CS236 Spring 2024

Lab Quiz 3

4 questions, 25 marks

Instructions

Before you begin, please check that you can see the following files in the Desktop/cs236 directory:

- This question paper `labquiz3.pdf`
- The original unmodified xv6 tarball. You can untar this file to obtain the xv6 code folder:
  `tar -zxvf xv6-public.tgz`
- A tarball `labquiz3_code.tgz` You can untar this file to obtain the required helper code for all the questions:
  `tar -zxvf labquiz3_code.tgz`
  This folder contains the code for each question in a separate subdirectory, with names `q1`, `q2`, `q3`, `q4`.

Questions 1–3 are based on xv6. Start with the original unmodified xv6 code for each question. Copy the modified files given to you into this folder. You can then make, build and run the xv6 code as you would do normally. Please remember to start with a fresh unmodified xv6 folder for each question.

Questions 4 deals with dynamic memory allocation. You are given template code that you must extend as explained in the question.

To submit your code, create your submission folder titled `submission_rollnumber` in the Desktop directory, where you must replace the string `rollnumber` above with your own roll number (e.g., your directory name should look like `submission_12345678`). Create separate subdirectories for each question in this folder with names `q1`, `q2`, `q3`, `q4`. Place the files you wish to submit for each question in these subdirectories.

For the xv6 questions, you must submit all source code files that you have changed. You must keep track of the files you have changed, and carefully submit all of them. We cannot grade your submission if some files have not been submitted. The questions have hints on which files you will be expected to change for each question.

For the question on dynamic memory allocation, you must modify the template code given to you and submit it. Please do not submit any other files.

Submission Instructions

1. Once you are ready to submit, please run the command `check` from your terminal. This command will create a tarball of your submission folder. If the folder is not in the proper place or is empty, this script will throw an error. In that case, please fix the errors and try again.

2. Next, call one of the TAs and ask them to run the `submit` command from your terminal. The TA will enter the password to upload your submission tarball to our remote submission server. Please note that we will only be grading the files that are submitted in this manner. So please ensure that you submit the correct files.

3. **IMPORTANT NOTE: Please ensure that you are submitting the correct files with the correct filenames. Test your code properly before submission. Strictly NO code changes will be allowed after the exam.**

Help for xv6 code

Listed below are the important xv6 files you will need to understand for solving this quiz.

- `user.h` contains the system call definitions in xv6 that are available to user programs.
- `usys.S` contains a list of system calls exported by the kernel, and the corresponding invocation of the trap instruction.
- `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.
- `defs.h` is a header file with function definitions in the kernel. You can use this file to store function signatures, so that functions written in one file can be called from another file.
- `proc.h` contains the `struct proc` structure.
- `proc.c` 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.
- `vm.c` contains implementations of various functions related to memory management, and some variables related to memory management. So most memory management functions are implemented here.

Exercises

1. **[5 marks]** In this question, you will implement a system call `pt_count` with the signature `void pt_count(int* arr)` that counts the number of pages used by page tables of a process. The count is split into two sub-counts, namely `user_pt_count` and `kernel_pt_count`:

   (a) `user_pt_count` is a count of all page table pages that aid in the translation of userspace virtual addresses of a process. You must include the outer page directory page also in this count.

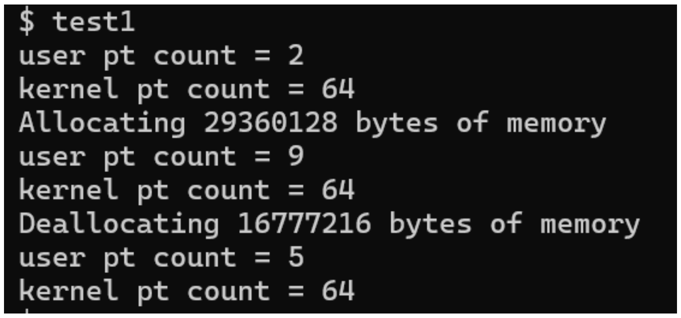
   (b) `kernel_pt_count` counts the number of inner page table pages that are used for translating kernel (high) virtual addresses from KERNBASE. You need not count the outer page directory here since it has been counted in the user sub-count.

   The pointer argument `arr` passed to the syscall points to a 2-element array in the user space program, and you have to return `user_pt_count` and `kernel_pt_count` in the first and second indices of the array respectively, i.e., `arr[0]` should carry `user_pt_count` and `arr[1]` should carry `kernel_pt_count`. You are already given the skeleton code for the system call in `proc.c`, which fills in a value of 0 for both counts. You must change this code to fill in the correct values of these counts.

   All your changes must be made in `proc.c` only. This is the only file we expect you to submit for this question.

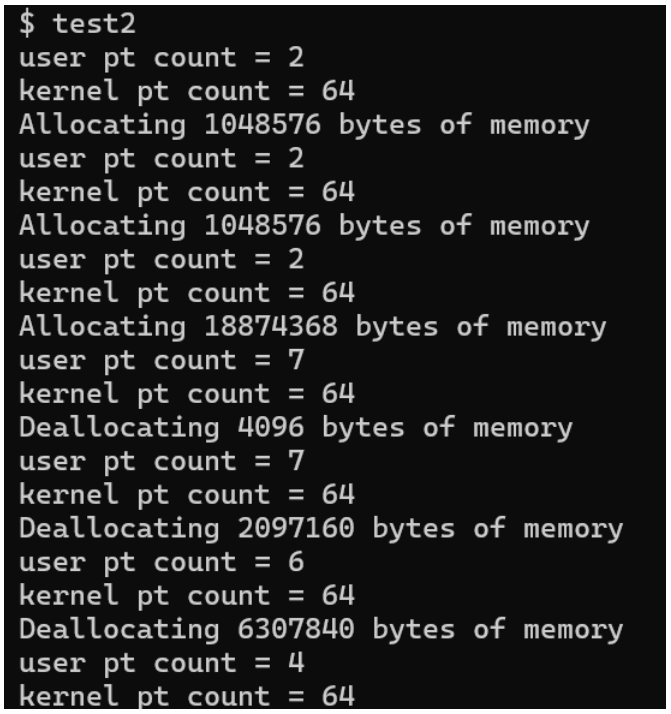
   Please note that a modified vm.c has been provided to you, in which the `deallocuvm` function has been changed so that it frees page table pages if there are no entries present in them. This change is being done so that the page table pages count changes across test cases, and we can easily test the correctness of your implementation. **Please use the modified vm.c given to you, and you will not need to make any changes to this file.**

   The testcases provided are self-explanatory and the outputs are supposed to be as follows:



```
$ test1
user pt count = 2
kernel pt count = 64
Allocating 29360128 bytes of memory
user pt count = 9
kernel pt count = 64
Deallocating 16777216 bytes of memory
user pt count = 5
kernel pt count = 64
```

Figure 1: Testcase 1 for Q1



```
$ test2
user pt count = 2
kernel pt count = 64
Allocating 1048576 bytes of memory
user pt count = 2
kernel pt count = 64
Allocating 1048576 bytes of memory
user pt count = 2
kernel pt count = 64
Allocating 18874368 bytes of memory
user pt count = 7
kernel pt count = 64
Deallocating 4096 bytes of memory
user pt count = 7
kernel pt count = 64
Deallocating 2097160 bytes of memory
user pt count = 6
kernel pt count = 64
Deallocating 6307840 bytes of memory
user pt count = 4
kernel pt count = 64
```

Figure 2: Testcase 2 for Q1

2. **[7 marks]** In this question, you will be implementing shared memory in `xv6`. To keep things simple, we will consider the scenario where the kernel has a set of 10 pre-allocated pages that can be used by processes for sharing memory. A user process can attach / detach any these shared pages in their virtual address space using the system calls shown below. In what follows, we will use the word **slot** to refer to the index of any of the 10 shared memory pages that the kernel provides, which is used as an argument to identify a page to attach. When two processes attach the same page by providing the same slot number as argument, they can communicate with each other by reading and writing into that page.

   You must implement the following two system calls. The skeleton for these system calls is already provided to you.

   - `shm_attach`: This syscall has the signature `char* shm_attach(int slot)`. The argument `slot` represents the shared memory slot that the process wants to attach through the syscall. Assume that slots are indexed starting from 0. Hence, if `slot` $>$ 9 or `slot` $<$ 0, the syscall must return 0 to indicate error. If `slot` is a valid number, then the kernel must attach the shared page corresponding to `slot` at the current break value of the process, rounded up to a multiple of `PAGESIZE`. The return value of the syscall must be the start address of the shared page in the virtual address space of the process.

     Note that the 10 shared pages have already been allocated in the modified files provided to you. You can find the physical addresses of the 10 pages in the array `slot_pas` declared at the start of `vm.c`. You must not allocate any new physical frames, and you must only attach these pre-allocated frames into the virtual address space of processes when requested.

   - `shm_detach`: This syscall has the signature `int shm_detach(char* shm_addr)`. It reverses the action of the previous syscall i.e., it removes the mapping corresponding to the shared page pointed to by `shm_addr` from the virtual address space of the process. After this syscall, any attempt by the process to access the shared page using `shm_addr` must result in a page fault. On a successful detachment, the syscall must return 0 and on a failure, it must return -1.

     Note that this syscall just "detaches" the share page from the process' virtual address space but the shared page will persist in memory, and it can still be attached to another process with `shm_attach`, i.e., you must not deallocate and free the shared pages at any point. You need not update the program break after detachment. We will not expect you to correctly handle gaps in the virtual address space caused by detachments in our test cases, so do not worry about those.

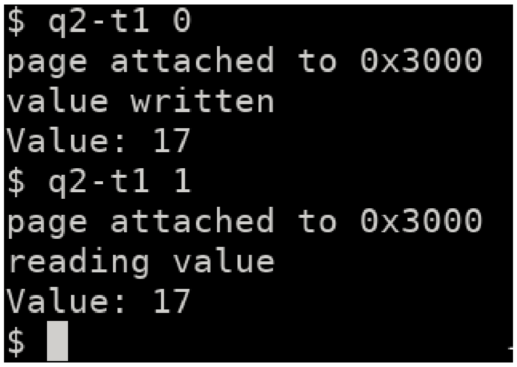
   We expect you to modify the following files for this question: `vm.c`, `sysproc.c`, and `defs.h`.

   Some hints to solve the question are given below. Please read them carefully.

   - Note that the skeleton functions for these system calls are given to you in `sysproc.c`. However, any code that accesses these shared pages must be written in `vm.c` itself, because the array `slot_pas` is a variable declared in this file. So you must parse the system call arguments in `sysproc.c` and call the functions implemented in `vm.c` to perform the actual attach / detach operations. You must add the signatures of these new functions you write in `vm.c` into the header file `defs.h` so that you can invoke them from `sysproc.c`.

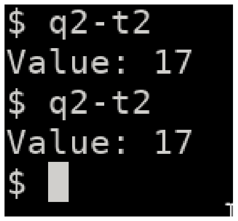
- You will need a new flag to identify these shared pages. We have already added a flag `PTE_S` in `mmu.h` to denote that the page is shared. Please set this flag for shared pages in the page table entries, and do not define your own flag for this purpose.
- We won't use `shm_attach(.)` or `shm_detach(.)` before calling `fork`. This is because copying shared pages in the parent memory image after fork may be complicated, and will have to be done differently. So you may assume that a process with shared pages will not fork after attaching the shared pages.
- Special care needs to be taken regarding shared pages when a process is exiting. When a process is reaped, `deallocuvm` is called which frees all the physical memory associated with the process. This will also include shared pages now. So there is a chance that the same physical page is freed multiple times which will make the kernel panic. To avoid these complications, we have already changed the exit function to handle shared page deallocation correctly. Our code in exit uses the `PTE_S` flag to identify shared pages, so you must set this flag suitably when you map a shared page to the virtual address space. Otherwise, the test cases may fail.

Two test cases are provided to you, and the expected output is shown below. The first test case takes two arguments indicating whether we wish to read or write from a shared page. You can read through the test cases for more details. We will also test your code with our own test cases during grading.



```
$ q2-t1 0
page attached to 0x3000
value written
Value: 17
$ q2-t1 1
page attached to 0x3000
reading value
Value: 17
$ 
```

Figure 3: Testcase 1 for Q2



```
$ q2-t2
Value: 17
$ q2-t2
Value: 17
$ 
```

Figure 4: Testcase 2 for Q2

3. **[8 marks]** In this question, we will implement shared memory in xv6, somewhat like the previous question. However, unlike the previous question, we will only have a single shared page in the system, which processes can attach and detach. Also unlike the previous question, we would want the shared memory page to be allocated on demand, and we have not allocated the page for you.

   To understand the task ahead, consider the following example sequence of events:

   (a) Initially no process has asked for any shared memory, so the kernel hasn't allocated memory for the shared page.

   (b) Process A requests to attach the shared page. The virtual address space of the process is changed, but note that physical memory for shared page is not allocated even now. Physical memory for the shared page is allocated only when some process accesses the page.

   (c) Process B requests to attach to the same shared page and attempts to read/write from/onto it.

   (d) This will result in a page fault since no physical memory is currently allocated for the shared memory page. While servicing the page fault, the kernel will allocate one free physical frame for the shared memory page, and make suitable changes to process B's page table so that the next read/write attempt to the shared memory slot by process B doesn't result in a page fault. Note that no change is made to process A's page table at this point.

   (e) Next, process A attempts to read/write from/to the shared memory page. This will result in a page fault because process A doesn't know the physical address of the page corresponding to the shared memory page. At this point, the kernel will update process A's page table appropriately to point to the already allocated shared memory page, so that the next read/write attempt to the shared memory slot by process A doesn't result in a page fault.

   To this end, you will implement a syscall `lazy_shm_attach` with the signature `char* lazy_shm_attach()`. It should return the virtual address through which the shared page can be accessed from the user space. In case of an error, the syscall must return 0. Note that the virtual address returned might not have a corresponding physical address because the shared page might not have been allocated at the time of calling the syscall.

   Next, you will implement functionality to detach (or even deallocate!) memory for the shared page. To understand the task ahead, consider the following example sequence of events:

   (a) Assume that processes A and B have the shared page attached in their virtual address space, and there is no other process that is currently using the shared page.

   (b) Process A attempts to detach the shared page from its virtual address space. Note that after this operation, the shared page must not be part of A's virtual address space, but must persist in memory because process B is still using it.

   (c) Process B attempts to detach the shared page. After this operation, memory corresponding to this shared page **must** be deallocated by the kernel because no process is currently using it.

   (d) Hereon, the behaviour of the rest of the execution must be similar to the preceding set of events (corresponding to `lazy_shm_attach` when shared pages are attached and be similar to the current set of events when shared pages are detached.

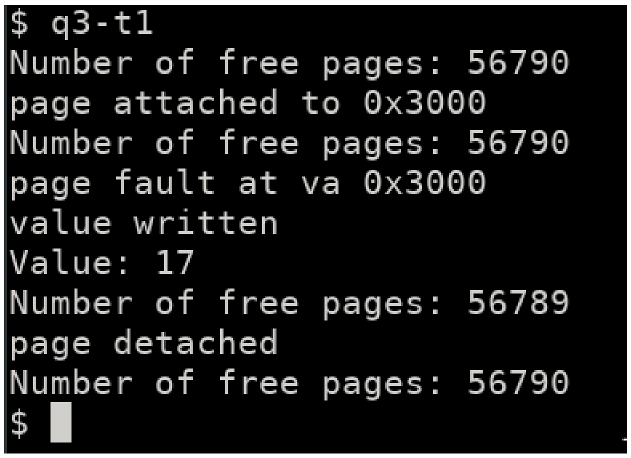
   To this end, you will implement a syscall `lazy_shm_detach` with the signature `int lazy_shm_detach(char* shm_addr)`. It should detach the shared page corresponding to `shm_addr` from the process's virtual address space and use the above mentioned policy for deallocation. An advisable way to do this is to maintain a reference count for the shared page and keep incrementing it on calls to `lazy_shm_attach` and decrementing it on calls to `lazy_shm_detach`. When the reference count becomes 0 for a shared page on a call to `lazy_shm_detach`, the kernel must deallocate the memory corresponding to this shared page.

For this question, we expect you to change `vm.c` to add the memory management code, `defs.h` to store function signatures, `trap.c` to handle page faults, and `sysproc.` to handle system calls. You need not change any other files.

Some hints to solve the question are given below. Please read them carefully.

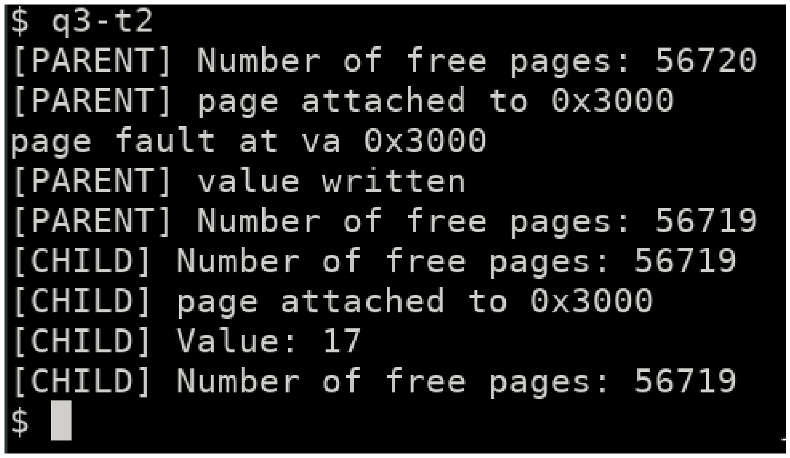
- Note that code and variables pertaining to virtual memory must be placed in `vm.c`, and the corresponding headers in `defs.h`, so that these functions can be accessed from other files like `sysproc.c` where the system call skeleton code is located.
- We won't use `lazy_shm_attach(.)` or `lazy_shm_detach(.)` before calling `fork`. This is because copying the shared page in the parent memory image after fork may be complicated, and will have to be done differently. So you may assume that a process with the shared page will not fork after attaching the shared page.
- You can also assume that we will always detach the shared page before the process exits. So you do not have to worry about cleaning up the shared page during the exist system call.
- You will need a new flag to identify the shared page in the page table entry, so that you can identify page faults for shared pages correctly. We have already added a flag `PTE_S` in `mmu.h` to denote that the page is shared. Please add this flag to page table entries of shared pages, and do not define your own flag for this purpose.
- To help you debug your code, we have added a new syscall named `num_free_pages()` which returns the number of free physical pages in the kernel free list. During lazy allocation, the number of free physical pages should decrease by 1 only when the page is accessed but not when it is attached.
- We will only allocate a single shared page for each process at the end of the virtual address space. So you need not worry about a "hole" being created in the virtual address space after calling detach. We guarantee the testcases will only allocate the shared page at the end of break value. Moreover no `sbrk` calls will be done in the testcases.

Two test cases are provided to you and their expected output is as shown below.



```
$ q3-t1
Number of free pages: 56790
page attached to 0x3000
Number of free pages: 56790
page fault at va 0x3000
value written
Value: 17
Number of free pages: 56789
page detached
Number of free pages: 56790
$
```

Figure 5: Testcase 1 for Q3



```
$ q3-t2
[PARENT] Number of free pages: 56720
[PARENT] page attached to 0x3000
page fault at va 0x3000
[PARENT] value written
[PARENT] Number of free pages: 56719
[CHILD] Number of free pages: 56719
[CHILD] page attached to 0x3000
[CHILD] Value: 17
[CHILD] Number of free pages: 56719
$
```

Figure 6: Testcase 2 for Q3

4. **[5 marks]** In this question, you will implement a memory manager with a simple buddy allocator. The memory manager will manage 4 KB of memory. The granularity of allocations will be 1 KB. You can also assume that allocations will be in powers of 2, i.e., the only possible sizes of allocations from user space are chunks of 1 KB, 2 KB, and 4 KB. For buddy allocation, you must keep in mind that you must coalesce adjacent free chunks only if they are buddies of each other. For example, if the first and fourth 1 KB blocks are occupied, then a user space allocation request for 2 KB cannot be serviced because there is **no** free 2 KB/4 KB block in the buddy hierarchy, even though there is a contiguous region of 2 KB available in the form of two 1 KB blocks.

   You are given header file `alloc.h` which has the following functions defined, which you have to implement in `alloc.cpp`.

   - `int init_alloc(void)` - This function requests 4 KB of memory from the OS. You can either use `malloc` or `mmap` to obtain this memory from the OS. It returns 0 on success and a negative value on failure.
   - `char* alloc(int)` - This function allocates a memory chunk of the requested size if possible. It returns the starting address of the allocated memory on success and `(void*) 0` otherwise. Please note that the memory block allocated should be the block with the lowest address, i.e., you must use the first fit allocation policy.
   - `void dealloc(char*)` - This function deallocates the block of memory pointed to by the virtual address given as argument.
   - `int cleanup()` - This function returns the 4 KB of memory back to the OS using `munmap` or `free`. It returns 0 on success and a negative value on failure.

   You have been given a file test_alloc.c to test your implementation. You should compile the test case along with the alloc code as follows.

   ```
   g++ test_alloc.c alloc.cpp
   ```

   The sample output file to be expected when you run your correct code is given to you along with the template code. You must write your code in `alloc.cpp` and submit. We will test your code with other test cases besides those given to you.