*A Report on*

**Dynamic process Prioritization and Monitoring**

*Submitted in partial fulfillment for the award of the degree of*

BACHELOR OF TECHNOLOGY

*in*

*ARTIFICIAL INTELLIGENCE*

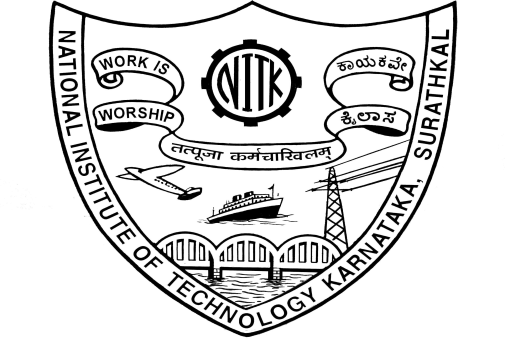
*by*

Sambhav Singh 231AI033

Kishora H Shetty 231AI014

Prajwal Meshram Krushna 231AI019

*IV Sem B.Tech (AI)*



Department of Information Technology

National Institute of Technology Karnataka, Surathkal.

*April 2025*

**CERTIFICATE**

This is to certify that the seminar entitled “**Dynamic Process monitoring and Prioritization**” has been presented by**Sambhav Singh (2310209) , Kishora H Shetty (2310232) and Prajwal Meshram Krushna (2310194)**students of **IV semester B.Tech (AI)**, Department of Information Technology, National Institute of Technology Karnataka, Surathkal, during the even semester of the academic year **2024 – 2025.** It is submitted to the Department in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Information Technology.

Place:

Date: (Signature the Examiner1)

**DECLARATION BY THE STUDENT**

We hereby declare that the Seminar (IT290) entitled Dynamic Process monitoring and Prioritization was carried out by us during the even semester of the academic year 2024 – 2025 and submitted to the department of IT, in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in the department of Information Technology, is a bonafide report of the work carried out by me. The material contained in this seminar report has not been submitted to any University or Institution for the award of any degree.

Place: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Name and Signature of the Student)

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| Sl. No | Chapter | Page Number |
|  | Abstract | 7 |
| 1 | Introduction | 8 |
| 2 | Objective | 9 |
| 3 | Methodology | 10 |
| 4 | System design block diagram | 14 |
| 5 | Implementation | 15 |
| 6 | Results and future work | 23 |
| 7 | References | 35 |

**LIST OF FIGURES**

|  |  |  |
| --- | --- | --- |
| Figure | Name | Page number |
| Figure 1 | Structure of c groups | 12 |
| Figure 2 | Overview of eBPF based tracing | 13 |
| Figure 3 | Test 1.1 | 23 |
| Figure 4 | Test 1.2 | 24 |
| Figure 5 | Test 1.3 | 25 |
| Figure 6 | Test 2.1 | 26 |
| Figure 7 | Test 2.2 | 27 |
| Figure 8 | Test 2.3 | 28 |
| Figure 9 | Test 3.1 | 29 |
| Figure 10 | Test 3.2 | 30 |
| Figure 11 | Test 4.1 | 31 |
| Figure 12 | Test 4.2 | 32 |

ABSTRACT

**LIST OF TABLES**

|  |  |  |
| --- | --- | --- |
| Table | Name | Page number |
| Table 1 | Linux Scheduling Algorithms | 10 |

**ABSTRACT**

This project primarily aims to create a framework where we can run our process and monitor and prioritize processor and memory utilization of the process without using system calls within the process . We prioritize the CPU usage of a process by modifying the scheduling policies and their priority values.

The novelty of the project lies in the prioritization of memory usage . A process can request more memory from OS using system calls . The problem lies when the process doesn’t have these system calls in its code but we want to allocate more memory to a process without changing its code . The parent process cannot call these system calls on behalf of the child and cant allocate more memory to the child without changing the child process’ code .We have achieved this by using Linux Control Groups (version 2) . Cgroups can be used to control the resource utilization of processes . We use cgroups to control process memory allocation from outside the process without changing process code .

To monitor the results of our programs we use eBPF (extended Berkeley Packet Filter ) which is a tool used for attaching user programs into the Linux Kernel . We attach our monitoring programs into specific hook points in the kernel. We use this information collected from kernel to plot the graph of a process’ running time and the information of the CPU and Memory usage of desired processes .

The results of this project include four main findings explained in detain in the results section . The first finding explain the relationship between lru isolates , page faults and page swap outs . The second finding is about the effect of changing niceness values in CFS and priority values in round robin . The third finding is about the effect of increasing the amount of allocated memory on the CPU utilization of the process . The fourth finding is about the comparative analysis of CPU utilization and memory statistics of processes that read and write in memory vs processes that only read from memory .

Apart from the key results we have also observed the working of virtual memory by analyzing the memory statistics of the process .

**CHAPTER 1**

**INTRODUCTION**

**INTRODUCTION**

In modern operating systems, processes need to share limited resources like CPU and memory. Usually, processes manage their own resource needs using system calls. For example, a process can ask the operating system for more memory or increase its priority for CPU time. But sometimes, we need to control these resources for a process without changing its code or adding system calls inside it. This is a challenge because traditional system calls can only be made by the process itself.

This project solves that problem by creating a framework to control and monitor the CPU and memory usage of a process externally, while the process is running, without modifying its code. We do this by combining Linux Control Groups (cgroups v2) and eBPF (extended Berkeley Packet Filter).

For CPU control, we change the scheduling policy and priority of the process using system calls from the parent process. This allows us to manage how much CPU time a process gets without touching its code.

For memory control, we use Linux cgroups. Cgroups are a kernel feature that allows dividing system resources among groups of processes. By assigning a process to a specific cgroup, we can limit or increase its memory allocation from the outside, without requiring the process to make system calls.

To monitor how much CPU and memory a process is using, we use eBPF programs. eBPF allows us to attach small programs to specific events inside the Linux kernel, like context switches and memory faults, without modifying the kernel itself. This lets us track CPU states, page faults, page swaps, and other memory-related statistics for a process in real-time.

Overall, this project provides a practical way to externally manage and monitor resource usage of running processes in Linux, opening possibilities for better system control, analysis, and optimization without requiring changes to existing application code.

**CHAPTER 2**

**OBJECTIVE**

The objective is to create a framework where we can run processes and can monitor and have complete control over resource utilization of all processes , in our case the resources are CPU and memory . Thus irrespective of executable code of a processes , we should be able to control the resource utilization of a process while it is running without changing the code of the process .

The monitoring is achieved by eBPF and the CPU prioritization of process can be done by modifying the scheduling policy of child process via system calls from parent process and passing the pid of child process. Controlling the memory utilization of a process externally is challenging and not possible using system calls from an external process .

Linux provides system calls for processes to request more resources. But controlling a process’s resource allocation externally—without modifying its code or calling system calls from within—is hard, especially for memory. Even shared memory can't help unless the target process is modified, which isn't possible at runtime.

Memory is allocated by the kernel and managed dynamically via swapping. While a root process can use calls like mmap(), brk(), or sbrk() to get more memory, only the target process itself can invoke them. No external process, even with root, can do it on its behalf.

Calls like mlock() can reduce swapping by locking pages, but again, only the process itself can invoke them. Doing otherwise would require kernel modifications.

This project’s core idea is to control memory allocation of a process externally. We achieved this using cgroups, detailed in the methodology section.

Finally we conduct tests to analyze the relation between different memory usage parameters and effect of changing memory allocation on the CPU usage . We have tested how read and write operations in the memory effect the CPU usage of a process . We have also tested the effect of changing scheduling policy and priority on CPU usage .

**CHAPTER 3**

**METHODOLOGY**

Methodology

This project builds a framework where the user can give an executable file as input and run it as a process and be able to monitor and change the CPU and memory usage of a process while the process is running without modifying the process code . We have created a set of programs which together form a framework that can achieve this goal . The project can be divided into 3 parts :

1. CPU prioritization : done by calling system calls for changing scheduling policy
2. Memory allocation control : using cgroups to control memory allocation
3. Monitoring process statistics : BY attaching hookpoints at scheduling and memory allocation related events .

**1.1 CPU prioritization :**

The Linux OS has the following CPU scheduling algorithms or policies :

|  |  |  |  |
| --- | --- | --- | --- |
| Policy | Type | Priority values | Used For |
| sched\_other | CFS | Niceness 19 to -20 | Default for user processes |
| sched\_batch | CFS | Niceness 19 to -20 | Background CPU intensive task |
| sched\_idle | CFS |  | Very low priority task |
| sched\_fifo | Real time | 1 to 99 | Real time non preemptive task |
| sched\_rr | Real time | 1 to 99 | Real time with preemption |
| sched\_deadline | Real time |  | Very important system processes |

Table 1 : linux scheduling algorithms

When a user program for example c program is run on terminal , the default scheduling policy is sched\_other which is CFS policy . The priority values for CFS are called Nice values that range from -20(highest priority) to 19(lowest priority).A user process starts with the default niceness value of 0. A user process with root privileges can use the setpriority() system call to increase its priority on CPU by decreasing the niceness value . The same can be done when a parent process calls this system call and gives the child pid as argument along with the nice value to be set .

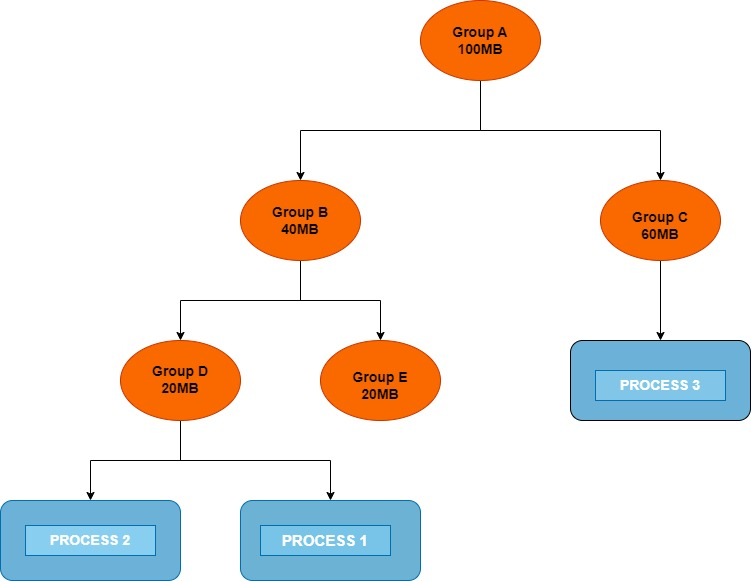
This will achieve some level of prioritization for child process but cant exceed when nice value is -20 . To prioritize a process even further , we can change the scheduling policy to sched\_rr(Real Time Round Robin) using the sched\_setscheduler() system call . After this , we can increase prioritization of child process by increasing the priority values ranging from 1 to 99 but not more than that .

**1.2 Memory allocation control :**

Controlling the amount of memory a process uses from within the process is easy since it can use system calls , but doing the same externally from a different process is not possible via system calls or other methods . The reason for this is explained in the objective section.

Hence we come up with a solution to control the amount of memory allocated to a process from another process , so the even if the code of the process doesn’t have the system call to request more memory , we can allocate more memory to it from another process .

We achieve this using Control Groups . Control groups (version 2 used in this project) in Linux are a kernel feature that lets you limit, account for, and isolate the resource usage (CPU, memory, disk I/O, network, etc.) of a group of processes. The cgroup v2 are created in a hierarchical manner in a tree structure such that a cgroup can have child cgroups which divide the resources of the parent . Processes can exist only in leaves of a cgroup . We can assign processes to certain cgroups and increase the memory limit of a cgroup at the expense of decreasing the memory limit of other cgroups , thus increasing the allocated memory for processes in that cgroup .

Figure 1 : Cgroup structure

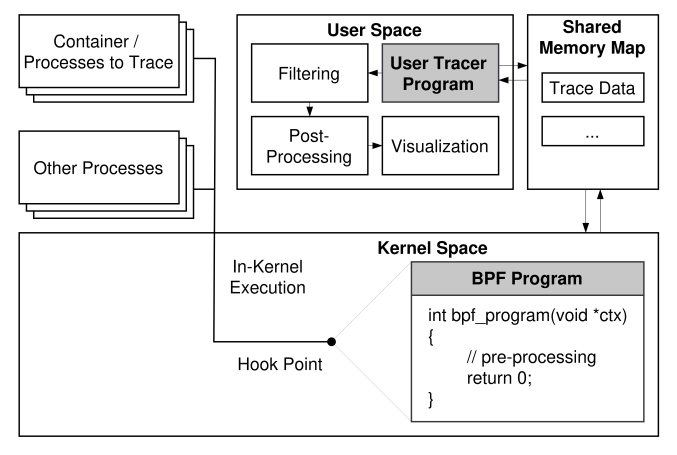
**1.3 Monitoring process statistics :**

We monitor the CPU and memory utilization using eBPF . eBPF(extended Berkeley Packet Filter) is a tool that allows us to attach programs in the kernel code without needing to change kernel code . There are events called as hookpoints which can be system calls or exceptions , before the kernel code runs to service the event , the attached program will run , thus helping monitor the event .

We monitor the percentage of amount time a process spends in the ready, running , blocked and suspended(swapped) state . We do this by attaching eBPF programs at the hookpoints , specifically tracepoints , related to context switch such as sched\_switch(triggered whenever a context switch occurs).

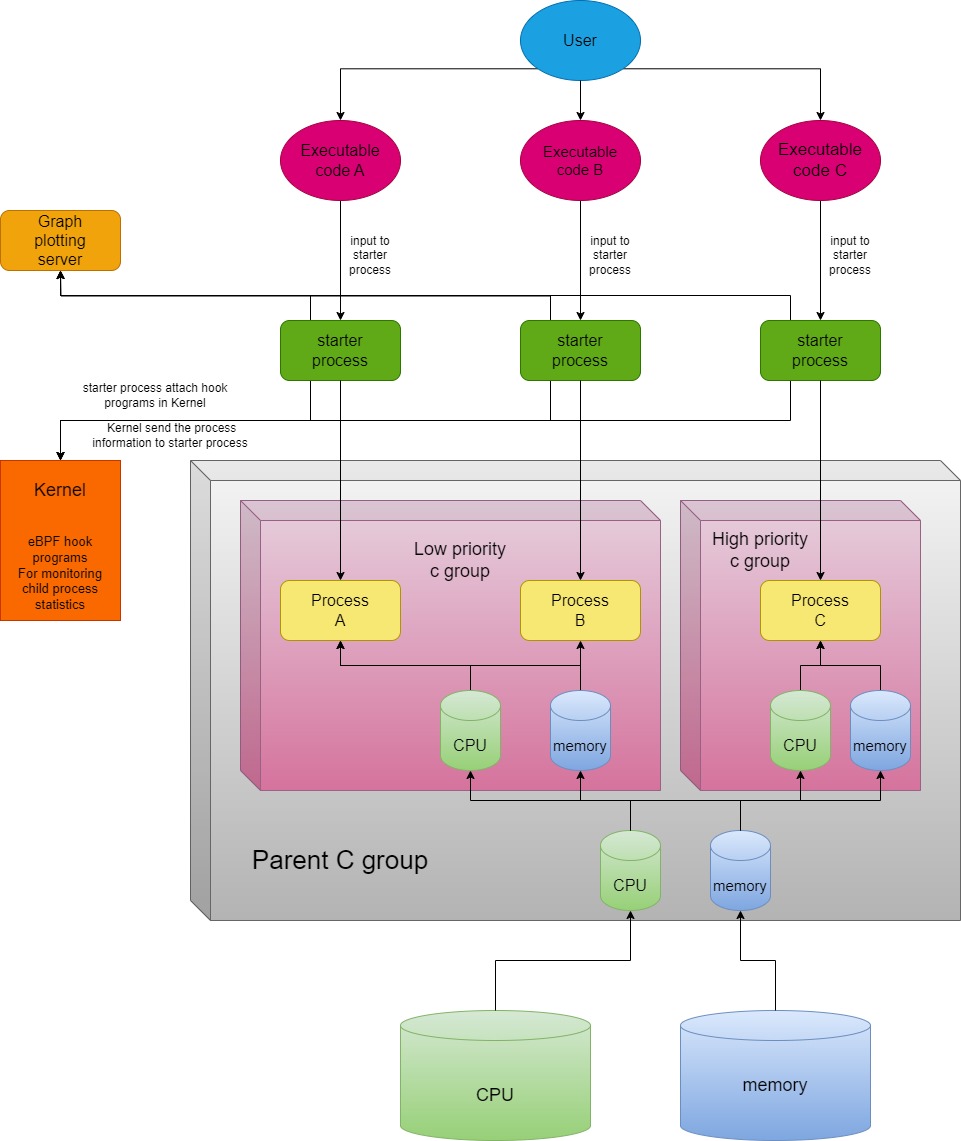
We monitor the number of page faults , page swaps , lru page isolates and fraction of physical memory allocated/ total size of the process . The first three are tracked via attaching eBPF programs in the respective tracepoints like page\_fault\_user . The fraction of memory allocated to a process / total size of process is found out using system calls .

We then plot these results as a time series graph for analysis .

Figure 1 : overview of eBPF based tracing

**CHAPTER 4**

**SYSTEM DESIGN BLOCK DIAGRAM**

****

**CHAPTER 5**

**IMPLEMENTATION**

**IMPLEMENTATION :**

The whole framework works in the following way :

1. The framework is first set up . The setting up includes creation of C groups . The parent C group has a memory limit of 30 MB and a CPU usage limit of 20% . There are 2 child C groups , one named low\_priority and one high\_priority ,each initially having equal share of memory i.e 15 MB. The user can start low priority processes in the low priority C group and high priority processes in the high priority C group .
2. The user can chose whether to execute a process in high priority C group or low priority C group . User gives executable file as input to a process called starter process which calls fork and lets the child process call exec() to start executing the input executable . The child process is now the process that executes the executable file the user had given . For testing purpose , we have created different executable files which require 30 MB of memory to run .
3. The user can chose whether to increase the CPU utilization or memory utilization of the child . If user chooses to increase CPU utilization , the niceness value of the child process is decreased by 5 . When user has chosen to increase priority 5 times , the niceness value become -20 , if the user again chooses to increase CPU usage , the scheduling policy is changed to round robin and each time user prioritizes the process , the priority is increased by 10 , this can continue till priority reaches 99 .
4. Each time user chooses to allocate more memory to the process , the memory limit of high priority C group is increased by 2 MB and memory limit of low priority C group is reduced by 2 MB . This is done till the low priority C group has 0 MB memory and complete 30 MB has been given to high priority process . Though the results show that below 1.5 MB of memory , all processes in the low priority C group are killed.
5. Two processes will run to plot the CPU and memory usage statistics for all processes running in the framework .

The methodology is divided in a step wise method from start to end . The steps are :

1. setting up cgroups
2. attaching eBPF programs
3. controlling CPU usage
4. controlling memory usage
5. setting up a graph plotting server
6. testing and analysis

**5.1 Setting up C groups :**

We create a bash script to create a parent C group with memory limit set to 30 MB and CPU limit sent to 20% . This ensures all processes in the C group must compete for the same 20% CPU and 30MB allocated to them . The parent C group is divided into 2 child subgroups : low priority C group with memory limit 15MB and high priority C group with 15 MB memory limit. The user can chose to allocate more memory to high priority C group by giving input to the starter process , in which case the starter process increases the limit of high priority C group by 2 MB and reduces low priority C group limit by 2 MB , thus obeying the memory limit of the parent C group which is 30 MB . Thus the total memory limit of both child C groups can never exceed the parent C groups memory limit .

**4.2 Attaching eBPF programs :**

We monitor the following information about our process :

1. fraction of time spent in ready state
2. fraction of time spent in blocked state
3. eBPF programs do the job of monitoring and storing CPU and memory information of the process to be monitored , which is the child process of starter process.

For every executable file the user wants to execute , it must execute a starter process and give the executable file as input , the starter process then calls fork and child calls exec() with path to executable file . The starter process uses the python library BCC (BPF Compiler Collection) to attach programs (written in C)in the kernel to monitor the child process .The starter process writes the pid of the child process in a BPF map to use as shared memory with the kernel .

We use 3 maps for shared memory with the kernel :

1. To store pid ,named target\_pid
2. To store CPU information , named CPU\_counter
3. To store memory information , named Memory\_counter

The tracepoints(hookpoints) and the programs attached to them are :

**5.2.1 TRACEPOINT\_PROBE(sched, sched\_wakeup\_new):**

This trace point is triggered whenever a new process is put in the ready state for the first time . It provides the pid of the process that is put in the ready state for the first time . We use this trace point to record the time the process actually got created and loaded in main memory .

Our program reads the pid of the child process from the target\_pid and checks if it matches with the newly created process id . If it doesn’t match , it does nothing . Else the program initializes and stores the following data structure in the CPU\_counter map :

struct SchedStats {

u64 total\_ready\_time;

u64 total\_blocked\_time;

u64 total\_swapped\_time;

u64 total\_running\_time;

u16 previous\_state;

u64 previous\_time;

u64 start\_time;

};

and the following data structure in Memory\_counter map :

struct PageEventStats {

u64 fault\_count;

u64 swap\_out\_count;

u64 lru\_isolate\_count;

};

It sets the start time to current time and the previous state to ready . Rest all are set to 0.

**5.2.2 TRACEPOINT\_PROBE(sched, sched\_switch) :**

This trace point is triggered whenever a context switch occurs . This trace point provides the following information about the context switch :

1. pid of process scheduled out
2. pid of process scheduled in
3. the next state of process scheduled out

We attach our program that loads the pid of the child process from the map and checks the if the child pid matches with the pid of the process that is scheduled out of CPU or scheduled on the CPU . If The process doesn’t match any of them , the program does nothing . Else if the child pid matches the pid of the process scheduled in the CPU , the total time of the previous state of the process is updated and previous state of the process in the map is updated to running . Else if the child pid matches the pid of the process scheduled out of the CPU , the total time in running state is updated and previous state is changed to the next state the process will be in .

Thus the starter process can read the total time spent by a process in each state .

**5.2.3 TRACEPOINT\_PROBE(exceptions, page\_fault\_user) :**

This trace point is triggered whenever a page fault occurs in the user space . This trace point provides the pid of the process that tried to access the page . Our program checks if the pid matched with target\_pid , if matches , increase the page\_fault count .

**5.2.4 TRACEPOINT\_PROBE(vmscan,mm\_vmscan\_memcg\_reclaim\_begin) :**

This trace point is triggered whenever the kernel starts to swap out pages of a process to free up memory space . We update the number of page swaps in the swap\_out\_count if pid matches.

**5.2.5 TRACEPOINT\_PROBE(vmscan,mm\_vmscan\_memcg\_softlimit\_reclaim\_begin) :**

This trace point is almost the same as the previous trace point, the only difference is that this is triggered when the kernel swaps out pages even when enough memory is present , just as a backup for the possibility of high memory requirement in future . This also increments the swap\_out\_count .

**5.2.6 TRACEPOINT\_PROBE(vmscan, mm\_vmscan\_lru\_isolate) :**

This trace point is triggered whenever the LRU algorithm of the kernel moves a page from the active list to inactive list . The attached program here updates the lru\_isolates field if pid matches .

**5.3 Controlling CPU usage :**

The starter process gives a menu to the user to chose the action . If the user chooses to prioritize the process CPU usage , the following algorithm runs :

check the current scheduling policy of child process

if the current policy is sched\_other (CFS) :

check the Niceness value of the child process

if the value is greater than -20 :

reduce the Niceness value of child process by 5 by using the setpriority() system call

else :

switch to real time round robin scheduling with priority 50 using the system call sched\_setscheduler()

else :

check the priority of the child process

if priority less than 99 :

increase the priority by 10 using setpriority() system call

**5.4 Controlling memory usage :**

If the user chooses to prioritize the memory usage of the target process , the c group memory limits are adjusted . The child process runs in the high priority c group , so to allocate more memory to child process we must increase the memory limit of high priority c group .

Since the total memory limit of parent c group is just 30 MB , allocating more memory to child process would require us the reduce memory limit of low priority process .

let delta be the amount of memory we want to increase for child process , in our case

delta = 2 MB

Thus the following algorithm runs :

* increase the memory limit of parent cgroup by delta
* increase the memory limit of high priority c group by delta
* reduce the memory limit of low priority c group by delta
* reduce the memory limit of the parent process by delta .

At the end of the algorithm we the parent c group has same memory limit as before but the memory limit of low priority c group reduced by delta and increased by delta for high priority c group .

**5.5 Setting up a graph plotting server :**

Every starter process is collecting information of their child processes but in order to compare the CPU and memory statistics of all processes in our framework , we need that each starter process must send the statistics about its child process to a common server process where the graph for all process statistics are plotted in the same graph for comparative analysis .

We set up 2 servers , one for plotting memory statistics and other for plotting processor statistics . The servers create a socket and bind to a port , the port number used in this case are 9000 and 9001 . All starter process continuously send their statistics to the server processes through a socket using these port numbers .

**5.6 Testing and analysis :**

The testing was done with the following parameters :

* There are 3 executable codes that we use in the test :
  1. read\_only\_process : generated from a c program . It calls malloc to create a character array of size 30MB . It runs an infinite while loop to access all elements of the array but doesn’t modify the values of the element . Thus only does read operations in memory and no write operations . In the loop there is a XOR operation to create CPU load .
  2. write\_process : generated from a c program that calls malloc to create an array of size 30 MB . It runs an infinit while loop to access all elements of the array and perform a XOR operation on the elements . Thus this process does both read and write operations on the whole array .
  3. no\_memory\_process : uses only 3 variables a,b,c and does an infinite while loop in which it does a XOR operation a = a^b^c . Thus this process only takes 4\*3 = 12 bytes of memory apart from its code section . This process doesn’t require much memory .

The c group structure used is as follows :

A parent c group with CPU usage limit 20% and memory limit 30MB . Two child c groups named low priority and high priority in the parent c group have memory limit 15MB each which we modify in our tests .

All tests have been done by binding the process to the same CPU core .

Four tests were conducted :

1. **TEST 1 :** First we ran the write\_process in low priority cgroup with memory limit 15MB , thus only half of the process memory requirement can be fulfilled . We then repeat the same for the read\_only process and no\_memory process.
2. **TEST 2 :** We run two write\_process processes , one in high priority c group and one low priority c group .The memory limit of both the c groups is 15MB and both hav equal share on the parents CPU limit which is 20% . We then increase the priority of the process in high priority c group in intervals of 5 seconds and measure the change in fraction of time spent by the processes in each state . We repeat the same for read\_only processes and no\_memory processes .
3. **TEST 3 :** We run two write\_processes , one in high priority c group and one in low priority c group . The memory limit of both c groups are initially 15MB . We increase the memory limit of high priority c group by 2MB and decrease the memory limit of low priority c group by 2 MB in intervals of 5 seconds . The effect of this change in memory allocated on the time process spends in each state is analyzed . We repeat the same for read\_only process .
4. **TEST 4 :** We run one write\_process and one read\_only process , once in the same c group and once in different c groups . When both run in same c group , both process must compete for same 15MB of memory . When they are run in different c groups , they have their own memory limit of 15 MB and don’t interfere with each others memory allocation .

**CHAPTER 7**

**Results and Future Work**

## **7.1.Results**

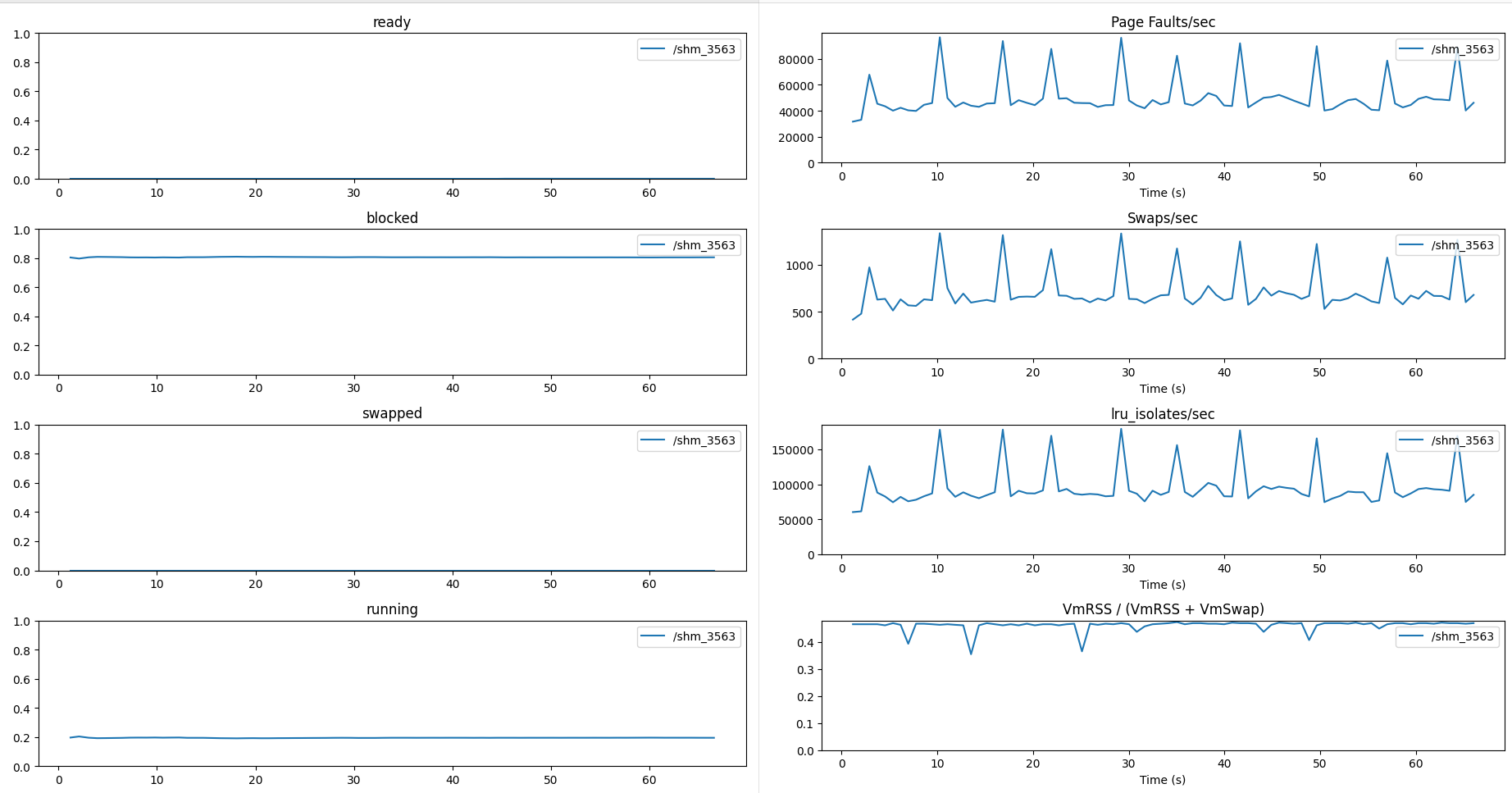
The results of the project present four main findings from each test:

**7.1.1. TEST 1 : The lru isolates , page faults and page swap outs appear to be related**

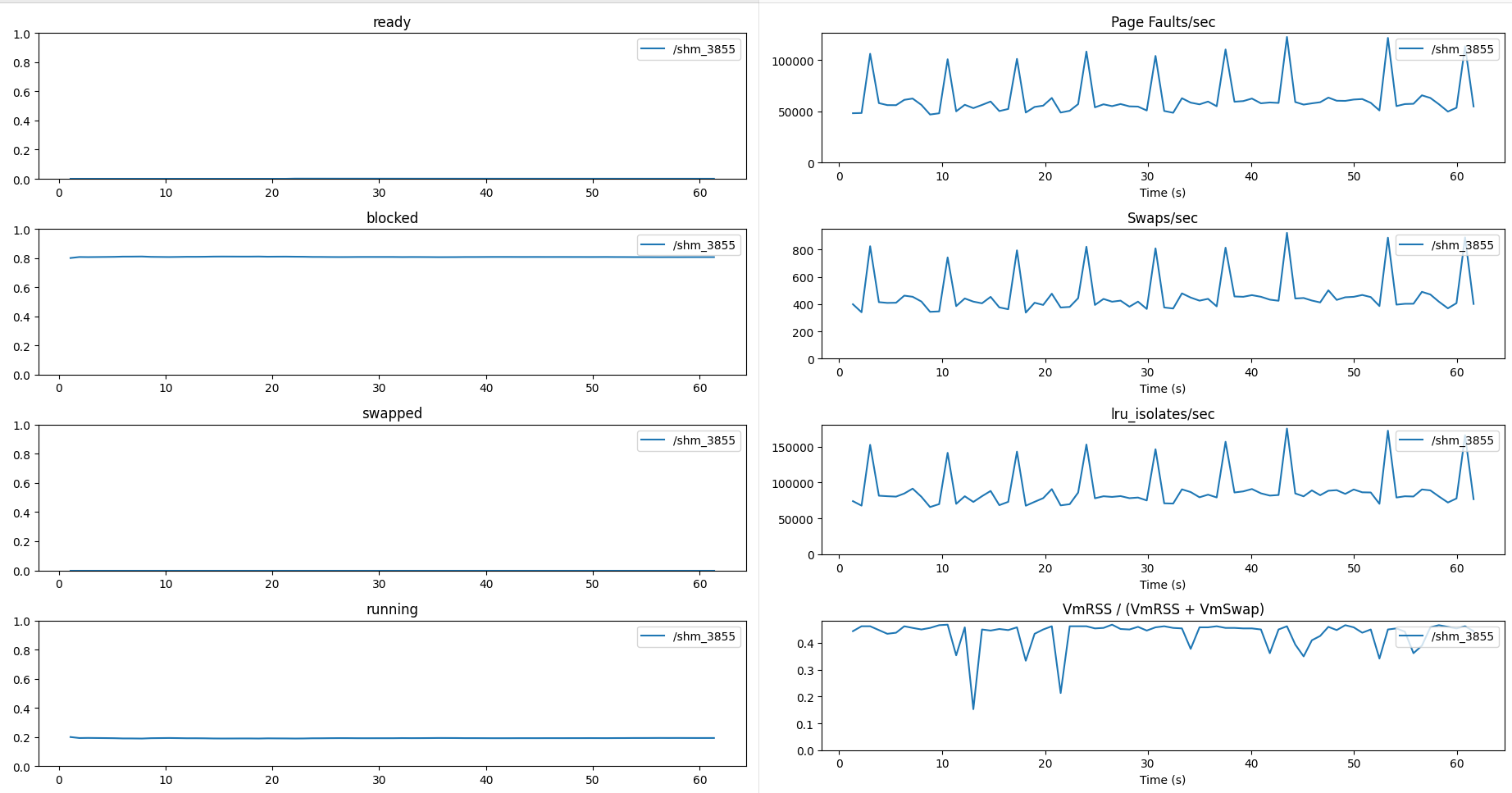
These are the time series graphs of the processes when test 1 was conducted :

here the VmRSS denotes amount of physical memory allocated and VmSwap is the amount of memory that is swapped to disk . This is because of the concept of virtual memory . Thus VmRSS + VmSwap = total virtual memory allocated to the process , which is little more than 30MB for read\_only and write\_process .

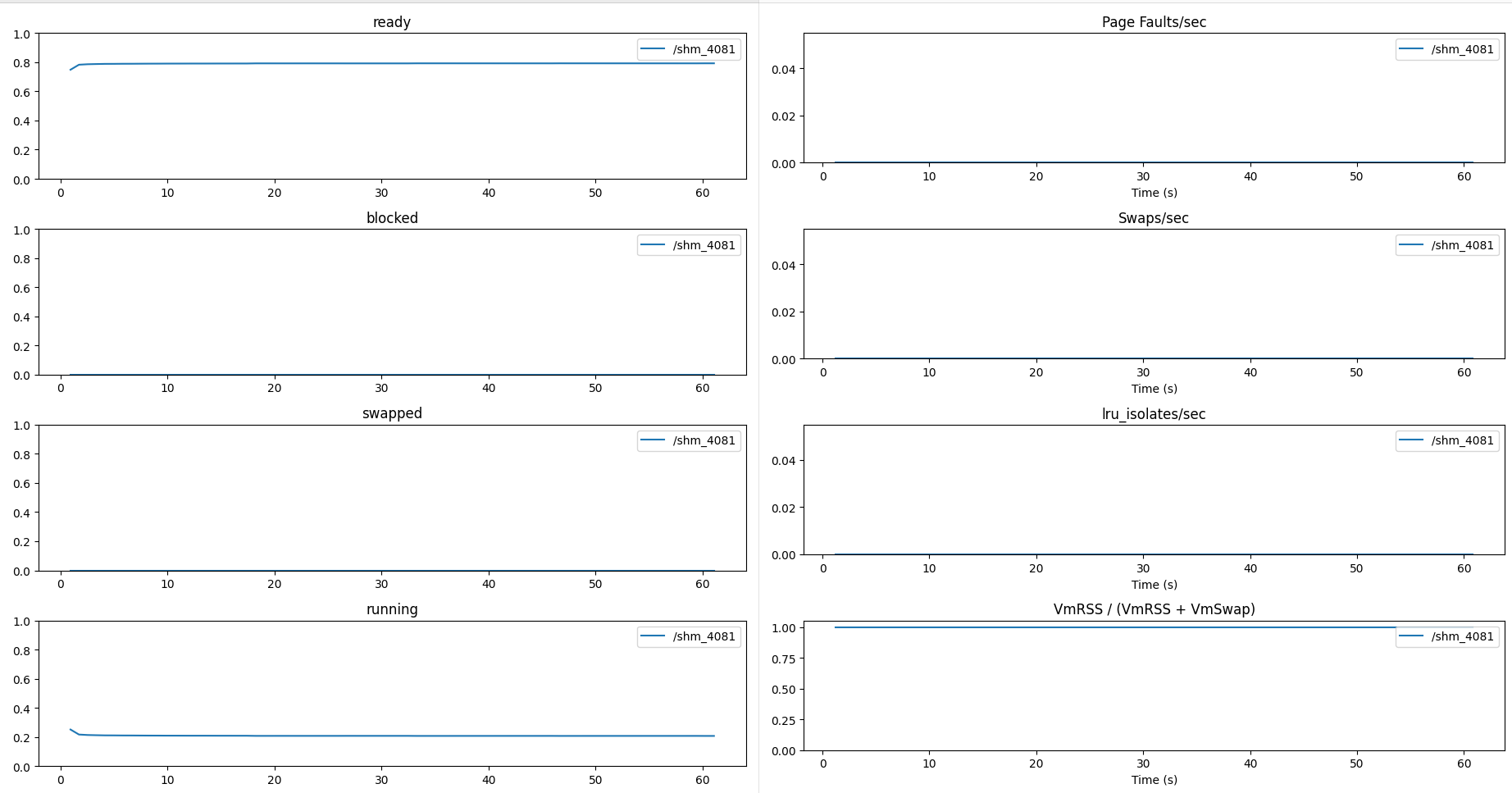
* read\_only process :

Test 1.1

* write\_process process

Test 1.2

* no\_memory process :

Test 1.3

Observations :

The no\_memory process has almost zero page faults, zero page swap outs and zero lru isolates .The value of (VmRSS/VmRSS+VmSwap) is very close to 1 , which indicates that 100% of the process is loaded in main memory . Thus the entire virtual memory allocated resides in the main memory .

For the write\_process and read\_only processes , the page faults , page swap outs and lru isolates all have a high value indicating that memory is not sufficient and pages are not being found in main memory and need to be swapped to accommodate required pages. This aligns with the fact that both processes need at least 30MB of memory but are given only 15MB , which is why the VmRSS/(VmRSS+VmSwap) is close to 0.45 indicating that only 45% of the vritual memory allocated is actually present in the main memory .

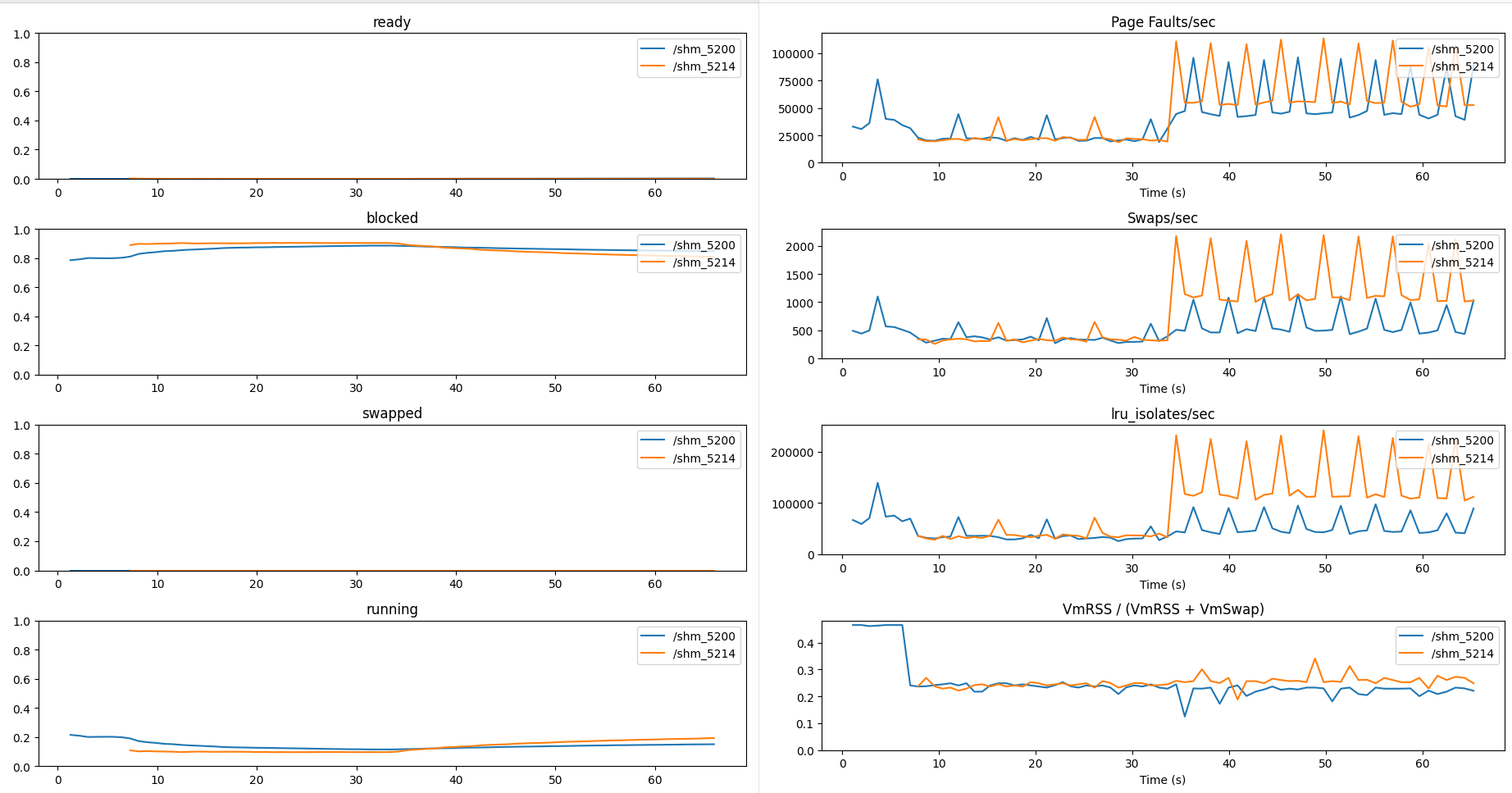
The no\_memory process spends 20% time in running state and 80% time in ready state while the write\_process and read\_only process spend 20% time in running state and 80% time in blocked state . This can be explained by the fact that both write\_process and read\_only process need more memory than allocated which causes page faults , to service a page fault the OS loads pages from disk to main memory which is an I/O operation , thus the process spend most of the time in blocked state .

A major observation is that the page\_faults/sec,page\_swap\_outs/sec and the lru\_isolates/sec all follow a similar trend and look almost similar in the graphs . The explanation for this is not done in the project currently and is a part of future work for this project . But we can infer that a possible relation exists between the three quantities .

**7.1.2. TEST 2 : CPU prioritization of Round Robin scheduling is much higher than CFS scheduling**

These are the graphs of processes when test 2 was conducted :

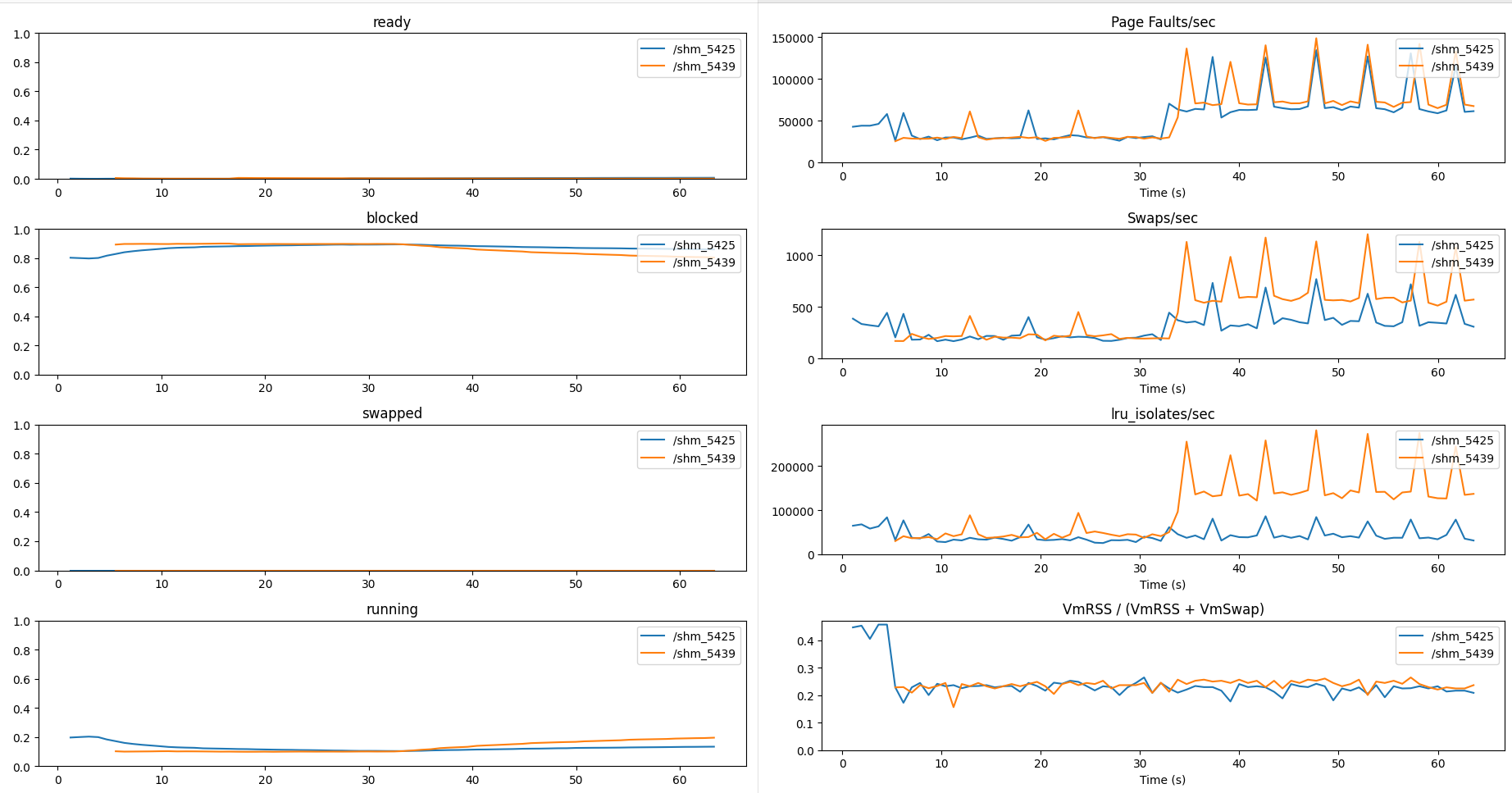
* read\_only process :

Test 2.1

From time = 10 to time = 30 , in intervals of 5 seconds we decreased the niceness value till -20 . However , we see no change in the CPU usage and the memory usage of the two processes .

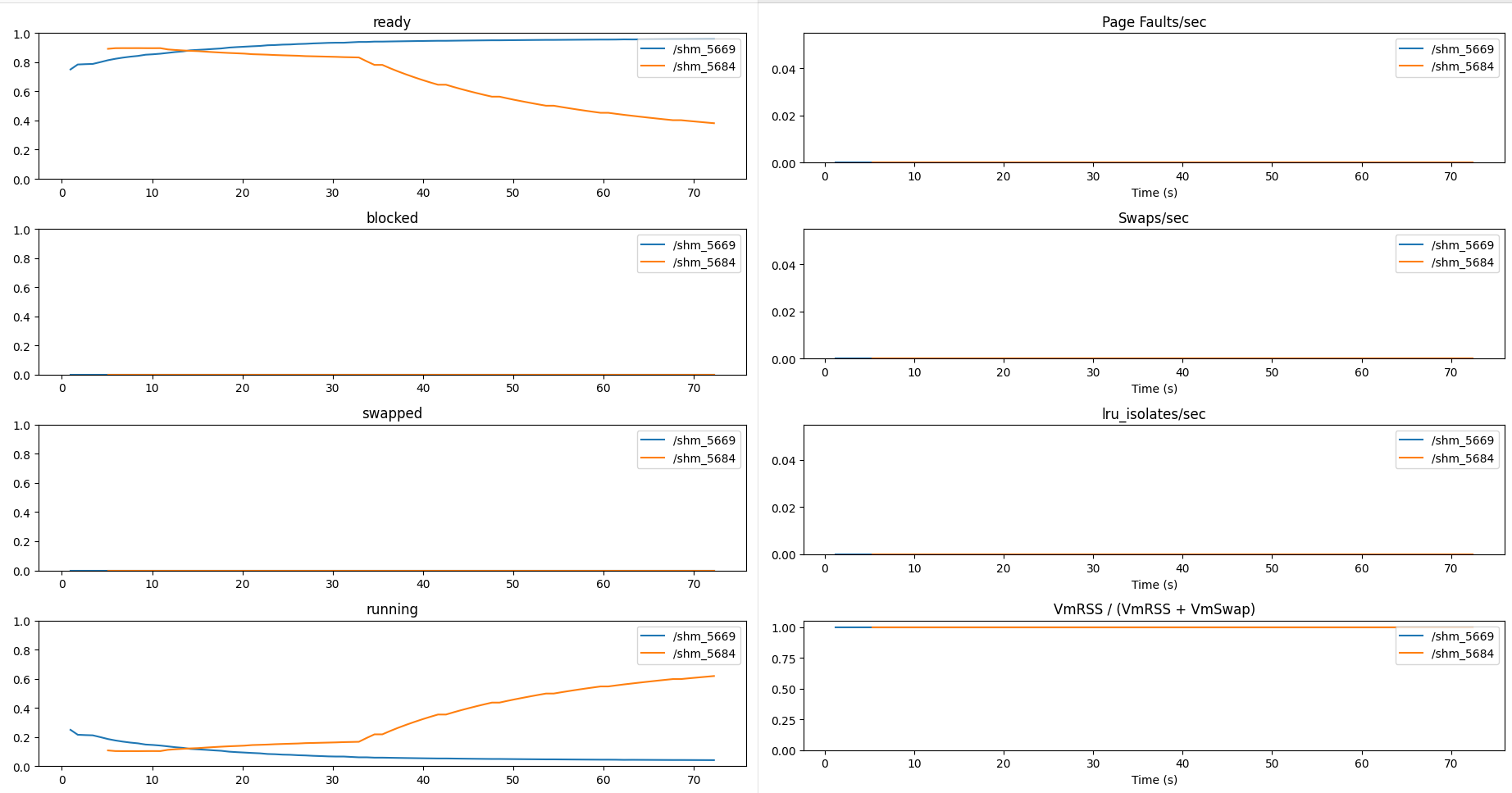
At time = 30 , we change the scheduling policy for the process represented in orange colour (pid 5214)to Round Robin scheduling . We see that the CPU utilization is still the same but the page faults per , lru isoltes and page swap outs have increased for both processes , higher increase is there in the orange process .

* write\_process

Test 2.2

The results are very similar to read\_only processes . Here also the process in orange (pid 5439) is prioritized .

* no\_memory process :

Test 2.3

Till time = 30 , we decrease the niceness value of the process in orange(pid = 5684) till niceness value becomes -20 . We can see a slight increase in the fraction of time the orange process spends in running state . At time = 30 , we change the scheduling policy for the process in orange to round robin . We can a drastic increase in the fraction of time the orange process spends in running state . The value rises from 20% at time = 30 to 70% at time = 70.

Observation :

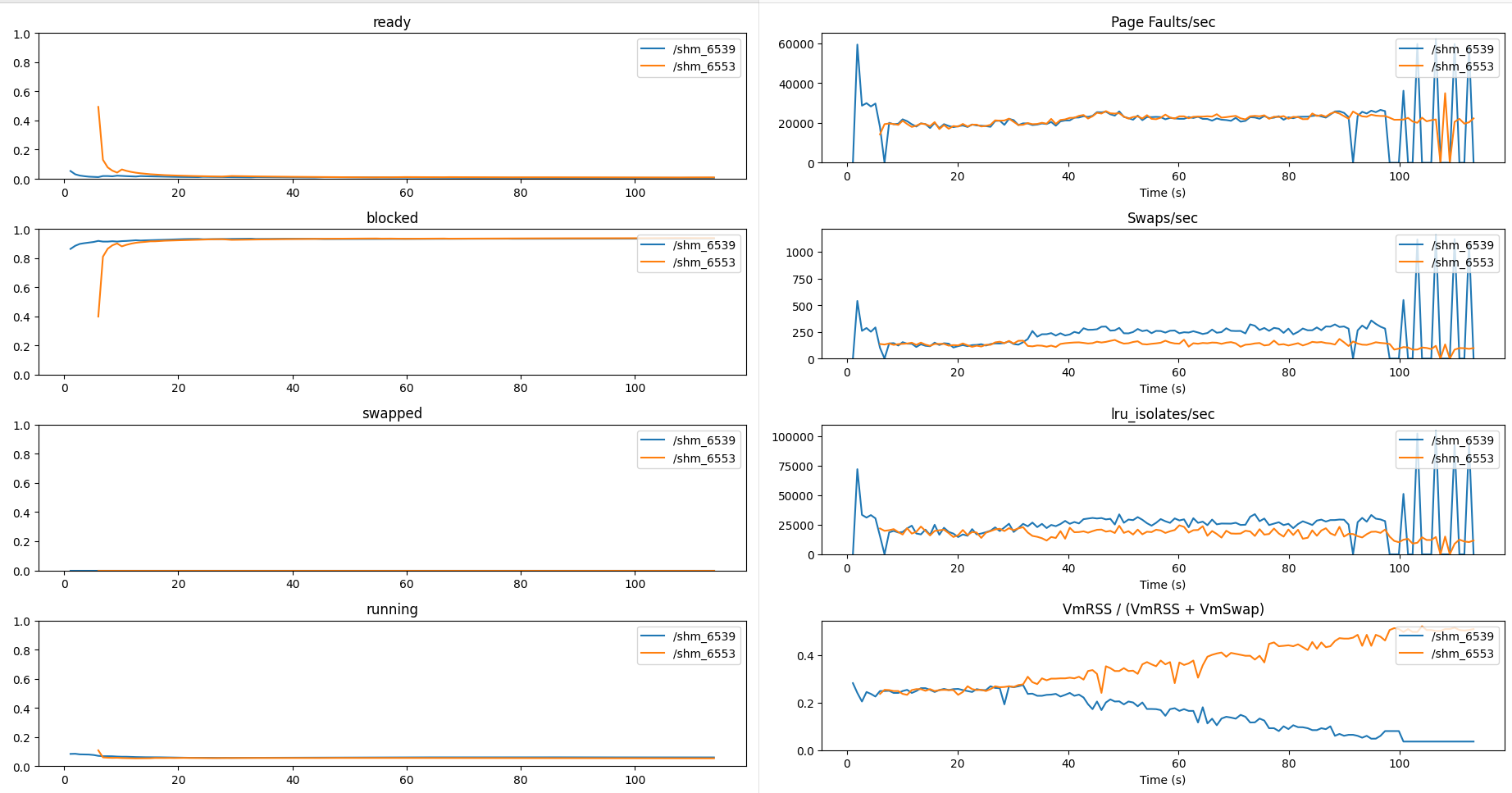
The write\_process and read\_only processes CPU state time share is almost unaffected by changing niceness values and changing from CFS to round robin scheduler . This is because both the process need twice the amount of memory they are allocated , thus most of the time these processes remain in blocked state due to the I/O operation of loading pages and into the memory .

The fraction of time the no\_memory process spends in running state increases by a small amount when the niceness value is decreased but it increases drastically when the process is switched to round robin scheduling . Thus changing the scheduling policy to round robin achieves higher level of prioritization compared to decreasing the niceness value .

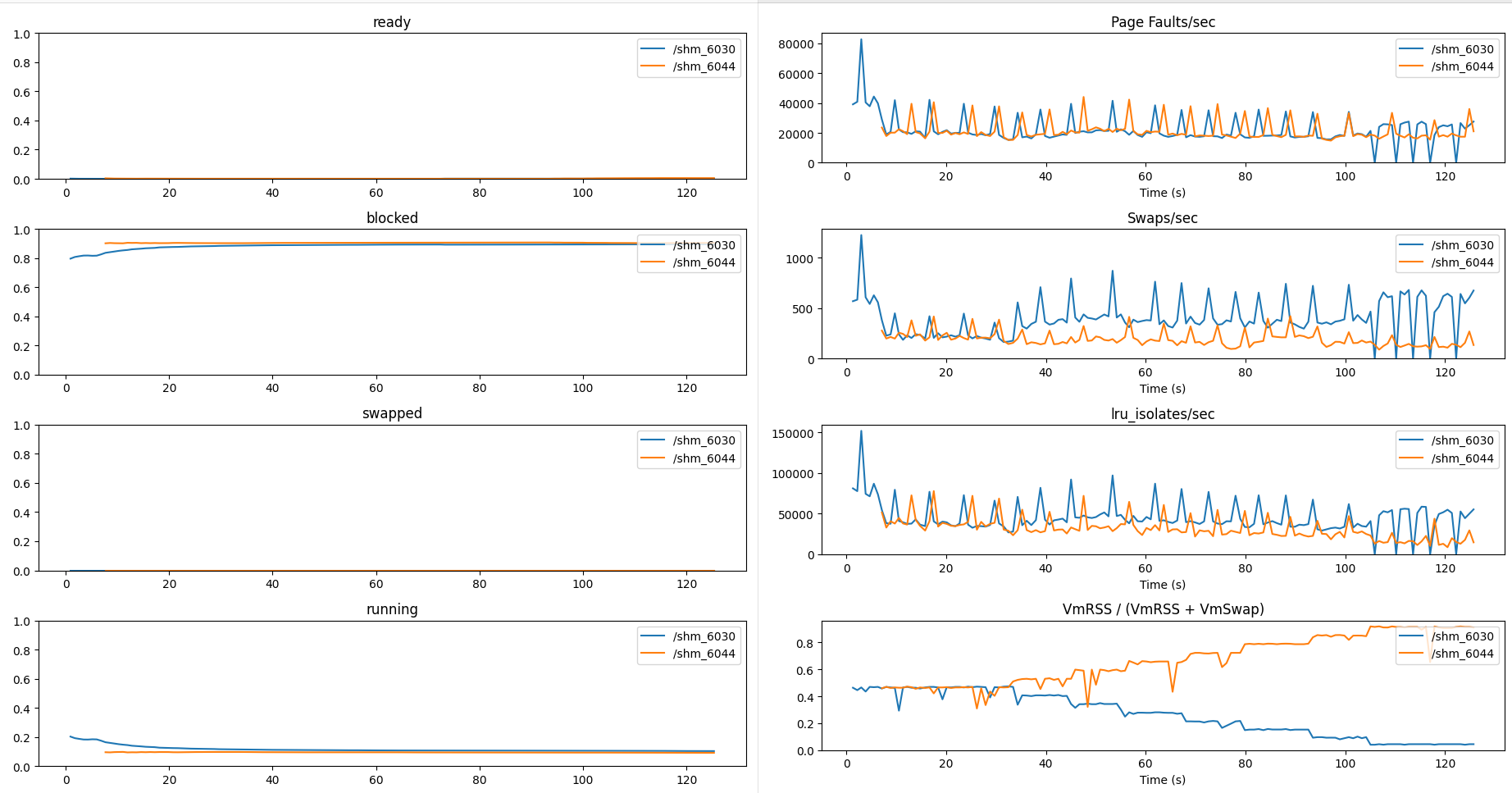
**7.1.3 TEST 3 : Allocating more memory has lesser effect on CPU usage than expected**

The graphs of processes in test 3 are as follows :

* write\_process process and read\_only processes in same c group :

Test 3.1

* read\_only process

Test 3.2

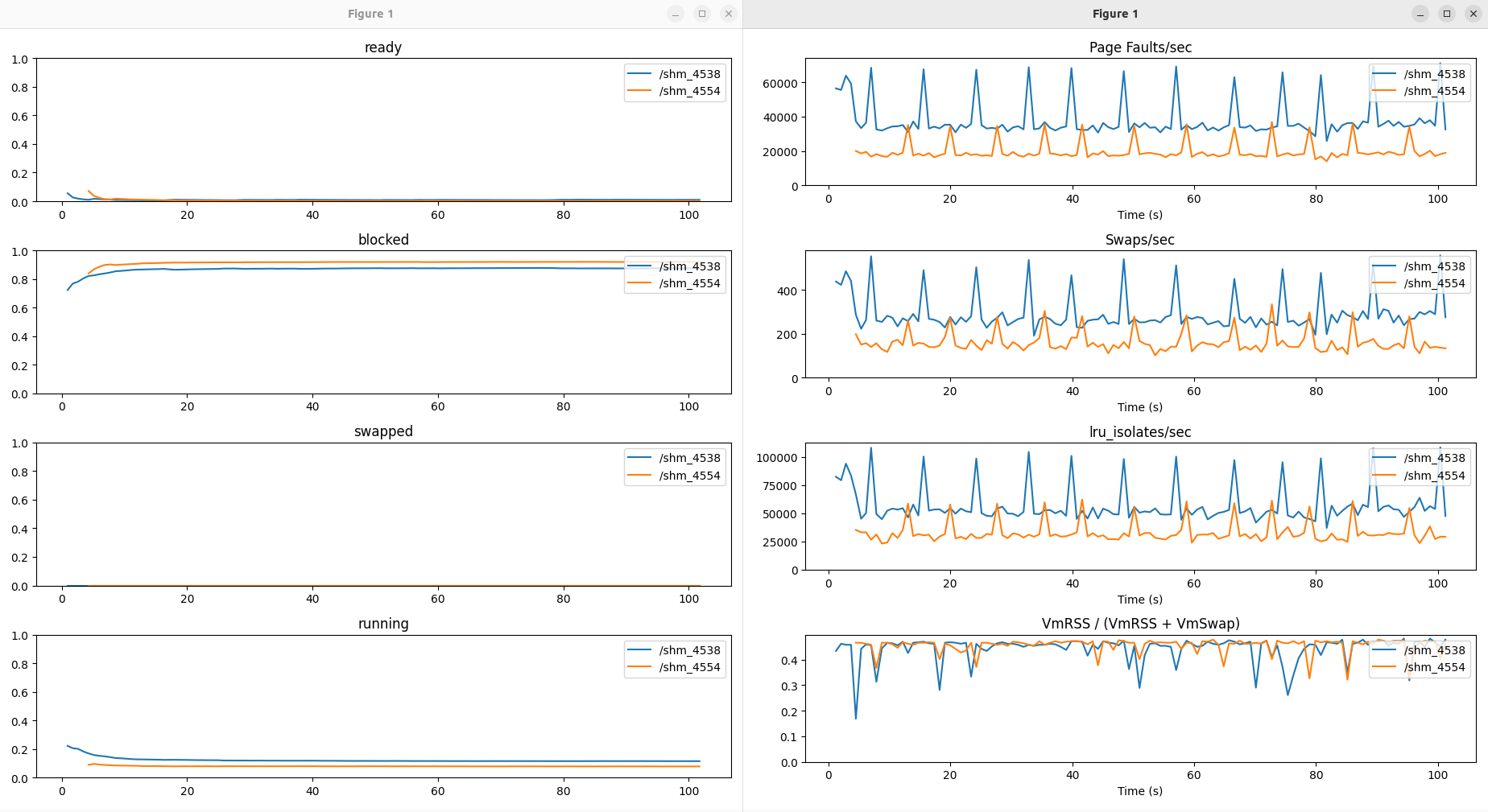
The process in orange (pid = 6044) is in c group high priority and process in blue(pid = 6030) is in c group low priority . Till time = 30 , both high priority and low priority c group have same memory limit = 15MB . From time = 30 to time = 120 , we increase the memory limit of high priority c group by 2 MB and decrease the memory limit of low priority c group by 2MB every 10 seconds . At time = 120 , the high priority c group has memory limit of 29MB and low priority c group has memory limit 1 MB . We see that changing the memory limit of c group has direct effect on the amount of VmRSS/(VmRSS+VmSwap) , which represents the fraction of physical memory upon the virtual memory . The value for orange process increases from 45% to 80% whereas it reduces from 45% to 10% for the blue process . This shows that we have controlled the memory allocated to process .

However , we see that increasing the amount of memory allocated to a process has lesser effect on the CPU usage than expected .

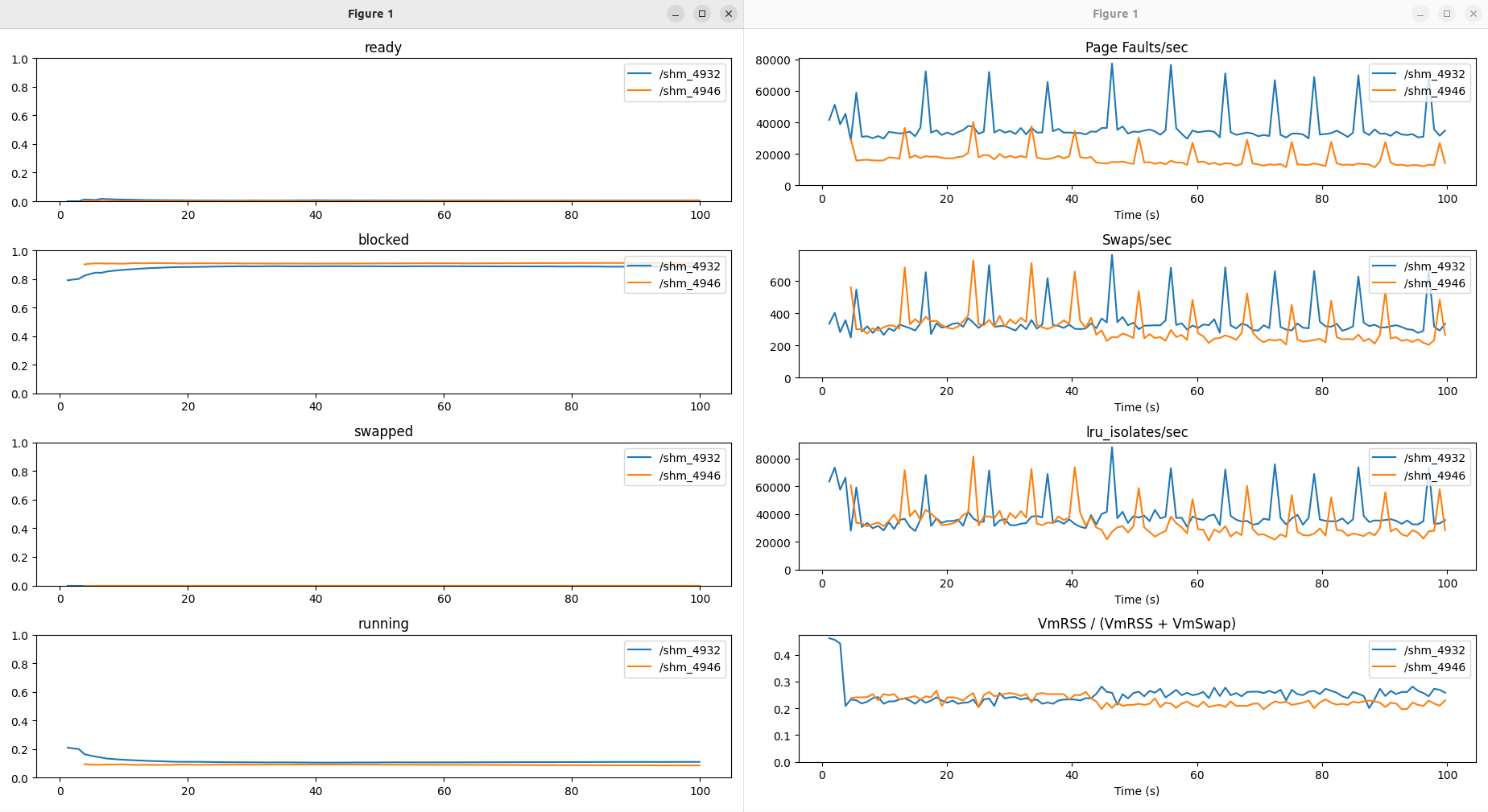
**7.1.4 TEST 4 : Process that reads and writes to main memory has higher page faults than process that only reads from main memory**

These are the graphs of read\_only process and write\_process when run simultaneously in test 4 :

* read\_only process and write\_process run in different cgroups :

Test 4.1

* read\_only process and write process run in same cgroups :

Test 4.2

Observations :

When both processes run in different c groups , each process has 15MB separately allocated to it .Thus 45% of the virtual memory allocated is present in the physical memory for both the process since the virtual memory size for both is 30 MB . The graph shows that the number of page faults per second for the read\_only process is less than that for write\_process process .

When both processes run in same cgroups , only 15 MB is available to each process , hence only 20-25% of virtual memory is actually present in main memory . Here also the graph shows that the read\_only process has lesser page faults than write\_process .

7.2 Future Work

The following are the list of future works that can be done to improve the project :

* Increase the number of information fields for the process .
* Extend the resource control of cgroups to I/O and networking .
* See the effect of swappiness value on the memory usage of process.
* Create a Huffman algorithm to automatically change c group structure and arrange higher priority process in higher level .
* Handling how to modify eBPF hooks if child calls fork .

**REFERENCES**

Books :

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, Operating System Concepts, 10th ed., Hoboken, NJ, USA: Wiley, 2018.

[2]L. Rice, Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security, 1st ed., Sebastopol, CA, USA: O’Reilly Media, 2021.

Research papers :

[1]“Dynamic Process monitoring and Prioritization"  
Authors: Craun, Hussain, et al. (2024)  
Proposes a system that allows for zero-untraced-overhead per-process eBPF tracing by modifying kernel virtual memory mappings to present per-process kernelviews.  
[DOI: 10.1145/3672197.3673431](https://dl.acm.org/doi/10.1145/3672197.3673431)