CS 342 Design Document
Project 3D – Page Eviction and Stack Growth

Rajat Kateja (r.kateja@iitg.ernet.in)
Maths and Computing
10012338

TA: Lalatendu Behera
Chetti Prasad

------ DATA STRUCTURES ------

This project required the introduction of a swap table to keep a track of the empty swap slots. The swap table was implemented as an array of boolean values. The array size is 1024, considering a swap disk of 4MB and slots equal to the page size i.e. 4KB. The value corrosponding to the array index identifies whether the swap slot is occupied or not.

bool swap_slot_occupied[1024];

Also introduced a last_evicted_frame integer value to store the last frame which was evited (in the implementation of the second chance replacement algorithm). This is similar to the clock hand in the clock algorithm.

int last_evicted_frame;

Added the folowing to the struct thread

int no_of_stack_pages;

This is to keep a track of the number of stack pages the process is using. It is also used in the heuristic used to identify whether the page fault is for an extra stack page.

------ ALGORITHMS ------

Stack Growth

For the stack growth, a heuristic had to be designed to identify the page fault as a page fault corrosponding for a stack page. This was done as follows: If the page fault address was greter than the stack pointer – 32 value, the page fault is identified as a request for an extra stack page. The reasoning behind this is that the instructions which push a value in the stack check for the addresses upto 32 below the stack pointer (on instruction checks for addresses 4 below the stack pointer and on checks for values 32 below the stack pointer).

If the check returned true, it meant that a page has to be allocated for the stack. Firstly a page from the user pool is asked for using the palloc_get_page function, and then this page is installed in the virtual address of the next stack page, which is calculated using the no_of_stack_pages field present in the thread data structure. If the page is installed successfully, the no_of_stack_pages field is incremented by one. Else the allocated page is returned back to the user pool usin palloc_free_page function.

Page Eviction Algorithm

For the page eviction algorithm, the swap disk was used to swap out the evicted page and to be brought back in when required again (on a page fault). The algorithm works as follows:

Whenever the function palloc_get_page returns a NULL pointer, which implies that there are no more free pages in the user pool, the function evict_and_allot (implemented by me) is called. The function works on the second chace algorithm to find a fram to evict. It iterates the frame table starting from the last_evicted_frame (clock) entry, and check if the page corrosponding to that frame has been accessed in the near past. If it has been accessed, its page table entry for the accessed bit is set to false (so as to keep only the relevant recent past). When the first page is found for which the accessed bit is false, it is chosen as the victim frame. The fram is now to be written on the swap space to be evicted. For that the swap table entries are iterated to look for a free swap slot. Then the data from the frame is written onto the swap alot using the disk_write function. The corrosponding page table entry is cleared, and the supplementary page table entry is updates to reflect that the page is in the swap slot. Also the last_evicted_frame entry is updated so as to start the iteration in the next failure from the corect frame number. An importatnt part to keep in mind is that user frames start only from frame number 654, so while iterating circularly, the frame numbers should start from 654 and go upto 1023.

While writing onto the disk, the disk write function deals with sector numbes rather than slot numbers (page numbers), so the appropriate translation has to be performed. One page corrosponds to 8 sectos, so the slot number must be multiplied by 8 to obtain the first sector number, and the disk_write function needs to be called 8 times, once for each of the 8 sectors corrosponding to the slot.

This completes the page eviction algorithm, but now in the case of a page fault, the page may also be present in the swap slot and the page must be swaped in from the swap slot. For that, a new page from the frame table is allocated to this process, and installed for the corrosponding virtual page number. Then the data is read from the swap disk into the frame using disk_read fucntion. Here again the appropriate translation from slot number to sector number has to be performed similar to that preformed for the disk write function.

The swap_table is upadated here and the supplementary page table and frame table are updated after the installation of the page in the install_page funciton.

------ SYNCHRONIZATION ------

There were no special race conditions to consider in this project and hence no extra synchronization efforts were required.

------ RATIONALE ------

1.) By keeping a track of the number of stack pages per process, the code can be easily modified to limit the total number of stack pages a process can have. This helps in extensibility of the code.
2.) Keeping the swap slot size same as the page size avoids any un-necessary overheads in translating from pages (in main memory) to swap slots.
3.) Swap table is just an array of boolean values, rather than a large data structure, thus saving space.
4.) In the current implementation, the second chance algorithm does not check for any already existing frame. This might be improved upon.