

Practice Problems: Memory Management in xv6

1. 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 a priori. 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 $U \leq \frac{V-K}{N+1}$.
2. 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.

3. 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.

4. Consider a process P in xv6 that has executed the `kill` system call to terminate a victim process V. If you read 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.

5. 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.

6. 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

7. 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.

8. 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.

9. 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

10. 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

11. 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

12. Consider the list of free memory frames maintained by xv6 in the free list. Whenever a process in kernel mode requests memory via the function `kalloc()`, xv6 extracts and returns free frames from this list. Which of the following things can be stored in the pages thus allocated from the free list?

- (a) New page table created for child process during fork
- (b) New memory image (code/data/stack/heap) created for child during fork
- (c) Kernel stack of new process created in fork
- (d) New page table created for the new memory image in `exec`

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

13. Consider a process P in xv6 that executes the `sbrk` system call to increase the size of the user part of its virtual address space from N1 bytes to N2 bytes. Assume N1 and N2 are multiples of page size, $N2 \geq N1$, and the difference between N1 and N2 is K pages. Which of the following statements is/are true about the actions that occur during the execution of the system call?

- (a) The OS assigns K free physical frames to the process and adds the frame numbers into the page table.
- (b) The OS does not assign any new physical frames to the process, but updates the page table.
- (c) The OS updates (N2-N1) page table entries in the page table of P.
- (d) The OS updates K page table entries in the page table of P.

Ans: (a), (d)

14. Consider a process P running in xv6. The high virtual addresses in the address space of P are assigned to OS code/data. Consider a virtual address V assigned to OS code/data. Which of the following statements is/are true?

- (a) If the CPU accesses address V in user mode, the MMU raises a trap to the OS.
- (b) The address V is translated to the same physical address by the page tables of all processes in the system.

- (c) The address V can be translated to different physical address by the page tables of different processes in the system.
- (d) The page table entry that translates address V to a physical address is present in the page table of P only when it is running in kernel mode.

Ans: (a), (b)

15. Consider a process running in xv6. The page table of the process maps virtual address V to physical address P . Which of the following statements is/are true? Assume $KERNBASE$ is set to 2GB in xv6.
- (a) $V = P + KERNBASE$ always
 - (b) $V = P - KERNBASE$ always
 - (c) $V = P + KERNBASE$ only for $V \geq KERNBASE$
 - (d) $V = P + KERNBASE$ only for $V < KERNBASE$

Ans: (c)

16. Consider a system running 32-bit xv6 with 4KB pages. Assume that the operating system image is loaded into the first 4MB of physical memory, and is mapped into the virtual address space of every process starting at high virtual address 2GB ($KERNBASE$). The total physical memory available to xv6 is 32MB ($PHYSTOP$). Two processes $P1$ and $P2$ have been created in this system, each having 3 pages (for the code, guard page, and stack, in that order) in the user part of their address space.

- (a) How many free frames are present in the free list of the OS initially, before any processes are created in the system? You may assume that all free physical memory that the OS can address is part of the free list initially.

Ans: 7168 (32MB = 8192 pages total, out of which 1024 are for OS, which means 7168 frames are part of the free list)

- (b) How many valid/present pages does the virtual address space of a process ($P1$ or $P2$) have, across the user and kernel parts of the address space?

Ans: 3 (user) + 8192 (all of 0 to phystop is mapped in kernel space) = 8195

- (c) Consider a physical frame at address 2MB, which holds some part of the OS image. At which virtual address is this frame mapped in the page tables of $P1$ and $P2$? List all mappings to this frame.

Ans: It is mapped at address $2GB + 2MB$ in the page tables of $P1$ and $P2$

- (d) Consider a physical frame at address $4MB + 8KB$ (i.e., the 3rd frame after the OS image), which is allocated to hold the stack page (i.e., the 3rd page in the user part of the memory image) of process $P1$. At which virtual address is this frame mapped in the page tables of $P1$ and $P2$? List all mappings to this frame.

Ans: It mapped at virtual address 8KB in $P1$, and also at $2GB+4MB+8KB$ in the page tables of $P1$ and $P2$

- (e) Consider the page table of $P1$. Recall that the inner page table pages are allocated on demand, i.e., only when there is at least one valid entry to store. How many pages does the OS allocate to build the page table of $P1$, across both the inner and outer levels?

Ans: 8 pages for kernel mappings (each holds 1024 PTE, so maps 4MB of address space) + 1 for user + 1 outer pgdir = 10 pages in total