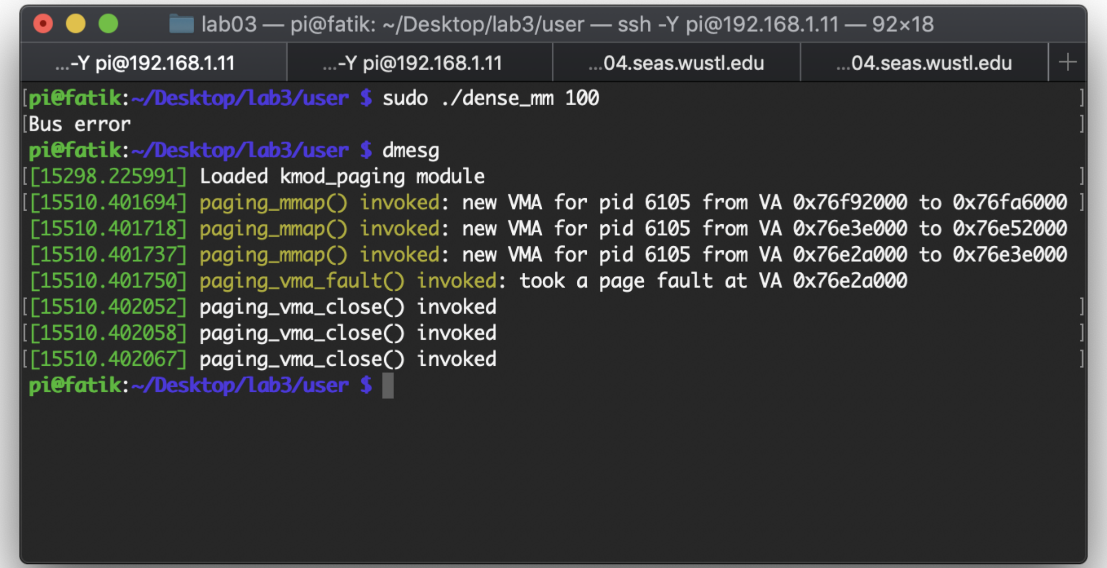
Funda Atik - fatik@wustl.edu
Ighor Tavares - f.tavares@wustl.edu

LKD book and lecture slides are used as reference.

CS422S: Lab 3 Report

1. The output of dmesg output and the output of the dense_mm program when executed on your Raspberry Pi.



```
pi@fatik:~/Desktop/lab3/user $ sudo ./dense_mm 100
Bus error
pi@fatik:~/Desktop/lab3/user $ dmesg
[15298.225991] Loaded kmod_paging module
[15510.401694] paging_mmap() invoked: new VMA for pid 6105 from VA 0x76f92000 to 0x76fa6000
[15510.401718] paging_mmap() invoked: new VMA for pid 6105 from VA 0x76e3e000 to 0x76e52000
[15510.401737] paging_mmap() invoked: new VMA for pid 6105 from VA 0x76e2a000 to 0x76e3e000
[15510.401750] paging_vma_fault() invoked: took a page fault at VA 0x76e2a000
[15510.402052] paging_vma_close() invoked
[15510.402058] paging_vma_close() invoked
[15510.402067] paging_vma_close() invoked
pi@fatik:~/Desktop/lab3/user $
```

2. We defined a new structure like this by using Linux Kernel's link list implementation.

```
struct page_list {
   atomic_t rf_count;
   struct page * m_page;
   struct list_head list;
};
```

The structure holds an atomic reference counter (rf_count), a page struct pointer (m_page), and a list_head structure. We used this structure for two purposes: a list head and a list entry (node).
In the mmap callback:

- We dynamically allocate the new structure and set its atomic reference counter to 1.
- Then, we mark this structure as a head node.
- Finally, we store this structure in the private field of the given vma.

In do_fault fuction:

- After a new page of physical memory is allocated, the program allocates a new data structure and holds a pointer to this new page in this list entry.
- Then, we add this list entry to the list which is accessed by the head list structure. Remember that we store the list head in vm_private_data.

3. Our page fault handler performs following operations:

- We allocate a new page of physical memory via the function alloc_page.
- Then, we update the process' page tables to map the faulting virtual address to the new physical address. The values for vaddr and and pfn is calculated like this:

```
page_ptr = alloc_page(GFP_KERNEL);
pfn = page_to_pfn(page_ptr);
vaddr = PAGE_ALIGN(fault_address);
```

- Finally, we allocate a new list entry structure in order to store the page that has been created. Then, we access the list via the list head which is stored in vm_private_data and add this new entry to the end of the list.

The output of user-level dense_mm program is: “Multiplication done”.

4. The close() callback function frees physical memory allocated previously and our tracker structure. For demand paging approach, when a page fault occurs, a new page is created and added to the list. We access this list via a head list structure which stored in the vma’s private data field, and m_page field of the list head structure does not point anything. However, for prepaging approach, we allocate all physical memory that we need in one large allocation, so we do not need to maintain a list of pages. We keep the pointer to this large block in the list head structure via m_page field. In addition to this, the head list’s list is empty.

The dmesg output after unloading the module:

```
[ 5170.805456] Number of pages allocated/freed: 384/384
```

5. To support prepaging memory mapping, we calculate how many pages that we need to allocate in one call. First, we find the length of the virtual address that is requested. Then, the number of pages can be found by dividing this length into PAGE_SIZE. The result is incremented if the remainder is not equal to zero. Moreover, in order to call pages_alloc function, we need to find the base 2 logarithm of the number of pages that we wish to allocate. We used my_get_order routine from Studio 12 to calculate order and vaddr is defined by using PAGE_ALIGN macro.

6. Plots for demand paging:

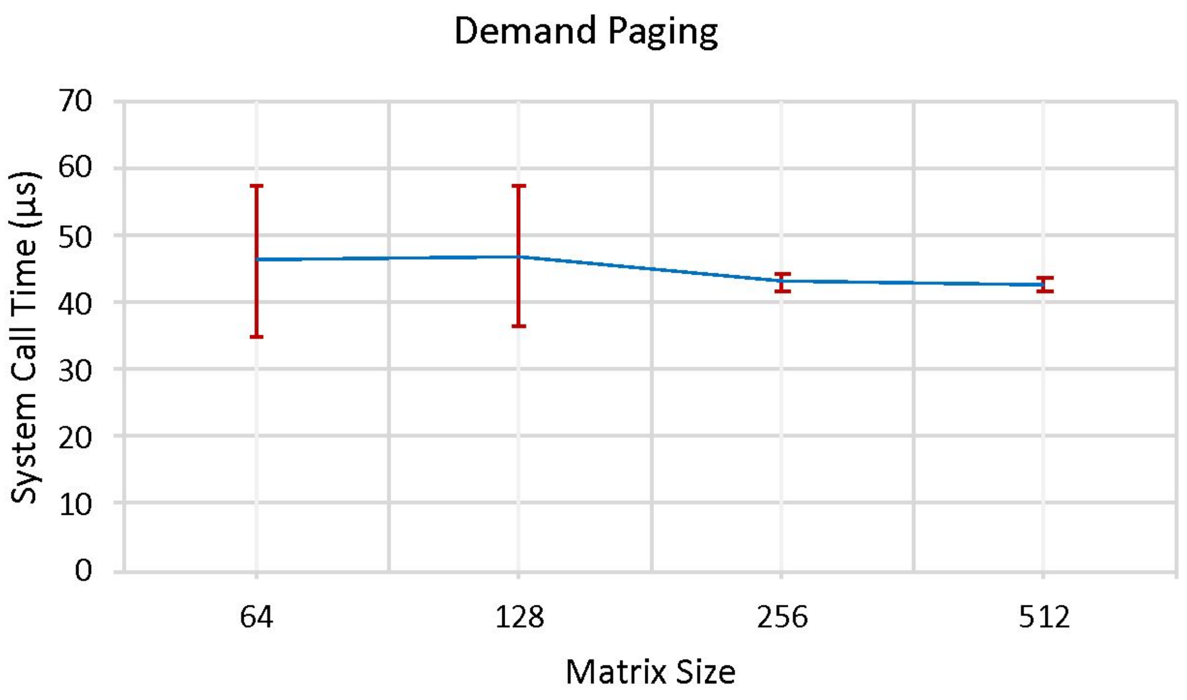


Figure 1. System call performance

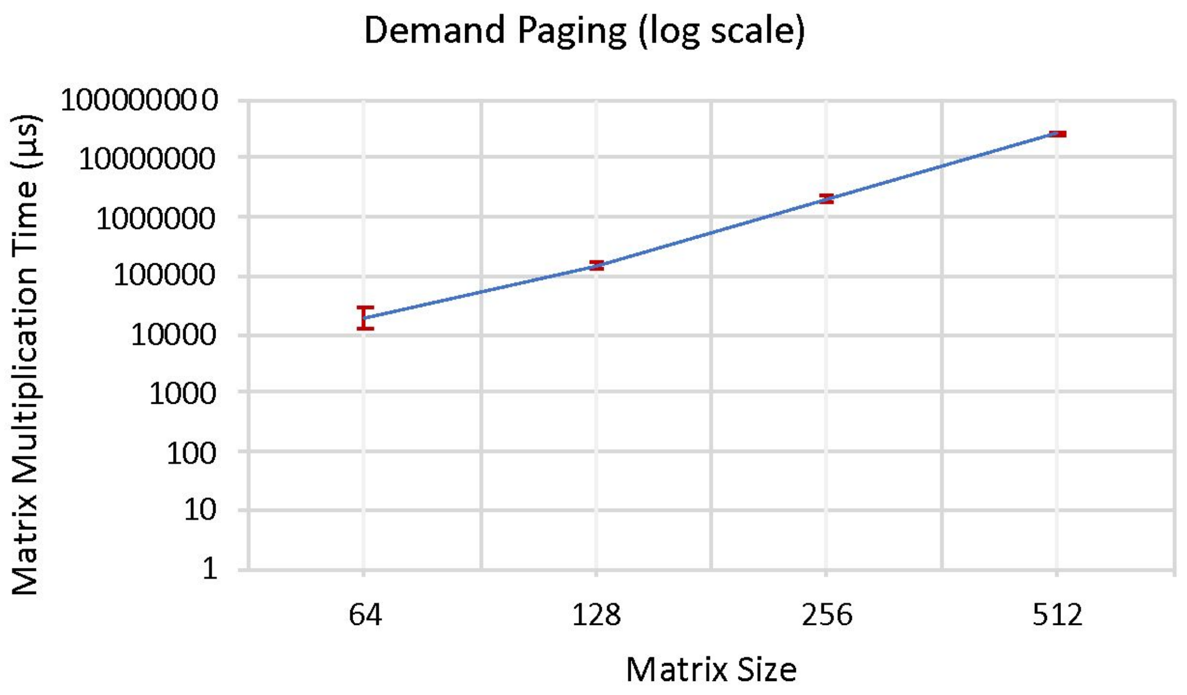


Figure 2. Multiplication Performance

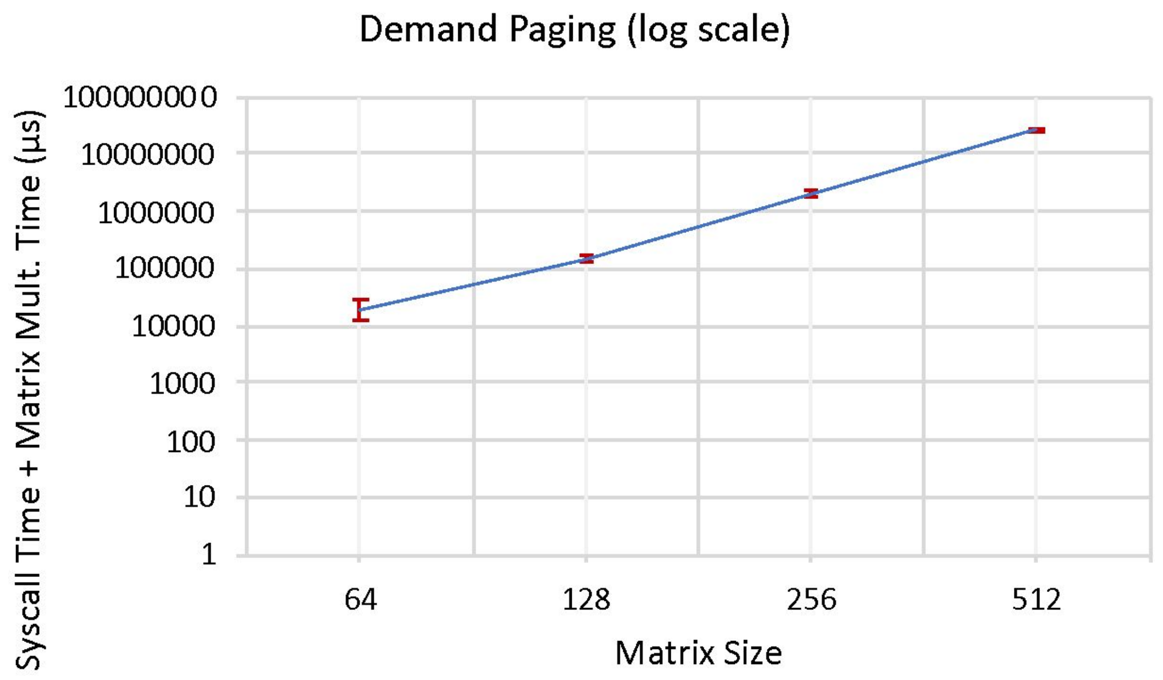


Figure 3. Overall performance for demand paging

Plots for pre paging:

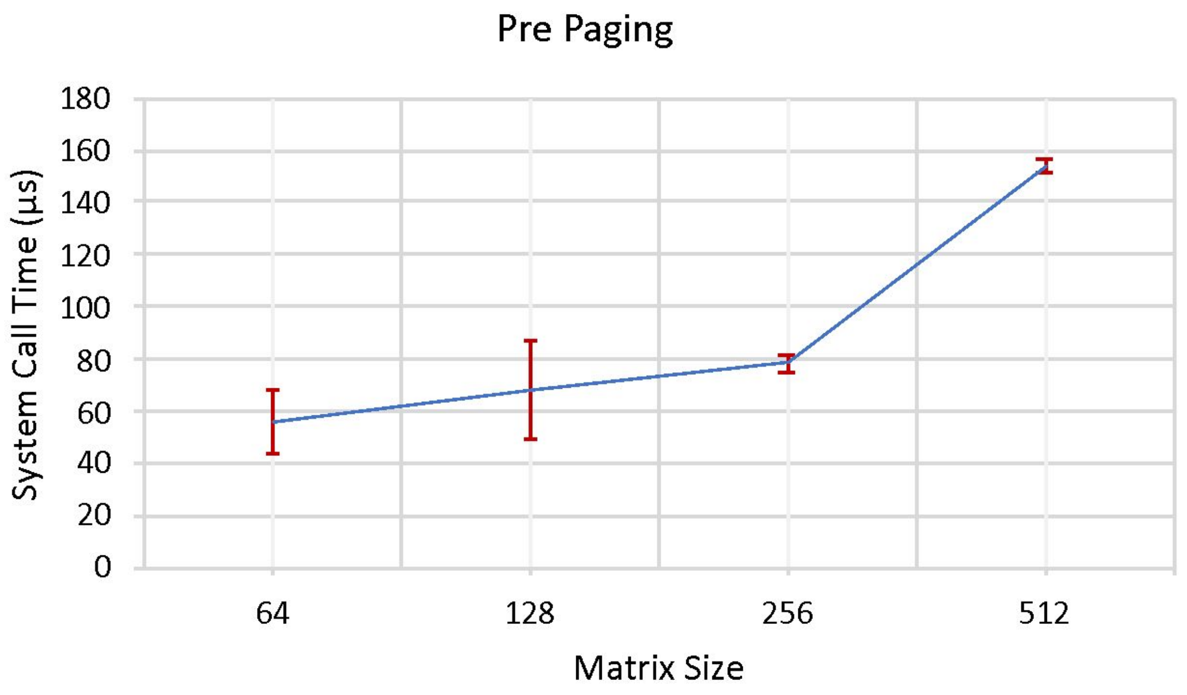


Figure 4. System call performance

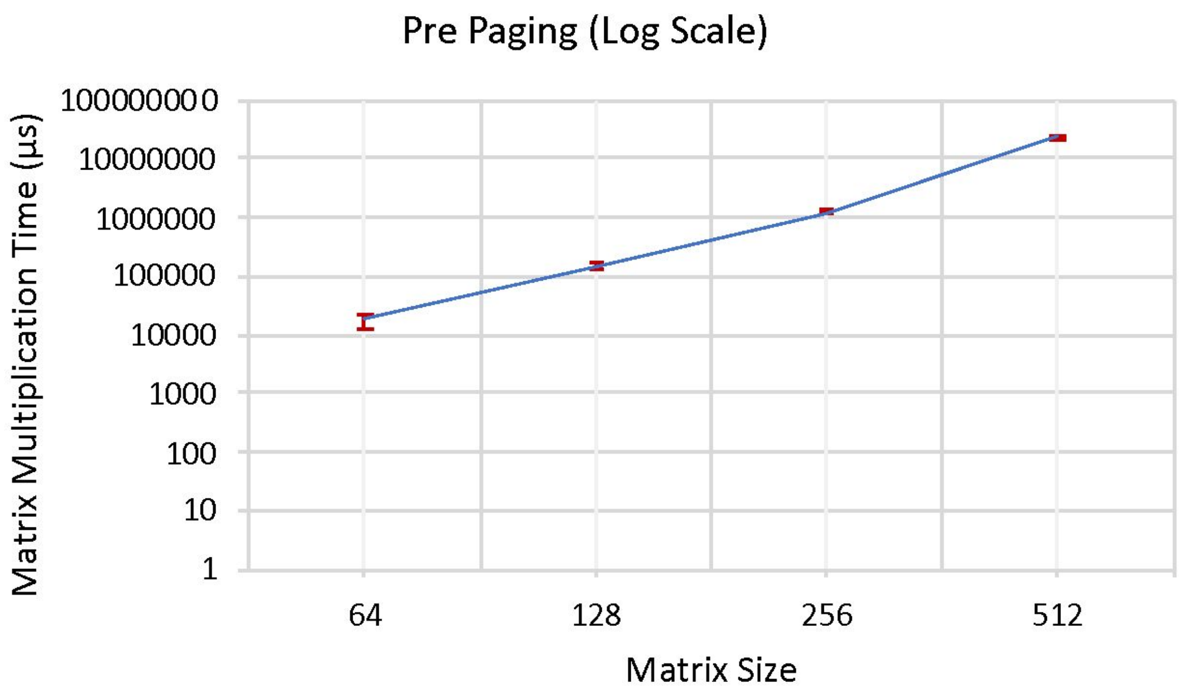


Figure 5. Multiplication Performance

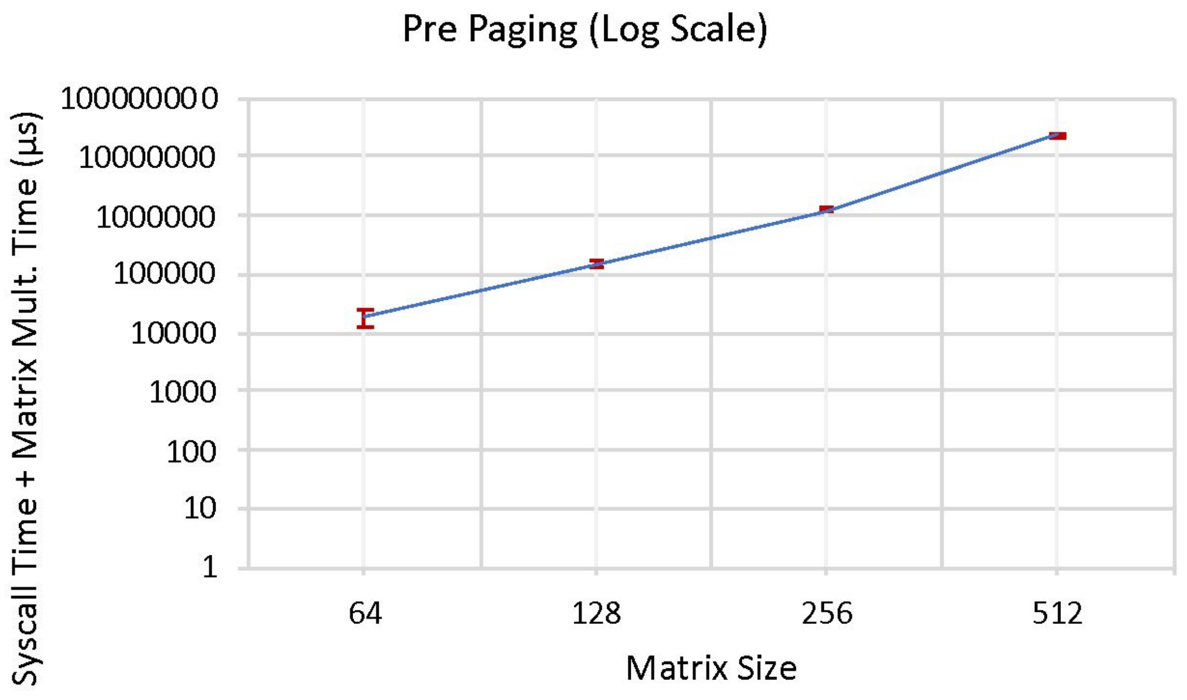


Figure 6. Overall performance

The initial overhead of prepaging approach is large because the program performs one large allocation to obtain all of the pages it needs in one single call. As can be seen in Figure 4, the system call takes long to finish prepaging due to the fact that pages other than the one demanded by the page fault are brought into memory. The program allocates all of the pages that need in a single call. The more resources the process request, the more pages will be allocated in with prepaging creating a larger overhead at initial because it tries to allocate as many pages as the VMA can support. With demand paging, however, the overhead seems to be consistent as expected because only the first page is allocated at the beginning. With demand paging, a page is brought into memory only when a location on that page is actually referenced at execution time. If the program does not access that location, the page fault does not occur, thereby avoiding unnecessary virtual to physical mapping. Although the standard deviation of the system call time is relatively larger for demand paging than for prepaging, it becomes small for both approaches when the matrix size becomes larger. Figures 2-5 demonstrate that when matrix size increases, time which is required to matric multiplication increases exponentially. Although demand paging incurs more overhead due to the allocation of a new page on-the-fly when a page fault occurs, this overhead is negligible when comparing to the amount of time is needed for matrix multiplication. This behavior can be seen in Figures 3-6. To sum up, we can say that if the program will access only a small portion of memory, then pre paging wastes lots of memory because it allocates one contagious block on the memory, which also increases the overhead of context switch. Prepaging saves time only when the large contiguous structure is used. Contrary to prepaging, demand paging does not load the pages that are never accessed, hence saves the memory for other programs.

7. Any insights or questions we have had while completing this assignment, any suggestions for improving assignment itself, etc.

Demand paging increases the degree of multiprogramming by demanding the pages on the fly. Most of the time, a program executes only a small portion of the code (90/10 rule) so large programs can run on the machine when demand paging is selected, even though there is no sufficient memory to run the program. Moreover, the operating system merges different pages if they point to the same content so pages will be shared by multiple programs. Although page sharing can be exploited as a side channel by some attacker programs, it is a good optimization to reduce memory footprints of the programs. In addition to page sharing, the page replacement policy and application reference pattern affect the performance of demand paging. Therefore, we believe, it will be good practice to se how different replacement policies or page sharing affect the performance of the program when the demand paging case is selected.

9. Approximate time spent
16 hours