1. **Name and Number of the lab:** Lab3 – Memory Management and Paging
2. **Name and email address of people:** Sayali Patil ([sayali.patil@wustl.edu](mailto:sayali.patil@wustl.edu))
3. **Attribution of source:**
4. LKD and LPI books, lecture slides
5. Linked Lists Tutorial: <https://www.geeksforgeeks.org/data-structures/linked-list/>
6. Mmap() sys call: <https://www.tutorialspoint.com/unix_system_calls/mmap.htm>
7. remap\_pfn\_range: <https://stackoverflow.com/questions/8788289/how-remap-pfn-range-remaps-kernel-memory-to-user-space>
8. list\_for\_each\_safe() : <https://stackoverflow.com/questions/9207850/why-do-we-need-list-for-each-safe-in-for-deleting-nodes-in-kernel-linked-list>
9. **Specific Report Sections:**
10. **Initial Configuration:**

pi@raspberrypi\_Sayali\_Patil:~/linux\_source/lab3 $ sudo ./dense\_mm 10

pi@raspberrypi\_Sayali\_Patil:~/linux\_source/lab3 $ sudo rmmod paging

pi@raspberrypi\_Sayali\_Patil:~/linux\_source/lab3 $ dmesg

[ 1826.108684] Loaded kmod\_paging module

[ 1834.413836] paging\_mmap() invoked: new VMA for pid 2148 from VA 0x76f6f000 to 0x76f70000

[ 1834.413861] paging\_mmap() invoked: new VMA for pid 2148 from VA 0x76f6e000 to 0x76f6f000

[ 1834.413876] paging\_mmap() invoked: new VMA for pid 2148 from VA 0x76f6d000 to 0x76f6e000

[ 1834.413891] paging\_vma\_fault() invoked: took a page fault at VA 0x76f6d000

[ 1834.414207] paging\_vma\_close() invoked

[ 1834.414214] paging\_vma\_close() invoked

[ 1834.414220] paging\_vma\_close() invoked

[ 1847.909942] Unloaded kmod\_paging module

1. **Datatype Design:**

Along with page structure and reference counter (mandatory fields), I decided to use a list structure to create a list of pages being allocated and deallocated and to traverse through.

The structure looks like this:

struct data\_struct{

atomic\_t ref\_counter;

struct page \*pm\_page;

struct list\_head h\_list;

};

In the do\_fault() function which is written in order to handle page fault, first a new page of physical memory is allocated using alloc\_page(). Once that is done then then a data structure of type data\_struct is declared and initialized. I am using this data structure to hold the pointer to newly created physical page in the list entry of the data structure. Head of the list is obtained by using vm\_private\_data as that’s where the list head is stored. After that, to keep a track of the pages being allocated, I am then adding the local list entry to the list handled by the main list. This assignment works because in the mmap() function I am declaring and allocating memory for the data\_struct and setting this data structure as the head of the list. To be accessed later, the structure is then stored in the vm\_private\_data filed of vma.

1. **Page Fault Handler:**

I allocated a new physical page using alloc\_page() and then updated the process' page tables to map the faulting virtual address to the new physical address allocated earlier. The values for vaddr and pfn were calculated using provided functions in the writeup as:

Vaddr = PAGE\_ALIGN(addr)

Pfn = page\_to\_fn(ptr)

To remember the created page I allocated a new list structure with head stored of the list stored in the vm\_private\_data field of the vma and then added the new page entry to the tail of the created list. Also, incremented the atomic variable holding the value for allocated pages to keep a count of the total number of pages being allocated.

Output at this stage was: Multiplication Done!

1. **Close Callback:**

In the case of demand paging, a new page is created when a page fault occurs and we store this page in the list to keep tracks of the pages being created. We store the head of the list in vm\_private\_data field of our vma. While freeing the memory we traverse through the list of pages using list\_for\_each\_safe function. I used list\_for\_each\_safe instead of using list\_for each to traverse through the list because of the reasons given in ref-5 listed under sources of attribution. By storing the current page in the list structure temp\_page, we free one page at a time using list\_for\_each\_safe() function. In case of pre-paging, since we allocate all the memory needed for required number of pages in a go, we can directly free up the pages using \_\_free\_pages(). In the end we also need to free up the memory allocated to hold vma structure.

**Output:**

[ 2213.452743] Number of allocated pages: 384

[ 2213.452760] Number of freed pages: 384

[ 2213.452769] Unloaded kmod\_paging module

1. **Pre-Paging:**

To calculate how many pages needed to be created we first get the length of the virtual address by accessing the vma fields and the dividing the difference by PAGE\_SIZE like this:

Nr\_pages = (vma->vm\_end - vma->vm\_start)/PAGE\_SIZE

If the virtual address length is not completely divided by the PAGE\_SIZE that means we need to add one more page to hold the remaining length. That’s why we check for that condition and if the condition is true, we add one more page in the total number of pages calcaluted earlier. Finally to allocate the pages using total number of orders, we used the same get\_order function we used in our studio exercise and performing the remaining operations in the same fashion as demand paging.

1. **Experiments:**

The values obtained for mean and standard deviation for sys-time, multiplication-time, total-time are as followed:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Paging Type | Matrix size | Mean(Sys-Time) | Mean (Mul Time) | Mean (Total Time) | SD(Sys time) | SD(Mul-time) | SD(Total time) |
| Demand-paging | 64 | 95.9 | 33253.7 | 33349.6 | 18.0 | 5669.2 | 5681.4 |
| 128 | 120.0 | 188494.0 | 188614.0 | 22.7 | 18189.6 | 18187.6 |
| 256 | 144.3 | 2422205.4 | 2422349.7 | 36.0 | 266829.8 | 266823.3 |
| 512 | 131.3 | 31442518.3 | 31442649.6 | 30.5 | 410101.1 | 410103.3 |
| Pre-paging | 64 | 143.7 | 30250.0 | 30393.7 | 44.0 | 6007.1 | 6043.5 |
| 128 | 172.6 | 176588.3 | 176760.9 | 52.5 | 18387.9 | 18411.8 |
| 256 | 209.0 | 1678239.7 | 1678448.7 | 101.7 | 16615.1 | 16591.0 |
| 512 | 402.4 | 30138036.7 | 30138439.1 | 18.1 | 109743.0 | 109746.7 |

The values of min and maximum times for system-time, multiplication time, and total time are as followed:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Paging Type | Matrix size | Min(Sys-Time) | Max(Sys- Time) | Min (Mul- Time) | Max(Mul time) | Min(Total-time) | Max(Total-time) |
| Demand-paging | 64 | 60 | 127 | 17512 | 37897 | 17572 | 38004 |
| 128 | 78 | 166 | 159747 | 213106 | 159856 | 213184 |
| 256 | 72 | 222 | 2111537 | 2860419 | 211759 | 2860564 |
| 512 | 78 | 174 | 30768589 | 32183509 | 30768677 | 32183650 |
| Pre-paging | 64 | 73 | 187 | 17110 | 36046 | 17197 | 36224 |
| 128 | 76 | 232 | 154472 | 208789 | 154686 | 208978 |
| 256 | 77 | 331 | 1649986 | 1716892 | 1650217 | 1716986 |
| 512 | 365 | 432 | 30007723 | 30353353 | 30008121 | 30353758 |

I am attaching following graphs with the report:

1. System time for demand and pre-paging: System-time.png
2. Multiplication time for demand and pre-paging: Mul-time.png
3. Total time for demand and pre-paging: Total-time.png

From the graph for system-time for demand and pre-paging it can be seen that the initially pre-paging takes longer system time complete the mmap() process as we know that in demand paging one page is added at a time when a page fault happens. But in pre-paging all the pages that may be needed by the application program are counted beforehand and added at once and that can delay the process of completion of memory allocation in case of pre-paging. The resulted graph supports out theory. Also, we can see that for pre-paging the jumps according to matrix size are considerable but for demand paging they aren’t. This could be because demand paging only allocates a page when a page fault occurs. If the page that we are looking for already exists, then demand paging doesn’t create a new page. But in case of pre-paging, since we are already creating all the required pages, the spikes are considerable with increasing number of matrix size. For both demand paging and pre-paging, the matrix multiplication time increases with increase in the size of matrices. Looking at both the paging behaviors, we can say that for small matrix sizes demand-paging could work better and for bigger matrix sizes pre-paging could work better.

1. **Questions:**

Which paging mechanism will be more helpful where the computations are complex and the variable sizes involved are bigger in size? If we add one more restriction saying the program should be run in parallel on different cores or using different threads then how does the earlier belief change or not?

1. **Suggestions:** None
2. **Approximate amount of time spent:** 21-22 hours