Memory management xv6

Before you begin

- We have modified some xv6 files for this lab, and these patched files are provided as part of this lab's code, which is a `Makefile` and a `testcase.c`. Before you begin the lab, copy the patched files into the main xv6 code directory.
- For this lab, you will need to understand the following files: syscall.c, syscall.h, sysproc.c, user.h, usys.S, vm.c, proc.c, trap.c, defs.h, mmu.h.

IMPORTANT: Refer to the system calls which are already there to edit these files accordingly. For example, you can refer to, say, `kill` system call. See how it is defined in all the above-mentioned files, how it is made related to `proc.c` (where you will actually define the functionality of the system calls), and how it is also defined in `defs.h`

Remember, always read completely a particular file, and make the changes wherever required, to make the new system call work perfectly fine.

- `user.h` contains the system call definitions in xv6, which are exported to userspace programs.
- `usys.S` contains code to invoke the trap instruction for each system call.
- `syscall.h` contains a mapping from system call name to system call number. Every system call must have a number assigned here.
- `syscall.c` contains helper functions to parse system call arguments and pointers to the actual system call implementations.
- `sysproc.c` contains the implementations of process-related system calls.
- `proc.c` contains the `struct proc` structure. It contains implementations of various process-related system calls, and the scheduler function. This file also contains the definition of `ptable`, so any code that must traverse the process list in xv6 must be written here.
- `defs.h` is a header file with function definitions in the kernel. All the functions defined in a file must also be defined here in the section of that file name. For example, if you make a new function in `vm.c`, then you also have to define the function in `vm.c` section of `defs.h`.

    - `mmu.h` is also a header file with various useful definitions pertaining to memory management. You do not have to change anything in this file but read it carefully.
    - The file `vm.c` contains most of the logic for memory management in the xv6 kernel, and proc.c contains process-related system call implementations. The functions written here are used in `proc.c` indirectly with the help of `defs.h`.
    - The file `trap.c` contains trap handling code for all traps including page faults.
    - Understand the implementation of the `sbrk` system call that spans all of these files.
    - Understand the `growproc` system call in `proc.c`.
- We have also provided a simple test program `testcase.c` as part of the code for this lab. This test program is compiled by our modified `Makefile` and you can run it on the xv6 shell by typing `testcase` at the command prompt. If you wish to include any other test programs in xv6, remember that the test program should be included in the Makefile for it to be compiled and executed from the xv6 shell. Understand how the sample testcase was included within the Makefile, and use similar logic to include other test programs as well. Note that the xv6 OS itself does not have any text editor or compiler support, so you must write and compile the code in your host machine, and then run the executable in the xv6 QEMU emulator.

PART-A: Understanding the memory information

You will first implement the following new system calls in xv6.

- `numvp()` should return the number of virtual pages in the user part of the address space of the process, up to the program stored in `struct proc.`
- `numpp()` should return the number of physical pages in the user part of the address space of the process. You must count this number by walking the process page table, and counting the number of page table entries that have a valid physical address assigned.

HINT:

➔ You can walk the page table of the process by using the `walkpgdir` function which is present in `vm.c`. You can find several examples of how to invoke this function within `vm.c` itself. To compute the number of physical pages in a process, you can write a function that walks the page table of a process in `vm.c`, and invoke this function from the system call handling code. You can count a page if its present bit is set.

➔ For `numvp`(), you can find the program size in the PCB of the process and divide it by `PGSIZE`.

PART-B: Allocating memory on demand (memory map)

IMPORTANT: Memory-mapped page means the pages that have been allocated by the `mmap` system call.

- You will implement a simple version of the `mmap` system call (just like `sbrk` system call) in xv6. Your `mmap` system call should take one argument: the number of bytes to add to the address space of the process. The number of bytes must be a positive number and a multiple of page size (the page size in xv6 should be known to you).
- The system call should **return** a value of **0** if any invalid inputs are provided. If a valid number of bytes is provided, the system call should expand the virtual address space of the process by the specified number of bytes, and **return** the starting virtual address of the newly added memory region.
- The new virtual pages should be added at the end of the current program break (the current program break is the location just after the heap section of the process) and should increase the program size correspondingly. However, the system call should NOT allocate any physical memory corresponding to the new virtual pages, as we will allocate memory on demand. You can use the system calls of the previous part to print these page counts to verify your implementation.
- After the `mmap` system call, and before any access to the mapped-memory pages, you should only see the number of virtual pages of a process increase, but not the number of physical pages. Physical memory for a memory-mapped virtual page should be allocated on demand, only when the page is accessed by the user. When the user accesses a memory-mapped page, a page fault will occur, and physical memory should only be allocated as part of the page fault handling. Further, if you have added more than one page in the virtual address using `mmap` system call, then physical memory should only be allocated for those pages that are accessed, and not for all newly added pages. Once again, use the virtual/physical page counts to verify that physical pages are allocated only on demand.
- We have provided a simple test program to test your implementation. This program invokes `mmap` multiple times and accesses the memory-mapped pages. It prints out virtual and physical page counts periodically, to let you check whether the page counts are being updated correctly. You can write more such test cases to thoroughly test your implementation.

Hints to solve this part

- Understand the implementation of the `sbrk` system call. Your `mmap` system call will follow a similar logic. In `sbrk`, the virtual address space is increased and physical memory is allocated within the same system call. The implementation of `sbrk` invokes the `growproc` function, which in turn invokes the `allocuvm` function to allocate physical memory. For your `mmap` implementation, you must only grow the virtual address space within the system call implementation, and physical memory must be allocated during the page fault. You may invoke `allocuvm` (or write another similar function) in order to allocate physical memory during a page fault.
- The original version of xv6 does not handle the page fault trap. For this assignment, you must write extra code to handle the page fault trap in `trap.c`, which will allocate memory on demand for the page that has caused the page fault. You can check whether a trap is a page fault by checking if `tf->trapno` is equal to `T_PGFLT`. Look at the arguments to the `cprintf` statements in `trap.c` to figure out how one can find the virtual address that caused the page fault. Use `PGROUNDDOWN(va)` to round the faulting virtual address down to the start of a page boundary. Once you write code to handle the page fault, do return in order to avoid the processing of other traps.
- *Remember*: it is important to call `switchuvm` to update the CR3 register and TLB every time you change the page table of the process. This update to the page table will enable the process to resume execution when you handle the page fault correctly.

—----------END—----------