence, the TTBR will also get switched to point to the new page table whenever a context switch occurs. From all the concepts introduced so far, we can illustrate our project as follows:

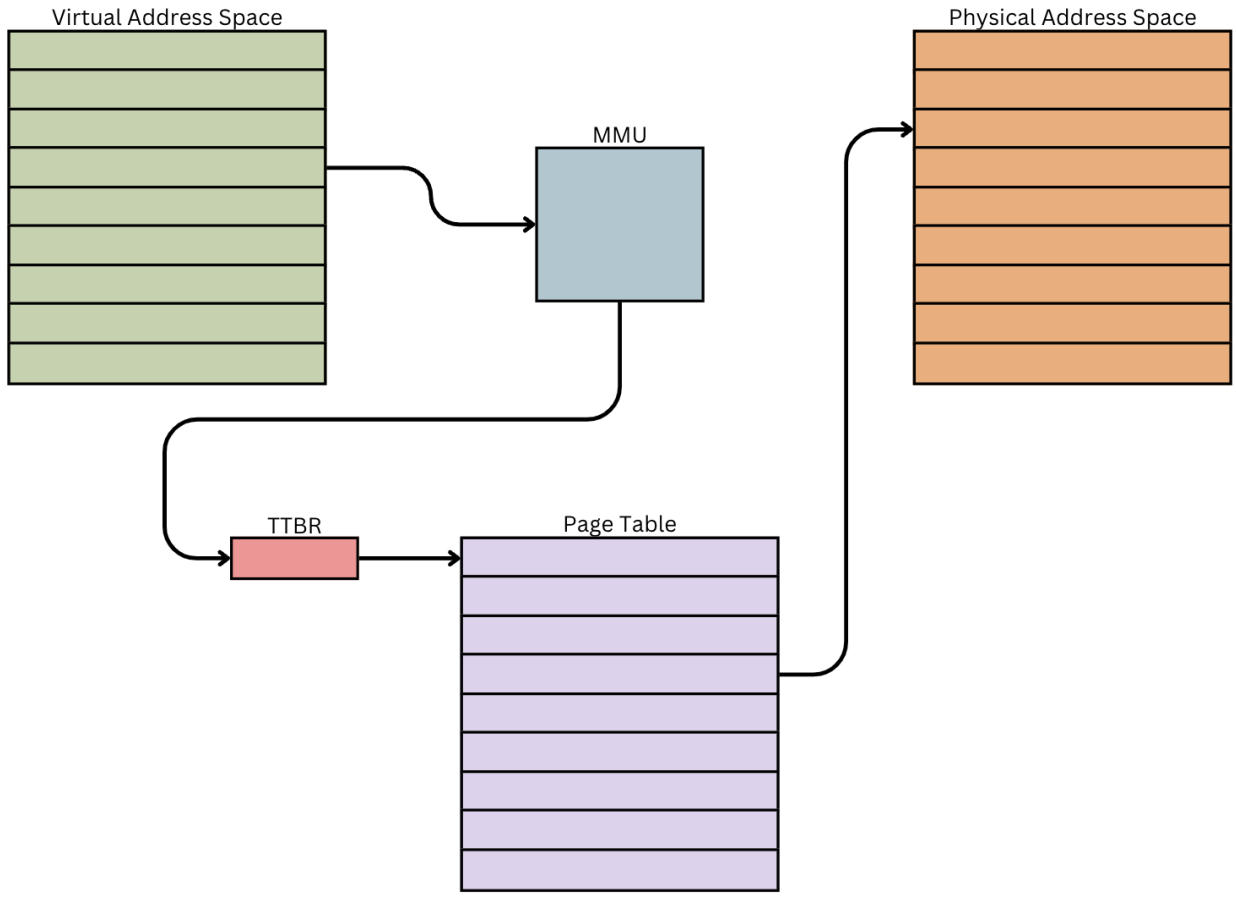

Figure 2. Scheme of VA, PA, MMU, TTBR, and a page table

To further improve the efficiency of the virtual memory, caching can also be implemented. This is done by a small cache located inside the MMU called the translation lookup table or TLB. TLB caches recent VA-to-PA translations to speed up future address translations. With the implementation of TLB, the MMU checks if the target VA is available in the TLB. If it exists, then it counts as a TLB hit, after which the MMU can retrieve the PFN and calculate the specific PA using the page offset. However, if the target VA is not available in the TLB, then it counts as a TLB miss, after which the MMU must perform a page table lookup. Once again, since each process has its own page table, it means that upon a context switch, the TLB is flushed. Another thing to keep in mind is that since TLB is fixed to a specific size (hence has limited entries), a replacement policy is required to swap a mapping in and out of the TLB. These policies include FIFO, LRU, and SCA. FIFO (First In First Out) evicts the oldest entry first, or in other words, the entry that has spent the most time being in the TLB. LRU (Least Recently Used), as the name suggests, removes the least recently used entry first. Meanwhile SCA (Second Chance Algorithm) provides a less strict eviction policy when used with LRU by giving a supposedly removed entry a second chance before being removed in the next case it is selected as the least recently used entry.

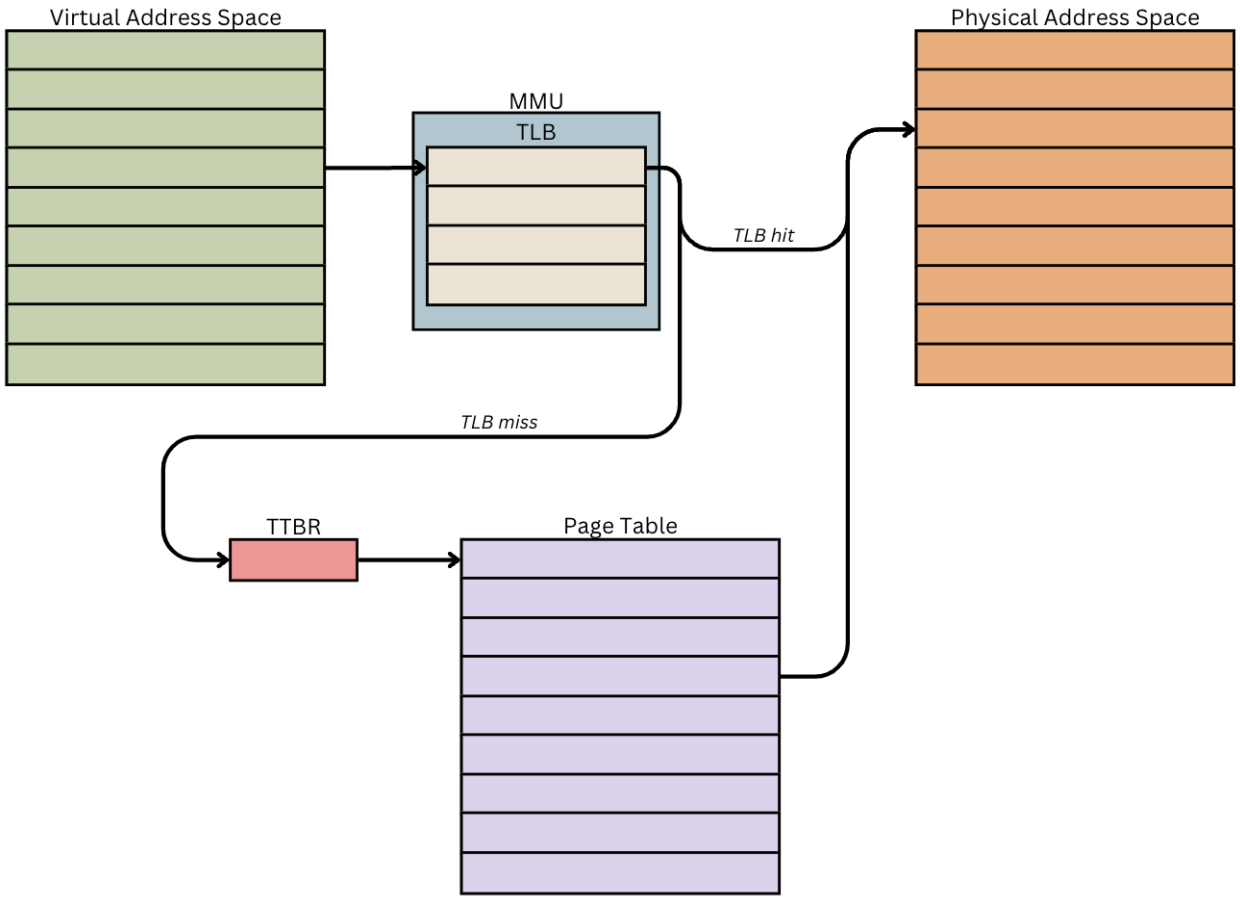

Figure 3. Full scheme with TLB

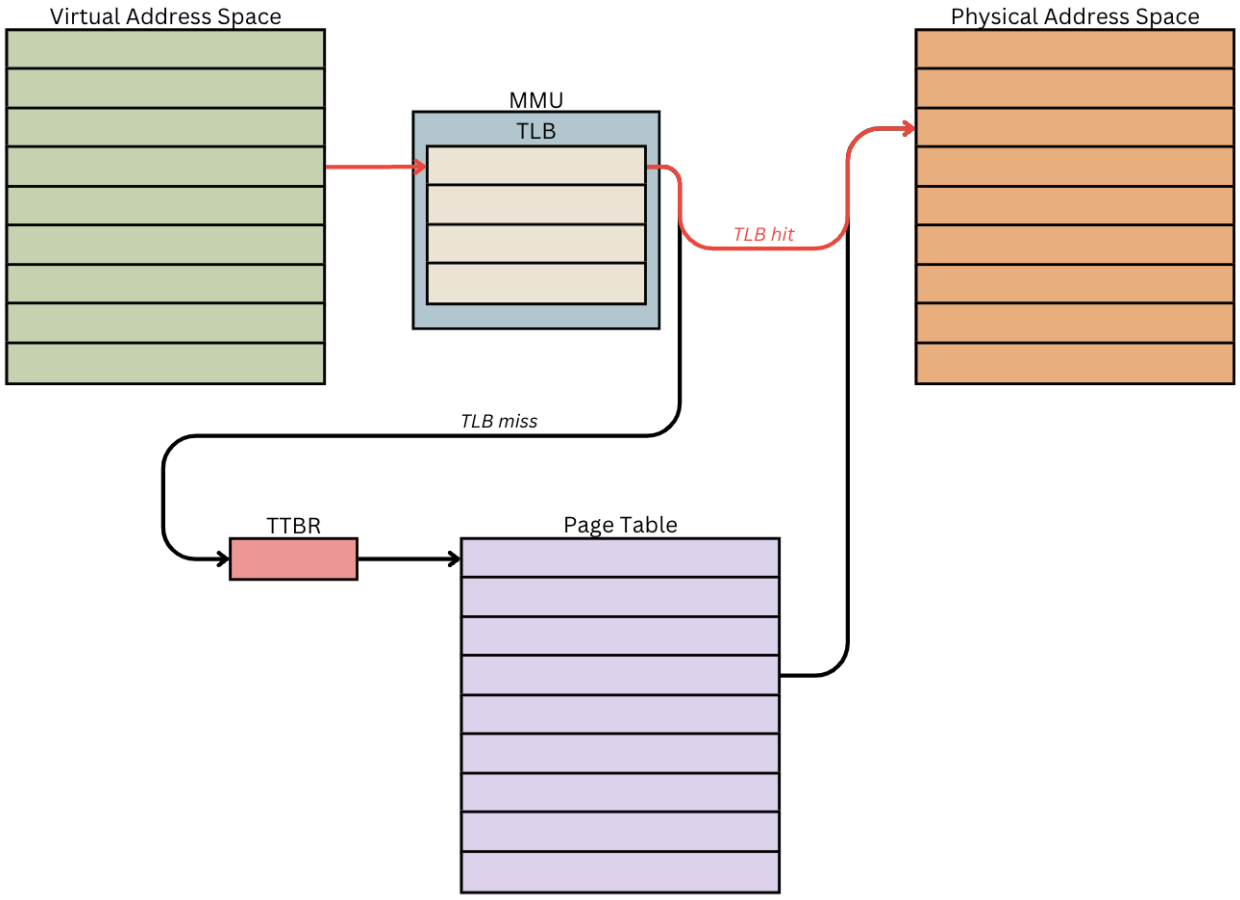

Figure 4. Flowchart of a TLB hit

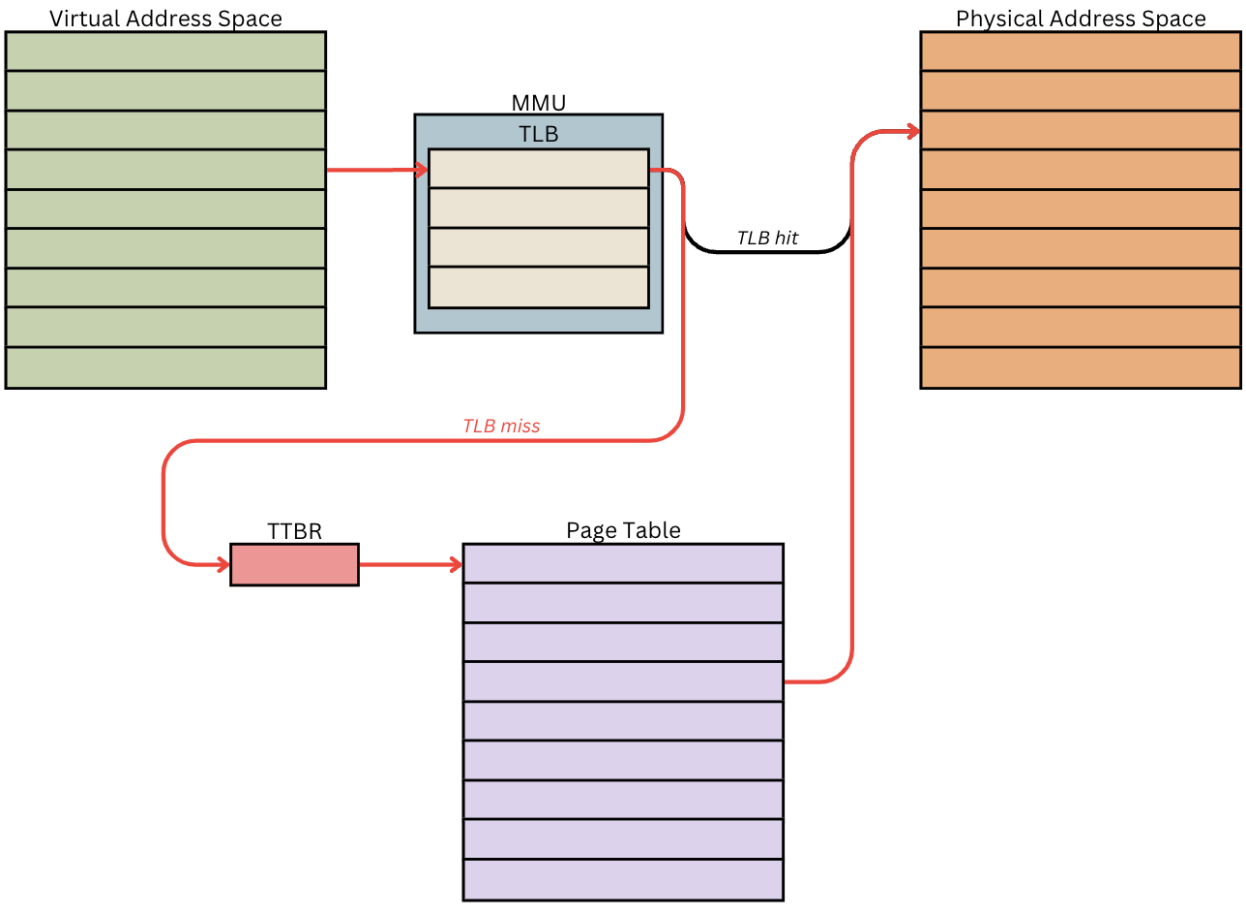

Figure 5. Flowchart of a TLB miss

So far, we have assumed the standard case that each process has a single page table. This concept is known as single-level paging. However, this scheme actually has some shortcomings in regards to size. For example, if we have a standard 32-bit virtual address (4GB virtual address space) with a page size of 4KB, then we would end up with $2^{32} \div 2^{12} = 2^{20}$ (1 million) entries. This is a huge cost in terms of memory space as a paging table resides in RAM. In order to address the scalability issues of single-level paging, we introduce the concept of two-level paging. As the name suggests, two-level paging divides the paging into two levels. The first level, also called the outer page table, has entries which represent coarse grained mapping. Each entry can be perceived as a pointer which points to a second-level page table. The second level, also known as the inner page tables, consists of tables which map a subset of the virtual address to the physical memory.

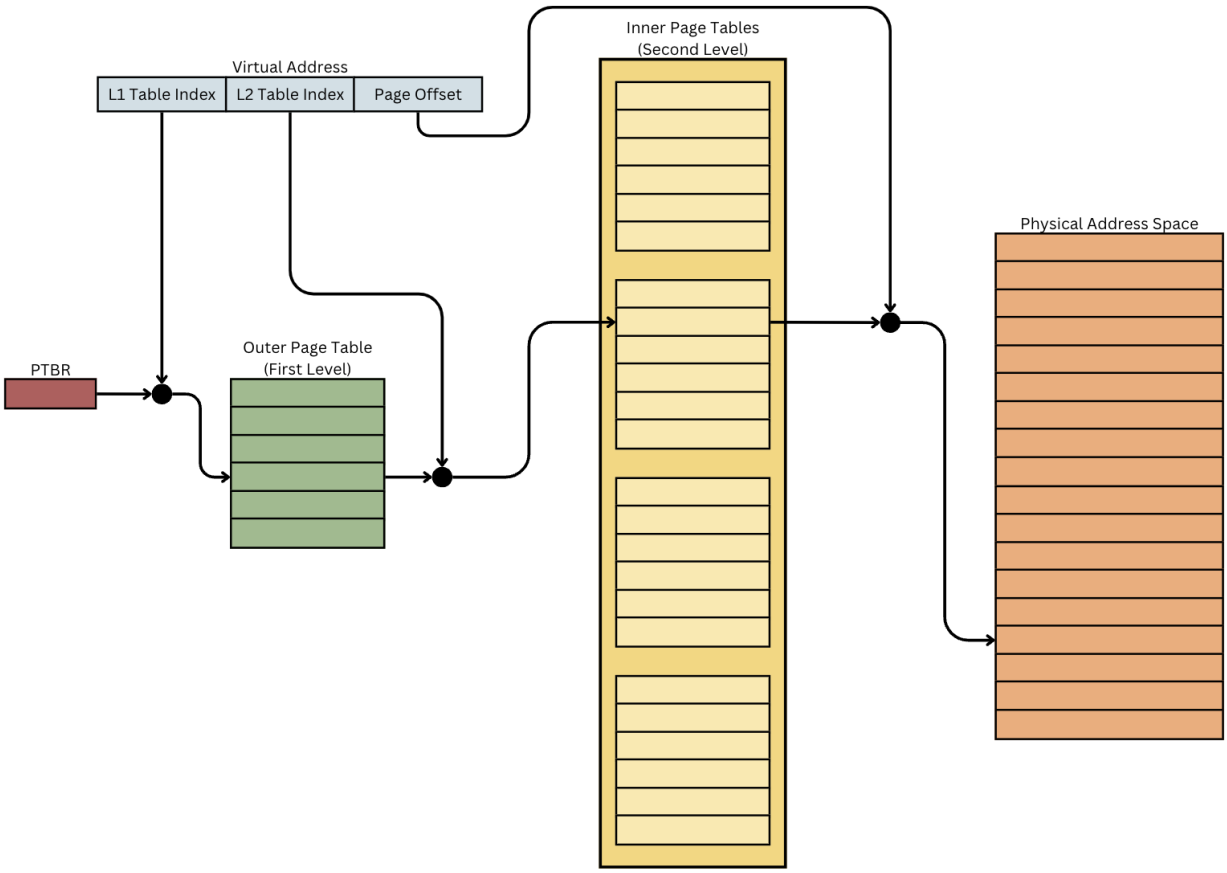

Figure 6. Two-level paging

In two-level paging, the virtual address is roughly divided into three main parts. The first part is the L1 table index, which can be seen as an index/offset for the first-level outer page table. The precise index is calculated by referring to the PTBR (Page Table Base Register) which has the beginning index of the outer page table, and then look at the L1 table index. Similarly, we calculate the index of the second level inner page table by looking at the base address of the second table based on the selected index of the first table. Next, we add it by the value of the L2 table index part in the virtual address to get the page frame on the physical address space. Lastly, we add it by the page offset value to get the final designated physical address.

For both first and second level page tables, allocating the entire table at the start is a waste of resources. To make this more efficient, we can adopt the concept of lazy allocation, which is known as demand paging. With demand paging, all mappings in the page table are initially invalid. This means that they can be considered as “free page frames” with no mapping to any address. Once a process begins and tries to access the memory (invalid mapping), the OS catches this and throws an exception which is known as page fault. In this case, the OS remembers the page number and searches for the data location in the backing store, which is a part of the disk that stores data which is not currently in the memory (RAM). The OS then finds a free page frame and copies the

data (contents of the page) from the backing store to the free page frame. This process is also commonly referred to as swapping, or in this case, “swap-in” to be precise. Next, the OS proceeds to map the page into the address space by filling in the page table. Once everything is done, the OS restarts the instruction that requests for memory access, now with a valid mapping. Another case when swapping is needed is when the main memory (RAM) is full, but we need to load in another page for a certain process. In this case, an existing page frame needs to be evicted (swap-out) and the requested page needs to be copied (swap-in). Similar to TLB, in regards to determining which page to swap out, there are some page eviction policies such as LRU (least recently used) which can be used alongside the second-chance algorithm, working set, and dirty bit.

Important Concepts & Considerations

I. The concepts & considerations from the previous project (Simple Scheduling) applies to this project as well.

II. The workflow of the program can be drawn as shown below:

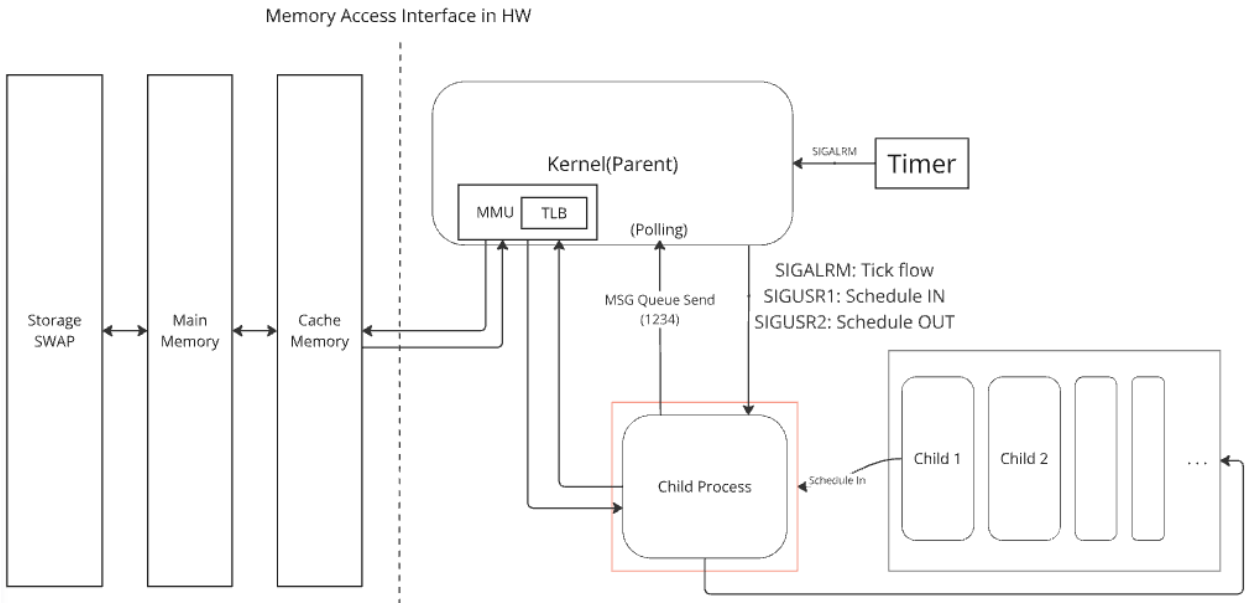

Figure 7. Project workflow

III. In this project, the eviction policy used for TLB entries is LRU.

IV. The program is comprised of several files:

Name	Description

child.c	Contain signal handlers and I/O request IPC for child processes.
scheduler.c	Contain interrupt, alarm, and I/O schedule handler, alongside scheduler initiation and run code.
message_queue.c	Contain message queue initiation and termination.
pid_queue.c	Contain initiation for PCB, PCB queue, and other queue-related operations.
process.c	Contain details for launching the scheduler, child, and parent process.
timer.c	Contain timer initiation and termination.
main.c	Contain result-determining declarations and launcher trigger.
page_manager.c	Manage incoming virtual address translation requests and memory access result
frame_list.c	Manage available page frames
paging.c	Manage page table creation, mapping, and cleanup
shared_memory.c	Manage shared memory initiation and cleanup
swap.c	Manage swapping operations and swap file
tlb.c	Manage TLB initialization and entry operations
tlb_test.c	Contain test loggings for TLB
random.c	Contain functions for VA generation

V. Excluding the ones mentioned in the previous Simple Scheduling project documentation, the following are the important functions used in this project:

Name	Return Type	Details
allocate_page()	void	Allocate a free page.
allocate_swap_slot()	void	Allocate a free slot in the swap.
cleanup_shared_memory(SharedMemory* shm)	void	Clean up the shared memory
create_l_pt()	l_pt_t*	Allocate memory for the lower page table.

create_u_pt()	u_pt_t*	Allocate memory for the upper page table.
free_page(int frame_number)	void	Free a page.
free_page_table()	void	Free a page table.
init_physical_memory()	void	Initialize the physical memory.
init_shared_memory()	SharedMemory*	Initialize the shared memory.
init_swap_space()	int	Create a swap file and a bitmap.
initialize_tlb(tlb_t* tlb)	void	Initialize the TLB.
map_page(u_pt_t* u_pt, uint16_t vaddr)	int	Handle mapping a virtual address to a physical frame.
page_manager()	void	Constantly check for a memory access request and send memory access result.
swap_in_page(u_pt_t* u_pt, uint16_t vaddr)	int	Swap in a page.
swap_out_page(u_pt_t* u_pt, uint16_t vaddr)	int	Swap out a page.
translate_address(u_pt_t* u_pt, tlb_t* tlb, uint16_t vaddr, uint32_t* paddr)	int	Perform a TLB lookup and translate a virtual address to a physical address.
tlb_add_entry(tlb_t* tlb, uint16_t vaddr, uint_32_t frame_number, int read_write, int user_supervisor)	void	Add an entry to the TLB by the designated policy.
tlb_add_entry_fifo(tlb_t* tlb, uint16_t vaddr, uint_32_t frame_number, int read_write, int user_supervisor)	void	Add an entry to the TLB by FIFO.
tlb_add_entry_lru(tlb_t* tlb, uint16_t vaddr, uint_32_t frame_number, int read_write, int	void	Add an entry to the TLB by LRU.

user_supervisor)		
tlb_add_entry_sca(tlb_t* tlb, uint16_t vaddr, uint_32_t frame_number, int read_write, int user_supervisor)	void	Add an entry to the TLB by SCA.
tlb_lookup(tlb_t* tlb, uint16_t vaddr, uint32_t* paddr)	int	Perform a TLB lookup.
unmap_page(u_pt_t* u_pt, tlb_t* tlb, uint16_t vaddr)	void	Invalidate a certain mapping (entry).

VI. While PTBR is used in theory, this project does not directly implement a PTBR, but rather utilizes a function that checks the PID of the current process and sets the page table for virtual address translation.

VII. In this project, all processes are assumed to strictly read from a certain memory address and no writing operations are done.

VIII. Page manager works as a standalone thread, which processes the virtual address translation request through the message queue.

IX. In our project, we include three different statistical distribution based approaches for virtual address generation:

A. Gaussian Distribution
   - Virtual addresses are generated according to a normal distribution.

B. Uniform Distribution
   - Virtual addresses are randomly generated across the entire address space.

C. Zipfian Distribution
   - A skewed distribution where certain addresses are more likely to be chosen, used to simulate real-world access patterns.

X. For each virtual address generation approach, three log files are available:

A. memory_access_log.csv
   - Provides logs for memory access in the form of a table (CSV).
   - Includes details such as timestamp, PID, PA, and VA

B. memory_access_pattern.txt
   - Provides logs for memory access in the form of a text file.
   - Includes details such as PID, PA, VA, and page table status.

C. tlb_cache_hit_log.csv
    - Provides logs for TLB lookups in the form of a table (CSV).
    - Includes details such as timestamp, PID, PA, VA, and TLB hit or miss (0 for hit, -1 for miss).

Build Configurations & Environment

The code in this project was written in C programming language using JetBrains CLion IDE and CMake.

Additional Acknowledgments and Notes

- The assistance of AI-based tools were used in the process of making the test files in tlb_test.c.

- Test codes are included for every 'util' package. However, the test codes do not work at the moment as the 'util' packages underwent a few changes which have not been followed by the modification of the test codes. Therefore, the test codes can be used as a simple reference but do not run it by any means.

- The final submission of the code is pushed to the branch under the name "32224560_dev_stage". For further development history of the project, please refer to the "32224560_hohyeon" branch.

Screen Captures

```
Queue current bytes: 0
Queue number of messages: 0
Queue max bytes: 16384
Page Manager: Received memory access request from PID: 91933, VADDR: 0x76C1
Page Manager: Page table is NULL for PID: 91933
Page Manager: Translated VADDR: 0x76C1 to PADDR: 0x000000C1 for PID: 91933


Page Manager: Received memory access request from PID: 91933, VADDR: 0x7E77
Page Manager: Translated VADDR: 0x7E77 to PADDR: 0x00000177 for PID: 91933


Page Manager: Received memory access request from PID: 91933, VADDR: 0x52AA
Page Manager: Translated VADDR: 0x52AA to PADDR: 0x000002AA for PID: 91933


Page Manager: Received memory access request from PID: 91933, VADDR: 0x97C3
Page Manager: Translated VADDR: 0x97C3 to PADDR: 0x000003C3 for PID: 91933


Page Manager: Received memory access request from PID: 91933, VADDR: 0x64D0
Page Manager: Translated VADDR: 0x64D0 to PADDR: 0x000004D0 for PID: 91933


Page Manager: Received memory access request from PID: 91933, VADDR: 0x9795
Page Manager: Translated VADDR: 0x9795 to PADDR: 0x00000395 for PID: 91933


Page Manager: Received memory access request from PID: 91933, VADDR: 0x8485
Page Manager: Translated VADDR: 0x8485 to PADDR: 0x00000585 for PID: 91933


Page Manager: Received memory access request from PID: 91933, VADDR: 0x7A1E
Page Manager: Translated VADDR: 0x7A1E to PADDR: 0x0000061E for PID: 91933


Page Manager: Received memory access request from PID: 91933, VADDR: 0xA0F7
Page Manager: Translated VADDR: 0xA0F7 to PADDR: 0x000007F7 for PID: 91933


Page Manager: Received memory access request from PID: 91933, VADDR: 0x6631
Page Manager: Translated VADDR: 0x6631 to PADDR: 0x00000831 for PID: 91933


Page Manager: Received memory access request from PID: 91934, VADDR: 0x76C1
Page Manager: Page table is NULL for PID: 91934
Page Manager: Translated VADDR: 0x76C1 to PADDR: 0x000000C1 for PID: 91934
```

Figure 8. Memory access pattern log

memory_access_log

timestamp	pid	vaddr	paddr
2	95743	2	2
3	95743	58	58
5	95743	86F	16F
7	95743	137	237
9	95743	28E	38E
10	95743	812	112
12	95743	29F3	4F3
14	95743	3DEC	5EC
15	95743	D3	D3
17	95743	800E	60E
18	95744	2	2
19	95744	58	58
20	95744	86F	16F
21	95744	137	237
22	95744	28E	38E
23	95744	812	112
24	95744	29F3	4F3
25	95744	3DEC	5EC
26	95744	D3	D3
27	95744	800E	60E
28	95745	2	2

Figure 9. Memory access log

tlb_cache_hit_log

timestamp	pid	vaddr	paddr	
0	91933	76C1	0	-1
2	91933	7E77	C1	-1
4	91933	52AA	177	-1
6	91933	97C3	2AA	-1
8	91933	64D0	3C3	-1
10	91933	9795	395	0
11	91933	8485	395	-1
13	91933	7A1E	585	-1
15	91933	A0F7	61E	-1
17	91933	6631	7F7	-1
19	91934	76C1	C1	0
20	91934	7E77	177	0
21	91934	52AA	2AA	0
22	91934	97C3	3C3	0
23	91934	64D0	4D0	0
24	91934	9795	395	0
25	91934	8485	585	0
26	91934	7A1E	61E	0
27	91934	A0F7	7F7	0
28	91934	6631	831	0
29	91935	76C1	C1	0
30	91935	7E77	177	0
31	91935	52AA	2AA	0
32	91935	97C3	3C3	0
33	91935	64D0	4D0	0

Figure 10. TLB lookup log

Metrics

- TLB Hit & Miss Rate
    - Hit rate can be defined as the number of times the program got a TLB hit divided by the number of memory accesses performed. Conversely, miss rate is the number of TLB misses divided by the number of memory accesses done.
    - Below are the result for each statistical distribution:
        - Gaussian
            - Hit Rate : 0.9630 = 96.30%
            - Miss Rate : 0.0370 = 3.70%
        - Uniform
            - Hit Rate : 0.9042 = 90.42%
            - Miss Rate : 0.0958 = 9.58%
        - Zipfian
            - Hit Rate : 0.9624 = 96.24%

        - Miss Rate : 0.0376 = 3.76%
    - It’s important to note that the values may slightly vary each time the program is run (element of randomness). The values displayed above are rounded to 4 decimals and can be seen as a representative average value.
    - From the representative values above, we can infer that the hit rate is the lowest when the virtual address generation is purely random (uniform distribution). Meanwhile when the virtual address generation follows normal distribution or Zipfian distribution (which comes close to real-world usages), the TLB hit rate can see a notable increase (by around 6%).

Conclusion

The project demonstrates how virtual memory works inside our everyday computers.With relatively basic implementations of virtual memory instruments such as two-level page tables, TLB, MMU, as well as concepts such as address space abstraction, paging, and swapping, we have allowed users to virtually have more space in their computer’s memory to work with.