Segmentation and Paging

Important things to know (Lecture 7, 8 revision):

- **Segmentation:** A flexible, logical division of process memory with per-segment protection and sharing, but suffers from external fragmentation.
    - Divide a process's address space into logical segments. Each segment is a contiguous portion of the process's address space (with a specific length). Examples: Code, Heap, Stack.
- Each segment can be placed separately in physical memory, grow and shrink independently, and have its own protection bits (read, write, execute).
- MMU maintains separate base and bounds registers for each segment. This allows independent translation and protection checks per segment.
- A logical address is divided into two parts: **Segment number** (top bits): selects which segment. **Offset** (low bits): position within that segment.
    - Example:

Digit	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Logical Addr.	**0**	**1**	0	0	0	0	0	1	1	0	1	0	0	0
Identifier	Segment		Offset											

- **Special case of Stack:** Stack often grows in the negative direction. MMU tracks whether a segment grows upward or downward. Address translation for the stack is slightly different from that for code/heap.
- **Code Sharing:** Segmentation allows the sharing of specific memory segments between processes. Each segment has hardware-level protection bits (read-write, read-execute) to control sharing.
- **Responsibilities of OS on Context-switch:** It must save/restore all segment base and bounds registers, handle growing/shrinking of segments (e.g. stack/heap expansion), manage allocation of new segments, and must deal with external fragmentation *(when the total amount of free memory is sufficient to meet a process's request, but it is scattered into small, non-contiguous blocks or "holes" that are too small to be useful)*.
- Free-list management algorithms may resolve this issue, ***but not always***:
    - **First-Fit** allocates the first partition that is large enough. It is the quickest method, but not the most efficient in terms of memory usage. **Best-Fit** checks all free partitions and allocates the smallest partition that can accommodate the process, minimizing wasted space. Aims for memory efficiency, but is slower. **Worst-Fit** allocates the largest available partition, aiming to leave large blocks free for future use, but it can still result in fragmentation.

- **<u>Paging:</u>**

- A challenge with segmentation is external fragmentation. The memory holes are too small and scattered. *Paging* is a solution that eliminates external fragmentation by using fixed-sized units.

- **Page (logical memory unit):** fixed-size block of a process's logical address space.

- **Page Frame (physical memory unit):** fixed-size slot in physical memory (to store the page). Each virtual page is *mapped* to a physical page frame.

- *Segmentation vs Paging:* Segmentation creates segments or variable-sized chunks. Paging creates fixed-sized chunks that are simpler to manage.

- **Page Table:** Per-process data structure maintained by the OS. Stores the **mapping between virtual pages and physical frames**. **Page-Table Base Register (PTBR):** holds the starting physical address of the page table.

- Page Table Entry Fields
    - **Present Bit (P):** whether the page is in physical memory or swapped to disk.
    - **Read/Write Bit (R/W):** permission for writes.
    - **User/Supervisor Bit (U/S):** user-mode accessibility.
    - **PWT/PCD/PAT/G Bits:** hardware caching control.
    - **Accessed Bit (A):** set if the page has been accessed.
    - **Dirty Bit (D):** set if page has been modified since loaded.

- Address Translation in Paging:
    - The CPU generates a **logical address**.
    - Logical/Virtual address split into: **Virtual Page Number (VPN)** (used to index into page table) and a **Page Offset** (copied directly into physical address).
    - Page table lookup finds the **frame number**.
    - **Physical address = frame number + page offset.**

- Disadvantage of Paging: It is slow as it requires memory access to fetch the PTE and then to access the actual data, causing a translation overhead.

- Solution? A **Translation Lookaside Buffer (TLB)** for faster translation and dramatically improved performance**.** This is a hardware cache of recent LA to PA translations.
    - On each memory reference, check the TLB first.
    - If entry is present (TLB hit): translation is fast, no memory access to the page table needed.
    - If not present (TLB miss): must consult the page table.

Questions:

Q1. Consider an operating system that uses 48-bit virtual addresses and 16KB pages. The system uses a hierarchical page table design to store all the page table entries of a process, and each page table entry is 4 bytes in size. What is the total number of pages that are required to store the page table entries of a process, across all levels of the hierarchical page table?

Ans: $2^{22} + 2^{10} + 1$

Virtual Address space is 48-bit and page size = 16 KB = 2^{14} bytes. So, the total number of page table entries = $2^{48}/2^{14} = 2^{34}$.

Each page table entry is 4 bytes, and the page size is 16 KB, so the number of entries stored in one page table = 16KB/4 = 2^{12} page table entries.

Lowest level: So, the number of pages at the lowest level of the hierarchical table would contain entries for all 2^{34} virtual pages, so the number of pages at the lowest level = $2^{34}/2^{12}$ = $\mathbf{2^{22}}$.

Second-lowest level: Now, pointers to all these lowest-level pages must be stored in the next level of the page table, so the next level of the page table has $2^{22}/2^{12}$ = $\mathbf{2^{10}}$ pages.

Third-lowerst level: Finally, these pointers must be stored in the next level of the page table, so this would have $2^{10}/2^{12} <$ **1**. Therefore, a single page can store all the 2^{10} page table entries, so the outermost level has one page.

So, the total number of pages that store page table entries is $\mathbf{2^{22} + 2^{10} + 1}$.

Q2. Consider a system with 8-bit virtual and physical addresses, and 16-byte pages. A process in this system has 4 logical pages, which are mapped to 3 physical pages in the following manner: logical page 0 maps to physical page 6, 1 maps to 3, 2 maps to 11, and logical page 5 is not mapped to any physical page yet. All the other pages in the virtual address space of the process are marked invalid in the page table. The MMU is given a pointer to this page table for address translation. Further, the MMU has a small TLB cache that stores two entries, for logical pages 0 and 2. For each virtual address shown below, describe what happens when that address is accessed by the CPU. Specifically, you must answer what happens at the TLB (hit or miss?), MMU (which page table entry is accessed?), OS (is there a trap of any kind?), and the physical memory (which physical address is accessed?). You may write the translated physical address in binary format. (Note that it is not implied that the accesses below happen one after the other; you must solve each part of the question independently using the information

(a) Virtual address 7, (b) Virtual address 20, (c) Virtual address 70, (d) Virtual address 80

Ans. (a) 7 = 0000 (page number) + 0111 (offset) = logical page 0. TLB hit. No page table walk. No OS trap. Physical address 0110 0111 (207) is accessed.
(b) 20 = 0001 0100 = logical page 1. TLB miss. MMU walks the page table. Physical address 0011 0100

(c) 70 = 0100 0110 = logical page 4. TLB miss. MMU accesses the page table and discovers it is an invalid entry. MMU raises a trap to the OS.
(d) 80 = 0101 0000 = logical page 5. TLB miss. MMU accesses the page table and discovers page not present. MMU raises a page fault to the OS.

Q3. Consider a system with several running processes. The system is running a modern OS that uses virtual addresses and demand paging. It has been empirically observed that the memory access times in the system under various conditions are: t1 when the logical memory address is found in TLB cache, t2 when the address is not in TLB but does not cause a page fault, and t3 when the address results in a page fault. This memory access time includes all overheads like page fault servicing and logical-to-physical address translation. It has been observed that, on average, 10% of the logical address accesses result in a page fault. Further, of the remaining virtual address accesses, two-thirds of them can be translated using the TLB cache, while one-third require walking the page tables. Using the information provided above, calculate the average expected memory access time in the system in terms of t1,t2, and t3.

Ans: Given: t3 occurs 10% of the time.

Of the remaining virtual address accesses (remaining 90% of the time), 2/3rds can be translated using the TLB cache (t1 occurs 60% of the time) and 1/3rd requires walking the page tables (t2 occurs 30% of the time).

This implies that the average expected memory access time in the system in terms of t1, t2, and t3 would be **0.6*t1 + 0.3*t2 + 0.1*t3**.

Q4. Consider a process P running in a Linux-like operating system that implements demand paging. The page/frame size in the system is 4KB. The process has 4 pages in its heap. The process stores an array of 4K integers (size of integer is 4 bytes) in these 4 pages. The process then proceeds to access the integers in the array sequentially. Assume that none of these 4 pages of the heap are initially in physical memory. The memory allocation policy of the OS allocates only 3 physical frames at any point in time to store these 4 pages of the heap. In case of a page fault and all 3 frames have been allocated to the heap of the process, the OS uses an LRU policy to evict one of these 4 pages to make space for the new page. Approximately what fraction of the 4K accesses to array elements will result in a page fault?
(a) Almost 100%, (b) Approximately 25%, (c) Approximately 75%, (d) Approximately 0.1%

Ans. Given:

- Page/frame size: 4KB
- Process has 4 pages in its heap
- Each page can store 4KB / 4 bytes = 1K integers
- Total integers in the array: 4K integers
- Memory allocation policy: Only 3 physical frames available
- Eviction policy: LRU (Least Recently Used)

The process has 4 pages of 4KB each, so the total heap size is 4 pages * 4KB/page = 16KB.

Each page can store 1K integers, so the total number of integers in the array is 4 pages * 1K integers/page = 4K integers.

Since only 3 physical frames are available at any given time, the process can have at most 3 pages of its heap in physical memory at once. Initially, no pages are in physical memory.

When the process starts accessing the array sequentially, it will cause page faults as it accesses pages that are not in physical memory. So, the process will generate a page fault for the first access to each page of the array. Since there are 4 pages, the process will generate a total of 4 page faults. The fraction of page faults is therefore (4 / 4000) * 100 = **0.1%**.

Q5. Consider a process running on a system with a 52-bit CPU (i.e., virtual addresses are 48 bits in size). The system has a physical memory of 8GB. The page size in the system is 4KB, and the size of a page table entry is 4 bytes. The OS uses hierarchical paging. Which of the following statements is/are true? You can assume 2^{10} = 1K, 2^{20} = 1M, etc.
(a) We require a 4-level page table to keep track of the virtual address space of a process.
(b) We require a 5-level page table to keep track of the virtual address space of a process.
(c) The most significant 9 bits are used to index into the outermost page directory by the MMU during address translation.
(d) The most significant 40 bits of a virtual address denote the page number, and the least significant 12 bits denote the offset within a page.

Ans: **(a), (d)**

(a)To calculate the number of levels of the page table required to address a given virtual address space, we can use the following formula:

Number of levels = $\log_2$(virtual address space size) / $\log_2$(page size)

In this case, the virtual address space size is 48 bits and the page size is 4KB (2^{12} bytes). Therefore, the number of levels of page table required is:

Number of levels = $\log_2(2^{48}) / \log_2(2^{12})$ = 48/12 = 4

Therefore, we require a 4-level page table to address a 48-bit virtual address space with a page size of 4KB. This means that (b) is false.

Page Offset: The number of bits needed to address an offset within a page is determined by the page size. Since the page size is 4KB (2^{12} bytes), we need 12 bits for the page offset because 2^{12} = 4,096, which covers all possible byte positions within a 4KB page.

The remaining 52-12 = 40 bytes of memory will be used for the page number. (d) is true.