CS314: Lab Report
Assignment 3 - Scheduling and UnixBench

B Siddharth Prabhu
200010003@iitdh.ac.in

Devdatt N
200010012@iitdh.ac.in

21 January 2023

1 Part I - MINIX3 Scheduler Swapping

Task: Modify the Minix3 source code such that:

- The string "Minix <roll-no> : PID <pid> swapped in" is printed, whenever a user-level process is brought in by the scheduler.

To do this, understand the working of the scheduler using online documentation of the same.

1.1 Testing and Running

The file `schedule.c` located at /usr/src/minix/servers/sched is modified by inserting the desired print statement in the `schedule_process()` function.

1.2 Screenshots

```
# sh arithoh.sh
Minix: PID 20479 created
Roll Number : 200010012 , PID 237 swapped in
Minix: PID 20480 created
Roll Number : 200010012 , PID 238 swapped in
Minix: PID 20481 created
Roll Number : 200010012 , PID 239 swapped in
Minix: PID 20481 exited
      10.21 real      10.03 user       0.18 sys
Minix: PID 20480 exited
arithoh completed
---
Minix: PID 20479 exited
# _
```

Figure 1: Process creation, exiting, and swapping in

```
# ls
Minix: PID 238 created
Minix 200010003: PID 214 swapped in
arithoh.sh      pipe.sh         syscall.sh
fstime.sh       spawn.sh        workload_mix.sh
Minix: PID 238 exited
```

Figure 2: Process creation, exiting, and swapping in

1.3 Some Remarks

- The files related to Scheduling are in /usr/src/minix/servers/sched, and we observe that the request to start scheduling a process occurs in `schedule.c` .
- As given in the reference textbook, Minix follows a multilevel queue for scheduling processes. Here, we have 16 queues from 0 to 15. To avoid system processes from coming in, we will only print out processes where `rmp>priority` is greater than `USER_Q` .
- A bash script `run.sh` is present in the submitted zip file, which copies the modified source files to the correct directories, and builds the system.

2 Part II - Analysis with UnixBench

Task: Conduct a study of the nature of the benchmarks in the UnixBench suite by analyzing the schedule orders. You may supplement your analysis by studying the source code of the benchmarks, and also studying online documentation of these benchmarks. Use commands such as "time". (Note: refrain from using the graphics benchmarks in the suite.)

2.1 About UnixBench

UnixBench is the original BYTE UNIX benchmark suite, updated and revised by many people over the years. The main purpose of this tool is to provide a basic indicator of the performance of a Unix-like operating system. UnixBench can be used to evaluate the performance of a system when running a single or multiple tasks. This is a system benchmarking tool, not just a CPU, RAM or disk benchmark tool. The results will depend not only on the hardware, but also on the OS, libraries, and even compiler. In this assignment, we will analyze and use this suite for scheduling.

2.2 Initial Remarks

- We will first gauge whether the given benchmark programs for different scripts are I/O-bound or CPU-bound. Then, we will construct workload mixtures and observe the swapping in of processes. The `top` command is used to make these initial remarks.
- We observe that `arithoh` runs more CPU-bound processes. This could be due to the fact that repeated arithmetic operations are rather computationally intensive, and don't have an I/O component.
- The `fstime` script appears to run I/O-bound processes, based on it taking less proportion of the CPU for running. It reads, writes and copies files.
- The `spawn` benchmark goes through 100002 iterations where it forks, and immediately reaps the child that is created. This seems like a memory-bandwidth and process creation benchmark where both the bandwidth of the underlying system and the process creation mechanism of the Unix-based OS is tested. Additionally, this is a CPU bound process.
- The `pipe` script seems to be more I/O-bound; it involves repeated reads and writes to a pipe (which is a mechanism for inter-process communication). It also has less CPU usage, due to its smaller bursts.
- The `syscall` script seems to mainly consist of CPU-bound processes, which does system calls and deals with file descriptors.

2.3 Screenshots

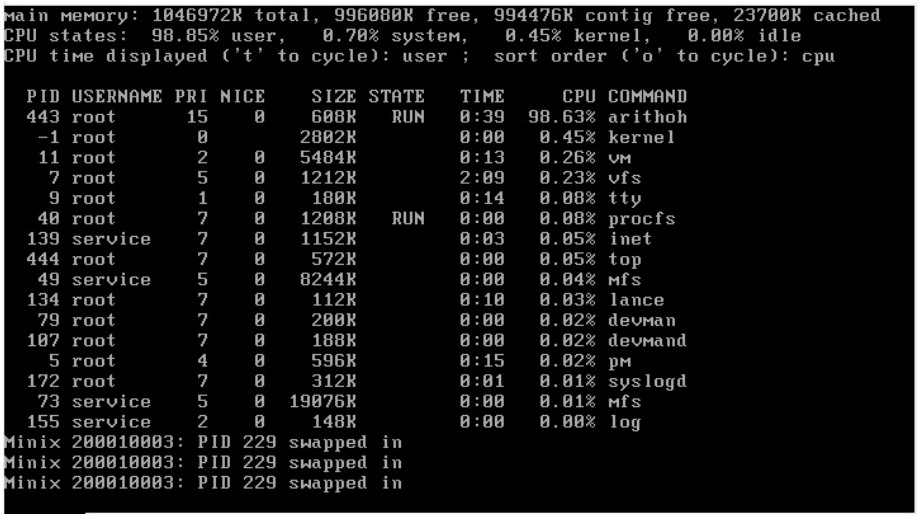

Figure 3: CPU Usage of arithoh

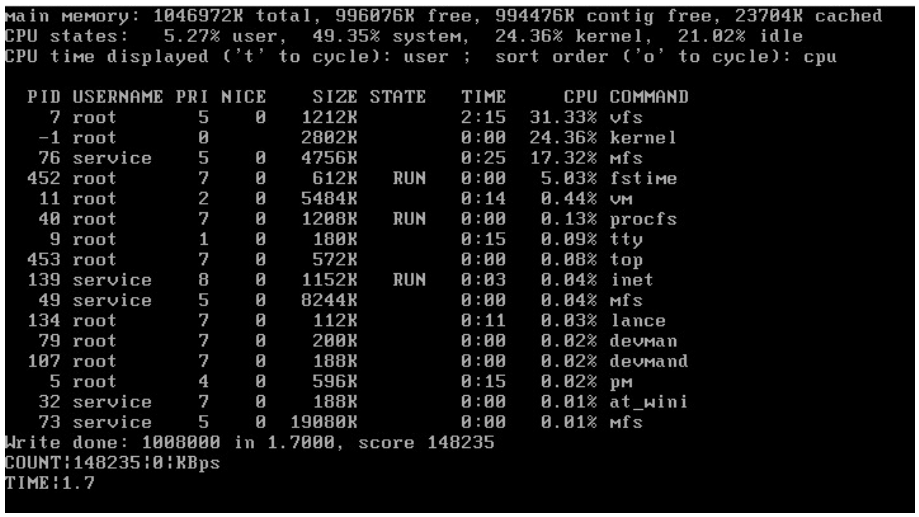

Figure 4: CPU Usage of fstime

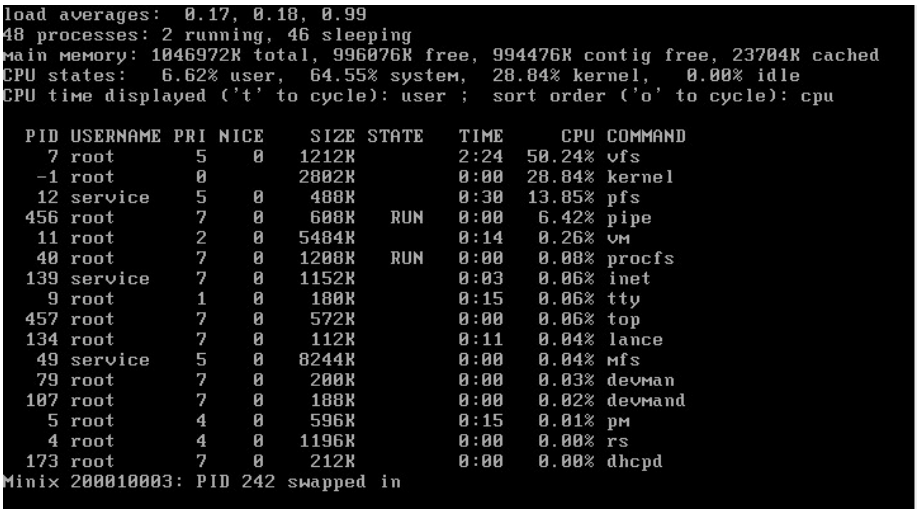

Figure 5: CPU Usage of pipe

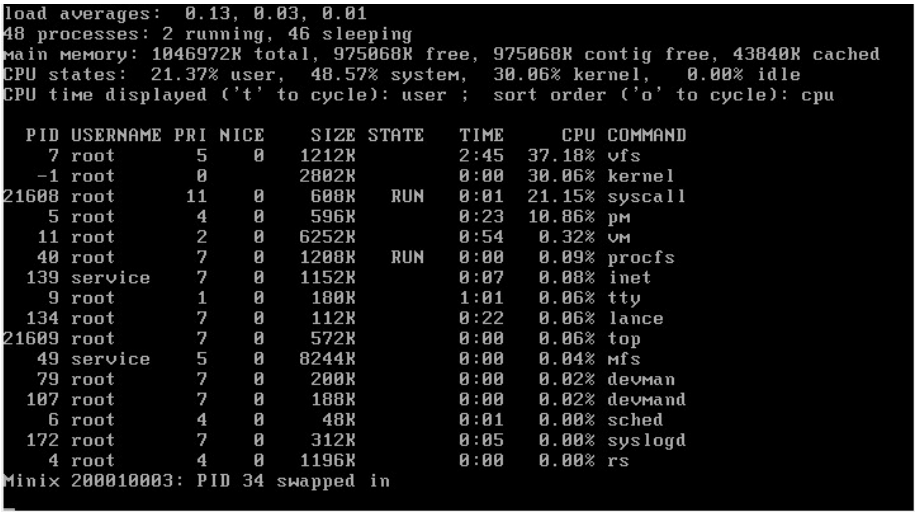

Figure 6: CPU Usage of syscall

2.4 Analysis of workload mixtures

2.4.1 arithoh and fstime

In this, we run a CPU-bound and an I/O bound process to observe the process swaps. The instructions of arithoh are CPU-intensive, while those of fstime are I/O-bound. We observe that arithoh swaps occur in the time where fstime waits for I/O. The code is as follows:

```
#!/bin/sh
./arithoh.sh &
./fstime.sh &
wait
```

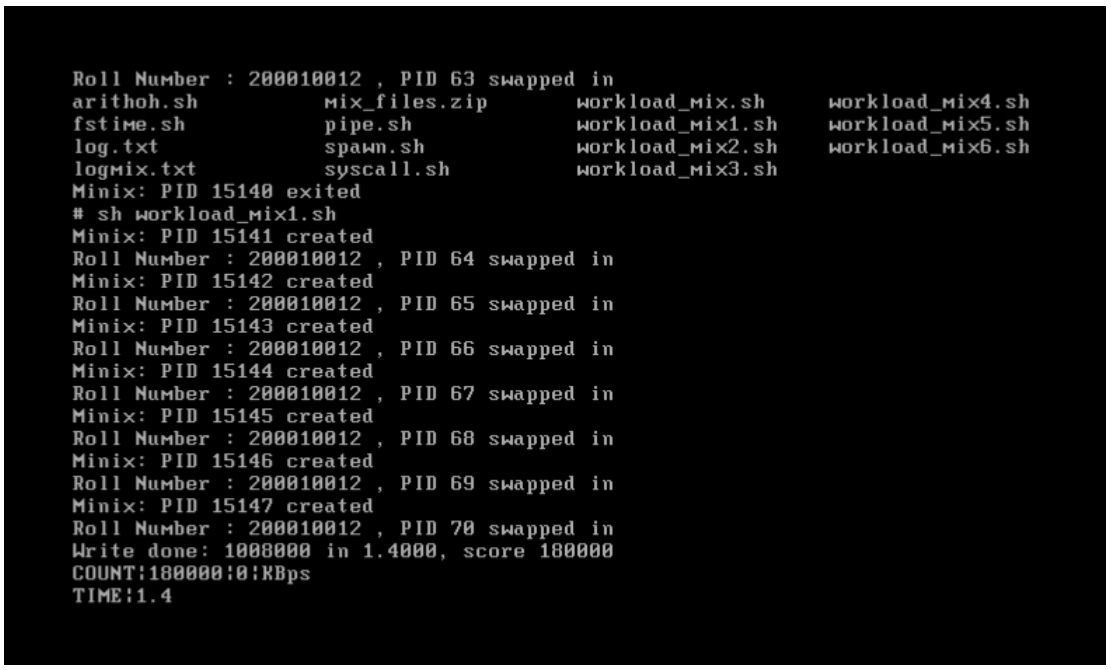

Figure 7: Swaps for workload 1

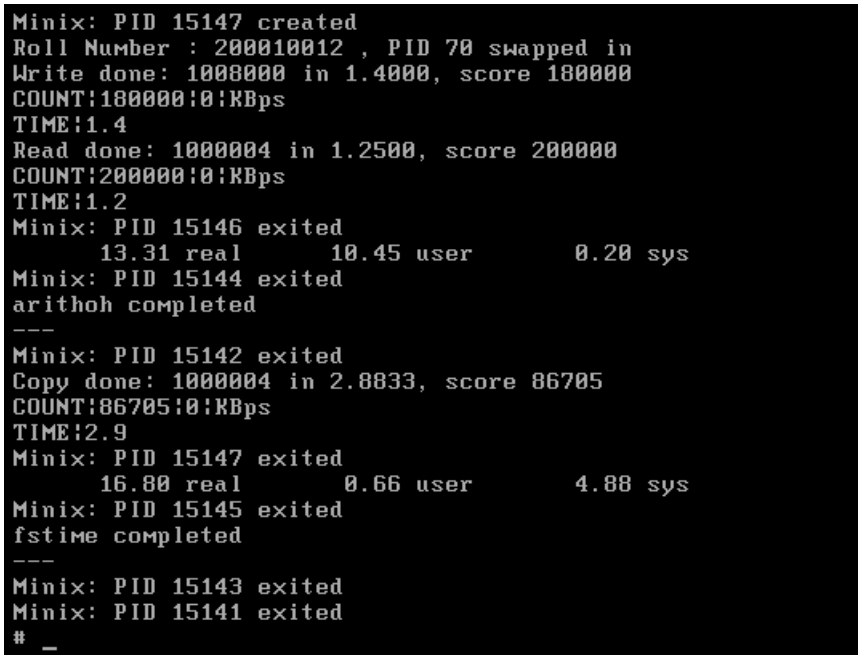

Figure 8: Swaps for workload 1

2.4.2 fstime and syscall

In this, we run a syscall and an I/O bound process to observe the process swaps. In this case, the CPU-intensive parts of the syscall code swap in when fstime waits for I/O. The code is as follows:

```
#!/bin/sh
./fstime.sh &
./syscall.sh &
wait
```

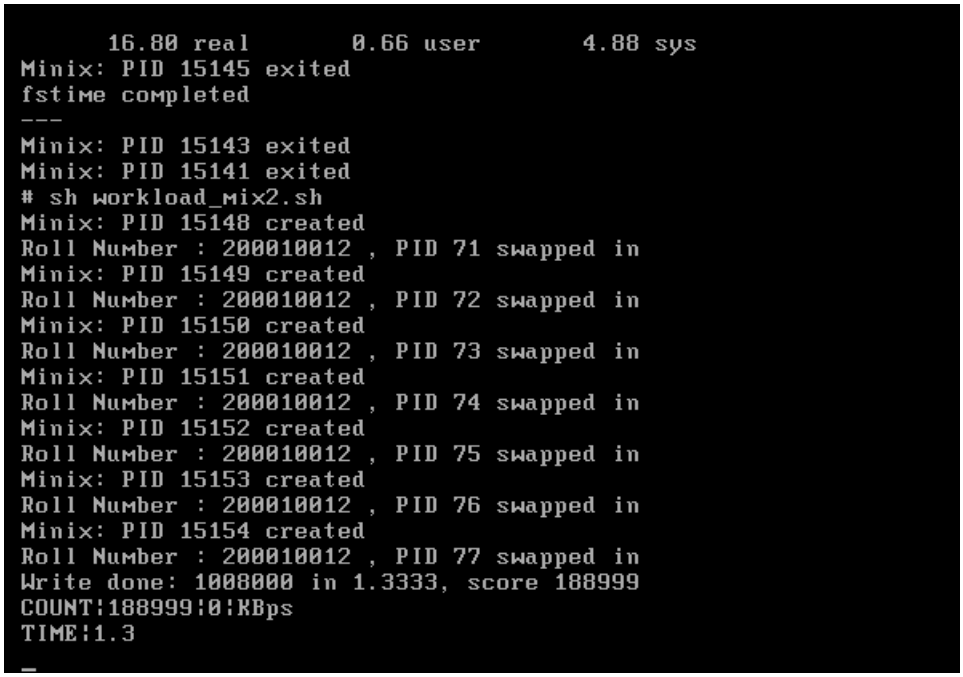

Figure 9: Swaps for workload 2

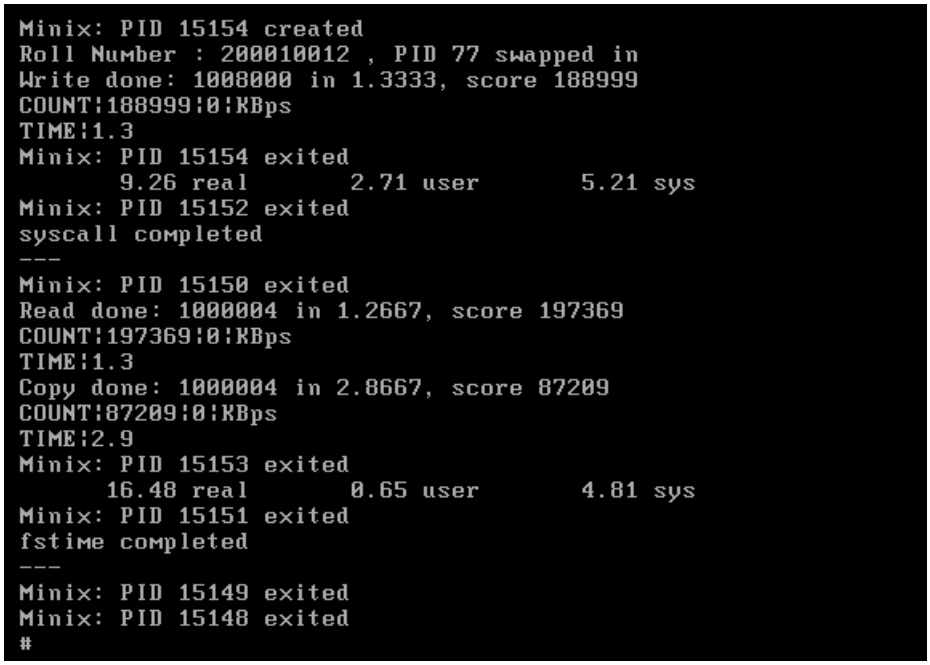

Figure 10: Swaps for workload 2

2.4.3 syscall and arithoh

In this, we run a syscall and a CPU-bound process to observe the process swaps. Both of these processes swap alternatively, since both are primarily CPU-bound processes. The code is as follows:

```
#!/bin/sh
./arithoh.sh &
./syscall.sh &
wait
```

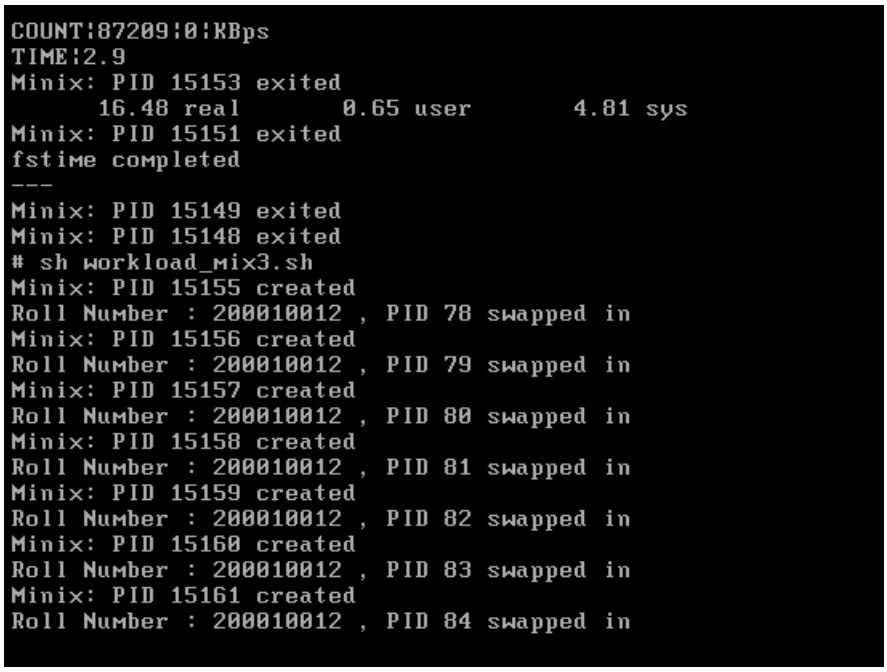

Figure 11: Swaps for workload 3

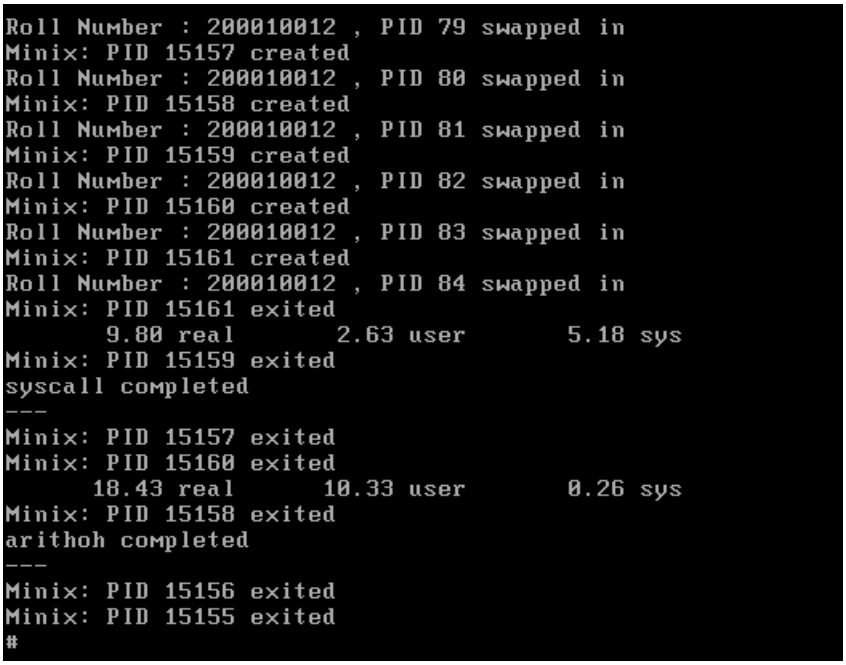

Figure 12: Swaps for workload 3

2.4.4 repeated syscall

In this, we run repeated syscalls to observe the process swaps. The code is as follows:

```
#!/bin/sh
./syscall.sh &
./syscall.sh &
./syscall.sh &
wait
```

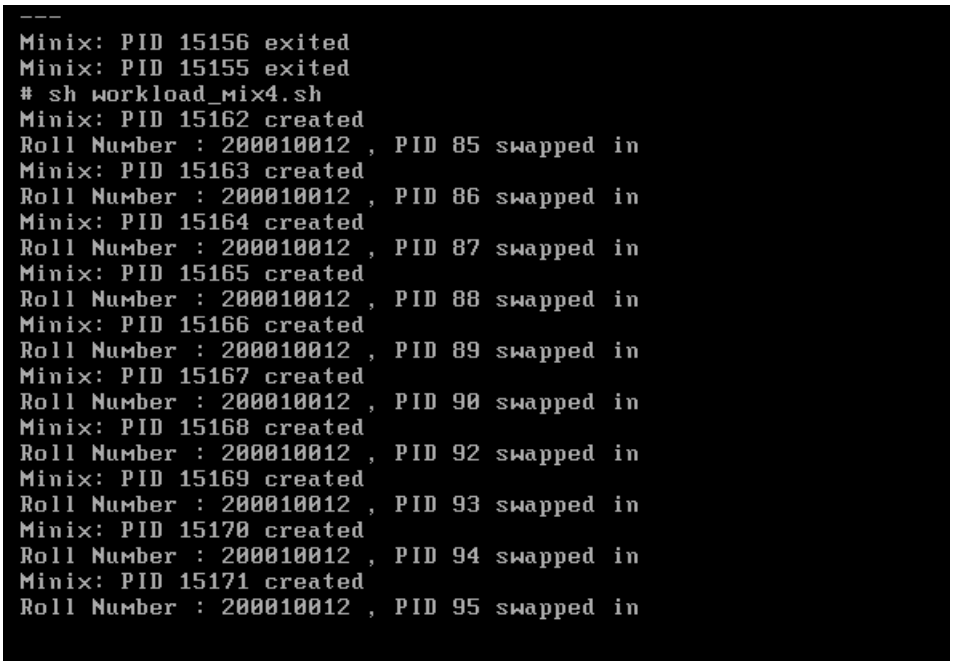

Figure 13: Swaps for workload 4

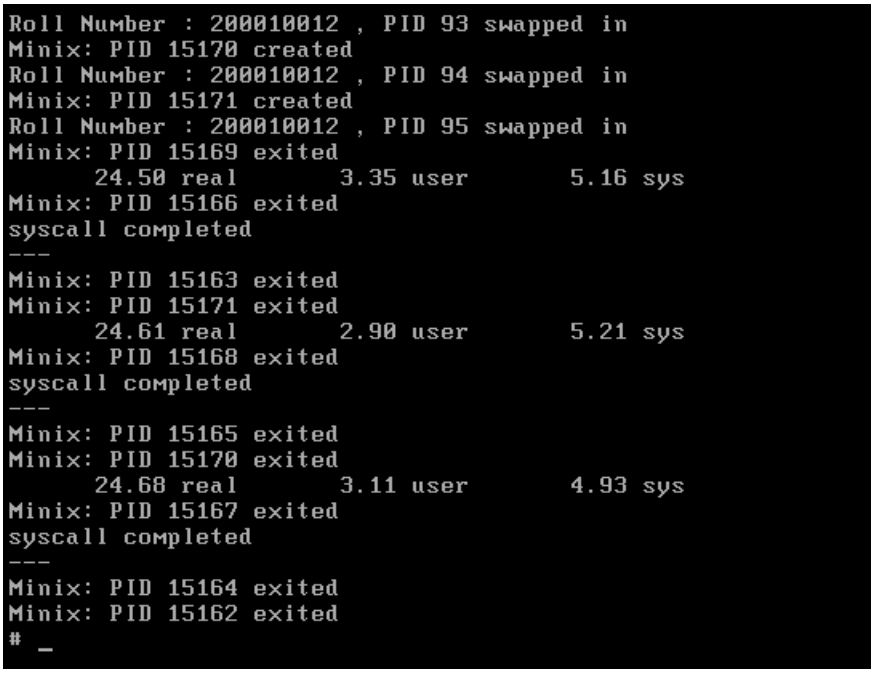

Figure 14: Swaps for workload 4

2.4.5 repeated arithoh

In this, we run repeated CPU-bound processes to observe the process swaps. The code is as follows:

```
#!/bin/sh
./arithoh.sh &
./arithoh.sh &
./arithoh.sh &
wait
```

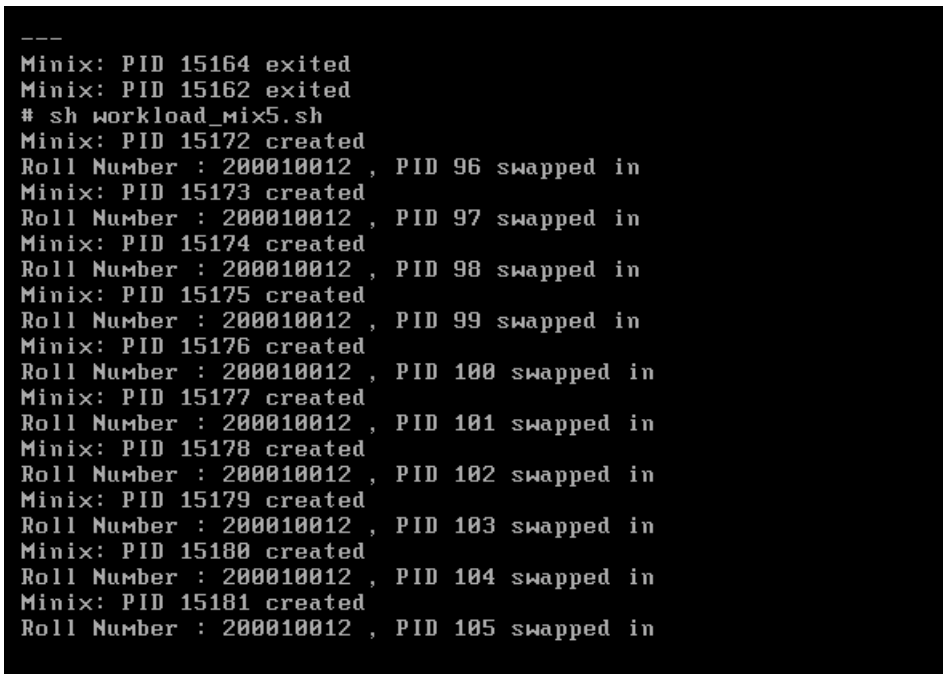

Figure 15: Swaps for workload 5

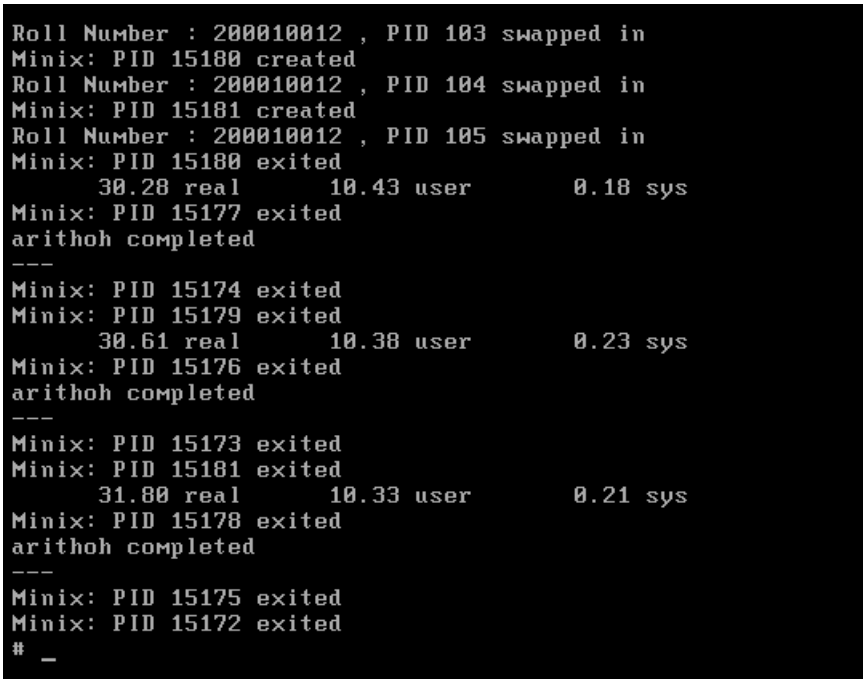

Figure 16: Swaps for workload 5

2.4.6 repeated fstime

In this, we run repeated I/O-bound processes to observe the process swaps. The code is as follows:

```
#!/bin/sh
./fstime.sh &
./fstime.sh &
./fstime.sh &
wait
```

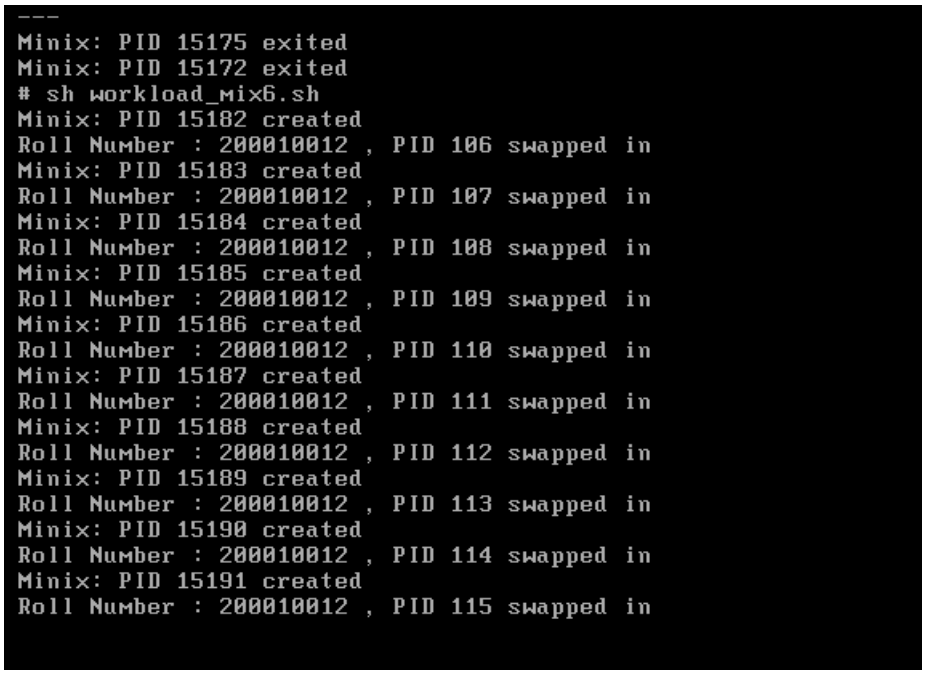

Figure 17: Swaps for workload 6

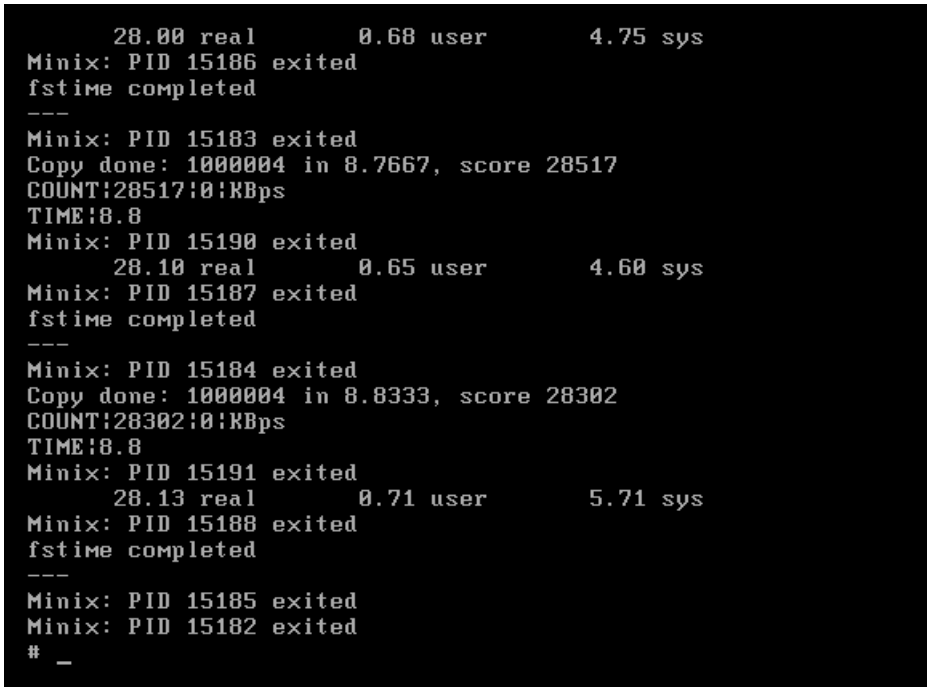

Figure 18: Swaps for workload 6

2.4.7 Other output logs

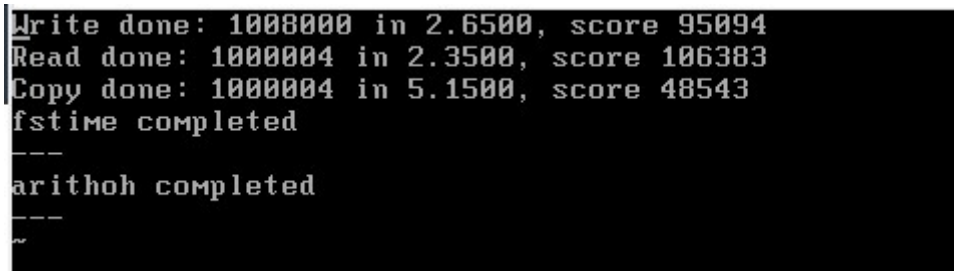

Figure 19: Output for workload 1

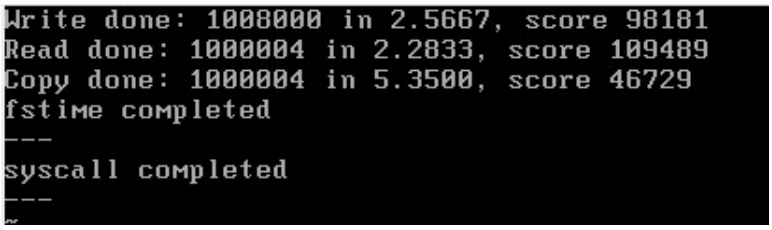

Figure 20: Output for workload 2

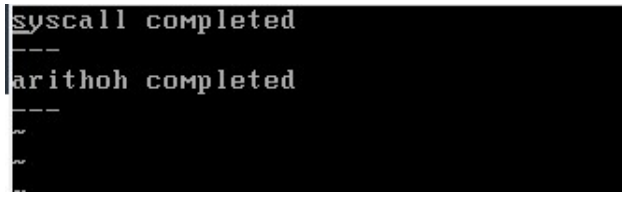

Figure 21: Output for workload 3

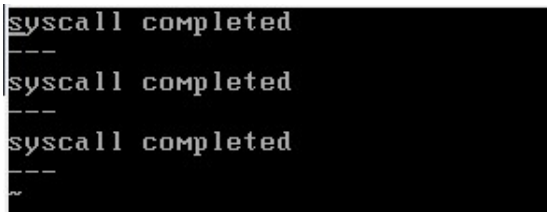

Figure 22: Output for workload 4

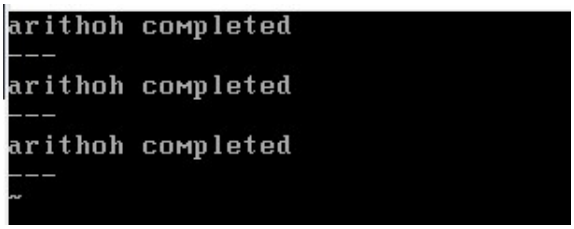

Figure 23: Output for workload 5

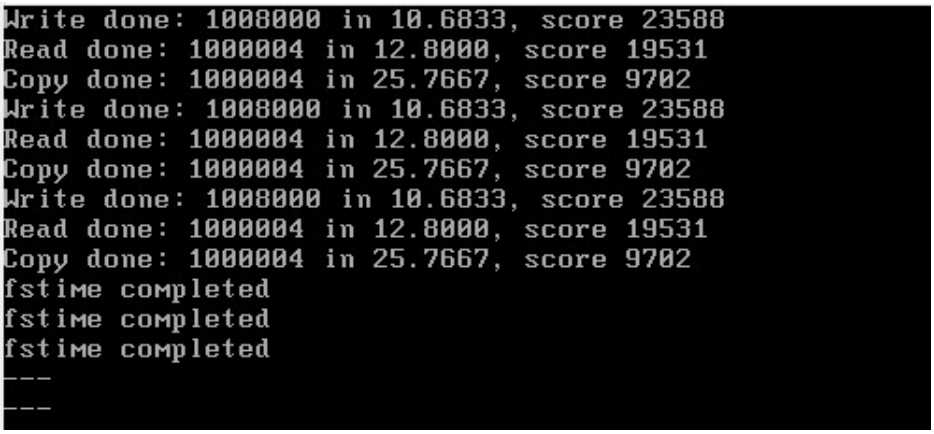

Figure 24: Output for workload 6