Lecture Notes on Operating Systems

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. Consider a program that memory maps a large file, and accesses bytes in the first page of the file. Now, a student runs this program on several machines running different versions of Linux, and finds that the actual physical memory consumed by the process (RSS or resident set size) varies from OS to OS. Provide one reason to explain this observation.

   Ans: Different operating systems may have different heuristics to allocate physical memory to a process. For example, systems may differ in the heuristics for prepaging, i.e., how to prefetch pages that may be accessed in the future. This leads to different values of RSS.

3. 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

4. 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

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

   Ans: T

6. 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

7. 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

8. Consider a process with 9 logical pages, out of which 3 pages are mapped to physical frames. The process accesses one of its 9 pages randomly. What is the probability that the access results in a TLB hit and a subsequent page fault?

   Ans: 0 (TLB hit implies a physical page has been mapped, so a page fault cannot occur)

9. 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

10. 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.

11. 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

12. 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

13. 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

14. 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

15. 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.

16. 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 $2^{20} \times 4 = 4$MB.

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.

17. 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.

18. 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 $t_h = \frac{t_m - t_x}{t_m - t_h}$

19. 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

20. 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

21. 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 $l = \text{ceil } \frac{v-k}{k-e}$

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

22. 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 $2^{22} + 2^{10} + 1$.

23. 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.

24. 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.

25. 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 results 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

26. 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 = $2^{42} + 2^{32} + 2^{22} + 2^{12} + 2^{2} + 1$

27. 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)

28. 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

29. 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.

30. 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

31. 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 = $2^{24} + 2^{13} + 2^2 + 1$

32. 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 = $2^{12} + 2^{12} + 2^2 + 1$.

33. 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.