MP3

Group Members: Edgar Cortez(ecortez2), Dashuai Qin(dqin3)

Implementation and Design Decision:

MP3_Init:

We initialized all things we would need, like the file directory, character device driver, our work_queue and work, as well as allocate continuous memory for our buffer.

Task_Structure:

We used a Linux list_head linked list to store PCB structure for all the task, we stored the current task struct using the given function in order for it to be retrieved later when looping through the linked list.

MP3_Write:

We parsed the arguments sent from the user space to either register or unregister the process.

Registration:

We initialized and allocated memory for each new process making space for our struct. This function would be called when a new process begins so we begin to insert into our linked list. If it was empty, I decided that it would automatically be the first in the list, so I would insert and then start the delayed work. If it's not empty, we simply added it to our linked list so it can be accessed during a work iteration. We made sure to put proper mutexes so when multiple processes are called at the same time there won't be memory issues.

Unregistration:

Deletes the process from the list, if we delete and it's the last in the linked list, we cancel any scheduled delayed work in order to flush and destroy the workqueue.

MP3_mmap:

This maps the physical page from of our buffer to virtual memory, so the user can access it without overhead time. We kept track of the offset of the starting position because we would have to map several page frames into the virtual address for the user. Therefore, we would keep calling the remap function until it reaches the end of our buffer.

Work_callback:

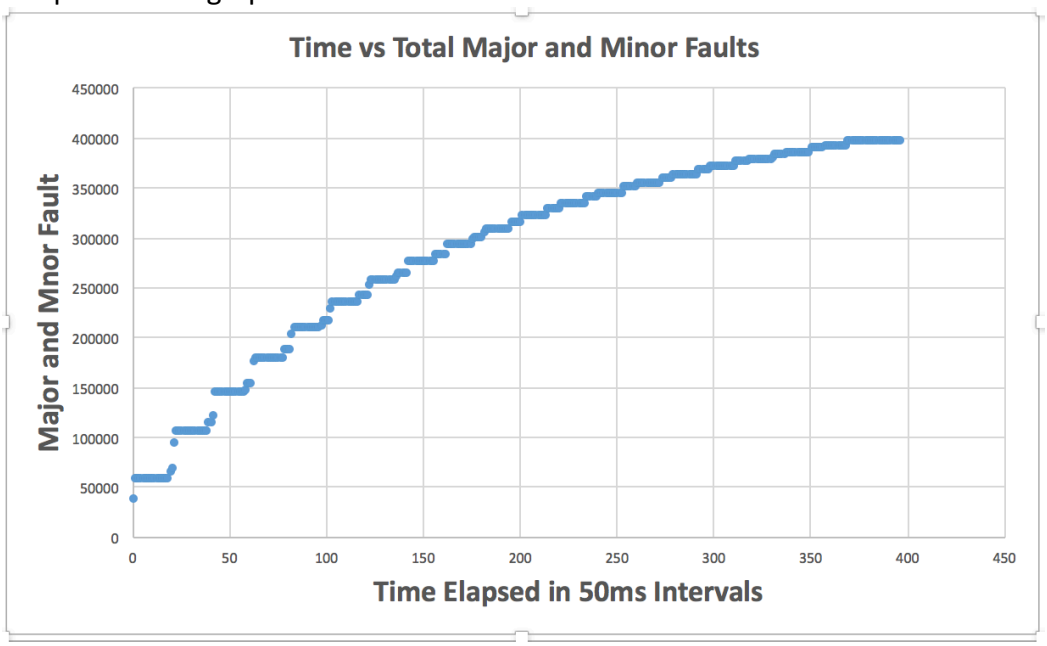
We looped through all our processes, and get the min_flt,maj_flt, utime and stime that accumulated since the previous invocation. We then store these variables into the buffer at the correct offset in order for the user to access when calling mmap. Then we add this same work on a delay after 50msecs so it can be called again.

MP3_exit:

We delete the character device driver and clean up the memory.

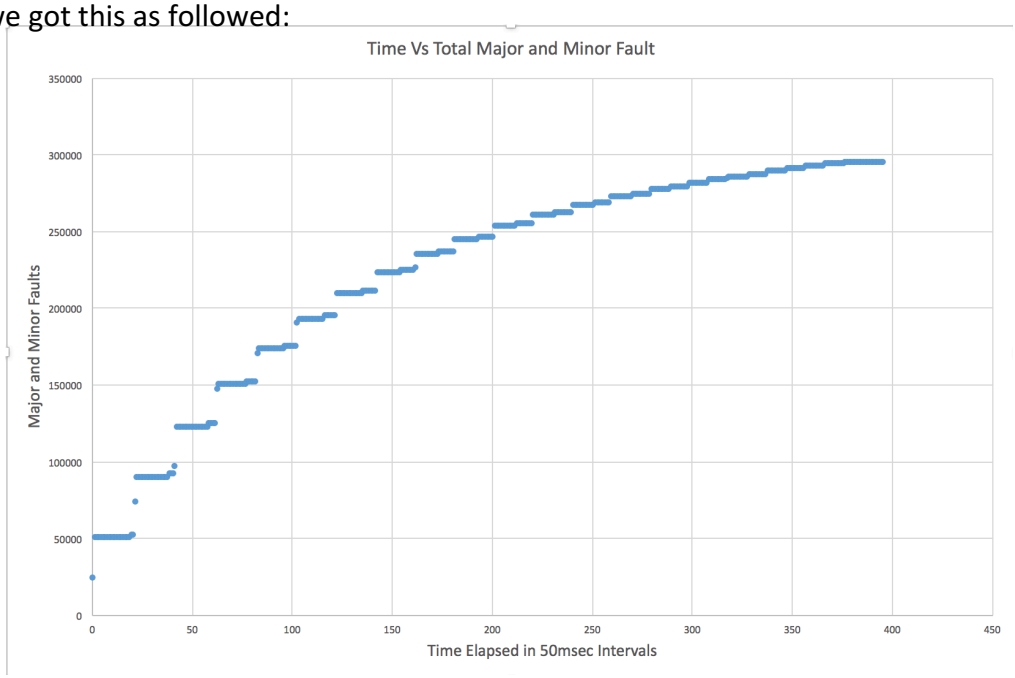
6.1 Case Study

using command: $ nice ./work 1024 R 50000 & nice ./work 1024 R 10000 &
We plotted the graph as follows:



Here we saw that as time progresses, the slope because less and less. This means that the number of new faults decreases as time progresses.

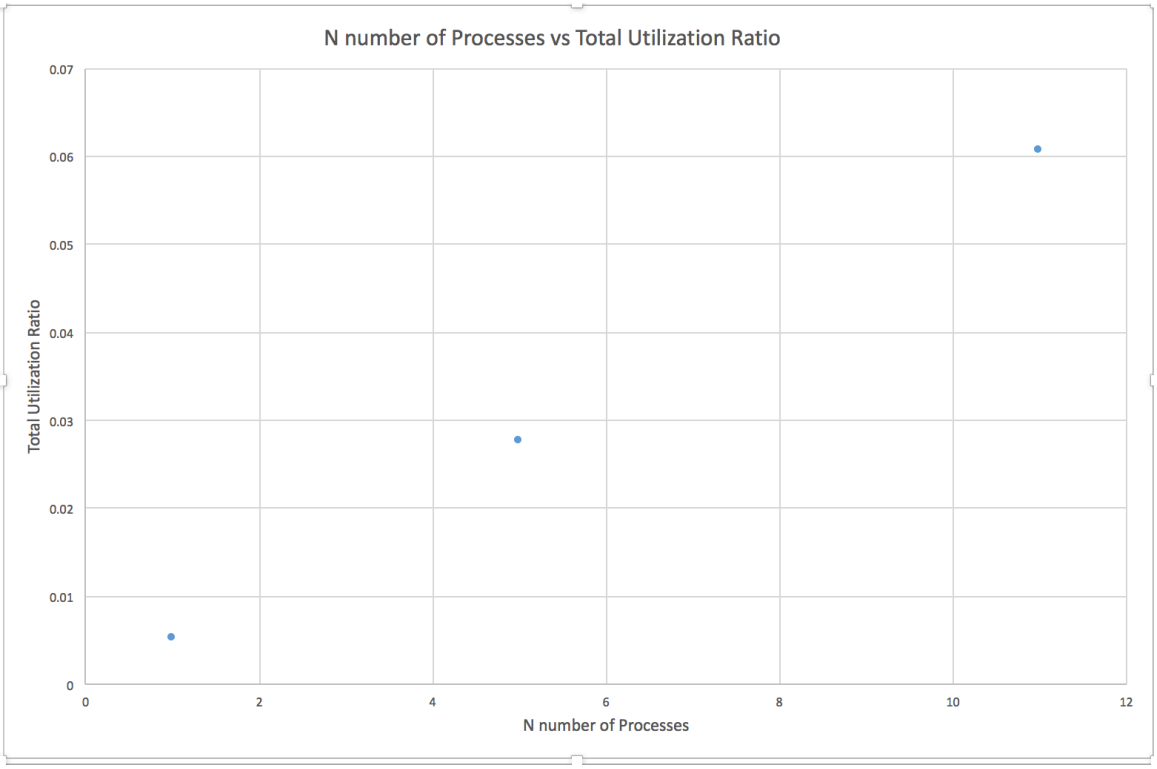
We then did the command: nice ./work 1024 R 50000 & nice ./work 1024 L 10000 &
And we got this as followed:



Here we see the same situation in that the rate of new faults decreases as time progresses. The differences between this and the previous one is that we used a Locality-based access. This reduced our overall total amount of major and minor faults in comparison. One thing to note is that it took the same amount to finish our calculations in this in comparison to the previous command.

6.2 Case Study

Running the command: nice ./work 200 R 10000 & (N number of times) produced the following graph for running it 1, 5 and 11 times.



We can see that the total utilization, which we calculated from adding each interval's total stime and utime divided by the difference of ending time and starting time of the provided user work function, increases linearly. We see that as you increase the number of processes to execute at the same time, we see a higher utilization rate of the CPU. This makes sense because with more processes, it contributes more "stime" and "utime" to the summation at each interval. One thing not shown in this graph that we saw is that as you increase the number of processes, it decreases the amount of time for our work function to finish.