A Scalable, Flexible and Configurable OOM Killer

Nikhil Abhyankar
nikhilsa@vt.edu

Gaurang Karwande
gaurangajitk@vt.edu

Radhika Nibandhe
radhikan@vt.edu

1 Introduction

Every program running in a computer system accesses memory in order to execute and generate results. The Linux Memory management subsystem is responsible for managing the memory. The work of this subsystem is not restricted just to allocating memory for user programs and kernel internals but also implementing virtual memory and demand paging. When a system is utilizing virtual memory to allocate memory to a user space program, the correspondence between virtual and physical memory is setup only after the first attempt to access the memory is made. Overcommit refers to the practice of giving out virtual memory with no guarantee for the committed amount of physical storage to exist. This is done to avoid the under utilization of the RAM. The overcommit can thus be harmful and cause the system to crash. The kernel is designed to perform memory overcommit handling heuristically by estimating the total available memory thus terminating requests for large chunks of memory. As a result of such a heuristic process, memory overload is possible. The overcommit policy in the kernel is set via the sysctl *vm.overcommit_memory*.

Out of Memory (OOM) is an error message which occurs when additional memory allocation is not possible for programs or the operating system. Thus, Out of Memory is an undesired state in a computer system. An OOM can error causes programs or even the entire computer system to shut down. The computer forcibly shuts down programs or the computer.

The kernels of operating systems try to recover from such an OOM condition by terminating processes. If an application derefernces the pointer and if the system has no memory space available, then the OOM Killer is invoked [1]. Thus, the OOM Killer is responsible to prevent the running processes from unitedly wearing out the host memory. Whenever a process tries to allocate more than the available memory, the process having the highest badness score will receive an out-of-memory signal, which dictates the process to quit or terminate one of its child processes. If neither is done, the process is terminated spontaneously by the OOM Killer. However, the controversial part is that, the process which is terminated may not be the same as the process triggering the OOM signal. It chooses a process that is using large memory but not long lived as a general assumption is made that long lived processes are not affecting the memory allocation process.

The victim selection mechanism of the kernel's OOM killer has been a point of contention in recent times. As it may select a victim process using more memory relative to other processes but at the same time is more critical than other processes running in the background. A very good example of such a process is the training and inference phase of a Machine Learning model. Machine Learning models are becoming more and more resource heavy with the recent success of Neural Networks. Deep Learning networks need a lot of data along with complicated model architectures which have resulted in tremendous need for resources being allocated to them. Currently, GPUs are used to handle this, however, an emerging trend is containerizing the ML models to solve the problem. Containerizing of the ML workflow requires putting the ML models in a container and then deploying it on a machine. With the increase in the number of applications of ML, the users have also gone up. Increased users containerisation has led to users migrate towards common cloud clusters for executing their works[2][3]. This can lead to a problem of shortage of resources on the servers hosting ML workflows requiring heavy computation. In case there is a memory over commit, some of the processes will be killed, often bringing down containers with them.

2 Motivation

The kernel OOM killer works on self-defined heuristics to select the victim process. The function

select_bad_process() chooses the process to be terminated [4]. Put very simply, it selects the process with the highest `oom_score`. How this `oom_score` is calculated has been a highly contended topic and has received fair amount of criticism on various online forums. The kernel calculates the `oom_score` for each individual process, and it can be viewed by running the following command -

$ **cat** /proc/<process-PID>/oom_score

The *select_bad_process*() acts like a debugger going over each running task to calculate the *badness*() which is responsible for killing the process. The badness is calculated as follows, note that the square roots are integer approximations calculated with *int_sqrt*().(figure1)

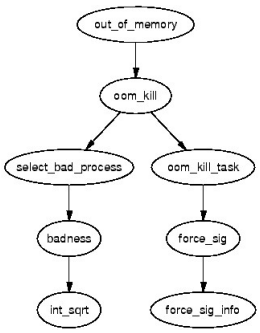

Figure 1: OOM Kill Graph [**4**]

The OOM Killer is invoked when the memory pressure reaches desperate levels. Its job is to kill one of the existing processes. Every process has an OOM $score\_adj$ value that is used to tweak the score relative to that of other processors. The OOM victim selection process is unpredictable as it may end up killing the processes in the user space than the kernel space to maintain system stability.

$$badness\_for\_task = total\_vm\_for\_task/(sqrt(cpu\_time\_in\_seconds) * sqrt(sqrt(cpu\_time\_in\_minutes)))$$ [4]

3 Related Work

Linux users have been occasionally troubled by the misbehaving OOM Killer by killing any random process. Facebook (now Meta) has been one of the biggest critiques of the OOM KIller as it has become a hindrance to their humongous servers running primarily on Linux[1]. However, the OOM Killer cannot be configured at the kernel level and hence a user-space alternative is necessary[5].
As a result, Facebook came up with OOMD [1], which is a OOM daemon for the User-Space with an idea of providing more flexibility to the kernel code. The daemon is triggered when the system is under pressure and can be invoked whenever certain threshold levels are breached. OOMD uses PSI and cvgroupv2 for system monitoring. OOMD takes the required action before the kernel space OOM is active. A custom code can be written to make the system configurable enabling user defined protection rules for safeguarding the running processes.
The EarlyOOM implementation checks the amount of available memory instead of free memory and memory available for swapping. By default, if both the memory values drop below 10%, then the the OOM killer is activated and the process with the highest OOM score is selected to get killed by the OOM killer. A similar implementation along with additional features has been done in Python.[6]
The work proposed in this project is inspired from the above contemporary works and aims to incorporate user-defined policies in the OOM Killer's victim selection mechanism. While we are inspired by memory consumption trends for the training loop of an ML model, we design a system that is modular and scalable to any processes deemed to be critical by the user.

4 System Design

The project is divided into three parts -

1. **Kernel Module**: To monitor system memory statistics
2. **OOM Daemon**: User-space program that handles victim selection and killing
3. **Test Programs**: To test and evaluate the OOM daemon

Kernel module

A Kernel module is an external segment of code that can be unloaded and loaded into the kernel as and when required without requiring to reboot the system. The first step in the project is to gather system memory statistics and present it to the user. A kernel module is thus created to monitor memory statistics information such as Total System Memory, the proportion of Available Memory, Total Swap Memory, proportion of Available Swap Memory and the per process individual memory. This information forms the basis of activating the User-space OOM Killer. The existing implementation raises a flag to summon the OOM daemon.

PSI interface In addition to the memory statistics, the PSI[7] is also monitored to send an activation signal to make a decision on which process to kill. PSI stands for Pressure Stall Information. It provides the information about the resource pressure on the system. The PSI, included in Linux Kernel 4.20 and above, helps to identify the impact memory crunches have on running processes. It provides the statistics about the impact memory shortage has on latency. It can give information about some processes or even the entire system. The PSI data can be found in the `/proc/pressure`. directory. The output is shown in figure 2.

```
1.  ➜  ~ ls -l /proc/pressure
2.  total 0
3.  -r--r--r-- 1 root root 0 Aug 28 07:06 cpu
4.  -r--r--r-- 1 root root 0 Aug 28 07:06 io
5.  -r--r--r-- 1 root root 0 Aug 28 22:22 memory
6.  ➜  ~
```

Figure 2: PSI Information [8]

Pressure metrics along with other tools help detect the resource shortages, killing the non essential processes and reallocating memory to the system. PSI provides fair warnings of resource shortages and warns the user beforehand, in case the memory starts becoming scarce. [9]

The *some* statistic gives the percentage of time some tasks were delayed as result of resource shortages. The *full* metric is indicates percentage of time when all the tasks are delayed.(figure.3)

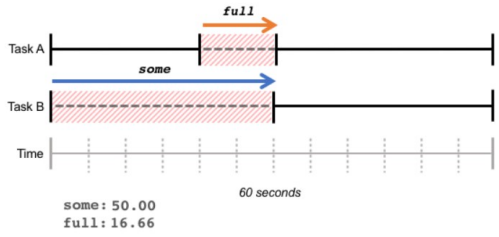

Figure 3: Pressure Metrics - PSI

The fields `cpu, memory, io` give corresponding memory information. The output of `/proc/pressure/cpu` gives the average times processes were starved of CPU. The output for PSI memory is shown in figure 4.

PSI is incorporated to read the `full avg10` i.e. the pressure on the memory after an interval of 10 seconds. Once the pressure percentage crosses the defined threshold of 30%, the OOM daemon is summoned to kill a process.(figure.5)

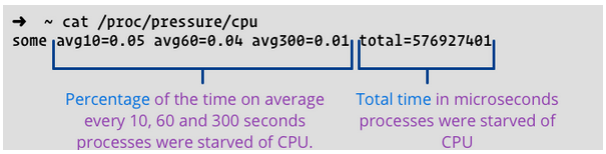

Figure 4: PSI Information [8]

Figure 5: Output of the PSI module

OOM Daemon

A daemon is a computer program that runs as a background process unlike being under direct control of the user. The User OOM Daemon continuously runs in the background, polling the activation signal from the Kernel Module. This User OOM daemon is triggered to kill a process on the basis of memory usage statistics and PSI. The PSI notifies the kernel module to summon the user space daemon when the available memory and PSI falls below a certain threshold. Contemporary works take into consideration the process with the higher OOM score as the one to be killed. The work presented is an extension of previous studies [10], [11], [1]. Our user-space OOM daemon will send either a `SIGTERM` or `SIGKILL` to the selected victim process. In this way the victim process gets an opportunity to correct its memory usage and not be terminated right away.
The work presented focuses on achieving the communication between the Kernel module and user-space OOM killer. The victim selection module of the user-space OOM killer solves an optimization problem with the objective of releasing the most memory pressure and at the same time achieving the best overall system utility. Thus we formulate the victim selection module as a variant of the Knapsack Problem. Dynamic programming techniques such as knapsack have been widely used in process scheduling in the CPU.[14]

Victim Selection Module The Knapsack problem is a problem in optimisation. For a given set of items, each with an associated weight and value, select the items in a collection such that the total weight is less than or equal to the given limit and the value is maximised. It is de-

rived from the idea of filling a sack of fixed size with the most valuable items [12]. In our case, the weight of a process is its OOM score and the value is a user-defined quantity which we call the process utility. The maximum weight threshold is also a user-defined quantity, however this can easily made dynamic by incorporating some heuristics about trends followed by individual process OOM scores.

Test Programs

In this module we run a batch of test processes to evaluate our implementation. Each process has an user assigned process utility. In simple terms, process utility is how much benefit or value we are getting by allowing that process to run through completion. We have created a replica of the */proc* directory as */tmp/user_processes* where information about the test processes are stored. The OOM daemon can access all the necessary information from this location.

5 Experimentation

The first phase of our experimentation work consisted of creating a OOM daemon to kill a process based on user-defined priority. Figure.10 shows the working of our user-space OOM killer when tested with two processes. Here we run two test processes 4904 and 4905 with priorities 100 and 20 and OOM score of 983 and 1150. Smaller the priority more critical is the process. These priorities are assigned by the user. We can then see that even though process 4904 has lesser OOM score, it is killed first as it is deemed to be less critical by the user. Then process 4905 is killed once the OOM condition is recreated.

As our initial motivation has been about taming the kernel's OOM killer for high memory utilization workloads like deep learning, we run a batch of processes consisting of computer vision tasks and some dummy processes. Computer Vision tasks involve complex matrix multiplications and are computationally quite heavy. The idea is to ensure smooth functioning of the ML processes with the heavy memory consuming dummy processes getting killed. We run a DeepFake image generation process, based on an AutoEncoder, called FaceSwap wherein the target face is superimposed on the source face over multiple epochs to create a fake image. The network consists of two decoders, one each for target and source and one common encoder. The network learns the common features of the source and target from the encoder by storing it in a latent variable. This is then extrapolated using the decoder to generate the face back. [13].

More precisely, we execute two FaceSwap processes which are Machine Learning Processes with more than

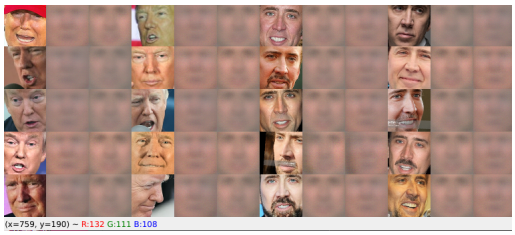

Figure 6: FaceSwap process output

one hundred thousand epochs each and two dummy processes. These dummy processes are designed to continuously allocate memory, hence available system memory and PSI metric quickly reach their thresholds and user-space OOM killer is activated. Dummy process with PID 1609 has utilty of value 8 and OOM Score 916. Dummy process with PID 1610 has utilty of 5 and OOM score 778. The ML process with PID 1607 has utility between 50 to 100 which is randomly generated and OOM Score 746. ML process with PID 1608 has a utility value that lies between 50 to 100 and is randomly generated and OOM Score 673. Figure.7 shows the test processes running in tandem. The first quarter is the dummy process 1609. Second quarter is the first ML process 1607. The third quarter is the dummy process 1610. And the last quarter is the second ML process 1608.

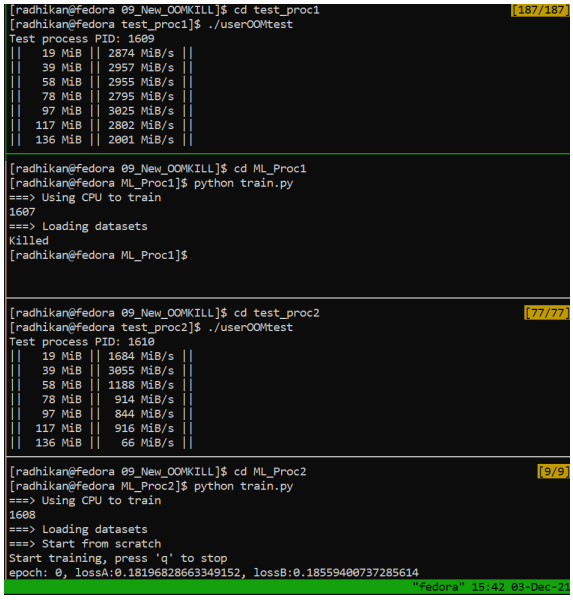

Figure 7: Test Processes

Figure.8 shows the output of the created kernel module. This module continuously monitors the system and outputs the Available Memory and Swap Memory usage and accordingly sends the signal to activate the user-space OOM Killer depending on comparison made with

the threshold values. The current threshold is set at 30% of the total available memory.

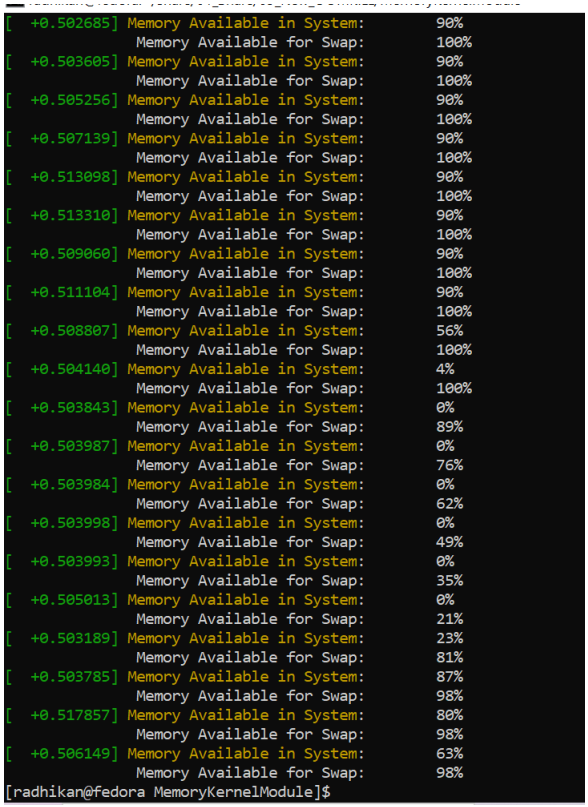

Figure 8: Memory Statistics: Kernel Module

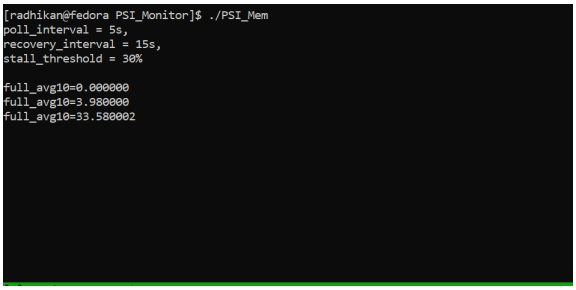

Figure 9: User Space:PSI Implementation

In addition to system memory levels we also incorporate the Pressure Stall Information (PSI) while deciding when to activate the user-space OOM killer. The implementation of the PSI in the system can be seen in figure. 9. The threshold is set at 30% to summon the OOM daemon earlier. The recovery interval the time for which the system forces all other processes to sleep while the existing process is getting killed. The Polling will take place to read the memory from proc file to check the pressure on memory.

6 Results

Figure.11 showcases the output of the User Space daemon along with PSI executed using a bash file `testproc.sh`.The execution includes running of 2 ML processes along with 2 dummy processes. However, as a result of running the processes using a bash script, the output is consolidated, not including the view for all processes separately. The victim selection module within the OOM daemon selects a batch of processes to kill such that it results in the overall highest user assigned utility and releasing the most memory pressure. Initially, we send a SIGTERM signal to the victim process. This gives the victim process to correct or reduce its memory use. We have purposely blocked the SIGTERM signal in all our test processes. Next the system sends SIGKILL to the victim process and the process is instantly terminated. We see that the dummy processes are killed first. Their utilities are 5 and 8, and their OOM scores are 674 and 1132. respectively. They have low utility (value) and a bad OOM score (weight). We set the OOM score threshold to 1000. Hence the victim selection module tries to select victim processes such that the summation of all running processes' OOM score is kept below this threshold. We also see that one of the non-dummy process is killed. Process 1115 is one of the two ML processes. It's utility is between 50 to 100 since the utility is randomly generated. And the OOM score at the time of killing is 906. Hence among the processes of interest it is a relatively 'bad' process. We see that after the first run of OOM Killer, the overall system OOM Score is reduced by 906, at the expense of a utility cost of 5. Then when the OOM condition is recreated the OOM Killer is invoked again. This time it results in the reduction of the OOM Score by 1806 at the expense of a utility cost of 98.

In Figure.8, we can see that the available System Memory and Swap Memory go on decreasing. The System Memory reaches 0% and the Swap Memory reaches 21%. Hence the memory falls below threshold and our OOM killer is invoked. After the execution of the OOM killer we can see that the memory levels increase and are back above the threshold.

Figure.11 displays a PSI Implementation for the pressure on memory. The display shows the PSI threshold which has been set to 30%. This threshold represents the value the processes have been on hold for 10 seconds. The display will print the full avg10 notifying the user and system about the pressure on the system every 10 seconds.

Hence, the user-space OOM killer is successful in selecting and killing processes based upon a user-define custom policy. We are effectively doing Out Of Memory management in user-space and the kernel's OOM

killer is stalled from coming into play. We have in no way changed the kernel's OOM implementation and the kernel's OOM killer will still work if our system fails to successfully release the memory pressure. However based upon our design this scenario will never arise.

7 Project Goals

The initial project goals were -

1. **75% Goal:** Create a module to monitor the current memory usage statistics and send activation signal to user-space OOM killer.
   Status: Completed.

2. **100% Goal:** Victim process selection by user-defined custom policy
   Status: Completed.

3. **125% Goal:** Formulation of the Knapsack problem and testing the User Space OOM killer with a batch of ML workloads and other non-critical processes
   Status: Updated the goal after experimentation.

We initially envisioned that the kernel module would be able to classify if a process is an ML process based upon the memory allocations trends. But after experimentation we realized that this was not the case. We tried monitoring the memory trends - the memory pressure, first and second gradients of the memory allocation, for making the classification. But the logic was too complex and not accurate. Hence the current OOM Killer that we have designed works irrespective of the type of the process. It can be extended to any process the user deems important and doesn't want to be killed by the Linux implementation of Out of Memory management.

8 Tasks Accomplished

1. Study of memory management techniques of the Linux kernel along with the implementation of the OOM Killer in the kernel-space.

2. Creation of a kernel module in order to periodically monitor the system memory statistics to result in generation and delivery of an activation signal to the User space OOM killer, based on comparison with a set threshold.

3. Implementation of the OOM daemon in the User Space to emulate the OOM Killer in the Kernel space. The OOM daemon is invoked before the Kernel space OOM Killer making the process flexible and configurable.

4. Customising the way to kill a process in the OOM daemon based on user-defined process utilities.

5. Implementation of PSI along with memory statistics information to summon the User space OOM daemon.

6. Formulation and implementation of an Knapsack Optimisation algorithm to decide the process to be killed.

7. Creation of an ML based resource (memory) heavy process and dummy test processes for checking the successful execution of the OOM killer.

8. Creation of a bash script to enable running all the processes simultaneously.

9 Conclusion and Future Work

The proposed system monitors the memory allocation and PSI for the processes and selects a process based on Knapsack Optimisation, such that the utility as well as the memory pressure released is maximized. The system successfully customises the killing of a process in the user space as against the kernel's controversial 'arbitrary' OOM killing.

The heuristics for victim selection is based upon the conventional knapsack problem. However this requires the process utilities to be defined by the user. This utilities can be instead computed dynamically based upon various factors such as the rate of change of a process OOM score and its individual memory utilization trends. We plan to investigate this further. Our initial motivation was to design a kernel module which can distinguish between an ML and non-ML workload. This however requires a large enough labelled training data-set. The experiments conducted suggest that availability of large dataset related to memory statistics, memory pressure and memory allocation can facilitate the use of Machine Learning to select a process to be killed. Additionally with the recent advent of Reinforcement Learning, a reward system can be designed in such a way that the OOM killer does victim selection based upon the past user trends. This can circumvent the need for a large labelled dataset. We plan to investigate this more in future work.

10 Member Contributions

- Nikhil:
  Making Memory Tracker Module, Making User Space OOM Killer, Adding Custom Policies, Literature review, Testing with ML processes, Report Writing

- Gaurang:
  Formulation and understanding of the problem, Understanding Kernel's OOM Score, Making User Space OOM Killer, Code Design, Report Writing, Literature Review, Debugging, Knapsack Optimisation for Process Killing
- Radhika:
  PSI information utilization and implementation, Making User Space OOM Killer, Adding Custom Policies, Debugging, Report Writing, Literature Review

Acknowledgement

The authors of the report would like to thank Dr. Changwoo Min and Mr. Christopher Jelesnianski for their guidance and support throughout the Fall 2021 semester for the 5414G: Advanced Linux Kernel Programming course.

References

[1] URL: https://github.com/facebookincubator/oomd.

[2] URL: https://www.bmc.com/blogs/machine-learning-containers/.

[3] URL: https://www.instana.com/blog/solving-the-out-of-memory-killer-puzzle/.

[4] URL: https://www.kernel.org/doc/gorman/html/understand/understand016.html.

[5] URL: https://github.com/rfjakob/earlyoom.

[6] URL: https://github.com/hakavlad/nohang.

[7] URL: https://www.kernel.org/doc/html/latest/accounting/psi.html.

[8] URL: https://unixism.net/2019/08/linux-pressure-stall-information-psi-by-example/.

[9] URL: https://facebookmicrosites.github.io/psi/docs/overview.

[10] URL: https://lwn.net/Articles/590960/.

[11] URL: https://lwn.net/Articles/552789/.

[12] URL: https://en.wikipedia.org/wiki/Knapsack_problem.

[13] URL: https://github.com/Oldpan/Faceswap-Deepfake-Pytorch.

[14] Neelakantagouda Patil. "A Knapsack Based CPU Process Scheduling Using Neelsack Algorithm". In: (2015), pp. 138–145. URL: http://ijseas.com/volume1/v1i7/ijseas20150715.pdf.

Appendix

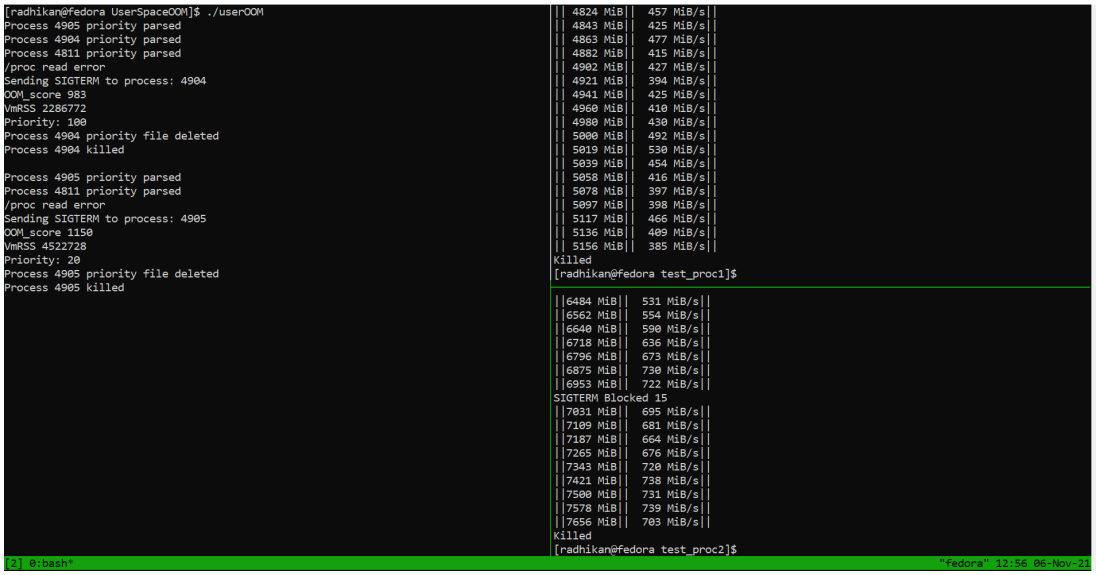

Figure 10: Output of running two dummy processes

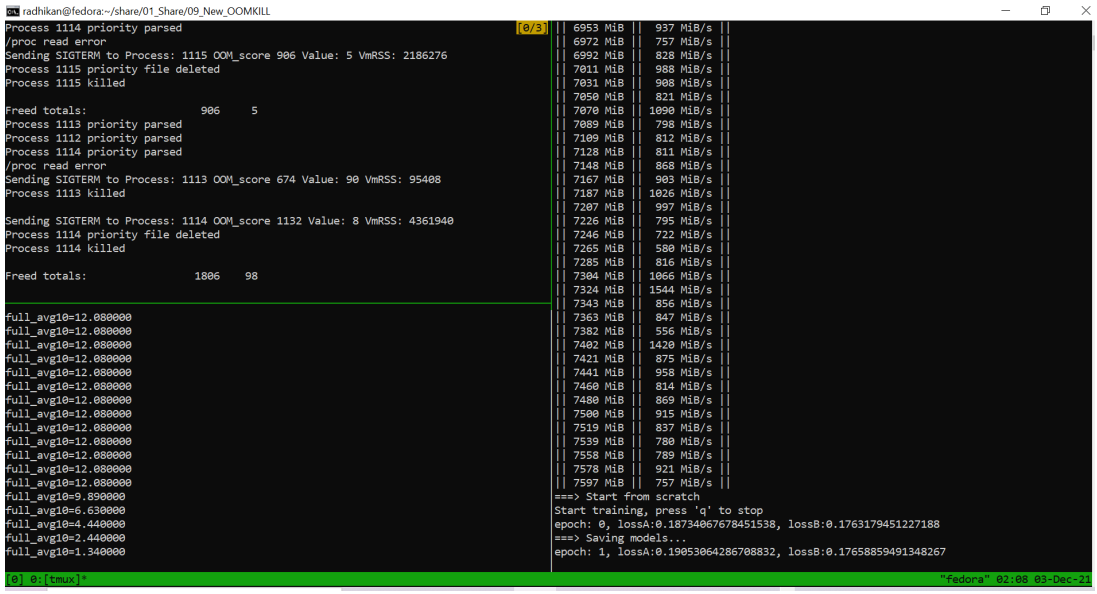

Figure 11: Output of running ML processes along with a dummy processes