Dooik Kim

Professor Kang

ELE3021

20 April 2024

Operation System Project02

The operation system should schedule given tasks according to the system designer's predesigned policies. The scheduler is an integral component of the operation system to ensure that the CPU is utilized efficiently. This project aims to substitute xv6's Round Robin scheduler with a scheduler implemented with Multi-Level Scheduler Queue(MLFQ) and Monopoly Queue(MoQ). A precise explanation of each queue's implementation will be discussed later.

The Round Robin scheduler of xv6 executes a "for loop" that iterates from the start of the ptable, an array of processes, to the end of the ptable until it meets a runnable process. However, several problems can be held: it can run at the time complexity of O(n) that can be considered slightly slow, and if a CPU-driven process is allocated at the front of the ptable, the process allocated behind the CPU-driven process will wait indefinitely and can lead to the starvation problem. A more efficient scheduler is needed for improved performance.

The usage of ptable would be a restriction to a better algorithm since the insertion and deletion process could not be faster than O(n) without any alteration. The Queue and Maxheap data structures could be good alternatives to ptable. Queue and Maxheap data structures accomplish the insertion and deletion process in constant time. Although the heapify operation is done in the time complexity of O(log n) and should be executed whenever the insertion and deletion function is called, it is still better than O(n).

Implementation of the Queue and Heap is needed as the xv6 does not have built-in Queue and Heap data structures. The implementation of Queue and Heap is shown below:

```c
//queue.h
#include "proc.h"
#include "defs.h"
#include "param.h"

struct queue {
    struct proc* proc[NPROC];
    int front;
    int rear;
};

void
initqueue(struct queue* q)
{
    q->front = 0;
    q->rear = 0;
}

int
sizeofqueue(struct queue* q)
{
    return (q->rear - q->front + NPROC) % NPROC;
}

int
isfullqueue(struct queue* q)
{
    if(q->front == 0 && q->rear == NPROC - 1)
        return 1;
    if(q->front == (q->rear + 1) % NPROC)
        return 1;
    return 0;
}

int isemptyqueue(struct queue* q)
{
    if(q->front % NPROC == q->rear % NPROC)
        return 1;
    return 0;
}


void
insertqueue(struct queue* q, struct proc* p)
{
    if(isfullqueue(q))
    {
        return;
    }
    q->proc[q->rear % NPROC] = p;
    q->rear = (q->rear + 1) % NPROC;
}

struct proc*
deletequeue(struct queue* q)
{
    struct proc* p;
    if(isemptyqueue(q))
    {
        return 0;
    }
    p = q->proc[q->front % NPROC];
    q->front = (q->front + 1) % NPROC;
    return p;
}
```

Picture 1. queue.h

```c
//heap.h
#include "proc.h"
#include "defs.h"
#include "param.h"

struct heap {
    struct proc* proc[NPROC];
    int size;
};

void
initheap(struct heap* h)
{
    h->size = 0;
}

int
isfullheap(struct heap* h)
{
    if(h->size == NPROC)
        return 1;
    return 0;
}

int
isemptyheap(struct heap* h)
{
    if(h->size == 0)
        return 1;
    return 0;
}


void
heapify(struct heap* h, int index)
{
    int largest = index;
    int left = 2 * index + 1;
    int right = 2 * index + 2;

    if (left < h->size && h->proc[left]->priority > h->proc[largest]->priority)
        largest = left;

    if (right < h->size && h->proc[right]->priority > h->proc[largest]->priority)
        largest = right;

    if (largest != index) {
        struct proc* temp = h->proc[index];
        h->proc[index] = h->proc[largest];
        h->proc[largest] = temp;
        heapify(h, largest);
    }
}


struct proc*
deleteheap(struct heap* h)
{
    if (isemptyheap(h))
        return 0;

    struct proc* root = h->proc[0];

    h->proc[0] = h->proc[h->size - 1];
    h->size--;

    heapify(h, 0);

    return root;
}
void
insertheap(struct heap* h, struct proc* p)
{
    if (isfullheap(h))
        return;

    h->proc[h->size] = p;
    h->size++;

    int i = h->size - 1;
    while (i != 0 && h->proc[(i - 1) / 2]->priority < h->proc[i]->priority) {
        struct proc* temp = h->proc[i];
        h->proc[i] = h->proc[(i - 1) / 2];
        h->proc[(i - 1) / 2] = temp;
        i = (i - 1) / 2;
    }
}
```

Picture 2.heap.h

MLFQ is structured with three queues and one heap. Since L0, L1, and L2 follow Round Robin policies with different time quantum, they are implemented with queues. L3 does priority scheduling, so it is implemented with a priority maxheap. Moq does an FCFS scheduling so it can be constructed with a queue.

```
//proc.c
struct {
  struct queue q[3];
  struct heap h;
} mlfq;

struct {
  struct queue q;
} moq;
```

Picture 3. MLFQ and MoQ

Let's delve deeper into the intricacies of MLFQ. When a process fully utilizes the allocated time quantum at a particular queue level, it's crucial to promote the process to the next tier of MLFQ based on its current level and pid. If the process is at the lowest tier of the MLFQ, it is promoted to L1 or L2 depending on whether its pid is odd or even. Specifically, if the pid is odd, the process is inserted into L1; if the pid is even, the process is inserted into L2. If the process is already at the highest level of MLFQ, its priority is decreased by one and stays at the highest level of MLFQ. Implementing these modifications requires potential adjustments to the tick incrementation process. However, in the xv6 operating system, the tick incrementation process is located in trapasm.S, operating at the register level, which poses a significant challenge for modification due to its out-of-scope nature in typical operating system development. Given these considerations, modifying the timer interrupts in a trap.c file emerges as a pragmatic choice. To accomplish this, several member variables need to be attached to the struct proc. These member variables, namely int level, int tq, and int priority, represent the current level in which

```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER){
    //cprintf("ticks: %d, pid: %d, priority: %d, level: %d\n", ticks, myproc()->pid, myproc()->priority, myproc()->level);
    if(myproc()->monopoly == 0) {
      if(ticks % 100 == 0) {
        yield();
      }
      int level = myproc()->level;
      int tq = ++myproc()->tq;
      if(tq >= 2 + level * 2) {
        if(myproc()->level == 3) {
          if(myproc()->priority > 0) {
            myproc()->priority--;
          }
        } else if(level == 1 || level == 2) {
          myproc()->level = 3;
        } else if(level == 0) {
          if(myproc()->pid % 2) {
            myproc()->level = 1;
          } else {
            myproc()->level = 2;
          }
        }
        myproc()->tq = 0;
        yield();
      }
    } else if(myproc()->monopoly == 1) {
      if(mycpu()->monopoly == 0) {
        yield();
      };
    }
  }
```

Picture 4. Handling time interrupt

```
int priority;                    //process priority
int level;                       //queue level
int tq;                          //time quantum
```

Picture 5. The member variables newly added

the process resides, the time quantum that the process has spent, and the priority of the process, respectively.

Let's delve into MoQ (Monopoly Queue) in detail. When the 'setmonopoly(int pid, int password)' function is invoked, the process's monopoly variable corresponding to the provided pid parameter is set to 1, and the process is subsequently inserted into MoQ. If the process is already designated as a monopoly process, it returns -3. Additionally, if the process itself attempts to set its monopoly variable to 1, it should invoke 'yield()' as this scenario occurs only when the scheduler is in unmonopolized mode. Upon invocation of the syscall 'monopolize()', the scheduler transitions to monopolized mode, exclusively scheduling processes from MoQ. To accomplish this specification, an auxiliary member variable ‘monopoly’ should be appended to ‘struct proc’ in ‘proc. c’. When MoQ becomes empty, the scheduler automatically reverts to the

unmonopolized mode by invoking ‘unmonopolize()’, resuming scheduling processes from MLFQ.

```
int monopoly;                    //monopoly queue check
```

Picture 6. Member variable ‘int monopoly’ in ‘struct proc’

```
//proc.c
int
setmonopoly(int pid, int password) {
  if(password != 2020099743) return -2;
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->pid == pid) {
      if(p->monopoly == 1) {
        cprintf("The process trying to be set monopolized is already monopolized\n");
        return -3;
      }
      p->monopoly = 1;
      p->level = 99;
      insertmoq(p);
      release(&ptable.lock);
      if(myproc()->pid == pid) {
        yield();
      }
      return sizeofqueue(&(moq.q));
    }
  }
  release(&ptable.lock);
  return -1;
}
```

Picture 7. setmonopoly(int pid, int password)

Monopolization and unmonopolization are implemented by introducing a new member variable, 'monopoly', within the 'struct cpu' in 'proc.h'. Additionally, whenever a transition occurs between monopolized and unmonopolized modes, 'pushcli()' and 'popcli()' must be invoked. This is crucial because any interruption during CPU access is severely restricted.

The issue of time interrupts becomes less critical when the scheduler is in monopolized mode, as the MoQ adheres to a First-Come, First-Served (FCFS) policy. However, a significant challenge arises when handling sleeping processes in MoQ. This occurs when all processes in MoQ are sleeping, causing the scheduler to incorrectly assume that MoQ is empty and revert to

unmonopolized mode. This problem can be solved by re-inserting the process whenever it invokes ‘sleep()’ and when the scheduler schedules the sleeping process.

```
// Per-CPU state
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?
  struct proc *proc;           // The process running on this cpu or null
  int monopoly;                // Is monopolized?
};
```

Picture 8. ‘Struct cpu’ in ‘proc.h’

```
//proc.c
void
monopolize(void) {
  pushcli();
  struct cpu *c = mycpu();
  c->monopoly = 1;
  popcli();
}

void
unmonopolize(void) {
  pushcli();
  struct cpu *c = mycpu();
  c->monopoly = 0;
  ticks = 0;
  popcli();
}
```

Picture 9. monopolze() and unmonopolize()

Before discussing the context-switching problem, it's necessary to provide a brief explanation of the advanced scheduler. Let's break down the scheduler into two parts: monopolized mode and unmonopolized mode. In unmonopolized mode, the scheduler traverses from L0 to L3, checking whether each queue is empty. If a queue is not empty and a runnable process is found, a context switch occurs. In monopolized mode, the scheduler also checks whether the MoQ is empty. If the MoQ is not empty and the first process in the queue is runnable, a context switch occurs.

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    int flag = 1;
    if(c->monopoly == 0)  {
      for(int i = 0; i < 3; i++){
        if(!isemptyqueue(&(mlfq.q[i]))){
          p = deletequeue(&(mlfq.q[i]));
          if(p->monopoly == 1 || p->state != RUNNABLE) {
            p = 0;
            flag = 0;
            break;
          }
          flag = 0;
          c->proc = p;
          switchuvm(p);
          p->state = RUNNING;
          swtch(&(c->scheduler), p->context);
          switchkvm();
          c->proc = 0;
          break;
        }
      }
      if(flag == 1) {
        if(!isemptyheap(&(mlfq.h))) {
          p = deleteheap(&(mlfq.h));
          if(p->monopoly == 1 || p->state != RUNNABLE) {
            p = 0;
            continue;
          }
          c->proc = p;
          switchuvm(p);
          p->state = RUNNING;
          swtch(&(c->scheduler), p->context);
          switchkvm();
          c->proc = 0;
        }
      }
    } else if (c->monopoly == 1){
      if(!isemptyqueue(&(moq.q))) {
        p = deletequeue(&(moq.q));
        if(p->state != RUNNABLE) {
          insertmoq(p);
          release(&ptable.lock);
          continue;
        }
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;
        swtch(&(c->scheduler), p->context);
        switchkvm();
        c->proc = 0;
      } else {
        unmonopolize();
      }
    }
    release(&ptable.lock);
  }
}
```

Picture 10. scheduler

The context-switching occurs under four conditions: when a process voluntarily relinquishes the CPU, such as by calling a system call like ‘sleep()’, when a process’s time quantum expires, when a process wakes up, and when a process terminates. In this project, a context switch also occurs when the global ticks reach one hundred and ‘priorityboosting()’ is invoked. Whenever each case except the process termination occurs, the currently running process should be enqueued because the process is not terminated yet, indicating that there are remaining instructions waiting to be executed. Nevertheless, the solution for this problem varies slightly between monopolized mode and unmonopolized mode.

In unmonopolized mode, the solution is to enqueue processes back to the MLFQ whenever they wake up by invoking ‘wakeup().’ If the process at the front of each queue is sleeping, the scheduler should bypass the process and reschedule to the next process. The rationale behind enqueueing sleeping processes when they wake up is that unmonopolized mode persists until the ‘monopolize()’ system call is invoked.

On the contrary, the solution is to enqueue processes back to the MoQ whenever they sleep, and when the scheduler schedules the sleeping process in the MoQ. This is because the scheduler determines whether to remain in monopolized mode or transition to unmonopolized mode by examining the queue's size. To ensure that the queue’s size remains greater than one until every process in MoQ is terminated, enqueuing the process in 'sleep()' and when the scheduler schedules the sleeping process is necessary.

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();
  if(p == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;
  if(p->monopoly == 1) {
    insertmoq(p);
  }
  sched();
```

Picture 11. ‘sleep()’

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  if(mycpu()->monopoly == 0) {
    if(myproc()->monopoly == 0)
      insertmlfq(myproc());
    if(ticks % 100 == 0) {
      priorityboosting();
      ticks = 0;
    }
  } else if(mycpu()->monopoly == 1) {
    if(myproc()->monopoly == 0) {
      insertmlfq(myproc());
    }
  }
  sched();
  release(&ptable.lock);
}
```

Picture 12. ‘yield()’

```
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->state == SLEEPING && p->chan == chan) {
      p->state = RUNNABLE;
      if(p->monopoly == 0) {
        insertmlfq(p);
      }
    }
  }
}
```

Picture 13. ‘wakeup()’

```
void
priorityboosting(void) {
  struct proc *p;
  for(int i = 0; i < 3; i++) {
    initqueue(&(mlfq.q[i]));
  }
  initheap(&(mlfq.h));
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->state == RUNNABLE && p->monopoly == 0) {
      insertqueue(&(mlfq.q[0]), p);
      p->tq = 0;
      p->level = 0;
    }
  }
}
```

Picture 14. ‘Priorityboosting()’

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;
  np->level = 0;
  np->tq = 0;
  np->monopoly = 0;
  insertmlfq(np);
  release(&ptable.lock);

  return pid;
}
```

Picture 15. ‘fork()’

For testing purposes, several system calls are required: ‘yield(),’ ‘getlev(),’ and ‘setpriority().’ Since the functionality of ‘yield()’ is quite obvious, it will be skipped here. ‘getlev()’ returns the queue level of the running process, while ‘setpriority()’ sets the priority of the process and returns 0 upon success.

```
int
getlev(void) {
  return myproc()->level;
}
```

Picture 16. ‘getlev()’

```c
//proc.c
int
setpriority(int pid, int priority) {
  if(priority < 0 || priority > 10) return -2;
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->pid == pid) {
      p->priority = priority;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

Picture 17. ‘setpriority(int pid, int priority)’

The result of the test 1 is below:

```
[Test 1] default
Process 5
L0: 9181
L1: 18448
L2: 0
L3: 72371
MoQ: 0
Process 10
L0: 6815
L1: 0
L2: 20870
L3: 72315
MoQ: 0
Process 8
L0: 12521
L1: 0
L2: 34821
L3: 52658
MoQ: 0
Process 7
L0: 13317
L1: 27838
L2: 0
L3: 58845
MoQ: 0
Process 6
L0: 15995
L1: 0
L2: 41834
L3: 42171
MoQ: 0
Process 9
L0: 17345
L1: 36120
L2: 0
L3: 46535
MoQ: 0
Process 4
L0: 16153
L1: 0
L2: 48396
L3: 35451
MoQ: 0
Process 11
L0: 17716
L1: 31322
L2: 0
L3: 50962
MoQ: 0
[Test 1] finished
```

Picture 18. Result of the test 1

It is shown that the processes with odd pids go to L1 and the processes with even pids go to L2.

The result of the test 2 is below:

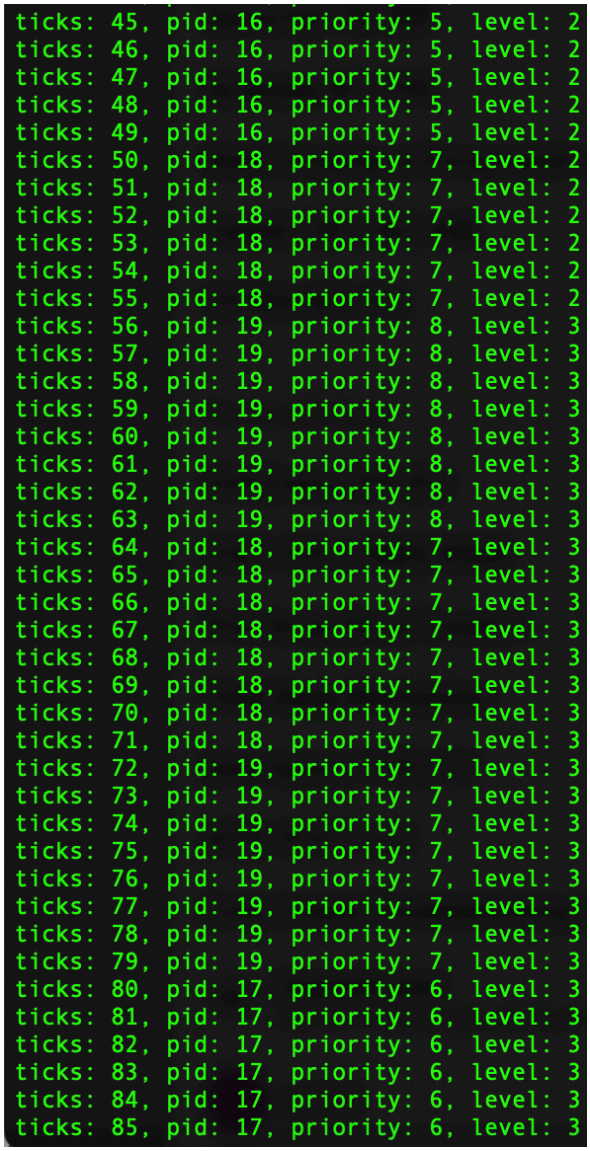
```
[Test 2] priorities
Process 19
L0: 9100
L1: 23214
L2: 0
L3: 67686
MoQ: 0
Process 17
L0: 15015
L1: 27933
L2: 0
L3: 57052
MoQ: 0
Process 16
L0: 14924
L1: 0
L2: 41793
L3: 43283
MoQ: 0
Process 12
L0: 17146
L1: 0
L2: 49046
L3: 33808
MoQ: 0
Process 13
L0: 18459
L1: 32537
L2: 0
L3: 49004
MoQ: 0
Process 14
L0: 17365
L1: 0
L2: 48791
L3: 33844
MoQ: 0
Process 15
L0: 20752
L1: 37227
L2: 0
L3: 42021
MoQ: 0
Process 18
L0: 11599
L1: 0
L2: 34456
L3: 53945
MoQ: 0
[Test 2] finished
```

Picture 19. Result of the test 2

It might seem that the processes end regardless of the priority. Therefore, the code below was added temporarily to ‘trap.c’ to allow observation of how the queue operates at each level.

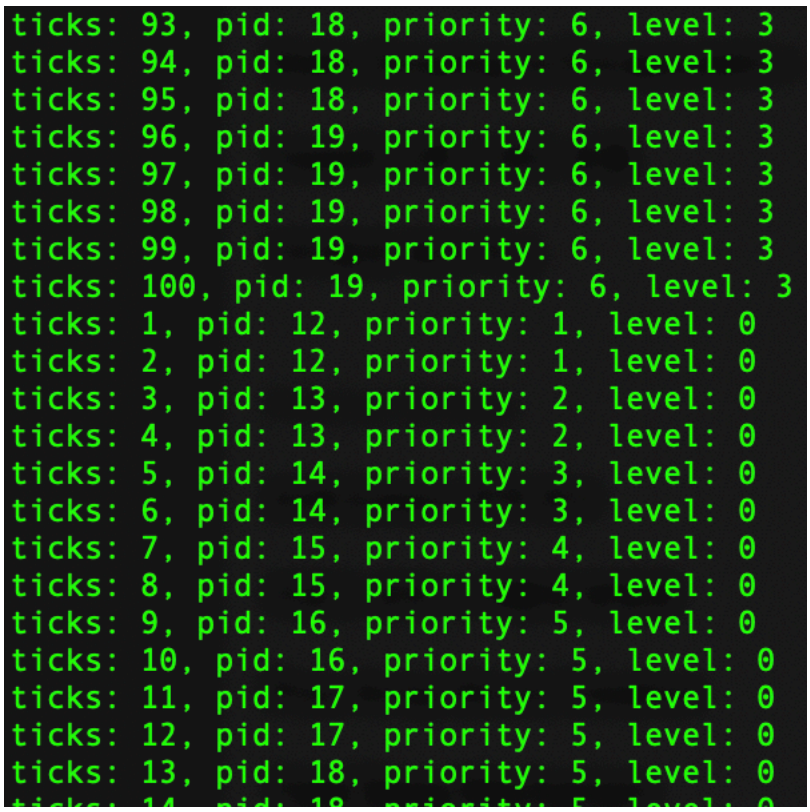
```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER){
    cprintf("ticks: %d, pid: %d, priority: %d, level: %d\n", ticks, myproc()->pid, myproc()->priority, myproc()->level);
```

Picture 20. Temporary code in ‘trap.c’



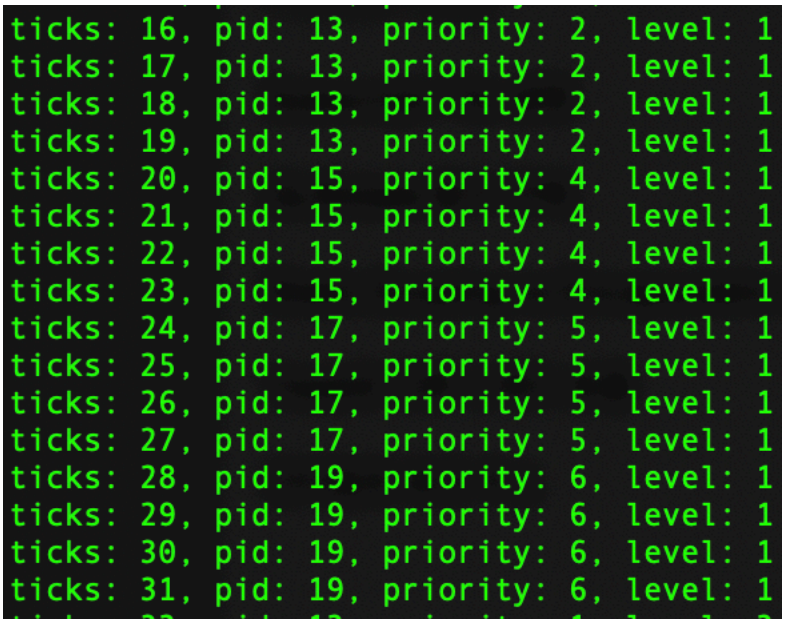
Picture 21. Temporary code result

As demonstrated, L3 employs priority scheduling, wherein a process's priority is decremented by one whenever it exhausts its time quantum. Also, the processes in L3 use their time quantum by 8 ticks and the processes in L2 use their time quantum by 6 ticks.



Picture 22. Priority boosting and L0

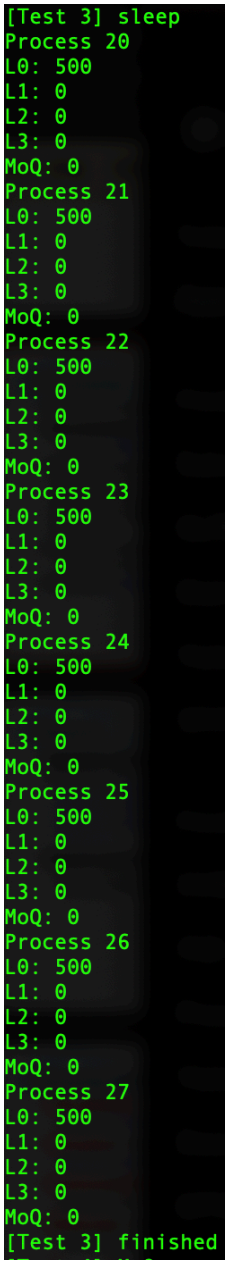
The priority boosting is happening whenever the global ticks become one hundred. Also, the processes in level 0 occupy the CPU by 2 ticks.



Picture 23. L1

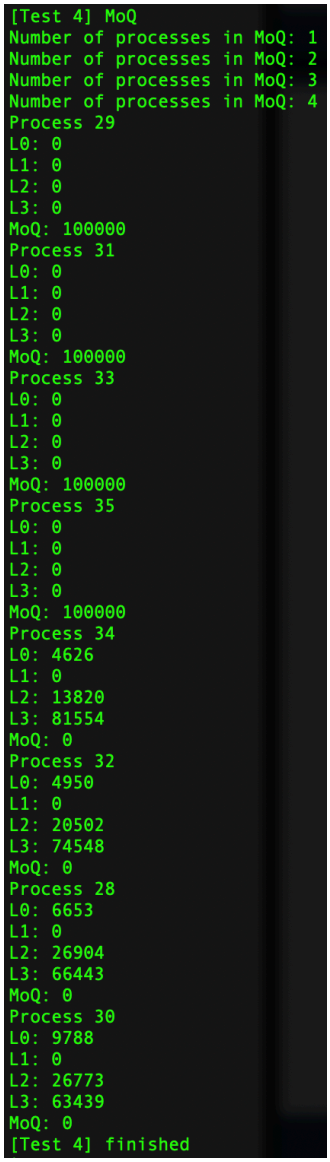
The processes in L1 occupy the CPU by 4 ticks.

The result of the test 3 is shown below:



Picture 24. Result of the test 3

The result of test 4 is demonstrated below:



Pictrure 25. Result of the test 4

However, the outcome may not always be consistent: in the test code, processes with odd pids promptly call 'sleep(10)', indicating that they will sleep for 10 ticks. If the program forks process 36 and invokes 'monopolize()' before the monopolized processes wake up, the monopolized processes would be removed from the queue and re-inserted at the back of the queue. This architecture is correct because the context switch still occurs when the process sleeps in a non-preemptive scheduling scheme.