Scheduling

The following Project was Executed in three Scheduling Policies:

1. Round-Robin (RR)
2. First Come First-Served (FCFS)
3. Multi-level Feedback Queue (MLFQ)

```
make clean
make qemu SCHEDULER=<Schedulertype>
```

Here < schedulertype > can be DEFAULT for RR , FCFS for FCFS, MLFQ for MLFQ

Round-Robin

In the round robin scheduling, each process gets a fixed time slice called as quantum. Once it is executed completely within its time slice, In this case it is 1 tick.

Implementation

In `Scheduler`:

```
for (p = proc; p < &proc[NPROC]; p++)
    {
      acquire(&p->lock);
      if (p->state == RUNNABLE)
      {
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);
        c->proc = 0;
      }
      release(&p->lock);
    }
```

In usertrap() and kerneltrap(), `yield()` is used to interrupt the process.

Output

```
Process 5 finished
Process 7 finished
Process 9 finished
Process 8 finished
Process 6 finished
Process 0 finished
Process 1 finished
Process 2 finished
```

```
Process 3 finished
Process 4 finished

Average rtime 11,  wtime 146
```

First Come First-Served

FCFS is a simple scheduling algorithm where processes are executed in the order they arrive. The Processes that come first will be executed first.

Implementation

In `usertrap` and `kerneltrap`: Yield is disabled inorder to handle preemption cause the process should run till completion without preemption.

```
usertrap()
#ifndef FCFS
  if (which_dev == 2)
    yield();
#endif
```

```
kerneltrap()
ifndef FCFS
  if (which_dev == 2 && myproc() != 0 && myproc()->state == RUNNING)
    yield();
#endif
```

In proc.c

The schedule policy is determined by ifdef in the scheduler function

```
#ifdef FCFS
  for (;;)
  {
    intr_on();
    struct proc *pres_proc = 0;
    for (p = proc; p < &proc[NPROC]; p++)
    {
      if (p->state == RUNNABLE && (pres_proc == 0 || pres_proc->ctime > p->ctime))
      {
        pres_proc = p;
      }
    }
    if (pres_proc != 0)
    {
      if (pres_proc->state == RUNNABLE)
      {
        acquire(&pres_proc->lock);
```

```
        pres_proc->state = RUNNING;
        c->proc = pres_proc;
        swtch(&c->context, &pres_proc->context);
        c->proc = 0;
        release(&pres_proc->lock);
      }
      release(&p->lock);
    }
  }
#endif
```

Here the ctime is compared which is the arrival time and the minimum is taken and was run. Since there is no preemption (Yield), It will run till its end.

Output

```
Process 5 finished
Process 6 finished
Process 7 finished
Process 8 finished
Process 9 finished
Process 0 finished
Process 1 finished
Process 2 finished
Process 3 finished
Process 4 finished

Average rtime 11,  wtime 123
```

Usage

To enable FCFS scheduling during the compilation of xv6, use the `FCFS` flag. For example:

```
make clean
make qemu SCHEDULER=FCFS
```

Multilevel Feedback Queue

Multi-Level Feedback (MLFQ) is a sophisticated scheduling algorithm that employs multiple queues with varying priorities (0 has highst priority and decreases till 3) to manage processes efficiently. These queues has their own time slices to send the running process to next level and also the other processes in the same Queue runs in Round Robin.

Graphs

For waittime of 10 ticks

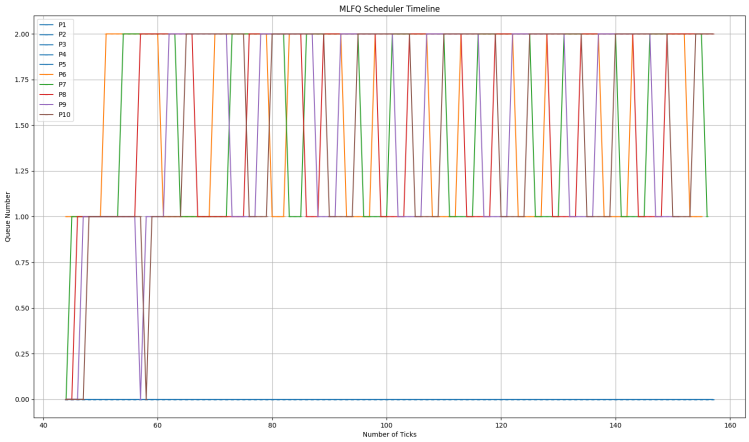

Figure 1: Waittime_10

`Output: Average rtime 11,  wtime 146`

For waittime of 20 ticks

`Output: Average rtime 11,  wtime 139`

For waittime of 30 ticks

`Output: Average rtime 11,  wtime 143`

For waittime of 40 ticks

`Output: Average rtime 11,  wtime 139`

Implementation

1. Process Structure Modifications

Several fields have been added to the `struct proc` to support MLFQ scheduling:

- `que_no`: Denotes the queue number where the process is placed.
- `proc_no`: Represents the position of the process in its queue.
- `que_on`: Indicates whether the process is in a queue or not.
- `wait_time`: Tracks the waiting time of the process.
- `stime`: Records the time the process has spent in the RUNNING state.

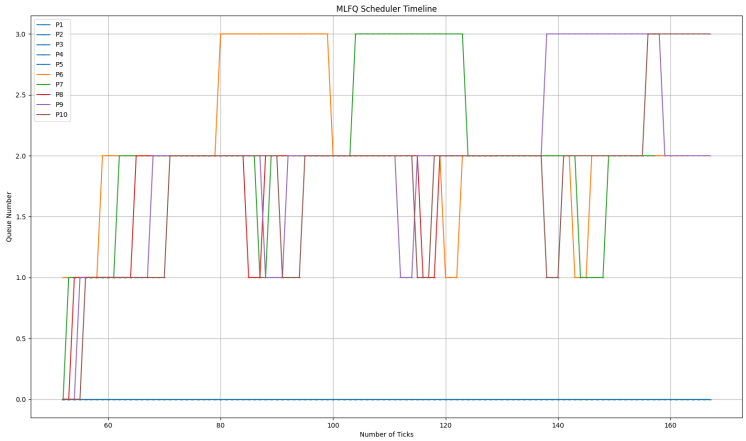

Figure 2: Waittime_20

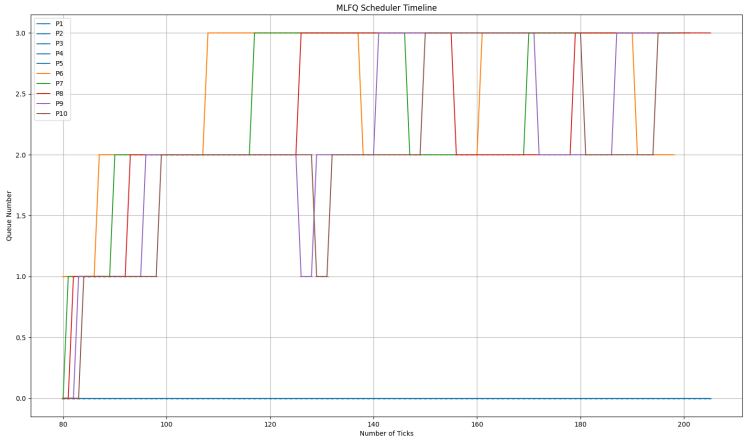

Figure 3: Waittime_30

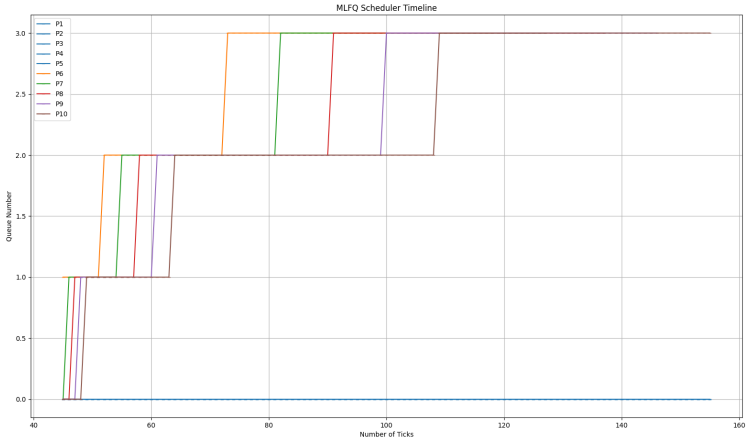

Figure 4: Waittime_40

2. MLFQ Queues

Four MLFQ queues are implemented, each with a different maximum time constraint {1, 3, 9, 15}. Processes move between these queues based on their behavior.

3. Initialization

The `init_que` function initializes the MLFQ queues with specific maximum time constraints given and que_on and que_no with 0.

4. Queue Operations

- `mlfq_push`: Adds a process to a queue at back.
- `mlfq_push_front`: Adds a process to the front of a queue.
- `mlfq_pop`: Removes and returns a process from a queue.
- `deque`: Removes the given process from a queue.

5. Scheduling Logic

In the `scheduler` function, processes are placed into queues based on their state (if RUNNING) and que-on status is updated to 1. Processes are moved between queues to ensure they meet their wait time constraints. Here the first process in the least priority is selected and popped, if it is runnable, insert it again in the front of queue to run it again after returning back from trap, when the running time goes greater than assigned time slice, the que_no is increased to push to

next level by popping it. When the waittime of a process is morethan given waittime limit, the que_no is decreased to get more priority.

Process:

In sechuduler function

a. The Runnable processes in the `proc` array is pushed at back of que_no (initialised with 0) by checking if it is already in queue. The que_on flag is set to 1 to indicate its presence in Queue.

```
struct proc *p = 0;
    for (p = proc; p < &proc[NPROC]; p++)
    {
      if (p->state == SLEEPING || p->state == UNUSED)
      {
        continue;
      }
      if (p->state == RUNNABLE && p->que_on == 0)
      {
        int que_no = p->que_no;
        mlfq_push(p, que_no);
        p->que_on = 1;
      }
    }
```

b. Selecting the first Runnable process in the first non empty queue by popping out queues continuously until a Runnable process was found. When found Push it again in the front of the Queue.

```
struct proc *pres_proc = 0;
for (int i = 0; i < 4; i++)
    {
      if (pres_proc != 0)
      {
        break;
      }
      while (mlf_Que[i].tail > 0)
      {
        pres_proc = mlfq_pop(i);
        pres_proc->que_on = 0;
        if (pres_proc->state == RUNNABLE)
        {
          mlfq_push_front(pres_proc, pres_proc->que_no);
          break;
        }
      }
    }
```

c. Running the found Runnable Process by context switching and CPU running process to present found process.

```
if (pres_proc != 0)
    {
      if (pres_proc->state == RUNNABLE)
      {
        acquire(&pres_proc->lock);
        pres_proc->state = RUNNING;
        c->proc = pres_proc;
        swtch(&c->context, &pres_proc->context);
        c->proc = 0;
        release(&pres_proc->lock);
      }
    }
```

In usertrap function

d. Check if the Running time of the Process of present Running process is more than time slice of present Queue. If it is greater pop the queue and increase the queue level (decrease priority) of the process such that it can be added to next level queue when it goes to scheduler again. Again reseting the runtime (stime) and also wait time starts from now. The que_no is left 3 it is in queue 3. As it always add it to last of Queue if que_on is 0.

```
  if (p->stime > mlf_Que[p->que_no].max_ticks)
  {
    if (mlf_Que[p->que_no].tail > 0)
    {
      mlfq_pop(p->que_no);
      p->que_on = 0;
    }
    if (p->que_no < 3)
    {
      p->que_no++;
    }
    p->stime = 0;
    p->wait_time = 0;
  }
```

e. Inorder to prevent ageing, check if the wait_time of the processes which are RUNNABLE and increase their priority (decraese queue level) if it more than some predefined threshhold (reset point) inorder to boost it to high priority by removing from present queue and again going to scheduler function.

```
  for (struct proc *pres_proc = proc; pres_proc < &proc[NPROC]; pres_proc++)
  {
```

```
    if (pres_proc != 0 && pres_proc->state == RUNNABLE && pres_proc->que_on == 1)
    {
      if (pres_proc->wait_time >= 30)
      {
        pres_proc->que_on = 0;
        deque(pres_proc->que_no, pres_proc);
        if (pres_proc->que_no > 0)
        {
          pres_proc->que_no--;
        }
        pres_proc->wait_time = 0;
      }
    }
  }
```

As the processes at least priority runs first, the processes run in Round robin as the next element of present running process will be first process if present running process goes to next queue or added at last.

6. Yield Behavior

The `yield` function ensures that processes go to RUNNABLE state from RUNNING. if they are in the SLEEPING state (got from premption as to reserve parent process until child completes), it is left in sleep state, preventing preemption errors.

```
void yield(void)
{
  struct proc *p = myproc();
  acquire(&p->lock);
  p->state = RUNNABLE;
#ifdef MLFQ
  if (p->state != SLEEPING)
  {
#endif
    p->state = RUNNABLE;
#ifdef MLFQ
  }
#endif
  sched();
  release(&p->lock);
}
```

7. Updating Process Statistics

In the `updatetime` function, process statistics such as `stime` and `wait_time` are updated to track the time spent in Running and Waiting Period.

Output for 30 Waittime

```
Process 8 finished
Process 5 finished
Process 9 finished
Process 6 finished
Process 7 finished
Process 0 finished
Process 1 finished
Process 2 finished
Process 3 finished
Process 4 finished

Average rtime 12,  wtime 145
```

Usage

To enable MLFQ scheduling during the compilation, use the `MLFQ` flag. For example:

```
make clean
make qemu CPUS=1 SCHEDULER=MLFQ
```

Overall Comparision among the Scheduler Policies:

1. FCFS is better among all for the average turn-around time.
2. Round Robin is better for average response-time.
3. MLFQ is better in the case of different time slots and waiting processes.