CS347: Operating Systems

Problem Set 3

1. 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 processes to run concurrently and uses a 10-bit number to represent the PID. (Assume 1K = 1024.)

   (a) 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.

   (b) Now assume the OS uses an inverted page table instead of hierarcical page tables. Every inverted page table entry needs to store the process identifier along with the page number. Assume that inverted page table entries are rounded up to the nearest byte. Calculate the memory required to store the inverted page table in the system.

2. 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.

3. Consider a system running the xv6 OS. The shell process P asks the user for a command, and forks a child process C. C then runs `exec` to execute the `ls` command typed by the user. Assume that process C has just returned from the `exec` system call into user space, and is ready to execute

the first instruction of the `ls` binary. Now, the memory belonging to process C would have been allocated and modified at various points during the execution of the `fork` and `exec` system calls. For each piece of memory that belongs to process C that is listed below, you must answer *when* the memory allocation/initialization described in the question occurred. Your possible choices for the answers are given below, and you must pick one of the choices.

- **Fork+allocproc**: In the `allocproc` function called from `fork` by the parent P.
- **Fork+...**: In a function (you must provide the name) called from `fork` by the parent P.
- **Fork**: In the `fork` system call execution by parent P.
- **Exec+allocuvm**: In the `allocuvm` function called by `exec` in child C.
- **Exec+loaduvm**: In the `loaduvm` function called by `exec` in child C.
- **Exec+walkpgdir**: In the `walkpgdir` function called by `exec` in child C.
- **Exec+...**: In a function (you must provide the name) called from `exec` in child C.
- **Exec**: In the `exec` system call execution by C.
- **Other**: Write down any other answer that you feel is correct but is not listed above.

Note that when a more specific choice is the correct answer, less specific choices will only get partial credit (e.g., if the correct answer is "fork+allocproc", the answer "fork" will only get partial credit).

(a) When was the the `struct proc` object of C assigned to C from an unused state?

(b) When was the memory page that holds C's kernel stack assigned to process C from the list of free pages?

(c) When were the memory pages that hold C's *current* page table entries allocated for this purpose from the list of free pages?

(d) When were the memory pages that hold the code of the `ls` executable allocated to C from the list of free pages?

(e) When were the memory pages that hold the code of C's `ls` executable populated with the `ls` binary content from the disk?

4. Consider a system with V bytes of virtual address space available per process, running an xv6-like OS. Much like with xv6, low virtual addresses, up to virtual address U, hold user data. The kernel is mapped into the high virtual address space of every process, starting at address U and upto the maximum V. The system has P bytes of physical memory that must all be usable. The first K bytes of the physical memory holds the kernel code/data, and the rest $P - K$ bytes are free pages. The free pages are mapped once into the kernel address space, and once into the user part of the address space of the process they are assigned to. Like in xv6, the kernel maintains page table mappings for the free pages even after they have been assigned to user processes. The OS does not use demand paging, or any form of page sharing between user space processes. The system must be allowed to run up to N processes concurrently.

(a) Assume $N = 1$. Assume that the values of V, U, and K are known for a system. What values of P (in terms of V, U, K) will ensure that all the physical memory is usable?

(b) Assume the values of V, K, and N are known for a system, but the value of P is not known apriori. Suggest how you would pick a suitable value (or range of values) for U. That is, explain how the system designer must split the virtual address space into user and kernel parts.

5. Consider a system running the xv6 OS. A parent process P has forked a child C, after which C executes the `exec` system call to load a different binary onto its memory image. During the execution of `exec`, does the kernel stack of C get reinitialized or reallocated (much like the page tables of C)? If it does, explain what part of `exec` performs the reinitialization. If not, explain why not.

6. The xv6 operating system does not implement copy-on-write during fork. That is, the parent's user memory pages are all cloned for the child right at the beginning of the child's creation. If xv6 were to implement copy-on-write, briefly explain how you would implement it, and what changes need to be made to the xv6 kernel. Your answer should not just describe what copy-on-write is (do not say things like "copy memory only when parent or child modify it"), but instead concretely explain *how* you would ensure that a memory page is copied only when the parent/child wishes to modify it.

7. Consider a process P in xv6 that has executed the `kill` system call to terminate a victim process V. If you recall the implementation of `kill` in xv6, you will see that V is not terminated immediately, nor is its memory reclaimed during the execution of the `kill` system call itself.

   (a) Give one reason why V's memory is not reclaimed during the execution of `kill` by P.

   (b) Describe when V is actually terminated by the kernel.

8. 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 $t_m > t_h$.

9. Consider the implementation of the `exec` system call in xv6. The implementation of the system call first allocates a new set of page tables to point to the new memory image, and switches page tables only towards the end of the system call. Explain why the implementation keeps the old page tables intact until the end of `exec`, and not rewrite the old page tables directly while building the new memory image.

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

11. 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.

12. Consider the list of free pages populated by the xv6 kernel during bootup, as part of the functions `kinit1` and `kinit2`. Which of the following is/are potentially stored in these free pages subsequently, during the running of the system?

Answer _____

A. Page table mappings of kernel memory pages.

B. Page table mappings of user memory pages.

C. The kernel bootloader.

D. User executables and data.

13. Consider the logical addresses assigned to various parts of code in the kernel executable of xv6. Which of the following statements is/are true regarding the values of the logical addresses?

Answer _____

A. All are low addresses starting at 0.

B. All are high address starting at a point after user code in the virtual address space.

C. Some parts of the kernel code run at low addresses while the rest use high addresses.

D. The answer is dependent on the number of CPU cores.