Lab5 Phase 2

Contributors: Suraj Aralihalli, Tejas Prashanth

December 2022

Contributions

1. Tejas Prashanth
2. Design of Kernel data structures for Virtual Memory Allocation and Dellocation in E2
3. Dellocation of paging structures when a process gives up acquired pages (Development and Testing)
4. Development and testing of page fault handler for frame access present in E2 (and not in E1).
5. Suraj Aralihalli
6. Design of Kernel data structures for Virtual Memory Allocation and Dellocation in E2
7. Dellocation of paging structures when a process is terminated (Development and Testing)
8. Development and testing of page fault handler for frame access not present in E2 (and not in E1 i.e page is accessed for the first time).

3. Organization of physical and virtual memory

3.3 Backing store and demand paging

```
struct fholderE1{
    unsigned int frame_pres : 1; /* frame is present? */
    pid32  owner_process;          /* Owner process ID */
    v32addr_t vaddr;               /* Virtual address */
    uint32 time_counter;
};

struct fholderE2{
    unsigned int frame_pres : 1; /* frame is present? */
    pid32  owner_process;          /* Owner process ID */
    v32addr_t vaddr;               /* Virtual address */
};
```

We have used fHolderListE2 array to track the frames in Region E2. This datastructure resides in Region B and formed as an array of struct as defined above (*fholderE2*). The fundamental purpose of this datastructure is to facilitate ease of frame allocation in region E2 when a process requests a frame that in not in E1 when E1 is full, and deallocation when a process gives up the frame. Here $fHolderListE2[i].frame\_pres$ indicates whether the frame is occupied or not. Note that i lies in the range [0,1047]. If the frame is present $fHolderListE2[i].owner\_process$ and $fHolderListE2[i].vaddr$ indicate the owner pid and virtual address (Vf) of the frame. The purpose of adding owner pid is to enable purging the frames owned by a proccess when the process terminates (acts like an inverted page table in region E2). With this datastructure we can elegantly classify the functionality of page fault handler into following cases (when frame is missing in E1):

1. Case 1: Frame is present in E2
   (a) E1 is free (bring frame from E2 to E1)
   (b) E1 is full (swap frames E2 and E1)
2. Case 2: Frame is not present in E2 (Page is accessed for the first time)
   (a) E1 is free (add new frame to E1)
   (b) E1 is not free
      i. E1 is full and E2 is not full (evict page from E1 into E2 and add new frame to E1)
      ii. E1 is full and E2 is full (add the process to PR_FRAME state)

7. Submission in two phases

7.1 Testing

We test the implementation of the virtual memory management unit in the following manner-

1. We test the ability of the kernel to move frames from E1 to E2 when E1 is full. We do this by creating two processes - processes A and B - and each process allocates 512 pages of virtual memory and writes to the pages (process A allocates pages, followed by process B). Thereafter, process A creates an additional page in virtual memory (VF). When the page is written to, a page fault is generated since region E1 already has 1024 frames allocated. Consequently, a frame is moved from region E1 to E2 according to the FIFO replacement policy defined. In this case, this corresponds to a frame allocated by the same process (i.e process A). The free frame in region E1 is assigned to the new page that was allocated. In order to test that the page was successfully moved from E1 to E2, we print the frame's contents (i.e the page's contents) before and after the movement and ensure that they match the value that was written to them respectively. Moreover, we check the number of frames allocated to processes A and B in regions E1 and E2 and verify that process A 512 frames in E1 and 1 frame in E2.

2. We test the ability of the kernel to move frames from E1 to E2 when E1 is full. The difference between 1) and 2) is that the frame that is moved corresponds to a different process. We create 2 processes, processes B and X, such that each process allocates 512 pages of virtual memory in VF and writes to all the pages (process B creates pages before process X). Thereafter, we allocate an additional page in process X and write to it, which results in a page fault. Based on the page replacement policy, a frame corresponding to process B is moved from region E1 to E2 and the free frame in E1 is allocated to the new page that process X demanded. This

test case verifies that frames in E1 not owned by the process demanding frames can be moved from region E1 to E2.

3. We test the ability of the kernel to move processes into state PR_FRAME when a new frame is demanded from E1 when E1 and E2 are full. We also test the ability of deallocation modules to make processes in state PR_FRAME into state PR_READY when frames in region E1 or E2 become available. We do this by creating three processes, processes C, D, E. Process C allocates 1024 pages in VF and writes to the pages. This ensures that region E1 is occupied by process C entirely. We print the content of the first and last page to verify that the write was successful. In addition, we also print the number of frames in regions E1 and E2 that have been occupied by process C. Thereafter, process D runs and allocates 1024 pages and writes to all the pages. Since region E1 is full, frames occupied by process C are moved to region E2 and given to process D when the page fault is handled. We print the number of frames in regions E1 and E2 occupied by process D and verify that it is equal to 1024 (corresponding to process D) and 1024 respectively (corresponding to process C). Thereafter, process E runs and allocates 100 pages in VF and writes to all the pages. During the writing process, multiple page faults are generated and frames in E1 occupied by process D (i.e according to FIFO replacement policy) are moved to E2 until E2 is full. Further frame allocation by process E results in it going to state PR_FRAME. When processes C and D die, they deallocate frames occupied in regions E1/E2 and move process E to state PR_READY. Process E then continues to claim frames from region E1 until it has the required number of frames it needs.

4. We test the ability of the kernel to move frames from E2 to E1 when a page in E2 is accessed/read. We do this by creating two processes, process F and process G, such that process F has a higher priority than G. Process F creates 1024 pages in VF and writes to all the pages, leading to multiple page faults. Consequently, 1024 frames are allocated in region E1 corresponding to process F. We verify this by printing the number of allocated frames (corresponding to process F) in regions E1 and E2. Thereafter, process G runs and creates 1024 pages in VF and writes to all the pages, leading to multiple page faults. Since region E1 is full, frames corresponding to process F in E1 are moved to E2 and free frames in E1 are assigned to process G. Then, process F performs reads a random page's contents (one page among the 1024 pages it allocated). During this process, a page fault is generated since the required page is in region E2. The page is brought to E1 by evicting a page corresponding to process G from E1 and using the free frame for process F. We print the content in process F's page to ensure that it corresponds to the data that was originally written in the page.

5. We test the ability of the kernel to bring a frame from region E2 to E1 when E1 is free. We do this by creating two processes, process H and I, such that process H allocates 1024 pages in VF and writes to all the pages, resulting in multiple page faults. After handling all page faults, we verify the region E1 contains frames owned entirely by process H. Thereafter, process I also allocates 1024 pages and writes to the pages. This causes all pages (corresponding to process H) to be evicted from region E1 to E2, after which region E1 will be used entirely by process I. Then, process I dies and process H tries to read a page's content. This causes the page content's to be moved from a frame in region E2 to E1. We verify that the page's content matches the value that was originally written to it.

6. We test the ability of the kernel to bring a frame from region E2 to E1 when both E1 and E2 are full. We do this by creating three processes, process J, K and L, such that process J allocates 1024 pages in VF and writes to all the pages, resulting in multiple page faults. After writing process J goes to sleep. Process K allocates and writes to 1024 pages before going

to sleep This results in evicting all the process J frames into E2. At this point we have 1024 frames in E1 that belong to process K and 1024 pages in E2 belonging to process J. We now have a third process that allocates and writes to 24 pages before going to sleep. At this point all the frames in E1 and E2 are completely occupied. Now process J wakes up from sleep and reads a random page's contents (one page among the 1024 pages it allocated). During this process, a page fault is generated since the required page is in region E2. The page is brought to E1 by evicting a page corresponding to process K from E1 and using the free frame for process J. We print the content in process J's page to ensure that it corresponds to the data that was originally written in the page.

Bonus

Please refer code.