Vamvas Ioannis A.M.: 2943

Gewrgoulas Vasilis A.M.: 2954

Step 1

Originally in the handle_vfs_reply() function of the archive usr/src/servers/pm/main.c, we can see PM_FORK_REPLY field that's where the execution of the main fork() happens in sched_start_user() of /usr/src/servers/pm/schedule.c. There It's the first change we're going to make to get the group through a process in the do_start_scheduling() of the archive usr/src/servers/sched/schedule.c. In sched_start_user(), we have the the function of sched_inherit(), which accepted 5 We changed her prototype through the archive. usr/src/include/minix/sched.h so that it now accepts 6 argument where the last argument will be the process group (rmp->procgrp). The function of the sched_inherit() is then implemented in the usr/src/lib/libsys/sched_start.c where we put the 6th argument in the field m9_l5 of the message that will be posted to the sched (which will be is the SCHEDULING_INHERIT). Weput it on the process. in the field of m9_l5 because we concluded that in the SCHEDULING_INHERIT don't use this field of message (more specifically we saw it through the archive usr/src/include/minix/com.h). Then we see that with themessage SCHEDULING_INHERIT'll end up in the archive. usr/src/servers/sched/main.c where from there we see that for the case SCHEDULING_INHERIT't end up with the function. do_start_scheduling() mentioned above, where there will be the initialization of the team of each process by taking the field m9_l5 of the message.

Step 2

In the archive usr/src/servers/sched/schedproc.h we added the 4 fields requested in the exercise, more specifically the pid_t procgrp (where is the team driver), the unsigned proc_usage which is the use

process, the unsigned grp_usage which is the use of the process group and the unsigned fss_priority which is the priority based on the algorithm of fair routing. These 4 fields, the initials in the do_start_scheduling() function of theusr/src/servers/schedul/schedule.cand more specifically, we initialize the procgrp field as m_ptr->m9_l5 right after the introduction of the process into the schedproc.hNG_INHERIT structure.

The field proc_usage initialize to 0 because the use of the process once it enters do_start_scheduling is 0 (because the "programming" of the process from the core has not yet beendone).

The field grp_usage initialize as much as the grp_usage of the 1st process (parent) because we know that all processes of the same group should have the same grp_usage.

The field fss_priority be initialized on the basis of the type, where the number_of_groups find it through the function num_of_grps(), which isreturned to number_of_groups.
num_of_grps() we first find all processes useful and
we kept in 1 table the drivers of the process group and in the
We've always counted the different group drivers we find.
within this tablewhere it will finally be and the
number_of_groups we're looking for.

Then the information of the 4 above fields is done in the do_noquantum() function of the usr/src/servers/schedul/schedule.c file, where we check if we have user processes and if we then have the followinginformation:

The field proc_usage process that finished its quantum, we increase it by RMP->time_slice if time_slice is equal to USER_QUANTUM otherwise we increase it by USER_QUANTUM.

Then we run all the processes we have in the structure of schedproc.h and increase all grp_usage of the processes that have the same team driver as the process that ended its quantum USER_QUANTUM.

Finally, for all user processes, we update theirdata as follows:

The proc_usage=proc_usage/2

The grp_usage=grp_usage/2

And the fss_priority based on the type fss_priority = proc_usage/2 + grp_usage*number_of_groups/4 + base, where base=0(is the same type that was climbed above step 2 in terms of the initialization of the fss_priority).

Step 3

To make the processes useful in 1 queue only (as you ask in the pronunciation), we changed the file usr/src/include/minix/config.h and now we put as NR_SCHED_QUEUES = 8 and MAX_USER_Q = MIN_USER_Q =USER_Q=7 so that the queues 0-6are the same as before, MAX_USER_Q MIN_USER_Q tail 7 be the tail useful and the tail 8 be the tail idle (as it was in the previous version of the minix tails). Then, in order to pass the fss_priority thecore, we take the six steps :

1) We change the prototype of the sys_schedule so that instead for 4 argument, accept 5, so that its^{5th} field is fss_priority. This was done through the usr/src/include/minix/syslib.h file.
2) In the usr/src/lib/libsys/sys_schedule.c file where you perform the sys_schedule() function we put in the free field m9_l5 of the message in fss_priority that we passed through step1 SYS_SCHEDULE _kernel_call sys_schedule.
3) Then after the call of a system the message ends in the functions do_schedule() and do_schedctl() (SYS_XXX->d o_xxx) (which are located in/usr/src/kernel/system and in there we put the fss_priority equal to the field of the message m9_l5 where we pass the fss_priority of step 2) where these

send the fss_priority to the sched_process of the archive system.c (where we changed the prototype of the function sched_proc() through the /usr/src/kernel/proto.h file so that the function sched_proc() no longer accept 5 argumentations, where the 5th argument will be the fss_priority) and there each process is useful he's going through the fss_priority by updating the board. percentage h.

Having now for each user process the fss_priority (this is ensured through the schedule_process() function of the usr/src/servers/schedul/schedule.c, where if we have a user process then we call once to send to the core the fss_priority of all the user processes) so that the core can choose to perform the user process with thesmallest fss_priority.

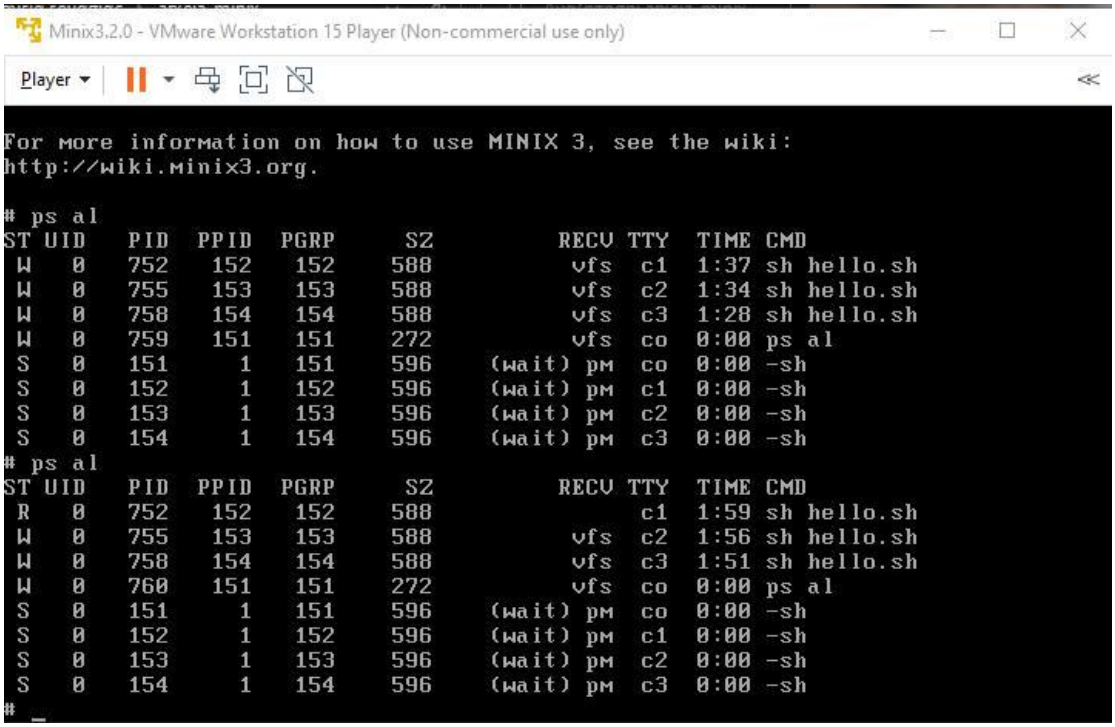
Then to select the kernel process with the lowest fss_priority we changed the pick_proc() function to /usr/src/kernel/proc.c. First we had before for each tail the 1st process of each tail that was ready to be executed, we added and have the last process of each tail that is ready to be executed(rdy_tail).
η Method does what it did before. If we find out we're in the
then we check if the1st process of the user's tail is non-current (i.e. it is ready for execution), if it is not then we continue to the next tail (essentially we leave the time) while if it is ready for execution then the min is done equal to the
p_fss_priority of the p_fss_priority 1st process that is ready for execution.
less than >p_nextready 1st process and if it is then we will perform some of these processes.
will be ready for execution). First the p will be equal to the 1 the process of the tail that is ready to be performed(rdy_head[q])

while the end will be equal to the last process of the tail that is ready to be performed(i.e. rdy_tail[q]).

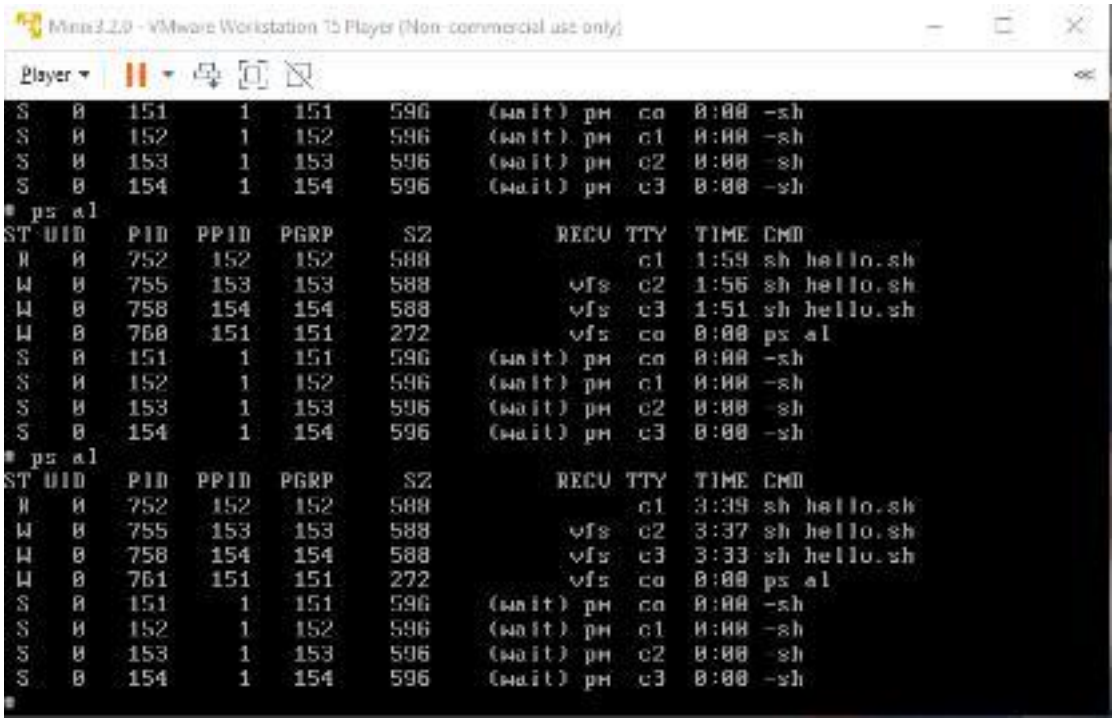
Step 4

We made 1 scrpt which we called hello.sh and ran it almost simultaneously in 3 different terminals (in the 2nd,3rd and 4th and 1st the terminal with the command ps al we saw the time of execution of the review in the terminals).

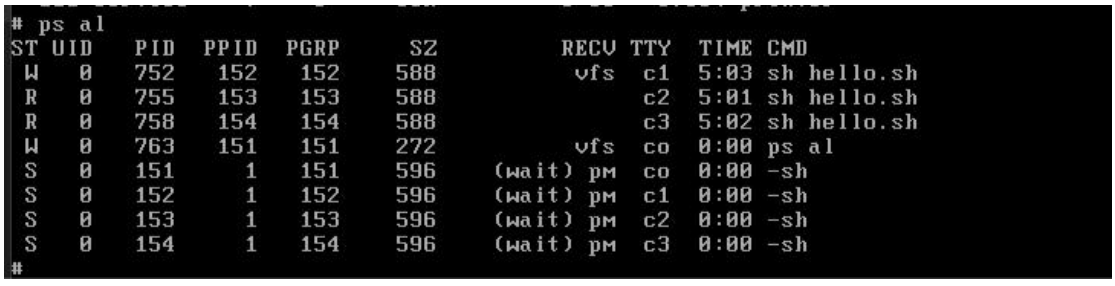
After 5 minutes (1 ps al) and 6 minutes of execution (2^{ps} al)
we had this effect here:



After 10 minutes (the last ps al shown below in the picture) we had this result here:



After 15 minutes of execution we had this result here:



As shown by the above images, time is divided between the 3 different terminals that we run the scrpt described above.