Akshita Mittel
CS3031
11 November 2015

Implementing Schedulers in Minix3

Operating Systems: Project

1. Introduction

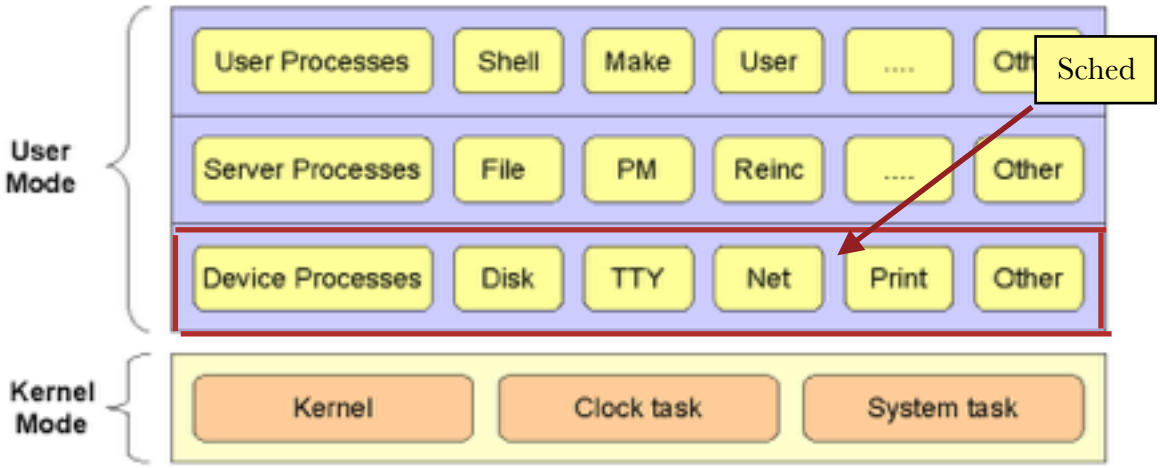
The main aim of this assignment is to understand the current Minix kernel and its scheduling algorithms. Based on the knowledge alter the current schedulers to implement our own algorithms, and test their performance in different situations.

This is possible, as the scheduler portion of the Minix kernel is placed in the User space as opposed to the conventional Kernel space.

2. The Minix Kernel

The server in the User space handles the policy to be followed while scheduling processes, and the kernel does the actual job of scheduling processes on the CPU and context switching.

The Minix kernel is split into 4 sections, with the top 3 being User mode and the bottom layer being the Kernal mode. The actual schedulers are present in the User mode in layer 2, thus allowing the user to run multiple schedulers, each with their independent policy.



The MINIX 3 Microkernel Architecture

The scheduler is in layer 2 under the director sched.

The way that the user interacts with the scheduler as follows:

2.1. The user makes a library call to the scheduler.

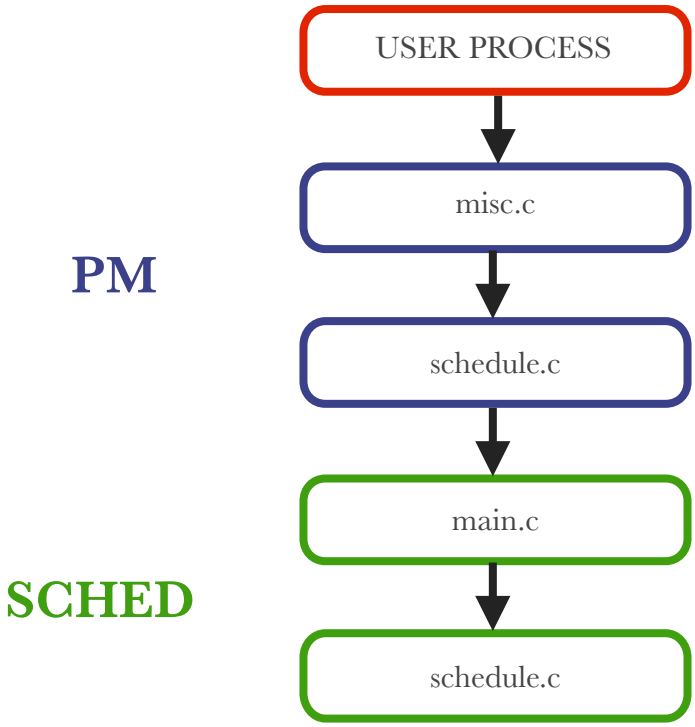
2.2. This goes to the system wrapper in misc.c

2.3. It then passes through the helper functions of the scheduler, stored in the PM directory, also named schedule.c

2.4. In the shed directory, where our scheduler actually lies, it goes into the main which selects which scheduler to call.

2.5.The scheduler is finally called.

The following flow chart gives an overview of the scheduler process, and how a user made scheduler is executed:



3.MULTILEVEL FEEDBACK QUEUE

3.1. The MFQ

The basic things to take into consideration when designing a multi-level feedback queue is:

- A new job entering the system is always given the highest priority, i.e. it is put it in the top queue.
- Once the job uses its quantum number its moved to the next layer and its priority is lowered. This happens in the 2nd queue as well.
- If the job is in the lowest queue and uses up its quantum number it is given a higher priority and it is moved to the topmost level. This means that after a certain time all the jobs are in the topmost queue.

3.2. The Implementation

In order to implement the MFQ, the following steps were followed:

• Adding an unsigned quantum in schedproc.h file. This file is found under usr/src/servers/sched/

• Added 3 queues, that are used for the different levels in the MFQ. These queues are added in /usr/src/include/minix/config.h. Where the original number of queues were changed from 16 to 19. Now the the number of NR queues are 19 whereas the number of user queues are 16.

CHANGES MADE IN THE MAIN SCHEDULER:

There were several modifications made in the schedule.c in the directory /usr/src/servers/schedule.c.

• **do_noquantum( ):** For processes between queue 16 and 18, increase their priority by 1, as soon as their quantum expires. For processes in queue 18, set the new priority as 16 .For all other processes, use default policy.

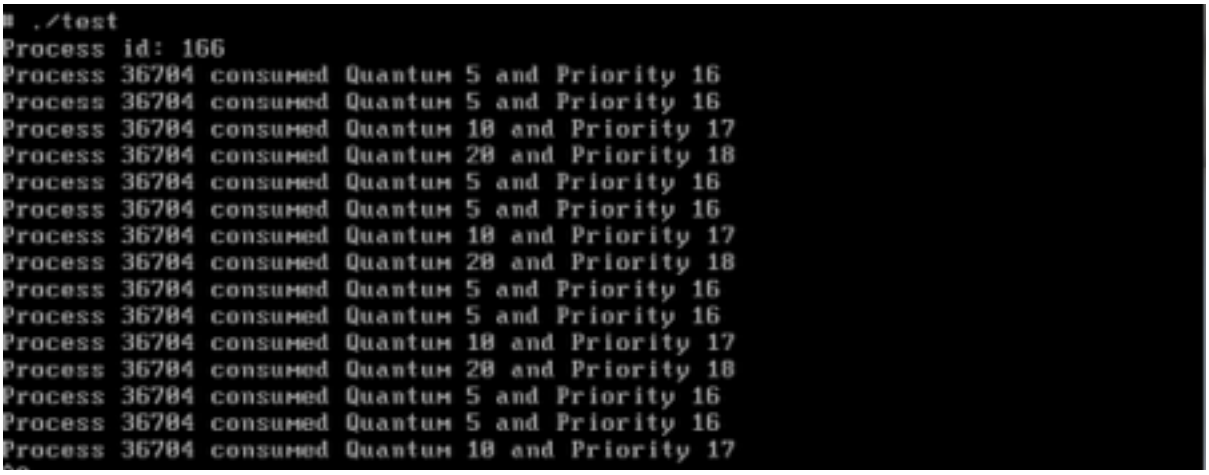
• **do_start_scheduling( ):** In this function we mainly initialise the priority, max_priority, quantum, and time_slice. Changing the value's when required.

• **do_nice( ):** If we cannot assign new priorities to the jobs in queues 16 -18, we assign previous priority value's. In case of an error we roll back.

• **balance_queues( ):** increase the priorities of all the jobs by one.

3.3. Testing and output

In order to test the MFQ, we used longrun1.c (processes that take a long time to execute. The output should be that the process starts from the first queue and alters to different queues as its quantum keeps changing. This can be shown as follows:



4. Priority Queue

4.1. The Priority queue

The basic things to take into consideration when designing a priority queue is:

- The priority of the job is predefined by the user.
- The job with the highest priority will be given access to the processor first.
- If two jobs have the same priority, the order of the execution of the jobs will be FCFS.

4.2. The Implementation

In order to implement the MFQ, the following steps were followed:

- The differences between the current minix scheduling algorithm and priority scheduling is minor. Following are some of the changes made to the existing scheduler.
- **do_no_quantum( ):** We disable the feedback mechanism. Here even if the priority gets above the MIN_USER_Q; we no longer lower the priority.

5. Performance:

5.1. MFQ:

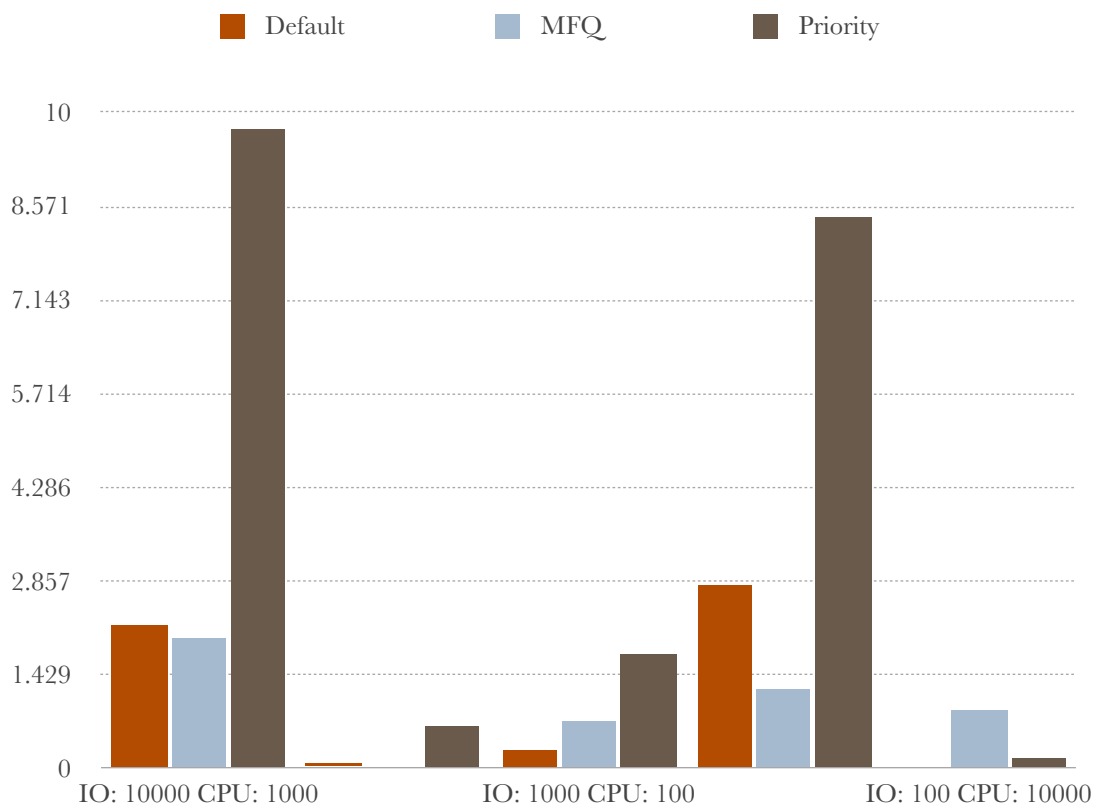
The advantages of using a MLFQ scheduling algorithm is that it assigns priority for a job based on the nature of its execution. Hence it achieves good results for both I/O and CPU bound processes.

5.2. Priority Scheduler:

Although priority based scheduling has a much greater performance as opposed to FCFS. The major drawback is starvation. Jobs with really low probabilities may have to starve indefinitely.

5.3. Comparison

In order to test the performance of the scheduling algorithms certain number of processes were generated (in this case 10). Both the I/O and CPU bound processes were simulated according to the parameters set initially. Here the I/O processes were required to write to the terminal. The turnaround time was calculated for different variants of CPU/IO ratio for each scheduler as shown below:



As expected, on an average the turnaround time for the original scheduler was the least. This may be due the fact that this scheduler can set higher priorities than the ones that can be set in the user mode. It is interesting to see that the MFQ worked almost as well as the default. This is because we allowed a job to increase its priorities after certain iterations, i.e. when its quantum got over in the bottommost queue and it was pushed to the topmost queue with an increase in priority. The priority scheduler had the worst turn around times amongst all of them. This may be due to the fact that a number of shorter jobs with relatively low priority would have to wait for the jobs which are longer but have higher priority. All this would increase the average turn around time.