MLFQ and Priority Scheduler in Minix

Written by Akilesh B, CS13B1042

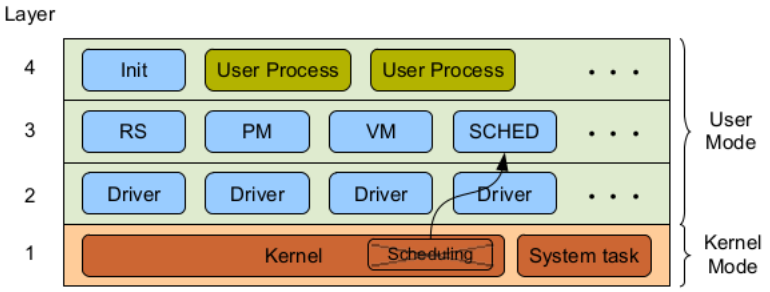
November 12, 2015

Objective:

To implement multi-level feedback based scheduler and a priority scheduler in Minix 3. The non-preemptive version of priority scheduler selects the process with highest priority among all the processes in the ready queue. The feedback scheduler does a similar thing using multiple queues. It all depends on the policy of the scheduler to move the processes across the queues based on the feedback. The feedback can be related to processes priorities, basically it is to allow I/O bound processes to be favoured by the scheduler like Linux kernel 2.6+.

Minix architecture:

MINIX 3 is a micro-kernel operating system structured in four layers. Some examples of services running in layers above the kernel are device drivers, networking and memory management. Since it has a user mode scheduler, it is possible to change the scheduling policy without modifying the kernel. Above all, it is possible to run multiple user mode schedulers each with its own policy. A user-mode scheduler in a multi-core environment can afford to spend more time evaluating scheduling decisions than the kernel can, since the system (and kernel) keep running on other cores.



Default scheduler in MINIX:

- It implements a simple pre-emptive RR scheduler. It always chooses the process at the head of queue with highest priority.
- If a process has exhausted its quantum, it is preempted and placed at the end of current priority queue with a new quantum.
- It is used in scheduling when the system starts. It is not starvation-free.

How I implemented?

Priority Scheduling:

- Every process is assigned a priority. Among the various processes in the run queue, the process with highest priority is allowed to execute.
- The priorities in this scheduler are internally defined. The priority of a process remains the same as its initial priority.
- The process can be either a pre-emptive or a non-preemptive process.

Changes made in code:

- The default scheduling policy in MINIX Scheduler is priority based pre-emptive RR scheduling with its default time quantum.
- Time quantum has to be changed. Increasing and decreasing the priorities have been taken care by the OS.
- To not preempt a process after it has consumed its time slice, increase its time quantum to infinity. This way processes are scheduled purely on the basis of priority.
- Remove priority in *do_noquantum* function.

The vicinity of the code after changes in *do_noquantum* looks as below:

```
rmp = &schedproc[proc_nr_n];

//Remove priority code here

if ((rv = schedule_process_local(rmp)) != OK) {
return rv;
}

return OK;
```

- Include this line at the starting: #define DEFAULT_USER_TIME_SLICE 0xffff
- In function *init_scheduling*, remove this: set_timer(&sched_timer, balance_timeout, balance_queues, 0);

Multiple level feedback queue scheduling:

MLFQ is interesting because instead of demanding a priori knowledge of the nature of a job, it instead looks at the execution of the job and prioritizes it accordingly. So, it manages excellent overall performance similar to SJF for short-running interactive jobs, is fair and makes progress for long running CPU-intensive workloads. Scheduing is done by following the rules below:

- When a job enters the system, it is placed at the highest priority (top most queue).
- Once the job uses its time quantum equal to 5 at a given level, its priority is reduced (moves down one queue).
- Once the job uses up its time quantum equal to 10 at a given level, its priority is reduced (moves down another queue).
- Once the job uses its time quantum equal to 20 at a given level, its priority is increased (moves up to highest priority queue).
- After some time period S, move all the jobs in the system to the topmost queue.

Hence, the algorithm chooses a process with highest priority from the queue and run that process preemptively. A process which waits too long in the lower-priority queue may be moved to a higher priority queue which can subsequently be moved to a highest-priority queue. This prevents starvation.

Algorithm:

Process can be separated into categories based on its need for the processor. Scheduling in each level of Multi-level Queue is done through Round Robin (RR).

- A new process which arrives is inserted at the end (tail) of top-level queue.
- The process at some point of time reached the head of the queue and is assigned to the CPU.
- If a given process completes its execution within the quantum of the given queue, it exits the system.
- If a process uses its entire quantum assigned to it, it is pre-empted and inserted at the tail of next lower level queue. This lower level queue will have a quantum time which is bigger than its previous higher level queue.
- These steps follow until the process finishes its execution or reaches the base queue after which it will be again put back into the top level queue and this repeats.

Changes in code:

Parameters used in the implementation of the algorithm are as follows:

- Number of queues - 3
- Scheduling algorithm in each queue - Preemptive Round Robin Scheduling.
- Priority number is used to determine when a process is promoted to a higher/lower priority queue.

Changes in do_noquantum:

```
rmp = &schedproc[proc_nr_n]; //The code above this line remains the same

    //Changes made
    int prio_val=rmp->priority;
    int end_pnt=rmp->endpoint;

    if(prio_val == MAX_USER_Q) {

    sprintf(buf,"Process %d consumed time-slice/quantum %d and priority is
%d\n",(int)rmp->endpoint,DEFAULT_USER_TIME_SLICE,MAX_USER_Q);

    rmp->priority=MAX_USER_Q+1;
    rmp->time_slice=2*DEFAULT_USER_TIME_SLICE;

    }

    else if(prio_val == MAX_USER_Q+1) {

    sprintf(buf,"Process %d consumed time-slice/quantum %d and priority is
%d\n",(int)rmp->endpoint,2*DEFAULT_USER_TIME_SLICE,MAX_USER_Q+1);

    rmp->priority=MAX_USER_Q+2; //move to third queue
    rmp->time_slice=4*DEFAULT_USER_TIME_SLICE;

    }

    else if(prio_val == MAX_USER_Q+2) {

    sprintf(buf,"Process %d consumed time-slice/quantum %d and priority is
%d\n",(int)rmp->endpoint,4*DEFAULT_USER_TIME_SLICE,MAX_USER_Q+2);

    rmp->priority=MAX_USER_Q; //move to first queue

    rmp->time_slice=DEFAULT_USER_TIME_SLICE;

    }

    else{

    sprintf(buf,"Process %d priority set to %d\n",(int)rmp->endpoint,MAX_USER_Q);

    rmp->priority=MAX_USER_Q;
    rmp->time_slice=DEFAULT_USER_TIME_SLICE;
```

```
}
printf("%s\n", buf);
//Code below this remains same as before
if ((rv = schedule_process_local(rmp)) != OK) {
return rv;
}
return OK;
```

Testing:

The scheduling algorithms were exhaustively tested for different ratios of I/O and CPU bound processes.

Code:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
//create 10 instances of longrun – 5 CPU bound and 5 IO bound
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%i)
execlp("./longrun00","./longrun00",processid,"100000","1000",NULL); //cre-
ate a CPU bound process
else
execlp("./longrun01","./longrun01",processid,"100000","1000",NULL); //cre-
ate an IO bound process
}
}
for(i=1;i<=10;i++) //wait for everything to terminate {
wait(NULL);
}
return 0;
}
```

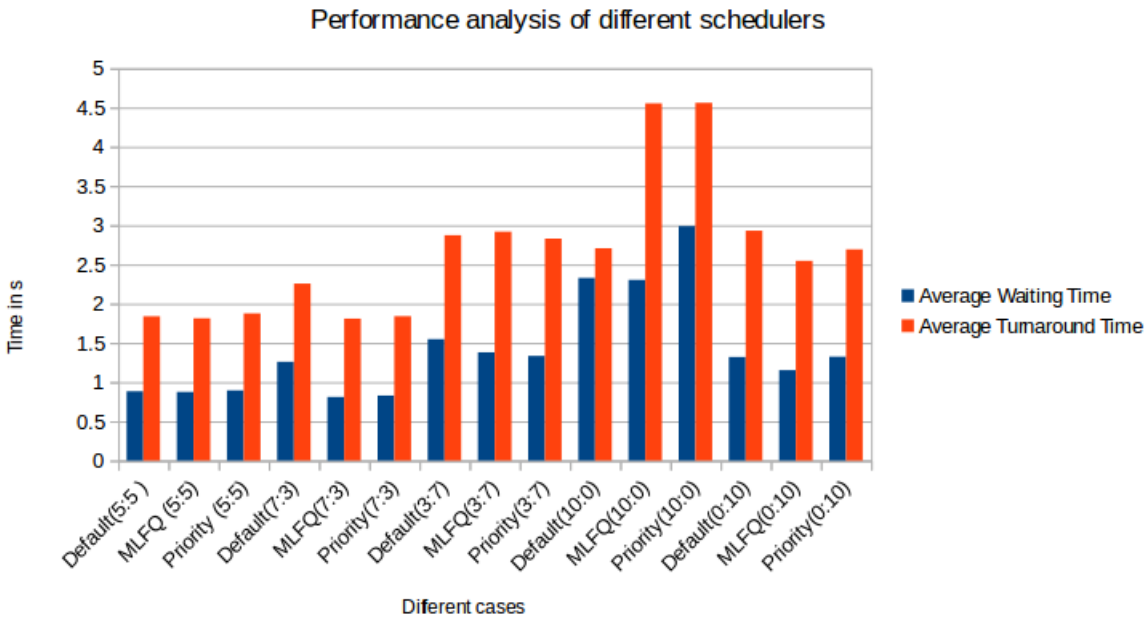
By changing if condition we get values for different ratios. (like if(i%2) for equal number of CPU and I/O bound processes).

Screenshot:

```
The IO bound final value of v is 0xc664ceec
Turnaround time = 4.666666 s
Waiting time = 2.299996 s
The IO bound final value of v is 0xa0a514ed
Turnaround time = 4.666666 s
Waiting time = 2.216668 s
The IO bound final value of v is 0x7ae55aee
Turnaround time = 4.666666 s
Waiting time = 2.433331 s
The IO bound final value of v is 0x5525a0ef
Turnaround time = 4.666666 s
Waiting time = 2.466664 s
The IO bound final value of v is 0x2f65e6f0
Turnaround time = 4.666666 s
Waiting time = 2.200002 s
```

Performance analysis of different scheduling algorithms:

The below graph shows the results obtained for various cases (Ratio is (I/O bound process : CPU bound process)).



- The average waiting and turnaround times depend specifically on whether they are IO bound or CPU bound processes in addition to total number of user processes.
- The algorithm with minimum waiting time is most efficient. The turnaround time depends on the kind of process.
- In most cases MLFQ scheduling performs the best. This is because of the fact that MLFQ is an algorithm which is the closest to the SJF algorithm. It ensures that the processes which have small completion time are set at the top of the Multi level queue.
- Default scheduling is the worst in most of the cases because of the time-slice quantum which increases the waiting times for the IO bound processes.
- It is evident from the graph that the average turnaround time is the highest for IO bound processes because of the fact that the IO bound processes wait more than the CPU bound processes.