MINIX 3 SCHEDULING

MLFQ and Priority Queue

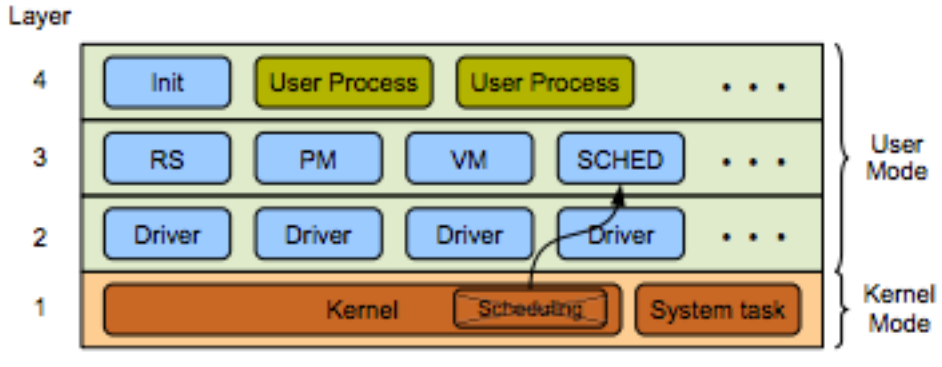
1.Goals of Project:

The goal of this assignment is to get everyone to learn how to modify MINIX 3 and to gain some familiarity with scheduling. In this assignment you are to implement a multi-level feedback-based scheduler and a lottery scheduler or priority scheduler.

2.Minix default scheduling:

The kernel defines 16 priority queues and implements a simple pre-emptive Round Robin scheduler. It always chooses the process at the head of the highest priority queue.

- When a process runs out of quantum, it is preempted but immediately placed at the end of its current priority queue with a new quantum.
- This is neither starvation free nor tries to be fair, but it is meant to schedule servers during the system startup, to avoid the problem of running out of quantum time before the scheduler starts at the start of the system.
- In minix3, the scheduling is added in the User mode of the kernel which gives us power to change the scheduling process.



3.MultiLevel Feedback Queue:

Instead of demanding a prior knowledge of the nature of a job, it observes the execution of a job and prioritizes it accordingly. In this way, it manages to achieve the best of both worlds: it can deliver excellent overall performance, and is fair and makes progress for long-running CPU-intensive workloads. If the process uses too much CPU time it will moved to a lower-priority queue. Similarly, a process that wait too long in the lower-priority queue may be moved to a higher-priority queue may be moved to a highest-priority queue.Thus it prevents starvation of a process.

It needs to schedule user processes by following rules:

MLFQ Procedure

Rule 1: When a job enters the system, it is placed at the highest priority (the topmost queue).

Rule 2: Once the job uses up its time quantum equal to 5 at a given level, its priority is reduced (i.e.,it moves down one queue).

Rule 3: Once the job uses up its time quantum equal to 10 at a given level, its priority is reduced (i.e., it moves down one queue).

Rule 4: Once the job uses up its time quantum equal to 20 at a given level, its priority is increased (i.e., it moves up to topmost (highest) priority queue).

Rule 5: After some time period S, move all the jobs in the system to the topmost queue.

My Implementation:

I changed the schdule.c file in 3 locations to change the existing round robin scheduling to MLFQ.

In **do_start_scheduling**()

I changed

```
rmp->max_priority = (unsigned) m_ptr->SCHEDULING_MAXPRIO;
rmp->priority   = USER_Q;
```

to

```
rmp->priority   =MAX_USER_Q;
mp->max_priority =MAX_USER_Q;
```

This is because, since we need to place the process in the max level queue from where we implement the MLFQ scheduling till 3 levels.

Since we need to change how a process is managed by scheduler once its time quantum is exhausted, in do_noquantum, I changed the body of the function and replaced it with.

do_no_quantum()

```
int priority =rmp->priority;
 int rm_endpt=(int)rmp->endpoint;
 if(priority==MAX_USER_Q)
 {
   printf( "Process [%d] has  priority %d and consumed quantum  :%d
\n",rm_endpt,MAX_USER_Q+1,DEFAULT_USER_TIME_SLICE );
       rmp->priority=MAX_USER_Q+1;/*INCREASING the priority and placing
in 17th */

       rmp->time_slice=2*DEFAULT_USER_TIME_SLICE;
    }
    else if(priority==MAX_USER_Q+1)
    {
        printf("Process [%d] has priority %d and consumed quantum  :%d
\n",rm_endpt,MAX_USER_Q+2,2*DEFAULT_USER_TIME_SLICE);
       rmp->priority=MAX_USER_Q+1;
```

```
        rmp->time_slice=4*DEFAULT_USER_TIME_SLICE;
        }
    else if(priority==MAX_USER_Q+2)
    {
         printf( "Process [%d] has priority %d and consumed quantum  :%d
\n",rm_endpt,MAX_USER_Q+2,4*DEFAULT_USER_TIME_SLICE);
        rmp->priority=MAX_USER_Q;
        rmp->time_slice=DEFAULT_USER_TIME_SLICE;
      }
    else
    {
         printf("Setting the process [%d]'s priorit y to
%d",rm_endpt,MAX_USER_Q);
        rmp->priority=MAX_USER_Q;
        rmp->time_slice=DEFAULT_USER_TIME_SLICE;
     }
```

This is because , whenever the quantum of a process expires if it is in the first level of MLFQ i.e MAX_USER_Q or the next, I increased its level and the quantum is made twice .If the quantum of the process in the third level expires, I pushed it back to the first level and changed the time quantum.

balance_queue()

I changed

```
rmp->priority -= 1;
```

to

```
rmp->priority=rmp->max_priority;
```

Whenever a process's priority increases above max_priority , we put it back to the level 1 of MLFQ. This is called for every 100 ticks to rebalance the queues. This function will find all processes that have been bumped down, and pulls them back up.

3.Priority Scheduling:

Every process in the system has a priority. Among the various processes in the run queue, the process with the highest priority is selected to execute in the CPU. If two processes has the same priority then FCFS is used for these processes.

In schedule.c ,

```
 /*
```

```
if (rmp->priority < MIN_USER_Q) {


        rmp->priority += 1;  lower priority
}*/
```

this part is commented out because, we should not change the level of the process that is preempted

In init_scheduling()

```
//set_timer(&sched_timer, balance_timeout, balance_queues, 0);
```

is commented out as there is no necessity for balancing the queues as the priority level is not changed.

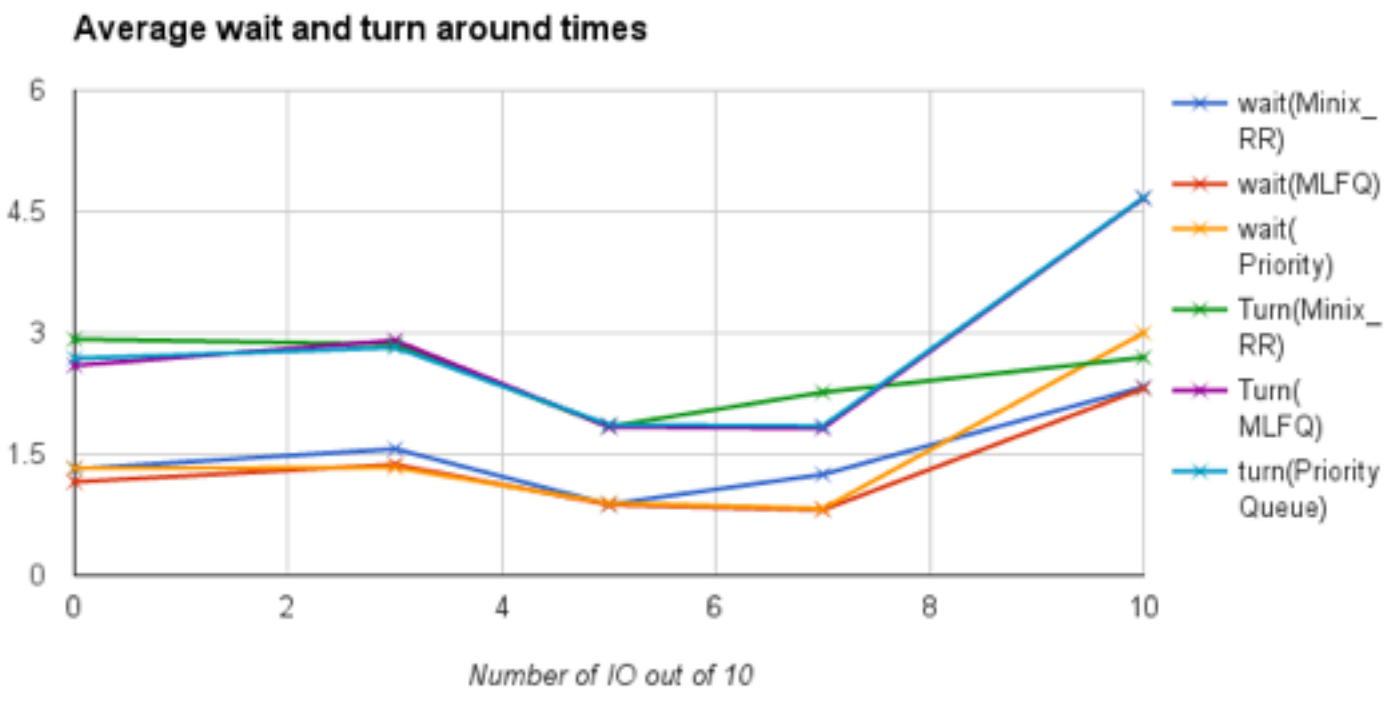
The following test file is used to get the time values.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
//Here , we are starting the IO and CPU bound processes in different ratios to
check the time variations
//longrun1 is a cpu bound and longrun2 is an io bound process
//Based on i%10 , we start either a CPU or IO bound process.
int main()
{
 int pid[15];
 int i;
 char processid[50];

    for(i=1;i<=10;i++)
    {
     pid[i]=fork(); //fork 10 processes
     if(pid[i]==0)
     {
          sprintf(processid,"%2d",i);
          if(i%10<3)
                execlp("./longrun0","./
longrun0",processid,"100000","1000",NULL); //create a CPU bound process
          else
                execlp("./longrun1","./
longrun1",processid,"100000","1000",NULL); //create an IO bound process
        }
    }
    for(i=1;i<=10;i++) //wait for everything to terminate
    {
     wait(NULL);
```

```
        }
        return 0;
    }

    /*
```



4.Analysis:

The above graph shows the variation of MLFQ,Priority queue and typical Minix round robin .

Observation on the number of IO processes:

During the first half, as the number of IO processes are increasing the waiting and turnaround time seem to reduce indicating that a good mix of CPU and IO bound processes will reduce the waiting and turn around times.

If we compare the initial and final graph values , all the time values are greater for IO processes compared to CPU bound processes. This shows that IO bound processes require more time to complete.

The slight raise from 0 to 3 might be because of the IO processes replacing the CPU processes which relatively take more time.

Observations on the time values for a particular ration:

Obviously, the turn around time will be more than the waiting time as few processes may not be preempted if they are short .But turn around time is the total time to get back to the preempted process.

In the graph for turn around time, as the IO bound processes are increasing, RR of minix emerges as the optimal .This is because there is more chance of preemption as the IO processes are long compared to the switching of processes.

But for waiting time , there isn't much difference between different scheduling algorithms.

From the above examples, for each and every combination of IO and CPU bound processes, MLFQ has almost the least time among all the scheduling algorithms. MLFQ is hence the preferable scheduling algorithm.This is because MLFQ reduces blocking by moving the longer process to lower priority and completing the shortest processes first which reduces the average waiting time.