

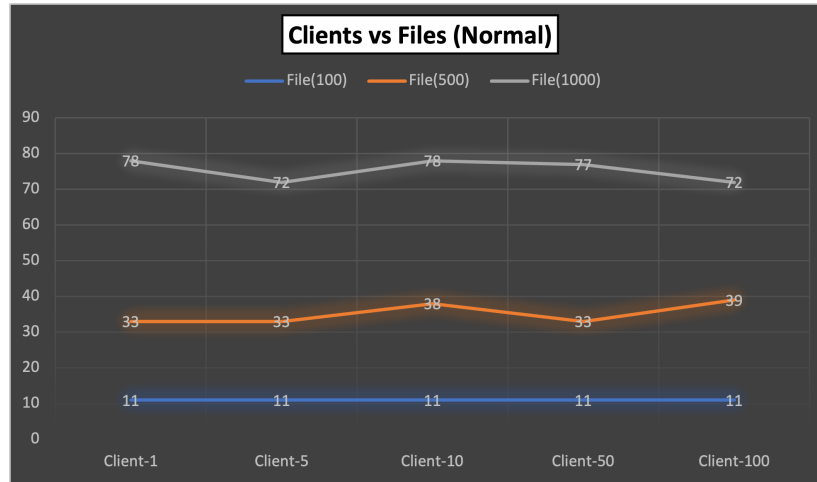
PROJECT 3

HENIL VISESH SHIVAM

November 2022

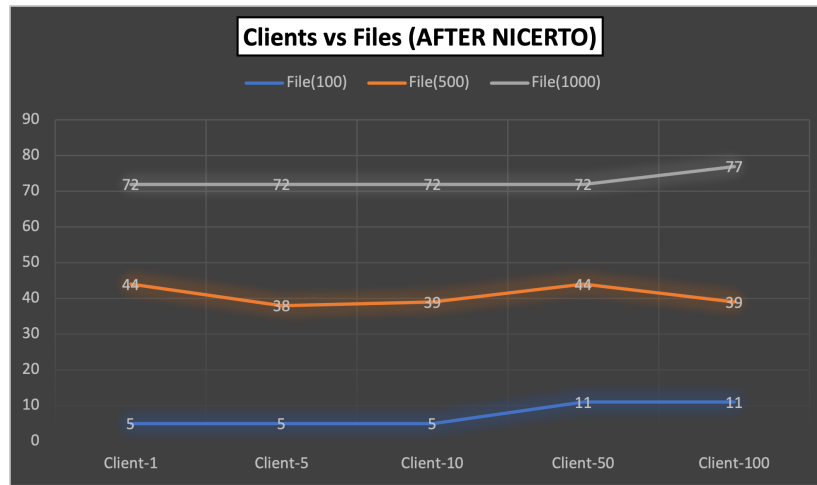
1 Analysis of Data

(I) In this section, first, I imported the line graph, the values in this graph we have taken from the last project; in this graph, the x-axis represents the number of clients, and the y-axis represents the number of files read by the clients, so here we get the time data in the microsecond view,



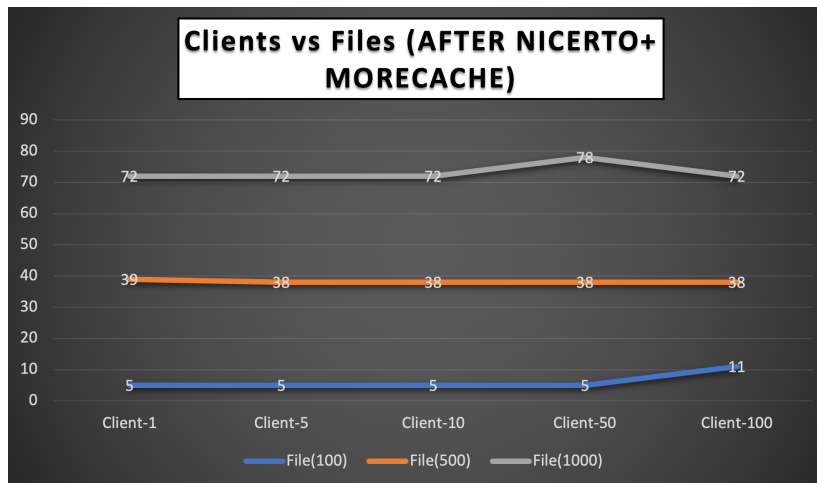
From the graph, we concluded that when a client reads the less number of files, then it shows a constant value in the chart.

(II) Now, In the second part, we have imported the `nicerto()` system call,

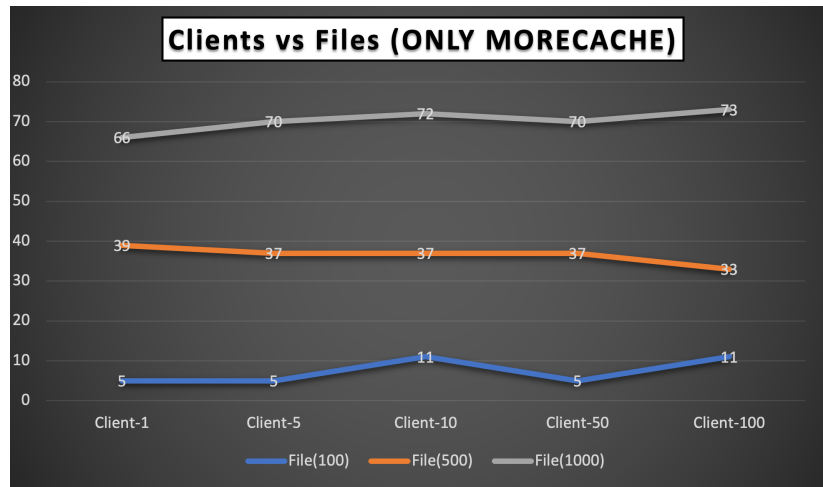


after that we have seen that when there is less number of clients performing the read-and-write operation, at that time, the time requirements are constant for the few numbers of files, but when the number of files is increased at that time, there is a fluctuation in the middle part.

(III) Now, In the third case, we do both operations together, like changing the priority and performing the more cache operations; then we found that there is minimal difference between time, which is used during the read-and-write operations, so if we use the same function during the execution time, so it reduces the overall performance time.



(IV) Now, If we increase the number of buffers assigned to the block, then the line graph looks like this,



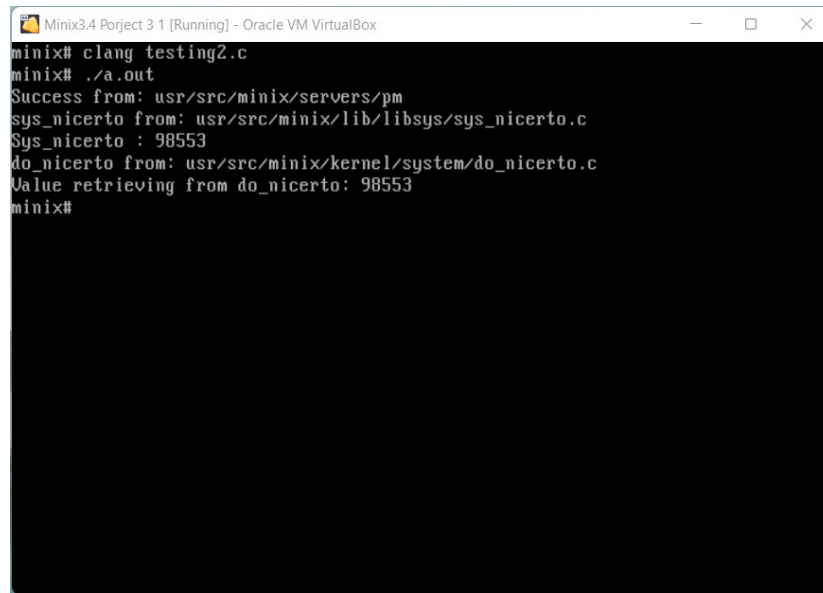
From the line graph, we can see that when there is less number of files, the time here is steady, then it increases suddenly and jumps back to a constant value (like 5) in this case; in the second condition, when the number of files is increased, then there is a steady flow, after then it gradually decreased in the other part.

Here, I attached snapshots of the system call for reference; also, we have attached the code folder for the part of the reference. As we implemented and tested the morezone system call in different systems and discussed it with classmates, we came to know that this system call throws an error when the blocks per zone are changed.

```

Minix3.4 Project 3 1 [Running] - Oracle VM VirtualBox
minix# clang testing1.c
minix# ./a.out
usr/src/minix/servers/pm/moreCache.c
Sending request from: usr/src/minix/minix/servers/vfs/request.c
Requested from: usr/src/minix/lib/libfsdriver/call.c
fsdriver_morecache: executed successfully
fs_morecache from usr/src/minix/fs/mfs/misc.c
minix#

```



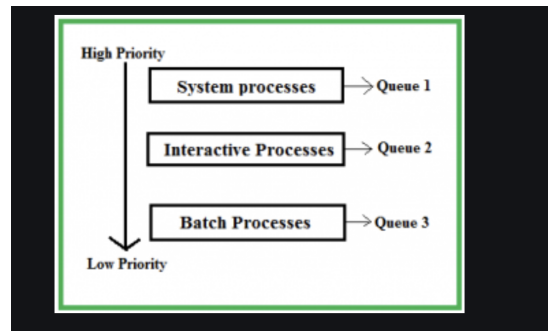
```
Minix3.4 Project 3 1 [Running] - Oracle VM VirtualBox
minix# clang testing2.c
minix# ./a.out
Success from: usr/src/minix/servers/pm
sys_nicerto from: usr/src/minix/lib/libsys/sys_nicerto.c
Sys_nicerto : 98553
do_nicerto from: usr/src/minix/kernel/system/do_nicerto.c
Value retrieving from do_nicerto: 98553
minix#
```

2 Why changing priority levels do or do not result in performance improvement

Priority scheduling means assigning priority to the processes, and the processes with a higher priority are executed first, compared to other processes. During the scheduling process, there are significant changes in the operating system, the tracking status of the processes, the allocation of the processor to processes, and the de-allocation of the processor to processes. Here, the operating system stores all the processes with the unique process ID. When we assign priority to the processes, it depends on the time limit, the memory requirement of the process, and the ratio of average i/o to average CPU burst time.

There are two methods for assigning priority to the processes, and the first one is a static priority; in this method, the priority gives to the processes during the design time, and this priority cannot change, and it remains constant for the lifetime of the process, the second one is the dynamic priority, here we assign the priority to the processes during the runtime. From the above two points, it can be concluded that static priority algorithms are simpler than dynamic priority algorithms.

Here, I attached the two techniques which help improve the performance of processes. The first one is the Multilevel queue method; in this method, the ready queue is divided into separate queues for each class of the process; in this method, we divided the queue into three parts, first is system processes, second is interactive processes, and last is batch processes. <https://www.geeksforgeeks.org/multilevel-queue-mlq-cpu-scheduling/>



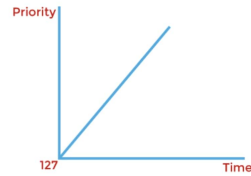
In the system processes, the CPU has its processes for execution. As a result, system processes have higher priority during the execution; the second one is Interactive processes, which means the interactive processes with the user, and batch processes collect the data and process it. <https://www.scaler.com/topics/operating-system/priority-scheduling-algorithm/>

Here, I discussed the drawback of changing the priority results into odd delays, which can lead to hangs and crashes. I explained this point through the example; suppose a higher priority process needs shared resources, but if another process uses the same shared resources, then the second process may wait until the first process can complete its task, so the overall process is called the priority inversion, and most of the time this problem is seen in the system-level processes. <https://www.cnet.com/tech/computing/>

3 How to Fix Starvation Problem

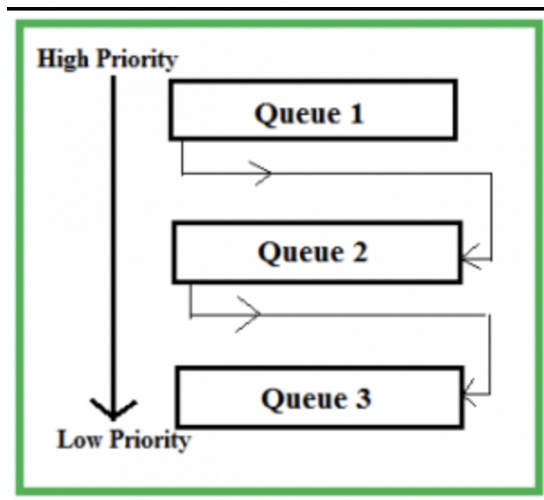
Starvation is a problem when processes with a higher priority are executed over time, and the processes with a lower priority are blocked for an infinite time. To overcome the starvation problem, I proposed three solutions; the first is Aging. In this method, after a certain time quantum period, we increase the priority of the lower process to a higher priority. I explained this point through the example; suppose there is one process with priority number 65 (Here, I consider the higher number as a lower priority); now, this process can wait a long period in a ready state to go into the running stage. Now here I defined the tie quantum period as 5ms, which means after every 5 ms, the process priority decrease by 7. Like, after 5 ms, the priority looks like 58, and the process goes up until either it goes to the higher priority or it can transfer from the ready to running stage. Two methods exist for altering the priority of a process. The first is Increasing the process, and the second is decreasing the process.

The main disadvantage of the Aging technique is that when we change the lower priority process to change the priority at that time, the higher priority process comes so that, the lower priority process may be blocked. <https://www.javatpoint.com/starvation-and-aging-in-operating-systems>.



Another solution is a Multilevel feedback queue scheduling algorithm. With this algorithm, for understanding purposes, we use the three queues for the process; here, processes move around the queues. In the first phase of the execution, the process can enter into any three queues based on priority; in the first case, the background process, such as the visual studio code application, Mx player, then this process is not allowed to enter into the queue 1 and 2. These processes are assigned directly to lower priority. In the second case, we assign a fixed number of units to the queue; let's take an example. Suppose process 1 requires six units to get the CPU, but the first queue executed only four units here. Then this process is transferred into queue two wards.

From queue two, it gets the remaining two units to get the CPU so that we can reduce the starvation of the process. In the third case, the process with lower priority will execute when the higher priority process queue is empty. Also, the lower priority process may be switched during the higher priority processes are coming. The disadvantage of this technique is that a lower-priority process may suffer from starvation when a short process takes the CPU. In contrast, this technique reduces the response time and is more useable than multi-level queue scheduling. <https://www.geeksforgeeks.org/multilevel-feedback-queue-scheduling-mlfq-cpu-scheduling/>



The third solution is starvation avoidance for the priority scheduling algorithm (SAF-PS). This model aims to use the available vacancies present in higher-priority queues for the process, those who wait a long time in the queue. Here

to support zero-copy. Here, I explained the shared buffer; this is implemented using the kiobufs. In this, kiobufs, the user application, allocates the buffer in user space and passes the virtual address to the kernel space. The kernel store the physical address of the page, and this value is stored in the kiobufs.

When a user makes a read system call using the shared buffer, the shared buffer is first cosy checked before being skipped in favor of copy to user. Cosy saves the task's physical page location, which is where the read data is located. Cosy uses the physical address stored in copy from a user if the user requests that any system calls be made in the same shared buffer. I referred to the one research paper for this purpose, so I attached it for reference. <https://www.am-utils.org/docs/cosy-perf/cosy.pdf>

Three configurations have been set up in the experimental set; the first is the VAN, which uses the regular system calls rather than the cosy. To eliminate memory copies between user and kernel space, we employed the COSY and COSY-FAST configurations for the second and third configurations, respectively.

Here, I attached the elapsed and system time with respect to the three configurations.

