CS314 : Operating Systems Lab

Lab 5

Sourabh Bhosale (200010004)
Dibyashu Kashyap (200010013)

February 5, 2023

1 Part 1

Problem

Prepare (at least 4) workload mixes having different characteristics, ranging from all compute-intensive benchmarks to all I/O-intensive benchmarks. Each workload should spawn around 5 processes. You can compose these workloads of benchmarks from the UnixBench suite, or write your own.

Answer

As the problem required us to encompass the time quanta spent in the CPU by the processes. For that, file `system.c` at location minix/kernel/ was changed slightly. In the function sched_proc(), the quanta allotted and used were extracted as follows:

```
printf("Minix : Allotted Quantum: %d ms\n
        Minix : Used Quantum: %d ms\n",
        p->p_quantum_size_ms, p->p_quantum_size_ms -
               cpu_time_2_ms(p->p_cpu_time_left));
```

To make appropriate changes we have written a runme1.sh file, which when run, copies the modified system.c file into appropriate location and rebuilds the minix OS. Upon executing runme1.sh, and successful build followed by a reboot, the changes should be reflected. The workloads used by us and our observations and inferences are presented in the sections below.

```
#!/bin/bash


cp system.c /usr/src/minix/kernel/system.c;
cd ../usr/src;
make build MKUPDATE=yes > log.txt;
```

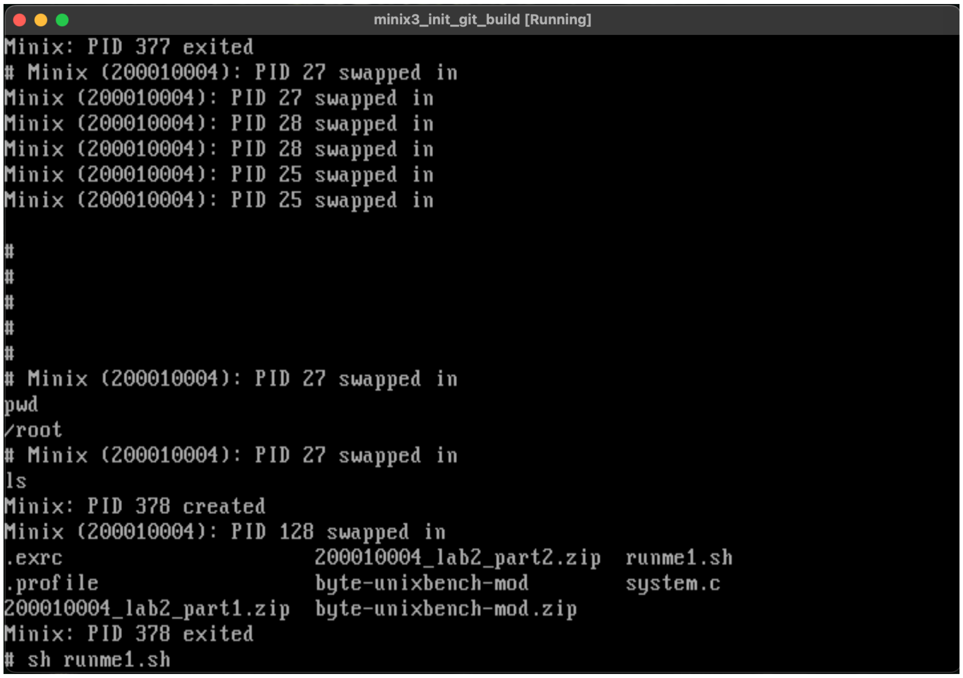

Figure 1:

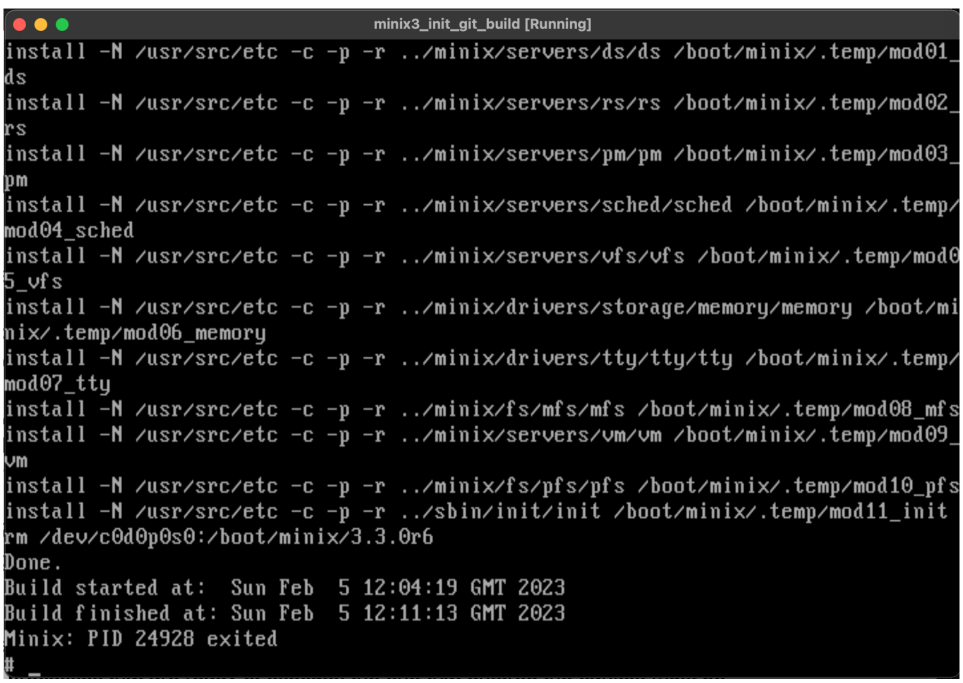

Figure 2:

workload_mix1.sh

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

Observations:

We see that the 5 processes are executed in parallel in the form of shifts. The time account, which is 200 by default, is used in full each time the process is scheduled.

Inference:

The CPU fairly schedules all processes on a rotational basis. Since these are CPU intensive processes, so they do not need to wait for I/O and run completely in the allotted quanta of time provided to them. These processes used the entire time quantum and placed at the end of the queue.

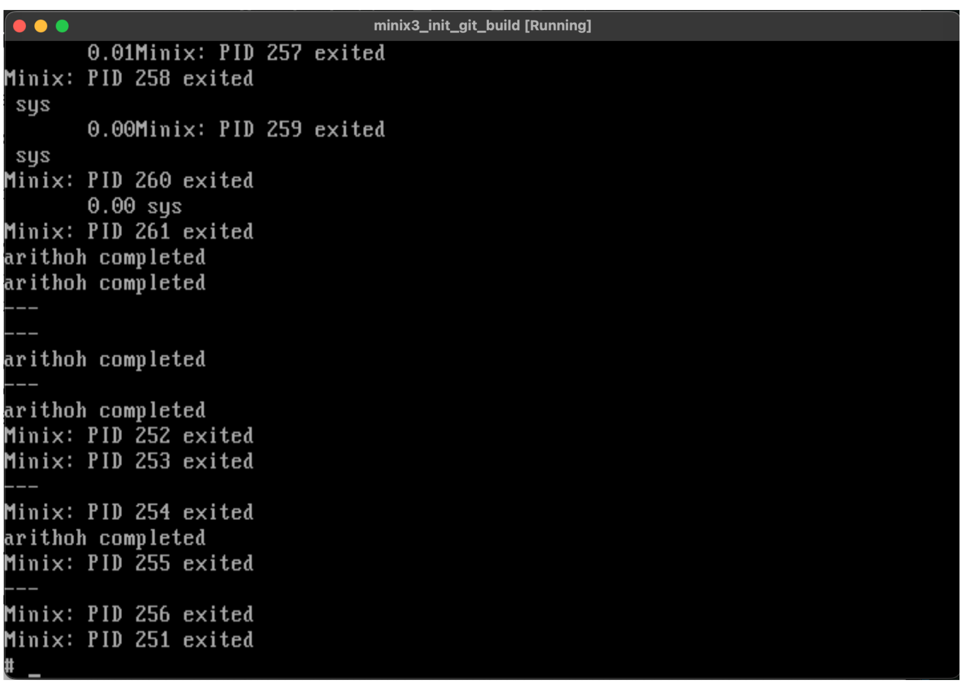

Figure 3:

workload_mix2.sh

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

Observations:

The 5 processes run in shifts. Also, since modifying files is a system call, it has higher quantum inorder to prevent malfunctioning.

Inference:

The fstime.sh process is I/O bound. Tasks don't use their quanta to the fullest as they don't consume much CPU as it got blocked waiting for I/O.

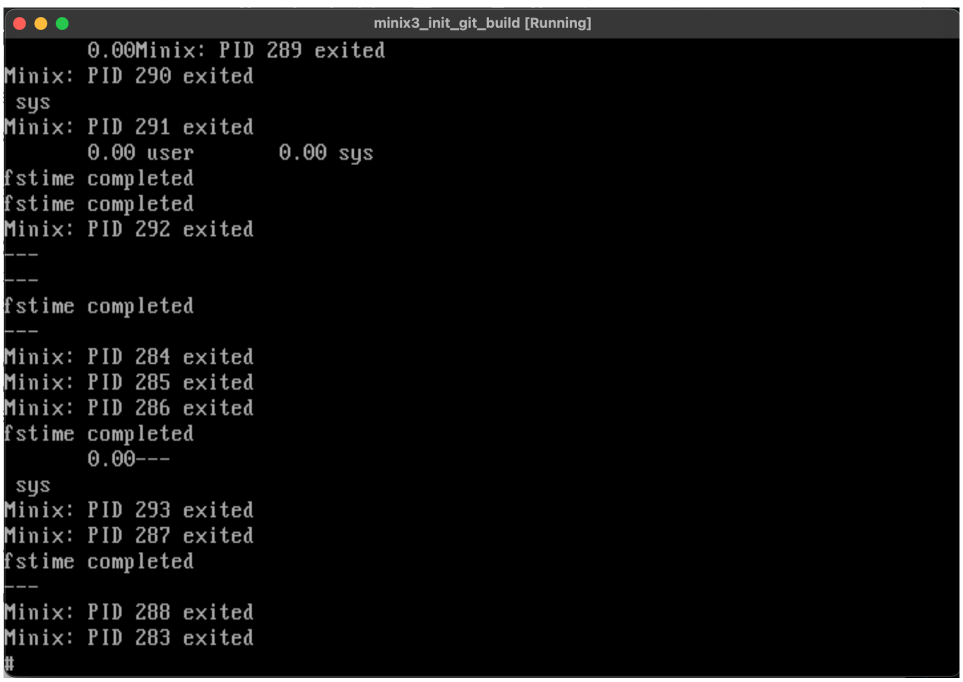

Figure 4:

workload_mix3.sh

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

Observations:

Syscall is a system call that is a request made by a program to the operating system's kernel. A system call can be either CPU bound or I/O bound.

We see that the 5 processes are executed in parallel in the form of shifts. The time account, which is 200 by default, is used in full each time the process is scheduled. Since syscall.sh is not intensive as arithoh.sh, all the quanta slots are not fully engaged every time the process is scheduled.

Inference:

The CPU fairly schedules all processes on a rotational basis. Since these are lesser CPU intensive processes, and run completely in the allotted quanta of time provided to them. Hence, in this workload mix3.sh, all process finish in almost same time while. partially utilizing their assigned quantum slots.

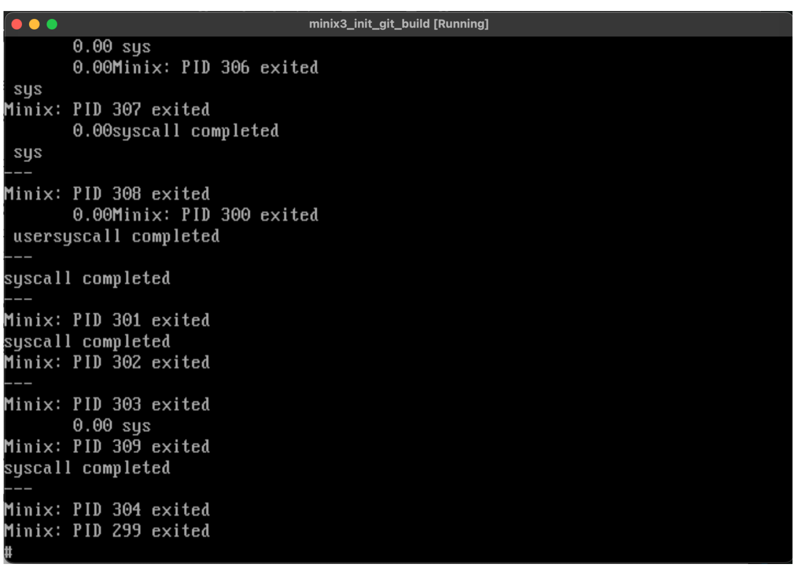

Figure 5:

workload_mix4.sh

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

Observations:

We see that the 3 CPU intensive processes and the 2 linked I/O processes run in parallel in a round-robin fashion. However, the other 2 linked I/O processes are waiting for their I/O work and are then scheduled to run on the CPU. So after completion these processes are put at the head of the queue with remaining time quantum.

Inference:

CPU-bound arithoh tasks use all of their quanta, while I/O-bound fstime tasks don't.

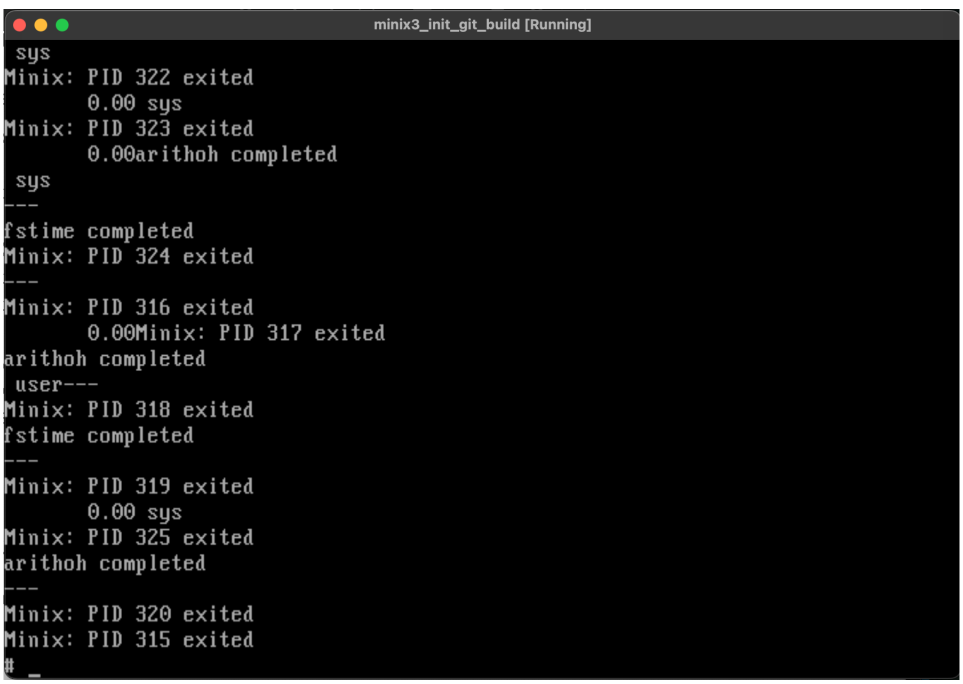

Figure 6:

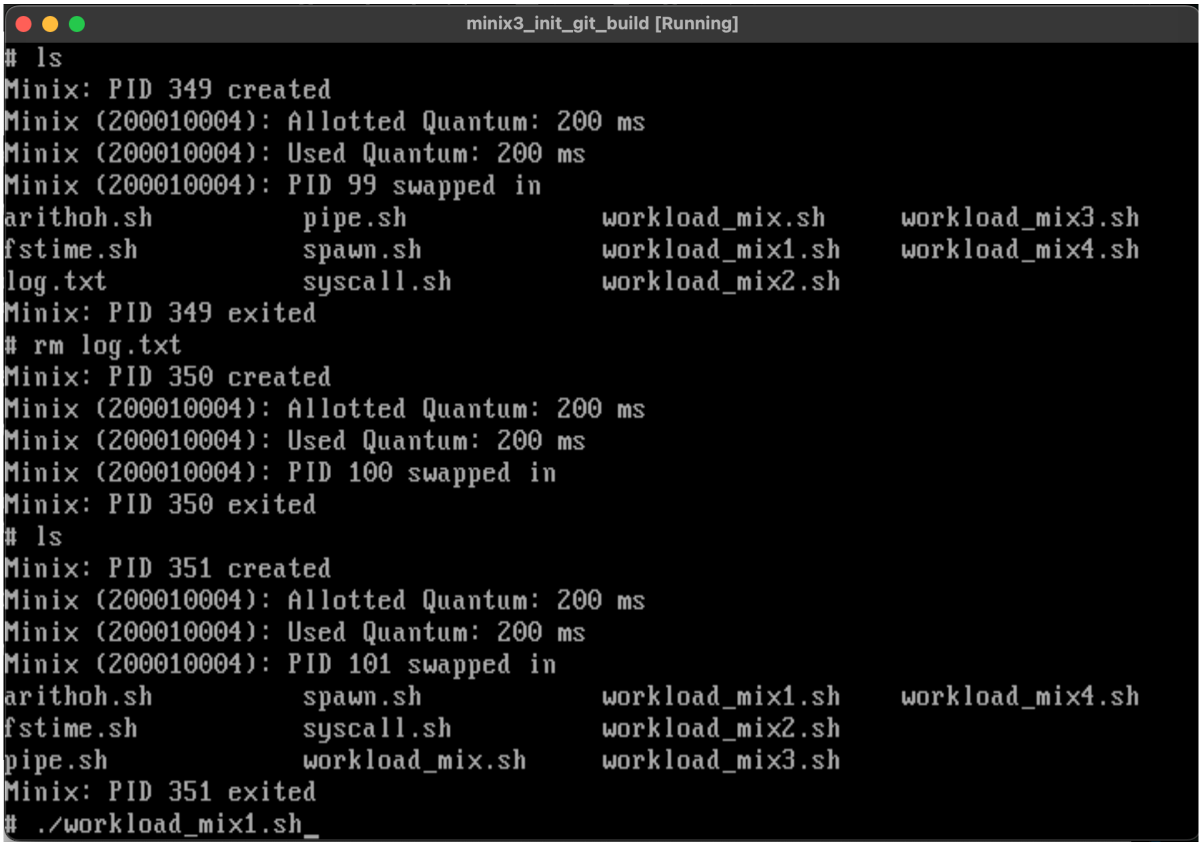

Figure 7:

2 Part 2

Problem

Modify the user-level scheduler in Minix3 to the following "Pseudo-FIFO" policy: among the user-level processes that are ready to execute, the one that entered the earliest must be scheduled.

Answer

The minix 3 scheduler by default following Round Robin policy inside each queue. Whenever a process uses complete quanta of time allocated to it, it will be send the subsequent lower priority queue. To avoid starvation, periodically, all the processes will get the priority boosted after certain interval of time. I/O bound processes run until they block but they are given higher quantum inorder to prevent malfunctioning. If a process has not used its entire quantum and blocked, then it means it got blocked waiting for i/o. After i/o is complete the process is put at the head of the queue with remaining time quantum. To implement Pseudo-FIFO, we increase the priority of the running process after it completes it's time quanta so that the process that was running first has the highest priority and is selected to run again. To do this, we have modified the do_noquantum() function of the `schedule.c` file present in the /minix/servers/sched/ directory as follows:

```
rmp->priority -= 1; /* higher priority */
```

Next, to avoid overflowing of queue , we have to ensure periodic priority boosting doesn't occur so that user processes don't invade the queues of drivers, servers, clock and system task. To do this we have commented out the following code present in the balance_queues() function of the same file.

```
// rmp->priority -= 1; /* increase priority */
```

To copy the modified file and re-build the OS, we have written a bash file named runme2.sh. Upon executing runme2.sh, and successful build, followed by reboot, the changes should be reflected. After this, we ran the workloads of part 1 again and our observations and inferences are presented in sections below.

```
#!/bin/bash


cp schedule.c /usr/src/minix/servers/sched/schedule.c;
cd ../usr/src;
make build MKUPDATE=yes > log.txt;
```

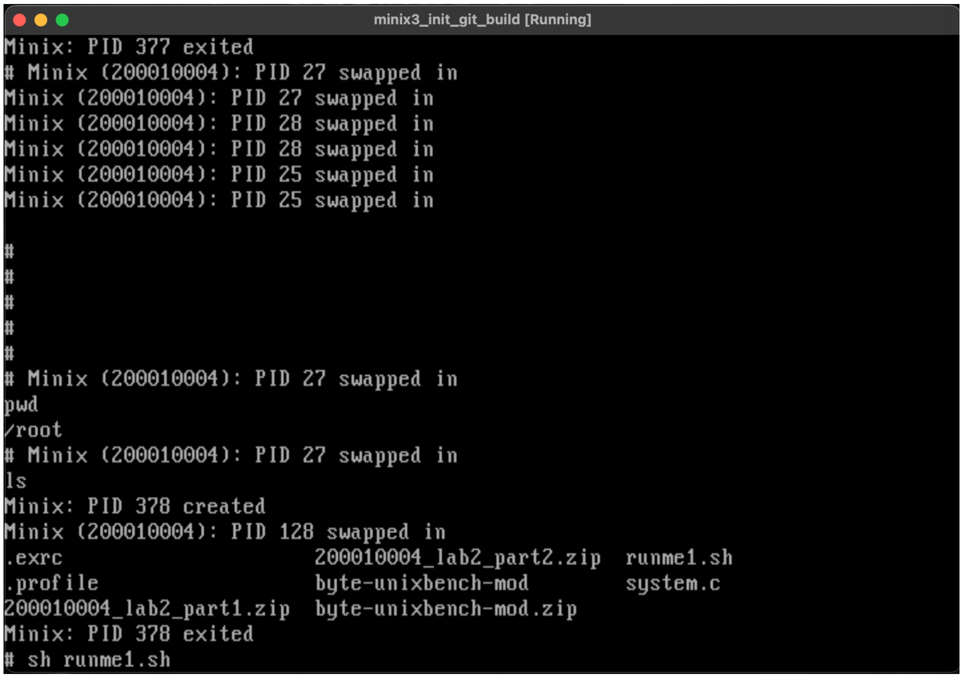

Figure 8:

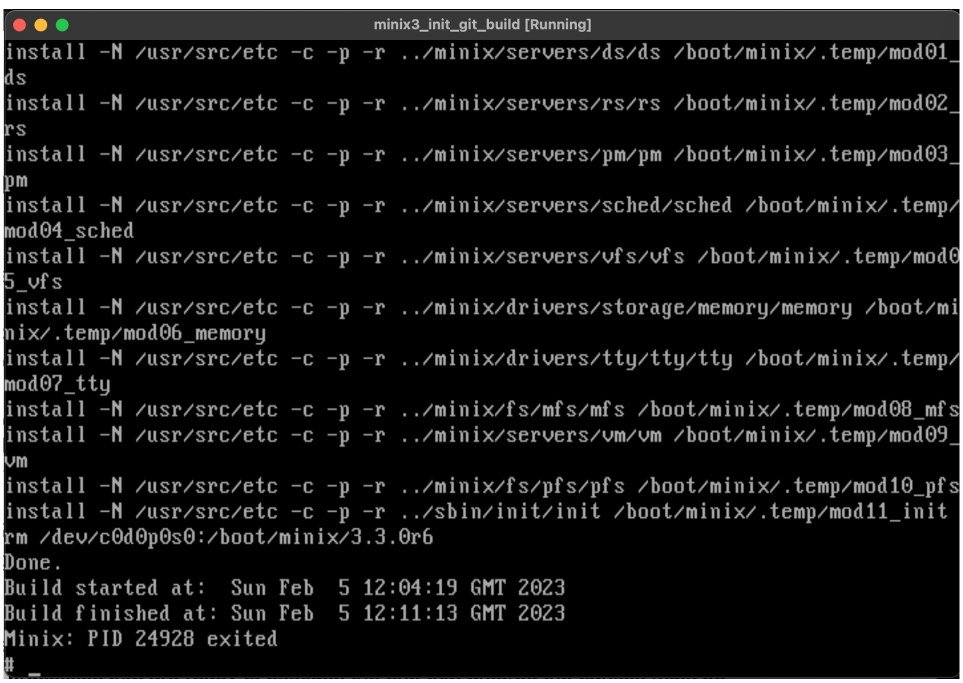

Figure 9:

workload_mix1.sh

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

This workload spawns 5 different instances of airthoh.sh process, which is CPU bound. Since all 5 are CPU Intensive processes, they run one after the other sequentially in first come first serve fashion. In the execution, they fully engage their quantum slot of 200 as they do not have to wait for I/O. All of these processes were allocated a time quanta of 200 ms by default. Since they are CPU bound, they utilize the complete time quanta. Unlike round robin manner, in the Pseudo-FIFO, one process starts its execution and runs till completion. In the figure 10, we can see process with same PID , is being scheduled back to back indicating a FIFO policy.

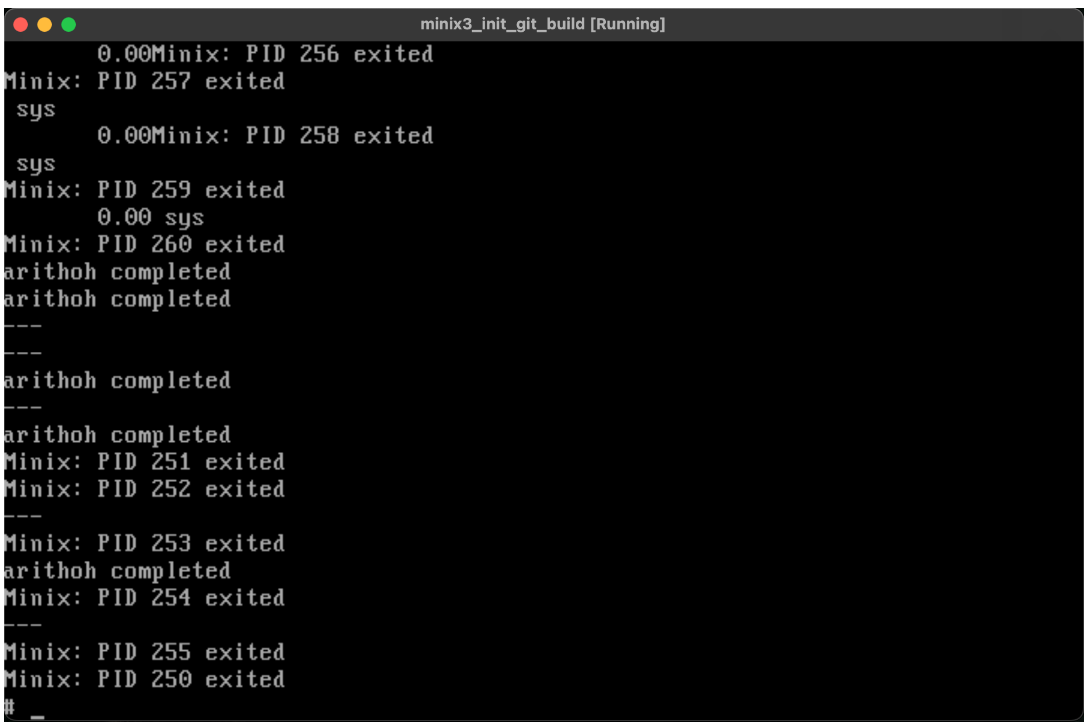

Figure 10:

workload_mix2.sh

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

This workload spawns 5 different instances of fstime.sh process, which is I/O bound.

Again in this workload we can observe the pseudo nature of the Pseudo FIFO policy. The fstime.sh is I/O bound process. By default these processes were allocated a time quanta of 200 ms, but don't completely utilize it. So next time they are allocated a time quanta of 500 ms. Unlike an actual FIFO policy, in our case when ever a process waits for I/O to respond, we simply preempt it. This was also observed in the case of round robin policy of default minix. Once the I/O responds, the processes are scheduled again until completion in a FIFO manner assuming they do not depend on I/O anymore. This is the reason we call our policy "Pseudo" FIFO. This policy follows FIFO, only in the case of CPU bound processes or when there is no dependency on I/O devices.

Since the I/O bound processes are sent to the waiting queue after requesting for I/O and are then placed back in the ready queue and scheduled to work on the CPU when I/O received. I/O bound tasks don't always utilize complete quanta slot of 500 allotted to them. Hence they then follow normal Round-Robin Order itself. There is the no change observed when Pseudo FIFO is applied rather than default Round-Robin in Minix3 in this case

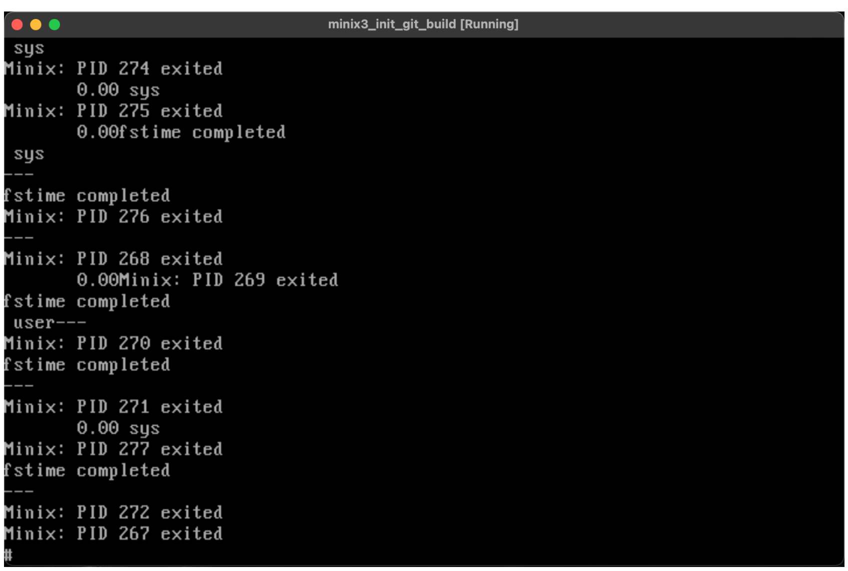

Figure 11:

workload_mix3.sh

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

The syscall.sh is CPU bound process. By default these processes have been allocated a time quanta of 200 ms. These processes run one after the other in a FIFO manner, unlike round robin manner of default minix. During such execution, processes fully engage their quantum slots of 200 as they do not wait for I/O. Only when one process's execution is terminated, next process is scheduled subsequently. From figure 12, we can see that process with same PID is being scheduled back to back until completion. Unlike round robin, here once a process completes its execution, only then the next process is scheduled.

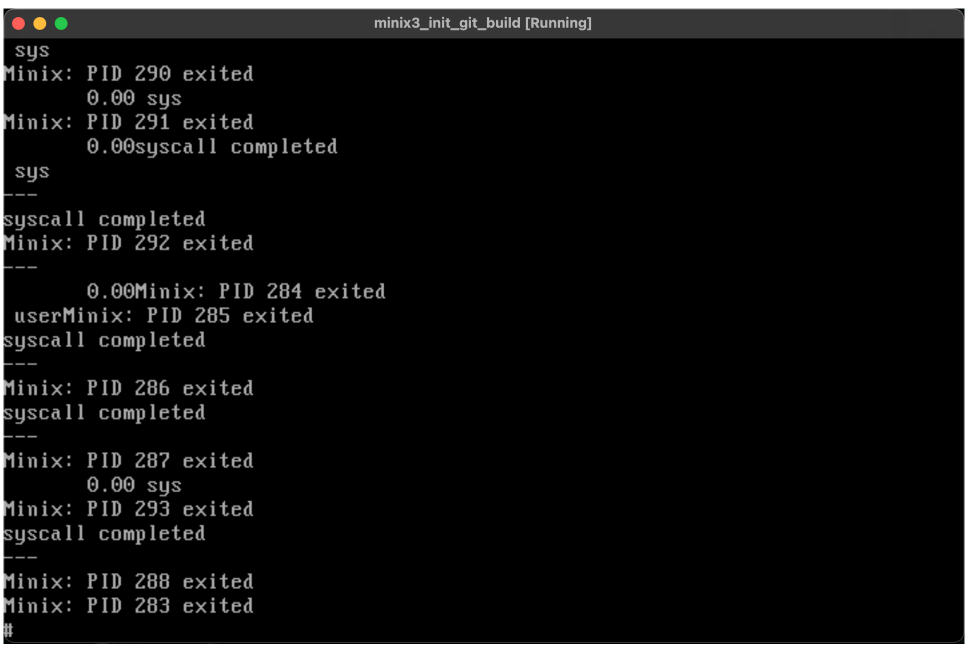

Figure 12:

workload_mix4.sh

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

This workload spawns 3 different instances of airthoh.sh process, which is CPU bound and 2 different instances of fstime.sh process which is I/O bound.

By default, all of these process have been allocated a time quanta of 200 ms. In this case, we can observe the working of actual "Pseudo" FIFO. In an actual FIFO policy, when an I/O bound process is scheduled, the CPU has to wait until I/O responds, while the I/O bound fstime.sh processes wait for I/O to respond, the scheduler schedules CPU bound airthoh.sh processes. But unlike the default minix scheduler, this scheduler follows a FIFO policy in case of CPU bound processes. As seen from figure 13, process with same PID is scheduled back to back indicating a FIFO policy. The fstime.sh processes being I/O bound do not completely utilize the allocated time quanta and they are allocated a time quanta of 500 ms next time. After the I/O responds, the fstime.sh processes are scheduled and completed.

So, we can conclude that after our appropriate changes in codes, CPU intensive were able to follow proper FIFO scheduling whereas I/O bound processes weren't. Due to this approximation and exceptions, we declare such scheduling as Pseudo FIFO.

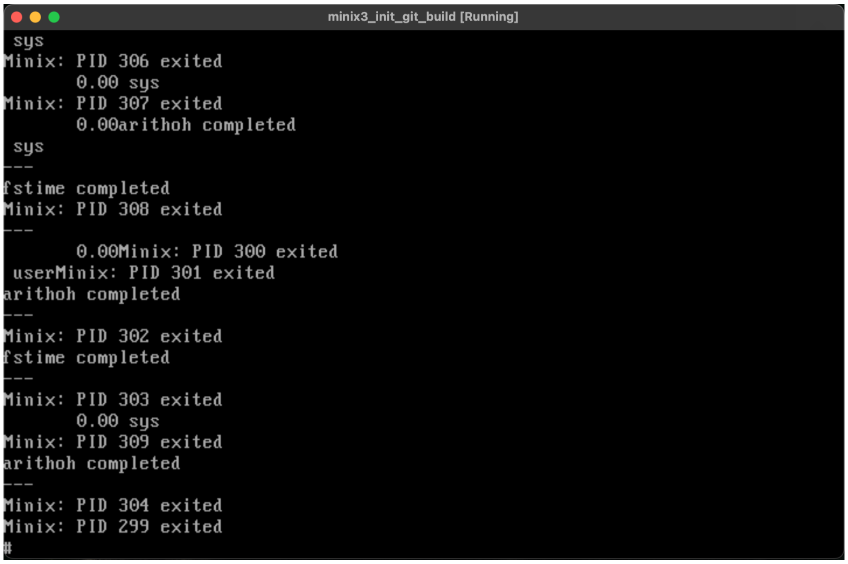

Figure 13: