컴퓨터소프트웨어학부 2022054702 김소정

OS Project02 WIKI _ 김소정

DESIGN

MLFQ DESIGN

mlfq를 수행할 큐가 4개 필요하다. L0, L1, L2, L3로 구성되며 숫자가 작은 큐에게 우선권이 있다. 각 큐 내에는 프로세스들이 저장되어 있으며, 이는 linked list로 구현한다.

큐를 구현하기 위해 큐 구조체를 만든다. 이때 구조체 안에는 linked list를 위한 head, tail 변수, 그리고 time quantum 변수 등이 있을 것이다.

또한 이미 proc.h에 있는 프로세스 구조체에도 몇 가지 변수를 추가한다. linked list를 위한 next 변수, running 시간, 큐에 들어있을 때 해당 큐의 level을 저장하는 변수 등이 있다.

처음 실행된 프로세스는 모두 L0에 들어가므로 userinit 함수와 fork 함수에 L0에 프로세스를 넣는 코드를 작성한다. allocproc 함수에는 process 구조체 내 변수들을 초기화해준다.

L0, L1, L2는 round robin 정책을 따른다. L0에 runnable한 프로세스가 있으면 해당 프로세스를 실행시키고, L0 time quantum 내에 끝나지 않으면 L1 또는 L2로 이동한다. 이는 해당 프로세스의 pid로 정해진다. L0의 runnable한 프로세스를 모두 훑으면 L1과 L2로 이동한다. L1과 L2에 runnable한 프로세스를 실행시키고, time quantum 내에 끝나지 않으면 L3로 이동한다.

L0, L1, L2는 모두 round robin이므로 같은 for문을 사용한다. for문을 모두 돌고 runnable한 프로세스가 없으면 L3로 넘어간다.

L3는 priority 스케줄링을 한다. L0, L1, L2는 큐의 head 프로세스부터 실행시키지만 L3는 해당 큐의 프로세스 중 priority가 가장 높은 프로세스 먼저 찾아 그 프로세스를 실행시켜야 한다.

이때 L3 큐에서 실행된 프로세스가 time quantum 내에 못 끝내면, L0, L1, L2와 달리 다른 큐로 이동하지 않고 priority를 하나 감소시키고 해당 프로세스의 time quantum만 초기화된다. (초기화는 다른 큐에서도 마찬가지이다.) 이때 starvation을 막기 위해 priority boosting을 구현한다. global tick이 100이 되면 L1, L2, L3의 모든 프로세스를 L0로 이동시킨다. 이때 global tick은 trap.c에서 사용되는 ticks를 이용하며, 여기서 priority boosting 함수를 호출한다.

컴퓨터소프트웨어학부 2022054702 김소정

MoQ DESIGN

monopolize 시스템콜이 호출되면 unmonopolize가 호출될 때까지 monopoly 큐에 있는 프로세스를 스케줄링한다. MLFQ와 구별하기 위해 global 변수 하나를 사용한다. monopolize 함수에서 해당 변수를 1로 set해주면 MoQ 스케줄링이 시작된다. unmonopolize 함수에서는 reset해준다. 그러면 다시 MLFQ로 돌아간다.

MoQ는 FCFS 정책을 따르기 때문에 큐의 head 프로세스부터 스케줄링해준다.

MoQ 큐를 스케줄링 하는 동안 priority boosting은 발생하지 않으므로 priority boosting은 global 변수가 0일 때만 작동하도록 해준다.

Implement

Queue, Process

```
struct Queue
{
    struct proc *head; // head process in Queue linked list
    struct proc *tail; // tail process in Queue linked list
    int time_quantum;
    int level; //0,1,2,3, 4
    struct spinlock queueLock;
};
```

linked list 구현을 위해 head, tail 함수를 사용한다. 각 큐에 대한 작업을 할 때 lock을 위해 queueLock 변수도 만들었다.

level 변수는 L0~L3는 0~3을, MoQ는 4로 지정하였다.

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)

  int priority; //0~10
  int queue_level;
  int current_tick;
  struct proc *next;
  int inMoQ; //is this process in MoQ??
};
```

기존에 있던 process 구조체에 몇 가지 변수를 추가하였다. 하위 5개 변수가 새로 추가한 변수이다. current_tick은 해당 프로세스가 실행된 time quantum을 나타낸다. next 변수는 각 큐의 프로세스 linked list에서 해당 프로세스 다음에 있는 프로세스를 나타내며, push, pop할 때 이용된다. inMoQ는 해당 프로세스가 monopoly 큐에 있는 지의 여부를 나타낸다.

priority는 해당 프로세스의 priority를 나타내며 L3에서 사용된다. queue_level은 해당 프로세스가 속해있는 큐의 레벨을 가리킨다.

```
void initializing_Queue(void)
{
    for (int i = 0; i < 4; i++)
    {
        L[i].level = i;
        L[i].time_quantum = 2 * i + 2;
    }

    MoQ.level = 4;
}
```

각 큐를 초기화해주는 함수로, scheduler 함수 가장 초반에 사용된다.

큐의 time_quantum을 2, 4, 6, 8로 초기화해주고 level을 설정해준다. MoQ의 level은 4이다.

큐에 새로운 프로세스를 넣어주는 pushQueue 함수이다.

```
void pushQueue(struct Queue *queue, struct proc *process)
{
    if (process->queue_level != -1)
        return;


    if (queue->level < 0 || queue->level > 4)
    {
        cprintf("MLFQ PUSH failed : level error\n");
        return;
    }

    acquire(&queue->queueLock);
    process->next = 0;
    if (queue->head == 0) // when queue is empty!
    {
        queue->head = process;
        queue->tail = process;
    }
    else // when queue is not empty, push process in tail..
    {
        queue->tail->next = process;
        queue->tail = process;
    }

    process->queue_level = queue->level;

    release(&queue->queueLock);
}
```

초반에는 예외처리를 해주고, 오류가 없으면 정상적으로 새로운 프로세스를 넣어준다.

프로세스를 넣어줄 큐가 비워져 있는 지의 여부에 따라 코드가 달라진다.

마지막에는 process의 queue_level을 넣어준 큐 level로 바꿔준다.

컴퓨터소프트웨어학부 2022054702 김소정

큐에서 프로세스를 제거해주는 popQueue 함수이다.

```
void popQueue(struct Queue *queue, struct proc *process)
{
    if (queue->level < 0 || queue->level > 4 || process->queue_level != queue->level)
    {
        cprintf("POP failed: level error\n");
        return;
    }
    if (queue->head == 0) //queue is empty
    {
        cprintf("POP failed : queue is empty\n");
        return;
    }

    acquire(&queue->queueLock);
    if (process == queue->head) // if proces is head of queue
    {
        if (process->next == 0) //this queue will be empty after pop
        {
            queue->head = 0;
            queue->tail = 0;
        }
        else
            queue->head = process->next;
    }
    else // process is not head process.
    {
        struct proc *prev_proc = queue->head;
        while (prev_proc->next != process) // find previous process in queue
            prev_proc = prev_proc->next;

        if (process == queue->tail) //process is in queue tail
        {
            queue->tail = prev_proc;
            prev_proc->next = 0;
        }
        else
            prev_proc->next = process->next;
    }
    process->next = 0;
    process->queue_level = -1;
    release(&queue->queueLock);
}
```

초반에 예외처리를 해준 후 예외가 없으면 정상적으로 처리해준다.

제거해줄 프로세스가 큐의 head에 있는 지의 여부에 따라 코드가 달라진다.

마지막에는 큐에 존재하지 않다는 것을 표기하기 위해 queue_level을 -1로 설정한다.

pushQueue, popQueue 모두 함수 내에서 해당 큐의 queueLock을 사용하여 동기화해준다.

프로세스가 속한 큐의 레벨을 반환해주는 getlev함수이다.

```
int getlev(void)
{
    if (myproc()->queue_level == 4) return 99; //process in MoQ
    return myproc()->queue_level;
}
```

현 프로세스가 MoQ에 속했을 경우에는 99를 반환한다. 참고로 MoQ 구조체의 level은 4이다.

컴퓨터소프트웨어학부 2022054702 김소정

새로운 프로세스를 L0에 넣어주기

새로운 프로세스는 무조건 L0에 넣어준다. 이를 위해 userinit, fork 함수에 pushQueue함수를 호출하였다. 그리고 allocproc 함수에서 프로세스를 초기화해준다.

userinit 함수

```
196
197       p->state = RUNNABLE;
198       pushQueue(&L[0], p); //push new process in L0
199       release(&ptable.lock);
200     }
201
```

fork 함수

```
259       pid = np->pid;
260       acquire(&ptable.lock);
261
262       np->state = RUNNABLE;
263       pushQueue(&L[0], np); //push child process in L0
264       release(&ptable.lock);
265
266       return pid;
267     }
```

allocproc 함수

```
123     found:
124       p->state = EMBRYO;
125       p->pid = nextpid++;
126
127       //initializing process!!
128       p->current_tick = 0;
129       p->next = 0;
130
131       p->queue_level = -1;
132       p->priority = 0;
133       p->inMoQ = 0;
134
```

컴퓨터소프트웨어학부 2022054702 김소정

MLFQ scheduler

MLFQ 스케줄링은 monopolize 시스템콜이 호출되지 않았을 때 시행된다. 이는 추후에 설명할 is_monopolize_called 전역변수에 의해 정해진다.

L0, L1, L2는 모두 round robin으로 기본적인 규칙이 같아 같은 for문에서 처리된다. L0부터 해당 큐가 안 비어있으면 runnable한 프로세스를 찾아 실행시킨다. 실행시킨 프로세스를 다음 큐로 넘길지의 여부는 해당 프로세스의 current_tick에 따라 정해지므로, switchuvm과 swtch 함수 사이에서 current_tick++를 해준다. running을 시켜준 후 프로세스의 상태가 zombie이면 지워주고, 상태가 runnable한데 current_tick이 해당 큐의 time_quantum을 넘었다면 다음 큐로 넘겨준다. 이때는 L0와 L1, L2가 코드가 다르다. 이 부분은 명세대로 해주었다.

```
else
{
   for (int i = 0; i < 3; i++) // L[0], L[1], L[2]
   {
       if (L[i].head != 0) //if L[i] is not empty.
       {
           for (p = L[i].head; p != 0; p = p->next)
           {
               if (p->state != RUNNABLE)
                   continue;
               //printQueue(&L[i], p);
               c->proc = p;
               switchuvm(p);
               p->state = RUNNING;
               p->current_tick++;
               swtch(&(c->scheduler), p->context);
               switchkvm();
               c->proc = 0;

               if (p->state == ZOMBIE)
                   popQueue(&L[i], p);

           else if (p->current_tick >= L[i].time_quantum && p->state == RUNNABLE)
           {
               p->current_tick = 0;

               if (i == 0)
               {
                   popQueue(&L[0], p);
                   if (p->pid % 2 == 0)
                       pushQueue(&L[2], p);
                   else if (p->pid % 2 == 1)
                       pushQueue(&L[1], p);
               }
               else if (i == 1 || i == 2){
                   popQueue(&L[i], p);
                   pushQueue(&L[3], p);
               }
           }

       }
   }
}
```

컴퓨터소프트웨어학부 2022054702 김소정

L3은 priority scheduling이기 때문에 따로 처리해주었다. L3는 L0, L1, L2를 모두 돈 후, L1과 L2가 비어있으면 스케줄링된다. L3는 L0, L1, L2와 달리 priority가 가장 높은 프로세스를 찾아 그 프로세스를 실행시켜줘야 한다. priority가 가장 높은 프로세스가 runnable하다면 실행시킨 후, process의 상태가 ZOMBIE면 제거해준다. process의 current_tick이 L3의 time_quantum을 넘기면 명세대로 해준다.

```
//L3 queue
if (L[1].head == 0 && L[2].head == 0 && L[3].head != 0)
{
    p = L[3].head;

    //finding process with highest priority in L[3]
    struct proc *highest_proc = L[3].head;
    int highest_priority = L[3].head->priority;
    struct proc *tmp = L[3].head;

    for (tmp = L[3].head; tmp != 0; tmp = tmp->next)
    {
        if (tmp->state != RUNNABLE || tmp == 0) continue;
        if (tmp->priority > highest_priority)
        {
            highest_proc = tmp;
            highest_priority = tmp->priority;
        }
    }

    p = highest_proc;
```

```
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;
        p->current_tick++;
        swtch(&(c->scheduler), p->context);
        switchkvm();
        c->proc = 0;

        if (p->state == ZOMBIE)
            popQueue(&L[3], p);
        else if (p->current_tick >= L[3].time_quantum)
        {
            p->current_tick = 0;
            if (p->priority > 0)
                p->priority--;

        }
    } //L3 end

} //else end.
release(&ptable.lock);
```

Priority Boosting

starvation 현상을 방지하기 위해 priority_boosting함수를 사용한다.

```
//priority boosting for starvation
void priority_boosting(void)
{
    if (is_monopolize_called) //not activate when monopoly scheduling
        return;

    struct proc *p;
    acquire(&ptable.lock);
    for (p = ptable.proc; p< &ptable.proc[NPROC]; p++)
    {
        if (p->queue_level == 1 || p->queue_level == 2 || p->queue_level == 3)
        {
             p->current_tick = 0;
             popQueue(&L[p->queue_level], p);
             pushQueue(&L[0], p);
        }
    }
    release(&ptable.lock);
}
```

priority_boosting은 monopoly 큐를 돌 때는 수행하지 않으므로 is_monopolize_called이 0일때만 수행된다. ptable을 돌면서 L1, L2, L3에 있는 프로세스를 모두 L0로 옮겨준다. 이때 각 프로세스의 current_tick은 다시 0으로 초기화해준다.

ptable을 돌아야 하므로 ptable.lock을 사용해 동기화해준다.

```
53      switch(tf->trapno){
54      case T_IRQ0 + IRQ_TIMER:
55        if(cpuid() == 0){
56          acquire(&tickslock);
57          ticks++;
58          if(ticks >= 100)
59          {
60            //cprintf("ticks: %d\n", ticks);
61
62
63            priority_boosting();
64            ticks = 0;
65
66            //cprintf("boosting done:)\n");
67          }
68          wakeup(&ticks);
69          release(&tickslock);
70        }
71        lapiceoi();
72        break;
```

이 함수는 trap.c 파일의 trap 함수에서 호출된다. ticks가 100이 되면 함수를 호출한 후 ticks를 다시 0으로 초기화해준다. trap함수는 interrupt나 exception이 발생할 때 호출되는 함수이다.

타이머 인터럽트를 관리하는 부분에서 ticks 관리를 해주므로 여기서 호출하였다.

MoQ

우선 monopoly 큐를 스케줄링한다는 것을 표현하기 위해 전역변수 is_monopolize_called을 선언 및 정의해주었다. 0으로 초기화해주었다.

```
// check if monopolize system call is called.
int is_monopolize_called = 0;
```

프로세스를 MoQ로 이동시키는 함수인 setmonopoly함수이다.

```
int setmonopoly(int pid, int password)
{
    if (password != 2022054702)
    {
        cprintf("MoQ PUSH failed : wrong password\n");
        return -2;
    }

    acquire(&ptable.lock);
    struct proc *p;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if (p->pid == pid)
        {
            popQueue(&L[p->queue_level], p);
            p->inMoQ = 1;
            pushQueue(&MoQ, p); //p->queue_level = 4;


            int MoQ_size = 0;
            for (struct proc *i = MoQ.head; i != 0; i = i->next)
            {
                if (i->state == RUNNABLE)
                    MoQ_size++;
            }
            release(&ptable.lock);
            return MoQ_size;
        }
    }

    release(&ptable.lock);
    cprintf("MoQ PUSH faild : wrong pid\n");
    return -1;
}
```

인자로 받은 암호가 틀리면 -2를 리턴하고 끝낸다. 암호가 맞으면 ptable에서 해당 pid를 가지는 프로세스를 찾는다. 프로세스가 존재하면 원래 그 프로세스가 있던 큐에서 MoQ로 이동시켜준다. 그 후 MoQ에 있는 runnable한 프로세스의 개수를 리턴해준다.

만약 해당 pid의 프로세스가 없으면 -1을 리턴한다.

이 함수에서는 ptable을 사용했으므로 ptable.lock을 사용해 동기화한다.

컴퓨터소프트웨어학부 2022054702 김소정

MoQ의 프로세스가 cpu를 독점하여 사용하도록 설정해주는 monopolize함수이다.

```
void monopolize(void)
{
    acquire(&MoQ.queueLock);
    is_monopolize_called = 1; //set
    release(&MoQ.queueLock);

}
```

is_monopolize_called를 set해줘서 scheduler 함수에서 MoQ 스케쥴링을 하도록 한다.

MoQ 구조체의 queueLock을 사용해 동기화한다.

MoQ 프로세스 사용을 멈추고 다시 MLFQ로 돌아가게 하는 unmonopolize함수이다.

```
//exit MoQ
void unmonopolize(void)
{
    acquire(&MoQ.queueLock);
    is_monopolize_called = 0;
    ticks = 0;
    release(&MoQ.queueLock);
}
```

is_monopolize_called를 reset해줘서 scheduler 함수가 다시 MLFQ 스케줄링을 하도록 한다. 또한 global tick을 0으로 초기화해준다.

여기서도 MoQ 구조체의 queueLock을 사용해 동기화한다.

MoQ Scheduler

is_monopolize_called가 1일 때 시행된다.

```
for (;;)
{
    sti();
    acquire(&ptable.lock);

    if (is_monopolize_called) //when monopolize system called
    {
        for (p = MoQ.head; p != 0; p = p->next)
        { //find runnable process (FCFS)
            if (p->state == RUNNABLE)
                break;
        }

        if (p == 0)
        {
            release(&ptable.lock);
            continue;
        }
```

```
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;

    swtch(&(c->scheduler), p->context);
    switchkvm();
    c->proc = 0;

    if (p->state == ZOMBIE)
    {
        popQueue(&MoQ, p);
        p->inMoQ = 0;
    }
    if (MoQ.head == 0)
        unmonopolize();

}
```

FCFS 정책을 따르므로 들어온 순서대로, 즉, MoQ.head 프로세스부터 실행시킨다. MoQ의 head프로세스부터 돌아 runnable한 프로세스를 찾은 후, 그 프로세스가 끝날 때까지 실행시킨다.

실행 후 프로세스의 state가 ZOMBIE면 MoQ에서 제거해준다.

만약 MoQ에 프로세스가 없다면 자동으로 unmonopolize을 호출한다. 그러면 MLFQ로 넘어간다.

System Call 등록

명세에 있는 6개의 시스템콜 함수의 wrapper 함수는 모두 sysproc.c에 작성하였다. 모든 wrapper 함수의 인자는 void, 리턴값은 int이다. wrapper함수를 작성해준 후 defs.h, syscall.h, syscall.c, user.h, usys.S에 6개의 함수를 등록하였다.

추가로 테스트로 사용할 mlfq_test.c 파일은 Makefile에 추가하여 실행이 가능하도록 하였다.

컴퓨터소프트웨어학부 2022054702 김소정

Result

실행환경: Ubuntu 20.04.6 LTS

실행과정: make clean -> make -> make fs.img -> ./bootxv6.sh (qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512)

실행결과:

1. [Test 1]

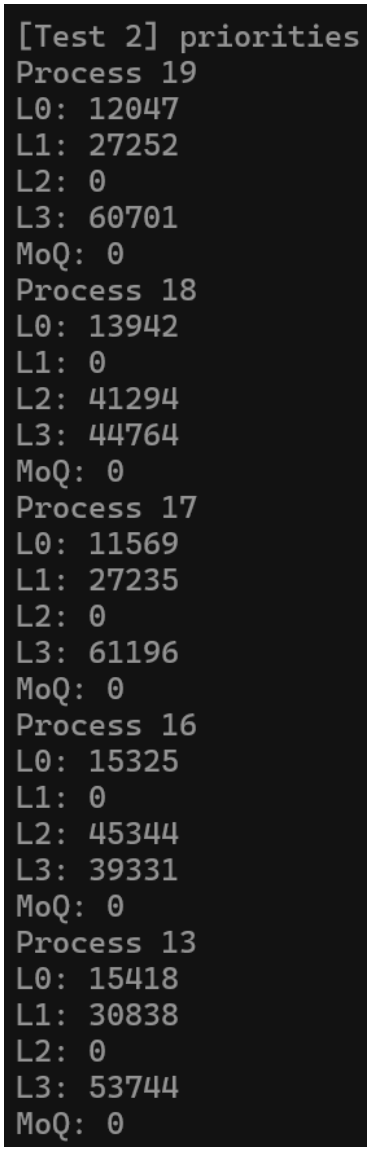
```
$ mlfq_test
MLFQ test start
[Test 1] default
Process 5
L0: 7303
L1: 15582
L2: 0
L3: 77115
MoQ: 0
Process 7
L0: 12363
L1: 22594
L2: 0
L3: 65043
MoQ: 0
Process 4
L0: 12937
L1: 0
L2: 41044
L3: 46019
MoQ: 0
Process 9
L0: 14965
L1: 30602
L2: 0
L3: 54433
MoQ: 0
Process 6
L0: 14551
L1: 0
L2: 45652
L3: 39797
```

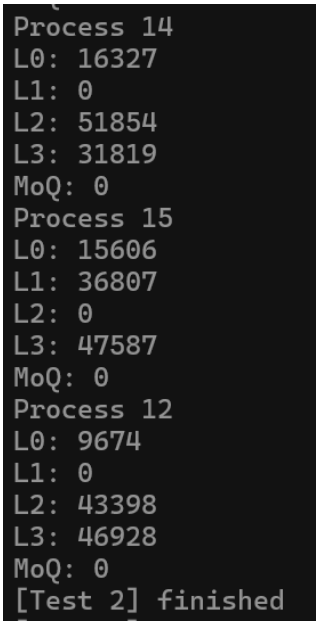
```
MoQ: 0
Process 11
L0: 15794
L1: 30912
L2: 0
L3: 53294
MoQ: 0
Process 8
L0: 16325
L1: 0
L2: 50133
L3: 33542
MoQ: 0
Process 10
L0: 16173
L1: 0
L2: 46649
L3: 37178
MoQ: 0
[Test 1] finished
```

프로세스가 MLFQ에 들어가 차례로 수행된다. L0에서 L1으로 가는 프로세스는 pid가 홀수인 프로세스이므로 pid가 홀수인 프로세스가 대체로 먼저 끝난다.

MoQ에는 들어가지 않으므로 모두 0이다.

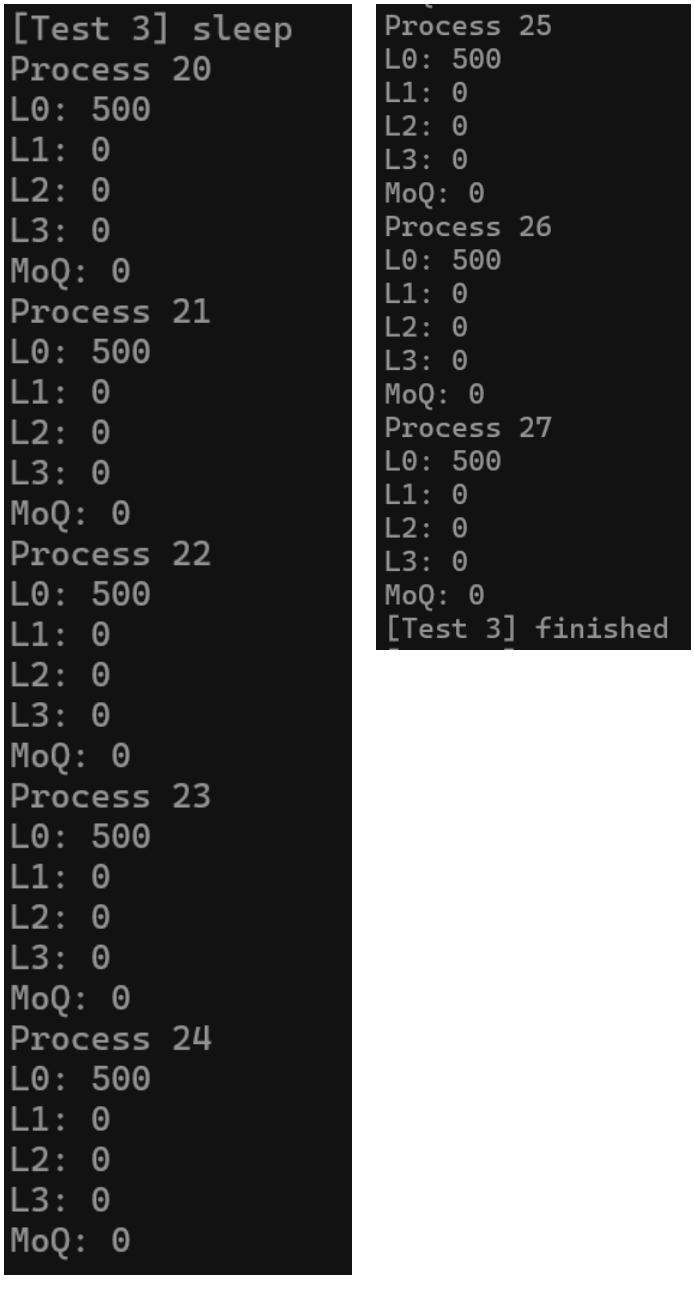
2. [Test 2]





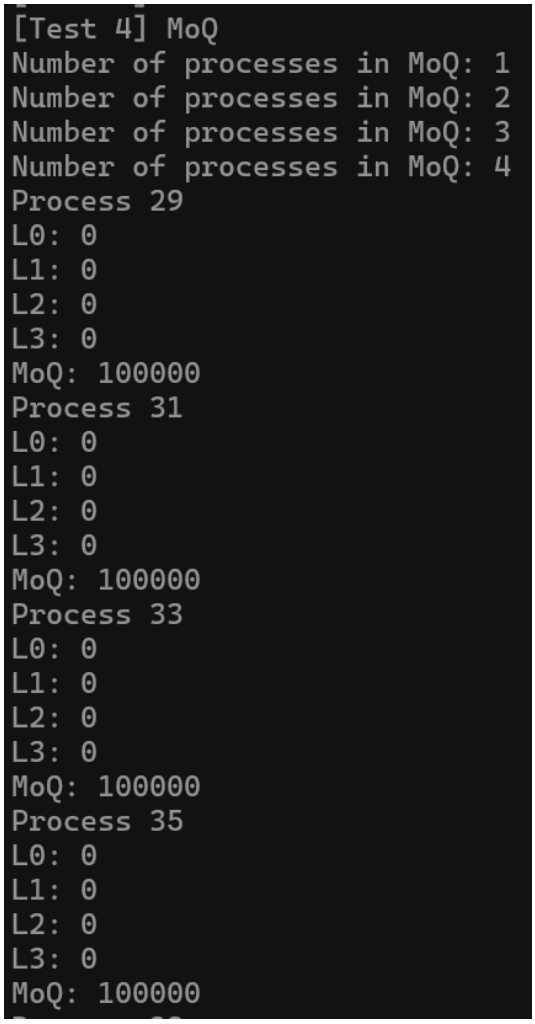
pid가 큰 프로세스에게 더 높은 우선순위를 부여한 경우이다. L3에서 우선순위가 높은 프로세스를 먼저 실행시키므로 pid가 큰 프로세스가 대부분 먼저 끝난다.

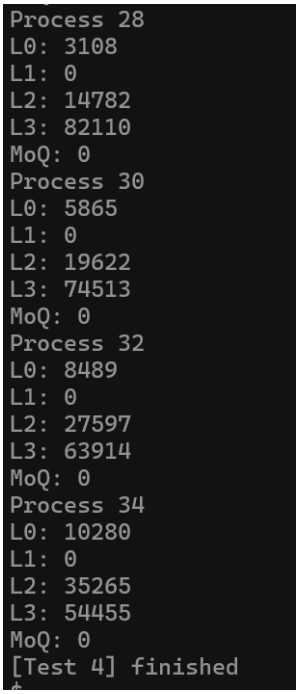
3. [Test 3]



pid가 큰 프로세스에게 더 높은 우선순위를 부여한 후 루프를 돌 때마다 바로 sleep 시스템 콜을 호출한다. scheduler에서 runnable한 프로세스만을 실행시키도록 하였으므로 sleep 상태에 있는 동안에는 스케줄링되지 않는다. 또한 sleep 상태인 프로세스는 다른 큐로 넘어가지 않도록 하였으므로 대부분 L0에 머문다. 또한 L3에 들어가지 않기 때문에 priority가 의미가 없다. 즉 들어온 순서대로 실행되기 때문에 pid가 작은 프로세스가 먼저 끝난다.

4. [Test 4]





monopolize를 호출한 경우이다. mlfq_test 코드를 보면 pid가 홀수인 프로세스만을 setmonopoly 시스템콜을 사용해 MoQ에 저장하였다. 저장할 때마다 Number of processes in MoQ를 출력하도록 하였으므로 처음에 출력된 것을 볼 수 있다. 이때 들어간 4개의 프로세스는 순서대로 29, 31, 33, 35의 pid를 가지는 프로세스이다. MoQ는 FCFS 정책을 따르므로 들어간 순서대로 끝난 것을 볼 수 있다. 또한 MoQ에서만 있으므로 나머지 큐에서 실행된 시간은 모두 0이다. unmonopolize가 호출된 후로는 다시 mlfq로 돌아가므로 Process 28부터는 다시 MLFQ에서 실행되는 것을 볼 수 있다.

Trouble shooting

1. Queue 구조체 내에 프로세스 배열

   처음에는 프로세스 배열을 linked list가 아닌 단순 배열로 하였다. 그때 배열 크기는 NPROC을 사용해 64로 지정하였다.

   그러나 배열로 하면 여러가지 단점이 존재하였다.

   우선 맨 처음 또는 끝에 있는 프로세스가 아닌 중간에 있는 프로세스를 제거할 경우 코드가 너무 복잡해진다. 또한 최대크기가 64라 하더라도 head와 tail이 이동하면 그거에 따라 문제가 생길 가능성이 있다.

   그래서 linked list 구조로 변경하였다.

2. 초반에 계속 xv6가 부팅이 되지 않는 문제를 겪었었다. 특히 no pgdir문제를 많이 겪었다.

```
L3
L3 process: 1
ng sh
push
global tick: 26
running L[3]
pid: 1
process tick: 20

lapicid 0: panic: switchuvm: no pgdir
 8010782d 80104850 8010301f 8010316c 0 0 0 0 0 0
```

해당 출력을 해주는 함수를 cscope을 통해 들어가 분석하였고, process에 문제가 있음을 깨달았다. 그래서 process가 처음 생길 때 초기화해주는 부분을 확인하였다. 처음에는 allocproc 함수에 pushQueue(&L[0], p)를 사용했었는데, 이를 지우고 allocproc 함수를 호출하는 userinit 함수에서 해주었다. 그리고 allocproc의 found 부분에서 process를 초기화해주었다. 추가로 fork 함수에도 pushQueue(&L[0], p)를 추가해주었더니 그제서야 xv6가 부팅되었다.

3. boosting이 처음 할 때는 잘되지만 그 후로는 문제가 생겼었다. priority_boosting함수에서는 pop, push 함수만 사용했기 때문에 그 부분을 자세히 다시 보았다. 원래는 pushQueue 함수에서 프로세스의 상태를 runnable로 바꿔주는 코드를 작성했었는데, 어차피 userinit, fork 등의 함수에서 state를 runnable로 바꿔주는 코드가 있기 때문에 이를 지웠다. 그랬더니 문제가 해결되었다.

4. 처음에는 L0, L1, L2, L3 큐 스케줄링 하는 코드를 모두 따로 작성하였다. 그랬더니 가독성이 너무 떨어지고, L0, L1, L2는 코드가 거의 동일하여 같은 코드가 반복되는 것 같아 for 문으로 다시 작성하였다. 또한 Queue 배열을 사용해 Queue L[4]로 정의하였다.

5. 그 외에도 정의를 여러 번 해주거나 올바르지 않는 줄에 적거나 하는 사소한 문제에도 모두 에러가 생겨 많은 시행착오가 있었다. 특히 xv6 부팅을 성공시킬 때까지 시간이 정말 오래 걸렸다. 시행이 된 후로는 오히려 문제점이 보였기 때문에 빠르게 진행이 되었다.