**Assignment 6 Report**

**Tanmay Goyal**

**AI20BTECH11021**

1. Demand Paging

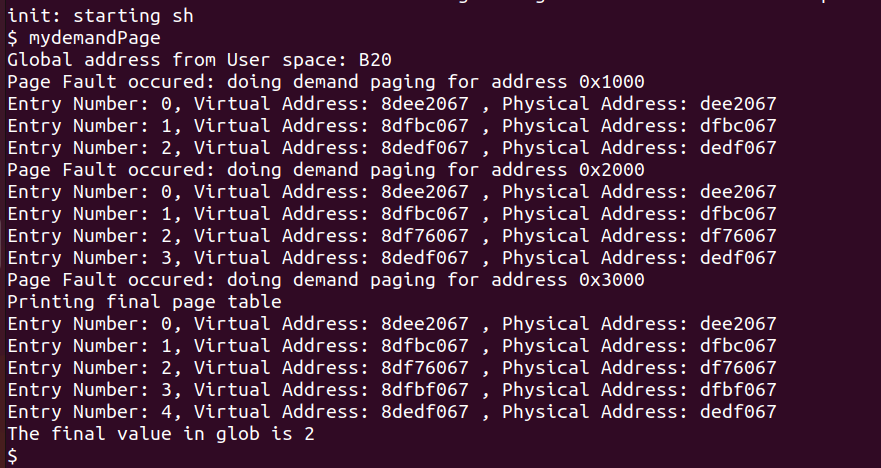
For demand paging, the main changes were in handling the page fault trap and how memory is allocated during exec.

While handling page fault traps, we check if the address at which the fault arises is not an incorrect address. If so, we return with an error. Else, we continue, allocate a new page, and map the faulty address into this new page.

Under the exec() process, we make sure that memory is being allocated only for read-only code and data. All the dynamic data is allocated on demand. This can be done using the fact that the program header of the ELF file has two information: filesz and memsz. Filesz is the memory segment’s in-file size, whereas memsz is the memory segment’s in-memory size. We have that memsz is at least filesz because memsz also contains information about a .bss, which contains uninitialized data and other dynamic data related information. Thus, we allocate memory only for filesz using the function allocuvm(), which returns the size of memory allocated, and then make sure to include (memsz – filesz) in the variable for memory size.

We also create a new function mappings() which basically returns the same value as mappages(). However, since mappages() is a static function, we ensure that this function can be accessed across files.

According to the instructions, we create a mydemandPage.c test file, and test demand paging for three values of array size: 3000, 5000 and 10000. The values are given below:

For N = 3000:

For N = 5000:

Text

Description automatically generated

For N = 10000:

Text

Description automatically generated

Text

Description automatically generated with low confidence

We modify the following files for demand paging:

1. *vm.c:* Add a function mappings() which is the same as mappages() but is not static
2. *user.h:* Add the function header for mydemandPage()
3. *mydemandPage.c*
4. *Makefile:* Added the call for mydemandPage
5. *trap.c:*  Added the code for dealing with page faults under the switch case.
6. *exec.c:* Modified the function exec() to make sure memory is only allocated for read-only code and data, but not for dynamic code.
7. *defs.h*
8. Copy On Write

For copy on write, we make sure that any page that when a child is forked from the parent, there is no new memory allocated. Instead, the child is handed a pointer to the same memory being used by the parent with a read-only tag. As soon as the child attempts to write to that memory, the child is handed its own memory.

For this, we can either change the fork() function in proc.c to use a new function defined by us, or we can modift the default implementation used in fork(), called copyuvm(). We choose the latter. We ensure that any page in copyuvm(), does not allocate new memory, and only maps the child’s pointer to the parent’s pointer, while resetting the writable bit. In case any child attempts to write, a page fault occurs, which is then handled by our custom function copyuvm\_cow().

We also include an integer array in kalloc.c that keeps track of the number of processes that are referring/mapped to a certain page, and create three functions to add, subtract and obtain the number of references to a page. We ensure that this array gets initialized to 0 under kinit2(), and during kalloc(), we initialize the array to 1. During kfree(), we only free the page completely if there is no page mapped to it. Under the page fault trap handling, we check if there are more than two processes referring to the same page, we create a new page and allow the child to use it. Else, it is given the writing permissions. To index the array, we do not need exact page boundaries. We can simply find the correct index by finding the page at which the address resides. This is done by dividing the address. By the page sizes.

I have referred to the following links. Each of them gave different perspectives and ideas to tackle the problem:

1. <https://www.cs.virginia.edu/~cr4bd/4414/F2018/paging-and-protection.html#tocAnchor-1-3>
2. <https://xiayingp.gitbook.io/build_a_os/labs/lab-5-copy-on-write-fork-for-xv6>
3. <https://pdos.csail.mit.edu/6.828/2022/labs/cow.html>
4. <https://www.cs.fsu.edu/~zwang/files/cop4610/Spring2014/project4.pdf>
5. <https://github.com/Tejesh-Raut/xv6-Operating-System-with-copy-on-write-fork/blob/master/report.pdf>
6. <https://github.com/dchandak99/copy-on-write>
7. <https://github.com/cristianomj/xv6>

Appropriate references have been mentioned in the code at appropriate positions as well.

We modify the following files for copy on write:

1. *vm.c:* Add a function copyuvm\_cow() to handle page faults. Also modified copyuvm() to ensure no new memory has been allocated and the pointer points to the parent source with a reset writable bit.
2. *user.h:* Add the function header for myCOW()
3. *myCOW.c*
4. *Makefile:* Added the call for myCOW
5. *trap.c:*  Deals with page fault using the copyuvm\_cow() function under the switch case.
6. *kalloc.c:* Adds an integer array to keep track of the number of references to each page. Added 3 functions to increase, decrease and get references respectively. Also modified kalloc(), kinit2() and kfree() to initialize the array and free pages according to the counts.
7. *defs.h*