Lecture Notes on Operating Systems

Practice Problems: Memory Management in xv6

1. 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?

Ans:

(a) fork+allocproc

(b) fork+allocproc

(c) exec+walkpgdir

(d) exec+allocvm

(e) exec+loaduvm

2. 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.

   Ans:

   (a) The kernel part of the virtual address space of a process should be large enough to map all physical memory, so $V - U \geq P$. Further, the user part of the virtual address space of a process should fit within the free physical memory that is left after placing the kernel code, so $U \leq P - K$. Putting these two equations together will get you a range for P.

   (b) If there are N processes, the second equation above should be modified so that the combined user part of the N processes can fit into the free physical pages. So we will have $N * U \leq P - K$. We also have $P \leq V - U$ as before. Eliminating P (unknown), we get $U \leq \frac{V-K}{N+1}$.

3. 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.

Ans: The kernel stack cannot be reallocated during `exec`, because the kernel code is executing on the kernel stack itself, and releasing the stack on which the kernel is running would be disastrous. Small changes are made to the trap frame however, to point to the start of the new executable.

4. 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.

   Ans: The memory pages shared by parent and child would be marked read-only in the page table. Any attempt to write to the memory by the parent or child would trap the OS, at which point a copy of the page can be made.

5. 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.

   Ans: Memory cannot be reclaimed during the kill itself, because the victim process may actually be executing on another core. Processes are periodically checked for whether they have been killed (say, when they enter/exit kernel mode), and termination and memory reclamation happens at a time that is convenient to the kernel.

6. 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.

   Ans: The `exec` system call retains the old page tables, so that it can switch back to the old image and print an error message if exec does not succeed. If exec succeeds however, the old memory image will no longer be needed, hence the old page tables are switched and freed.

7. 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?

   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.

Ans: ABD

8. 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?

   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.

   Ans: C

9. In a system running xv6, for every memory access by the CPU, the function `walkpgdir` is invoked to translate the logical address to physical address. [T/F]

   Ans: F

10. Consider a process in xv6 that makes the `exec` system call. The EIP of the `exec` instruction is saved on the kernel stack of the process as part of handling the system call. When and under what conditions is this EIP restored from the stack causing the process to execute the statement after `exec`?

    Ans: If some error occurs during exec, the process uses the eip on trap frame to return to instruction after exec in the old memory image.

11. Why does the function kinit1 called by the xv6 main() map only free pages within the first 4MB of RAM? Why does it not collect free pages from the entire available RAM?

    Ans: Because it needs some free pages for page tables before it can map the entire memory.

12. Why does the page table created by the entry code (entrypgdir) add a mapping from virtual addresses [0,4MB] to physical addresses [0,4MB]?

    Ans: So that the instructions between turning on MMU and jumping to high address space run correctly.

13. Suppose xv6 is running on a machine with 16GB virtual address space, out of which 0-8GB are reserved for user code, and the remaining addresses are available to the kernel. What is the maximum amount of RAM that xv6 can successfully manage and address in this system? Assume that you can change the values of KERNBASE, PHYSTOP and other such constants in the code suitably.

    Ans: 8GB

14. Consider a process in an xv6 system. Consider the following statement: "All virtual memory addresses starting from 0 to $N-1$ bytes, where N is the process size (`proc->sz`), can be read by the process in user mode." Is the above statement true or false? If true, explain why. If false, provide a counter example.

    Ans: False, the guard page is not accessible by user.

15. When an xv6 process invokes the `exec` system call, where are the arguments to the system call first copied to, before the system call execution begins? Tick one: user heap / user stack / trap frame / context structure

    Ans: user stack

16. When a process successfully returns from the `exec` system call to its new memory image in xv6, where are the commandline arguments given to the new executable to be found? Tick one: user heap / user stack / trap frame / context structure

    Ans: user stack

17. After the `exec` system call completes successfully in xv6, where is the EIP of the starting instruction of the new executable stored, to enable the process to start execution at the entry of the new code? Tick one: user heap / user stack / trap frame / context structure

    Ans: trap frame