Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

# Practice Problems: Memory

1. Provide one advantage of using the slab allocator in Linux to allocate kernel objects, instead of simply allocating them from a dynamic memory heap.

   **Ans:** A slab allocator is fast because memory is preallocated. Further, it avoids fragmentation of kernel memory.

2. In a 32-bit architecture machine running Linux, for every physical memory address in RAM, there are at least 2 virtual addresses pointing to it. That is, every physical address is mapped at least twice into the virtual address space of some set of processes. [T/F]

   **Ans:** F (this may be true of simple OS like xv6 studied in class, but not generally true)

3. Consider a system with $N$ bytes of physical RAM, and $M$ bytes of virtual address space per process. Pages and frames are $K$ bytes in size. Every page table entry is $P$ bytes in size, accounting for the extra flags required and such. Calculate the size of the page table of a process.

   **Ans:** M/K * P

4. The memory addresses generated by the CPU when executing instructions of a process are called logical addresses. [T/F]

   **Ans:** T

5. When a C++ executable is run on a Linux machine, the kernel code is part of the executable generated during the compilation process. [T/F]

   **Ans:** F (it is only part of the virtual address space)

6. When a C++ executable is run on a Linux machine, the kernel code is part of the virtual address space of the running process. [T/F]

   **Ans:** T

7. Consider a Linux-like OS running on x86 Intel CPUs. Which of the following events requires the OS to update the page table pointer in the MMU (and flush the changes to the TLB)? Answer "update" or "no update".

   (a) A process moves from user mode to kernel mode.
   **Ans:** no update

   (b) The OS switches context from one process to another.
   **Ans:** update

8. Consider a process that has just forked a child. The OS implements a copy-on-write fork. At the end of the fork system call, the OS does not perform a context switch and will return back to the user mode of the parent process. Now, which of the following entities are updated at the end of a successful implementation of the fork system call? Answer "update" or "no update".

(a) The page table of the parent process.
**Ans:** update because the parent's pages must be marked read-only.

(b) The page table information in the MMU and the TLB.
**Ans:** update because the parent's pages must be marked read-only.

9. A certain page table entry in the page table of a process has both the valid and present bits set. Describe what happens on a memory access to a virtual address belonging to this page table entry.

   (a) What happens at the TLB? (hit/miss/cannot say)
   **Ans:** cannot say

   (b) Will a page fault occur? (yes/no/cannot say)
   **Ans** no

10. A certain page table entry in the page table of a process has the valid bit set but the present bit unset. Describe what happens on a memory access to a virtual address belonging to this page table entry.

    (a) What happens at the TLB? (hit/miss/cannot say)
    **Ans:** miss

    (b) Will a page fault occur? (yes/no/cannot say)
    **Ans** yes

11. Consider the page table entries within the page table of a process that map to kernel code/data stored in RAM, in a Linux-like OS studied in class.

    (a) Are the physical addresses of the kernel code/data stored in the page tables of various process always the same? (yes/no/cannot say)
    **Ans:** yes, because there is only one copy of kernel code in RAM

    (b) Does the page table of every process have page table entries pointing to the kernel code/data? (yes/no/cannot say)
    **Ans:** yes, because every process needs to run kernel code in kernel mode

12. 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 of time, to the 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 a LRU policy to evict one of these pages to make space for the new page. Approximately what percentage 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:** (d)

13. Given below are descriptions of different entries in the page table of a process, with respect to which bits are set and which are not set. Accessing which of the page table entries below will always result in the MMU generating a trap to the OS during address translation?

    (a) Page with both valid and present bits set
    (b) Page with valid bit set but present bit unset
    (c) Page with valid bit unset
    (d) Page with valid, present, and dirty bits set

    **Ans:** (b), (c)

14. Which of the following statements is/are true regarding the memory image of a process?

    (a) Memory for non-static local variables of a function is allocated on the heap dynamically at run time
    (b) Memory for arguments to a function is allocated on the stack dynamically at run time
    (c) Memory for static and global variables is allocated on the stack dynamically at run time
    (d) Memory for the argc, argv arguments to the main function is allocated in the code/data section of the executable at compile time

    **Ans:** (b)

15. Consider a process P in a Linux-like operating system that implements demand paging. Which of the following pages in the page table of the process will have the valid bit set but the present bit unset?

    (a) Pages that have been used in the past by the process, but were evicted to swap space by the OS due to memory pressure
    (b) Pages that have been requested by the user using mmap/brk/sbrk system calls, but have not yet been accessed by the user, and hence not allocated physical memory frames by the OS
    (c) Pages corresponding to unused virtual addresses in the virtual address space of the process
    (d) Pages with high virtual addresses mapping to OS code and data

    **Ans:** (a), (b)

16. Consider a process P in a Linux-like operating system that implements demand paging. For a particular page in the page table of this process, the valid and present bits are both set. Which of the following are possible outcomes that can happen when the CPU accesses a virtual address in this page of the process? Select all outcomes that are possible.

    (a) TLB hit (virtual address found in TLB)
    (b) TLB miss (virtual address not found in TLB)
    (c) MMU walks the page table (to translate the address)
    (d) MMU traps to the OS (due to illegal access)

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

17. Consider a process P in a Linux-like operating system that implements demand paging using the LRU page replacement policy. You are told that the i-th page in the page table of the process has the accessed bit set. Which of the following statements is/are true?

    (a) This bit was set by OS when it allocated a physical memory frame to the page

    (b) This bit was set by MMU when the page was accessed in the recent past

    (c) This page is likely to be evicted by the OS page replacement policy in the near future

    (d) This page will always stay in physical memory as long as the process is alive

    **Ans:** (b)

18. Which of the following statements is/are true regarding the functions of the OS and MMU in a modern computer system?

    (a) The OS sets the address of the page table in a CPU register accessible to the MMU every time a new process is created in the system

    (b) The OS sets the address of the page table in a CPU register accessible to the MMU every time a new process is context switched in by the CPU scheduler

    (c) MMU traps to OS every time an address is not found in the TLB cache

    (d) MMU traps to OS every time it cannot translate an address using the page table available to it

    **Ans:** (b), (d)

19. Consider a modern computer system using virtual addressing and translation via MMU. Which of the following statements is/are valid advantages of using virtual addressing as opposed to directly using physical addresses to fetch instructions and data from main memory?

    (a) One does not need to know the actual addresses of instructions and data in main memory when generating compiled executables.

    (b) One can easily provide isolation across processes by limiting the physical memory that is mapped into the virtual address space of a process.

    (c) Using virtual addressing allows us to hide the fact that user's memory is allocated non-contiguously, and helps provide a simplified view to the user.

    (d) Memory access using virtual addressing is faster than directly accessing memory using physical addresses.

    **Ans:** (a), (b), (c)

20. Consider a process running on a system with a 52-bit CPU (i.e., virtual addresses are 52 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, and so on.

    (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)

21. Consider the following line of code in a function of a process.
`int *x = (int *)malloc(10 * sizeof(int));`

When this function is invoked and executed:

(a) Where is the memory for the variable `x` allocated within the memory image of the process? (stack/heap)

**Ans:** stack

(b) Where is the memory for the 10 integer variables allocated within the memory image of the process? (stack/heap)

**Ans:** heap

22. Consider an OS that is not using a copy-on-write implementation for the `fork` system call. A process P has spawned a child C. Consider a virtual address $v$ that is translated to physical address $A_p(v)$ using the page table of P, and to $A_c(v)$ using the page table of C.

(a) For which virtual addresses $v$ does the relationship $A_p(v) = A_c(v)$ hold?

**Ans:** For kernel space addresses, shared libraries and such.

(b) For which virtual addresses $v$ does the relationship $A_p(v) = A_c(v)$ not hold?

**Ans:** For userspace part of memory image, e.g., code, data, stack, heap.

23. Consider a system with paging-based memory management, whose architecture allows for a 4GB virtual address space for processes. The size of logical pages and physical frames is 4KB. The system has 8GB of physical RAM. The system allows a maximum of 1K (=1024) processes to run concurrently. Assuming the OS uses hierarchical paging, calculate the maximum memory space required to store the page tables of *all* processes in the system. Assume that each page table entry requires an additional 10 bits (beyond the frame number) to store various flags. Assume page table entries are rounded up to the nearest byte. Consider the memory required for both outer and inner page tables in your calculations.

**Ans:**

Number of physical frames = $2^{33}/2^{12}$ = $2^{21}$. Each PTE has frame number (21 bits) and flags (10 bits) $\approx$ 4 bytes. The total number of pages per process is $2^{32}/2^{12}$=$2^{20}$, so total size of inner page table pages is $2^{20} \times 4$ = 4MB.

Each page can hold $2^{12}/4$ = $2^{10}$ PTEs, so we need $2^{20}/2^{10}$ PTEs to point to inner page tables, which will fit in a single outer page table. So the total size of page tables of one process is 4MB + 4KB. For 1K process, the total memory consumed by page tables is 4GB + 4MB.

24. Consider a simple system running a single process. The size of physical frames and logical pages is 16 bytes. The RAM can hold 3 physical frames. The virtual addresses of the process are 6 bits in size. The program generates the following 20 virtual address references as it runs on the CPU: `0, 1, 20, 2, 20, 21, 32, 31, 0, 60, 0, 0, 16, 1, 17, 18, 32, 31, 0, 61`. (Note: the 6-bit addresses are shown in decimal here.) Assume that the physical frames in RAM are initially empty and do not map to any logical page.

    (a) Translate the virtual addresses above to logical page numbers referenced by the process. That is, write down the reference string of 20 page numbers corresponding to the virtual address accesses above. Assume pages are numbered starting from 0, 1, ...

    (b) Calculate the number of page faults genrated by the accesses above, assuming a FIFO page replacement algorithm. You must also correctly point out which page accesses in the reference string shown by you in part (a) are responsible for the page faults.

    (c) Repeat (b) above for the LRU page replacement algorithm.

    (d) What would be the lowest number of page faults achievable in this example, assuming an optimal page replacement algorithm were to be used? Repeat (b) above for the optimal algorithm.

    **Ans:**

    (a) For 6 bit virtual addresses, and 4 bit page offsets (page size 16 bytes), the most significant 2 bits of a virtual address will represent the page number. So the reference string is 0, 0, 1, 0, 1, 1, 2, 1, 0, 3 (repeated again).

    (b) Page faults with FIFO = 8. Page faults on 0,1,2,3 (replaced 0), 0 (replaced 1), 1 (replaced 2), 2 (replaced 3), 3.

    (c) Page faults with LRU = 6. Page faults on 0, 1, 2, 3 (replaced 2), 2 (replaced 3), 3.

    (d) The optimum algorithm will replace the page least likely to be used in future, and would look like LRU above.

25. Consider a system with only virtual addresses, but no concept of virtual memory or demand paging. Define *total memory access time* as the time to access code/data from an address in physical memory, including the time to resolve the address (via the TLB or page tables) and the actual physical memory access itself. When a virtual address is resolved by the TLB, experiments on a machine have empirically observed the total memory access time to be (an approximately constant value of) $t_h$. Similarly, when the virtual address is not in the TLB, the total memory access time is observed to be $t_m$. If the average total memory access time of the system (averaged across all memory accesses, including TLB hits as well as misses) is observed to be $t_x$, calculate what fraction of memory addresses are resolved by the TLB. In other words, derive an expression for the TLB hit rate in terms of $t_h$, $t_m$, and $t_x$. You may assume $t_m > t_h$.

    **Ans:** We have $t_x = h * t_h + (1 - h) * t_m$, so $t_h = \frac{t_m - t_x}{t_m - t_h}$

26. 4. Consider a system with a 6 bit virtual address space, and 16 byte pages/frames. The mapping from virtual page numbers to physical frame numbers of a process is (0,8), (1,3), (2,11), and (3,1). Translate the following virtual addresses to physical addresses. Note that all addresses are in decimal. You may write your answer in decimal or binary.

(a) 20

(b) 40

**Ans:**

(a) 20 = 01 0100 = 11 0100 = 52

(b) 40 = 10 1000 = 1011 1000 = 184

27. 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 an 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:** 0.6*t1 + 0.3*t2 + 0.1*t3

28. Consider a system where each process has a virtual address space of $2^v$ bytes. The physical address space of the system is $2^p$ bytes, and the page size is $2^k$ bytes. The size of each page table entry is $2^e$ bytes. The system uses hierarchical paging with $l$ levels of page tables, where the page table entries in the last level point to the actual physical pages of the process. Assume $l \geq 2$. Let $v_0$ denote the number of (most significant) bits of the virtual address that are used as an index into the outermost page table during address translation.

    (a) What is the number of logical pages of a process?

    (b) What is the number of physical frames in the system?

    (c) What is the number of PTEs that can be stored in a page?

    (d) How many pages are required to store the innermost PTEs?

    (e) Derive an expression for $l$ in terms of $v$, $p$, $k$, and $e$.

    (f) Derive an expression for $v_0$ in terms of $l$, $v$, $p$, $k$, and $e$.

    **Ans:**

    (a) $2^{v-k}$

    (b) $2^{p-k}$

    (c) $2^{k-e}$

    (d) $2^{v-k}$ / $2^{k-e}$ = $2^{v+e-2k}$

    (e) The least significant $k$ of $v$ bits indicate offset within a page. Of the remaining $v-k$ bits, $k-e$ bits will be used to index into the page tables at every level, so the number of levels $l$ = ceil $\frac{v-k}{k-e}$

    (f) $v - k - (l-1) * (k-e)$

29. 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:** Page size = $2^{14}$ bytes. So, the number of page table entries = $2^{48}$/$2^{14}$ = $2^{34}$. Each page can store 16KB/4 = $2^{12}$ page table entries. So, the number of innermost pages = $2^{34}$ / $2^{12}$ = $2^{22}$.

    Now, pointers to all these innermost pages must be stored in the next level of the page table, so the next level of the page table has $2^{22}$ / $2^{12}$ = $2^{10}$ pages. Finally, 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 $2^{22} + 2^{10} + 1$.

30. Consider a memory allocator that uses the buddy allocation algorithm to satisfy memory requests. The allocator starts with a heap of size 4KB (4096 bytes). The following requests are made to the allocator by the user program (all sizes requested are in bytes): ptr1 = malloc(500); ptr2 = malloc(200); ptr3 = malloc(800); ptr4 = malloc(1500). Assume that the header added by the allocator is less than 10 bytes in size. You can make any assumption about the implementation of the buddy allocation algorithm that is consistent with the description in class.

    (a) Draw a figure showing the status of the heap after these 4 allocations complete. Your figure must show which portions of the heap are assigned and which are free, including the sizes of the various allocated and free blocks.

    (b) Now, suppose the user program frees up memory allocations of ptr2, ptr3, and ptr4. Draw a figure showing the status of the heap once again, after the memory is freed up and the allocation algorithm has had a chance to do any possible coalescing.

    **Ans:**

    (a) [512 B][256 B] 256 B free [1024 B][2048 B]

    (b) [512 B] 512 B free, 1024 B free, 2048 B free. No further coalescing is possible.

31. 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 provided above.)

    (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 is accessed.

(b) 20 = 0001 0100 = logical page 1. TLB miss. MMU walks page table. Physical address 0011 0100

(c) 70 = 0100 0110 = logical page 4. TLB miss. MMU accesses page table and discovers it is an invalid entry. MMU raises trap to OS.

(d) 80 = 0101 0000 = logical page 5. TLB miss. MMU accesses page table and discovers page not present. MMU raises a page fault to the OS.

32. Consider a system with 8-bit addresses and 16-byte pages. A process in this system has 4 logical pages, which are mapped to 3 physical frames in the following manner: logical page 0 maps to physical frame 2, page 1 maps to frame 0, page 2 maps to frame 1, and page 3 is not mapped to any physical frame. The process may not use more than 3 physical frames. On a page fault, the demand paging system uses the LRU policy to evict a page. The MMU has a TLB cache that can store 2 entries. The TLB cache also uses the LRU policy to store the most recently used mappings in cache. Now, the process accesses the following logical addresses in order: 7, 17, 37, 20, 40, 60.

    (a) Out of the 6 memory accesses, how many result in a TLB miss? Clearly indicate the accesses that result in a miss. Assume that the TLB cache is empty before the accesses begin.
    **Ans:** 0,1,2, (miss) 1,2 (hit), 3 (miss)

    (b) Out of the 6 memory accesses, how many result in a page fault? Clearly indicate the accesses that result in a page fault.
    **Ans:** last access 3 result in a page fault

    (c) Upon accessing the logical address 60, which physical address is eventually accessed by the system (after servicing any page faults that may arise)? Show suitable calculations.
    **Ans:** 60 = 0011 1100 = page 3. 3 causes page fault, replaces LRU page 0, and mapped to frame 2. So physical address = 0010 1100 = 44

33. Consider a 64-bit system running an OS that uses hierarchical page tables to manage virtual memory. Assume that logical and physical pages are of size 4KB and each page table entry is 4 bytes in size.

    (a) What is the maximum number of levels in the page table of a process, including both the outermost page directory and the innermost page tables?

    (b) Indicate which bits of the virtual address are used to index into each of the levels of the page table.

    (c) Calculate the maximum number of pages that may be required to store all the page table entries of a process across all levels of the page table.

**Ans**

(a) ceil (64 - 12)/(12 - 2) = 6

(b) 2, 10, 10, 10, 10, 10 (starting from most significant to least)

(c) Innermost level has $2^{52}$ PTEs, which fit in $2^{42}$ pages. The next level has $2^{42}$ PTEs which require $2^{32}$ pages, and so on. Total pages = $2^{42} + 2^{32} + 2^{22} + 2^{12} + 2^2 + 1$

34. The page size in a system (running a Linux-like operating sytem on x86 hardware) is increased while keeping everything else (including the total size of main memory) the same. For each of the following metrics below, indicate whether the metric is *generally* expected to increase, decrease, or not change as a result of this increase in page size.

    (a) Size of the page table of a process

    (b) TLB hit rate

    (c) Internal fragmentation of main memory

    **Ans:** (a) PT size decreases (fewer entries) (b) TLB hit rate increases (more coverage) (c) Internal fragmentation increases (more space wasted in a page)

35. Consider a process with 4 logical pages, numbered 0–3. The page table of the process consists of the following logical page number to physical frame number mappings: (0, 11), (1, 35), (2, 3), (3, 1). The process runs on a system with 16 bit virtual addresses and a page size of 256 bytes. You are given that this process accesses virtual address 770. Answer the following questions, showing suitable calculations.

    (a) Which logical page number does this virtual address correspond to?

    (b) Which physical address does this virtual address translate to?

    **Ans:** (a) 770 = 512 + 256 + 2 = 00000011 00000010 = page 3, offset 2

    (b) page 3 maps to frame 1. physical address = 0000001 00000010 = 256 + 2 = 258

36. Consider a system with 16 bit virtual addresses, 256 byte pages, and 4 byte page table entries. The OS builds a multi-level page table for each process. Calculate the maximum number of pages required to store all levels of the page table of a process in this system.

    **Ans:** Number of PTE per process = $2^{16}/2^8 = 2^8$. Number of PTE per page = $2^8/2^2 = 2^6$. Number of inner page table pages = $2^8/2^6$ = 4, which requires one outer page directory. So total pages = 4+1 = 5.

37. Consider a process with 4 physical pages numbered 0–3. The process accesses pages in the following sequence: 0, 1, 0, 2, 3, 3, 0, 2. Assume that the RAM can hold only 3 out of these 4 pages, is initially empty, and there is no other process executing on the system.

    (a) Assuming the demand paging system is using an LRU replacement policy, how many page faults do the 8 page accesses above generate? Indicate the accesses which cause the faults.

    (b) What is the minimum number of page faults that would be generated by an optimal page replacement policy? Indicate the accesses which cause the faults.

    **Ans:**

    (a) 0 (M), 1 (M), 0(H), 2 (M), 3 (M), 3(H), 0(H), 2(H) = 4 misses

    (b) Same as above

38. Consider a Linux-like operating system running on a 48-bit CPU hardware. The OS uses hierarchical paging, with 8 KB pages and 4 byte page table entries.

   (a) What is the maximum number of levels in the page table of a process, including both the outermost page directory and the innermost page tables?

   **Ans:** ceil (48 - 13)/(13 - 2) = 4

   (b) Indicate which bits of the virtual address are used to index into each of the levels of the page table.

   **Ans:** 2, 11, 11, 11

   (c) Calculate the maximum number of pages that may be required to store all the page table entries of a process across all levels of the page table.

   **Ans:** Innermost level has $2^{35}$ PTEs. Each page can accommodate $2^{11}$ PTEs. Total pages = $2^{24} + 2^{13} + 2^2 + 1$

39. Consider the scenario described in the previous question. You are told that the OS uses demand paging. That is, the OS allocates a physical frame and a corresponding PTE in the page table only when the memory location is accessed for the first time by a process. Further, the pages at all levels of the hierarchical page table are also allocated on demand, i.e., when there is at least one valid PTE within that page. A process in this system has accessed memory locations in 4K unique pages so far. You may assume that none of these 4K pages has been swapped out yet. You are required to compute the minimum and maximum possible sizes of the page table of this process after all accesses have completed.

   (a) What is the minimum possible size (in pages) of the page table of this process?

   **Ans:** Each page holds $2^{11}$ PTEs. So $2^{12}$ pages can be accommodated in 2 pages at the inner most level. Minimum pages in each of the outer levels is 1. So minimum size = 2 + 1 + 1 + 1 = 5.

   (b) What is the maximum possible size (in pages) of the page table of this process?

   **Ans:** The $2^{12}$ pages/PTEs could have been widely spread apart and in distinct pages at all levels of the page table. So maximum size = $2^{12} + 2^{12} + 2^2 + 1$.

40. In a demand paging system, it is intuitively expected that increasing the number of physical frames will naturally lead to a reduction in the rate of page faults. However, this intuition does not hold for some page replacement policies. A replacement policy is said to suffer from *Belady's anomaly* if increasing the number of physical frames in the system can sometimes lead to an increase in the number of page faults. Consider two page replacement policies studied in class: FIFO and LRU. For each of these two policies, you must state if the policy can suffer from Belady's anomaly (yes/no). Further, if you answer yes, you must provide an example of the occurrence of the anomaly, where increasing the number of physical frames actually leads to an increase in the number of page faults. If you answer no, you must provide an explanation of why you think the anomaly can never occur with this policy.

   *Hint: you may consider the following example. A process has 5 logical pages, and accesses them in this order: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. You may find this scenario useful in finding an example of Belady's anomaly. Of course, you may use any other example as well.*

(a) FIFO

**Ans:** Yes. For string above, 9 faults with 3 frames and 10 faults with 4 frames.

(b) LRU

**Ans:** No. The N most recently used frames are always a subset of N+1 most recently used frames. So if a page fault occurs with N+1 frames, it must have occurred with N frames also. So page faults with N+1 frames can never be higher.

41. Consider a system with 32-bit addresses, 4KB pages, and 4-byte page table entries, using a two-level page table. A process in this system has the following layout in its virtual address space. The first 8MB of the virtual address space contains compile-time code and data. The next 1MB is the heap. Next, there is an unused (invalid) gap of 30MB, followed by a user stack of 1MB. Starting at virtual address 3GB, the next 20MB of the virtual address space is mapped to kernel code and data. The rest of the virtual address space is unused. The inner page table pages of the process are allocated on demand, that is, when they contain at least one valid page table entry. The outer page table is always allocated.

    (a) How many inner page table pages are required to store the page table mappings for the 8MB of compile-time code and data of the process?

    **Ans:** 8MB / 4KB = 2K PTEs. Each inner page table page can store 4KB/4 = 1K entries. So we need 2 pages.

    (b) How many inner page table pages are required to store all the page table entries corresponding to userspace code and data (compile-time code and data, stack, heap) of the process?

    **Ans:** 2 pages for the compile time code + 1 page for heap (partly filled) + 1 page for stack (partly filled) = 4 pages. There are 6 pages with only invalid entries between heap and stack that are not allocated.

    (c) Consider the inner page table page containing the page table entries for the user stack. How many page table entries in this page are invalid?

    **Ans:** The inner page table page corresponding to the user stack has invalid page table entries for 3MB of unused address space, followed by valid entries for 1MB of the stack. So 3/4th of the 1024 page table entries in this page are invalid, which is 768.

    (d) How many inner page table pages are required to hold the page table mappings for the kernel code and data?

    **Ans:** Each inner page table page holds 1024 mappings, covering 4MB of address space. So the kernel mappings will require 5 pages.

    (e) How many valid entries does the outermost level of the page table have?

    **Ans:** 4 valid entries for userspace + 5 for kernel space = 9

42. Consider a toy system with 64-byte pages and 12-bit virtual addresses. The first 10 page table entries of a process in this system are shown below on the left. For each page table entry, you are given a frame number (in binary, where valid), the valid bit and the present bit, in that order. For each of the virtual addresses (in decimal) shown below on the right, state whether the virtual address can be translated by the MMU or leads to a trap to the OS. If the address can be translated, write down the physical address after translation. If it leads to a trap, explain how the OS is likely to handle the trap. You can write your translated physical address in binary format.

| | | |
|---|---|---|
| 101100 | 1 | 1 |
| 100101 | 1 | 1 |
| 000000 | 1 | 0 |
| 000000 | 0 | 0 |
| 001100 | 1 | 1 |
| 101110 | 1 | 1 |
| 011001 | 1 | 1 |
| 000000 | 0 | 0 |
| 000000 | 1 | 0 |
| 000001 | 1 | 1 |

(a) 56

**Ans:** page number = 0, offset = 111000, PA = 101100 111000

(b) 156

**Ans:** page number = 10, offset = 011100. Valid but not present, leads to a page fault. OS allocates a new physical frame and restarts the process.

(c) 256

**Ans:** page number = 100, offset = 000000, PA = 001100 000000

(d) 356

**Ans:** page number = 101, offset = 100100, PA = 101110 100100

(e) 456

**Ans:** page number = 111, offset = 001000, Invalid page, leads to page fault. OS terminates process.

43. Consider a process with an array of 4K (that is, 4096) integers, each of 4 bytes. The array starts at a page boundary, and the page size in the system is 4KB. All the pages of the array have been allocated physical memory by the OS (and will remain in memory throughout this question), but none of the page table entries are initially in the TLB. The TLB uses the LRU policy when it needs to evict entries.

(a) The program now accesses each element of the array once. What is the TLB miss rate (fraction of accesses that cause a TLB miss), as computed over these 4K accesses? Assume that the TLB does not evict any newly added entries during this period.

**Ans:** The array spans 4 pages, each with 1K elements. The first element accessed in a page will lead to a TLB miss, and the rest within the page will be TLB hits. So TLB miss rate = 1/1024

(b) In the previous question, how many new TLB entries are added for the translation of the virtual addresses corresponding to the array accesses?

**Ans:** The array spans 4 pages, so 4 new TLB entries will have come into the TLB.

(c) Suppose the process accesses the array repeatedly, going over all 4K elements over and over again from beginning to end. Assume that the TLB has enough space, and does not evict any entries corresponding to this process. What is the TLB miss rate for accessing this array in steady state, computed across repeated array accesses over a long period of time?

**Ans:** 0. If no entries are evicted, then all 4 entries will be in TLB, so every access will lead to a TLB hit.

(d) Suppose the TLB can only accommodate one entry corresponding to the pages in the array. Repeat the previous question now with this constraint. What is the TLB miss rate for the array accesses in steady state?

**Ans:** There will be a TLB miss Same as part (a). One miss per 1024 elements in a page. So 1/1024

(e) Suppose the TLB now can accommodate 3 entries for the pages in the array. When this limit of entries for the array is exceeded, one of the existing array entries is evicted based on LRU policy. Repeat question (c) in this scenario. What is the TLB miss rate for the array accesses in steady state?

**Ans:** The TLB can accommodate only 3 entries, but the array spans 4 pages. In this scenario also, every new page access leads to a TLB miss, because every page access evicts the TLB entry of the next page access by LRU policy. So the miss rate is once again 1/1024

44. Consider a simple process with 5 pages, numbered 0 to 4. Shown below are the accesses made to these pages during the execution of the process. The times at which each page was accessed (in milliseconds, relative to some starting point t = 0) is shown in parentheses next to the page number. None of the pages was in memory at time t = 0, and the page is brought into memory at its first access.

1 (1 ms), 0 (4 ms), 0 (5 ms), 0 (6 ms), 3 (8 ms),
4 (12 ms), 1 (13 ms), 2 (17 ms),
1 (23 ms), 4 (25 ms), 3 (29 ms)

Now, the OS needs to pick one of these five frames to evict at time t = 30 ms. For each of the page replacement policies shown below, identify the page that will be picked by the OS as the victim page to be evicted. You may assume that every page table entry has an "accessed" bit which is set by the MMU on every page access shown above. The OS periodically inspects and resets these bits in the page table entries.

(a) Ideal LRU policy

**Ans:** Page 0

(b) An approximate LRU policy where the OS resets and inspects the accessed bits every 10 milliseconds (at time 0, 10 ms, 20 ms, ...), and uses the knowledge of accessed bits set in the past 10 ms to pick a victim page. If the OS has a choice of multiple pages, list all of them.

**Ans:** Pages 0 and 2. Both of these show up as unaccessed in the last 10 ms.

(c) Repeat the previous question for a modified approximate LRU policy, where the OS uses information of the accessed bits set over the preceding 20 ms.

**Ans:** Page 0, since OS can see that it was not accessed in past 20 ms.

(d) First In First Out (FIFO) policy

**Ans:** Page 1 (since it was fetched first into memory)

(e) OS uses information of the accessed bits over the past 30 milliseconds (which are inspected and reset every 10 milliseconds as described above), to pick the least frequently used (LFU) page. If multiple pages are tied with LFU, the LRU policy is applied after LFU to break ties.

**Ans:** Both page 0 and page 2 show up as having one access, and LRU is used to pick page 0. Note that even though page 0 is accessed multiple times, it will show up as only one access when the accessed bit is inspected.

45. Consider an OS that implements an approximate LRU policy in the following manner. Every page table entry has an "accessed" bit which is set by the MMU when the page table entry is accessed. Every 10 milliseconds, the OS inspects these bits for all pages, archives this information, and resets these bits in the page table entries, so that they may be set afresh in the next round. The OS uses the history of the accessed bit for each page over the past 5 rounds (i.e., over the past 50 milliseconds) to pick the least recently used page. If the OS has a choice of multiple pages that appear as LRU by this method, ties are broken arbitrarily.

(a) Now, when the OS needs a free page and is looking for a victim page to evict, it finds there is only one page which appear as LRU by the above algorithm. Is this page guaranteed to be the true LRU page according to an ideal "exact" (not approximate) LRU algorithm that has visibility into all page accesses? Answer yes/no and provide a justification.

**Ans:** Yes, if only one page is identified as LRU by this algorithm, then this has to be the true LRU page. There is no way the true LRU page will have accessed bit set in a later round as compared to this page identified by the approximate LRU algorithm.

(b) Now, suppose we modify the page replacement policy to make it a Least Frequently Used (LFU) policy. The OS uses information of the accessed bits over the past 50 milliseconds (which are inspected and reset every 10 milliseconds as described above) to count the number of times the accessed bit is set for each page across the past 5 rounds. This information is then used to pick the least frequently used page. At some point, when trying to find a victim page, the algorithm finds that one page has its accessed bit set in only one of the rounds (while all other pages have their accessed bits set in 2 or more rounds) and hence appears as LFU by the algorithm above. Is this page guaranteed to be the true LFU page according to an ideal "exact" (not approximate) LFU algorithm that has visibility into all page accesses? Answer yes/no and provide a justification.

**Ans:** No. Multiple accesses within a 10 millisecond interval show up as just one access when we inspect the accessed bit. So a page that has accessed bit set in only one round (but was accessed, say, 10 times in that interval) may get picked as LFU, while the true LFU page could have been accessed twice in two different epochs, and missed being identified as LFU by this algorithm.

46. Consider the following pseudocode for the implementation of the `fork` system call in a simple OS, where the OS replicates the user part of the virtual address space of the parent in the child. Note that this is not the optimized copy-on-write variant. In the code below, the variable `ppt` is the page table of the parent process invoking fork, the variable `newpt` is the new page table allocated for the child, and `size` denotes the size (in bytes) of the user part of the virtual address space of the parent that is copied to the child. The constant `PGSIZE` denotes the page size (in bytes). Other variables in the code are self-explanatory.

The code uses the following functions. `lookup_pte(pt, v)` returns page table entry corresponding to a virtual address `v` in the page table `pt`. The functions `get_pa` and `get_flags` return the physical address (starting physical address, in bytes) and flags (permission bits etc.) of a page table entry respectively. The function `alloc_page` allocates a new physical frame and returns its starting physical address. The function `memcpy(src, dst, num)` copies `num` bytes from physical address `src` to physical address `dst`. The function `add_pte(pt, va, pa, flg)` adds a new page table entry in the page table `pt`, mapping one page of virtual addresses starting from `va` to the corresponding physical addresses starting from `pa`, with flags set to `flg`.

Using the explanation provided above, fill in the blanks in the code below. Write your answers in the space provided below the code.

```
for(int addr = 0; addr < size; addr += PGSIZE) {
  pte = lookup_pte(ppt, _(a)_)
  ppa = get_pa(pte)
  flags = get_flags(pte)
  newpa = alloc_page()
  memcpy(_(b)_, _(c)_, PGSIZE)
  add_pte(newpt, _(d)_, _(e)_, flags)
}
```

(a) **Ans:** `addr`

(b) **Ans:** `ppa`

(c) **Ans:** `newpa`

(d) **Ans:** `addr`

(e) **Ans:** `newpa`

47. Continuing on the previous question, you must now complete the pseudocode of the `sbrk` system call shown below, which expands the virtual address space of a process from `oldsz` bytes to `newsz` bytes by adding one or more pages between these virtual addresses, with the flags (permission bits) set to `flags`. The OS also allocates physical memory corresponding to these new virtual pages. The page table of the process is in the variable `pt`. You may assume that `oldsz` and `newsz` are both multiples of `PGSIZE` and `newsz` > `oldsz`.

```
for(int addr = oldsz; addr < newsz; addr += PGSIZE) {
  newpa = _(a)_
  _(b)_ ( _(c)_, _(d)_, _(e)_, flags)
}
```

Note that blanks (a) and (b) are names of functions while (c), (d), and (e) are variables / arguments. You must use the function names from the previous question.

(a) **Ans:** `alloc_page()`

(b) **Ans:** `add_pte`

(c) **Ans:** `pt`

(d) **Ans:** `addr`

(e) **Ans:** `newpa`

48. Consider the following pseudocode of the `mmap` system call to map an anonymous page into the address space of a process in a simple OS. Fill in the **five** blanks below. An explanation of the functions and variables used is provided after the code.

```
mmap(start, size, flags):
  offset = 0
  while(____________) { //fill
    pa = ______________ //fill
    add_pte( pt , ________ , _________ , flags) //fill 2
    offset += PGSIZE
  }
  return __________ //fill
```

The arguments to the function are as follows. `start` denotes the starting virtual address where the memory-mapped area starts. Assume that this virtual address space is free to be assigned. `size` denotes the size in bytes of the memory mapped area requested by the process, and `flags` is the permission flags requested on the allocated memory. Assume that this implementation of `mmap` allocates physical memory along with the virtual address space. The system call returns the starting virtual address of the newly mapped memory. You may assume that `size` is a multiple of the page size (denoted by the constant `PGSIZE` in the code).

The other variables in the code are as follows. `offset` is a local variable to track how much of the memory allocation is complete. `pa` is a local variable to hold the physical address of a page. `pt` is the page table of the process making the system call.

The functions invoked in the code are as follows. The function `alloc_page()` allocates a new physical frame and returns its starting physical address. The function `add_pte(pt, va, pa, flg)` adds a new page table entry in the page table `pt`, mapping one page of virtual addresses starting from virtual address `va` to the corresponding physical frame starting from physical address `pa`, with flags set to `flg`.

**Ans:** `offset < size`
`alloc_page()`
`start+offset`
`pa`
`start`

49. Consider the following (jumbled) set of events that occur during a memory access by the CPU.

**E1:** MMU traps to the OS

**E2:** Main memory is accessed with the translated physical address

**E3:** MMU walks the page table of a process to translate a virtual address

**E4:** MMU looks up the TLB to translate a virtual address

**E5:** OS services page fault and updates the page table mappings of the process

For each of the scenarios given below, write the events from the list above that occur in the scenario, in the chronological order in which they occur (earliest to latest).

(a) TLB hit during address translation
**Ans:** E4, E2

(b) TLB miss but no page fault during address translation
**Ans:** E4, E3, E2

(c) Page fault during address translation
**Ans:** E4 E3 E1 E5 E2

50. Consider a process P that forks a child process C in a simple OS. After fork completes, the parent and child processes separately map a shared memory segment of one page into their virtual address space. This shared memory page starts at virtual address $S_P$ in process P, and at $S_C$ in C. The kernel code and data are mapped starting at virtual address $K$ in both processes. Let $A_Q(V)$ denote the physical address corresponding to virtual address $V$ when computed using the page table of process Q. State whether the following statements are true or false.

(a) We will always have $S_P = S_C$. **Ans:** False

(b) We will always have $A_P(S_P + X) = A_C(S_C + X)$ for every offset $X$ in the shared memory page. **Ans:** True

(c) We will always have $A_P(V) = A_C(V)$ for all valid virtual addresses V. **Ans:** False

(d) We will always have $A_P(V) = A_C(V)$ for all valid virtual addresses V in the user part of the address space. **Ans:** False

(e) We will always have $A_P(K + X) = A_C(K + X)$ for every valid offset $X$ in the kernel part of the address space. **Ans:** True

(f) We would have had $S_P = S_C$ if the shared memory segment was mapped in the parent process before the fork system call was executed. **Ans:** True