Lecture Notes on Operating Systems

Problem Set: 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 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.

10. 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.

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

12. 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

13. 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

14. 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) $v - k - (l-1) * (k-e)$

15. 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 $2^{22} + 2^{10} + 1$.