컴퓨터소프트웨어학부
2022074057
홍바다

Project02 - Wiki

I. Design

A. MLFQ & MoQ

기본적으로 round-robin scheduling을 따르는 4-level feedback 큐를 이용하여 priority scheduling을 구현하고 monopoly 큐를 위해 FCFS scheduling을 구현하기 위해 이론 수업 때 배운 내용들을 복습해 보았다. 먼저, FCFS 스케줄링의 최대 장점은 공평성이다. 먼저 오는 프로세스에게 먼저 CPU를 주는 것이다. 하지만 nonpreemptive하기 때문에 시간이 오래 걸리는 프로세스로 인해 딜레이가 많이 생길 수 있다. 이러한 특성을 고려하여 MoQ에서 해당 스케줄링 방법을 사용하는 목적이 이해가 됐다. 그 다음, priority scheduling은 보통 다른 스케줄링 방법과 함께 사용되는 것이 유용하기 때문에 round-robin 스케줄링에 덧붙여 사용하는것이 이해됐다. 여기에서 starvation 상황이 일어날 수 있기에 우선순위를 조정하는 명세도 중요한 포인트였다. Round robin은 FCFS를 기반으로 시간 기한을 정해주어 starvation 상황을 줄여주는 방법이다. 그리고 각각 다른 큐들 사이를 이동하며 프로세스를 관리하기 위해 multilevel feedback queue를 사용하는 것이다. 과제 명세대로 구현해야 할 여러 레벨의 큐와 그들이 따르는 스케줄링 방식에 대해 복습을 하여 중요한 점들을 복기했다.

과제 명세에 있는 필수 함수들을 프로젝트1에서 구현한 것과 같이 시스템 콜로 만들기 위해 proc.c에서 함수 구현을, sysproc.c에서 wrapper 함수 구현을, 그리고 메인 함수가 있는 유저프로그램에서 사용할 수 있도록 각 시스템 콜을 defs.h, syscall.c, user.h, usys.S에 등록해주어야 한다.

Xv6 운영체제에서 어느 부분에 코드를 작성해야 스케줄링이 구현이 되는지 몰라 시작하는데에 어려움이 있었다. xv6를 파악하기 위해 먼저 process에 관한 함수들을 다루는 proc.c 소스파일의 코드를 보았다. 기본적으로 알고 있는 fork(), yield() 함수들이 포함 되어있는 것을 볼 수 있었고, 특히 context switch가 일어나는 yield() 함수에서 sched()가 사용되는 것을 알았다. sched() 함수를 보니 context switch를 할때 scheduler라는 것이 사용되었다. scheduler가 무엇인지 찾기 위해 grep -rn "scheduler" 라는 command를 사용해 어디에서 사용되는지 보았다. 이 또한 proc.c에서 구현되어있는 것을 관찰할 수 있었고, 해당 함수에서 context switch 과정이 일어나는 것을 볼 수 있었다. 이곳에서 스케줄링을 구현해야겠다고 생각했다.

또한, starvation을 막기 위한 priority boosting을 구현하기 위해 추가적인 함수가 필요하다고 생각했다. 그리고 MoQ가 독점하는 상황인지를 구분하기 위해 monopoly라는 변수가 필요했고, cpu와 관련되어있으니 proc.h에 글로벌 변수로 선언하여 다른 파일에서도 사용할 수 있게 하였다. proc.h에는 cpu와 proc 구조체에 대한 정보가 있었다. 추가로 proc에 필요한 필드(level, priority, timeQuantum)를 넣어주었다. level은 해당 프로세스가 속해 있는 큐의 레벨이고, priority는 L3에서 사용하는 우선순위를 넣어주고, 프로세스가 cpu를 이용할 수 있는 시간 할당량을 timeQuantum에 저장해주었다.

다음으로는 스케줄러를 구현하기 위해 큐에 대한 헤더파일이 필요하다고 생각했다. 배열을 이용하여 큐를 관리하는 것이 용이하다 생각했고, 그에 관련된 함수들, enqueue(), dequeue(), isEmpty(), isFull() 등이 기본적으로 필요했다. 이에 대한 헤더파일을 만들기로 하였고, MLFQ는 L0~L3을 관리하는 큐 배열을 만들고, Moq는 구분하기 위해 다른 큐를

사용하기로 했다. 이 큐들을 그냥 proc.c에 선언해도 되지만, ptable 구조체 안에서 관리하기 위해 이 구조체 안에 mlfq에 대한 큐 배열과 moq에 대한 큐를 선언해주었다. 그리고 proc.c를 분석하다보니 pinit()에서 ptable.lock에 대해 spinlock()을 불러오는 것을 확인하여 여기에서 큐들을 초기화해주면 되겠다고 생각했다.

II. Implement

A. Proc.h 수정

```
extern struct cpu cpus[NCPU];
extern int ncpu;
extern int monopoly; //indicates which queue is monopolized
```

```
int monopoly = 0; //indicates which queue is monopolized
```

위에서 언급한대로 mlfq와 moq의 독점 상태를 구분하기 위해 정수 타입의 monopoly 변수를 proc.h 헤더파일에서 선언해주었다. 다른 파일에서도 해당 변수를 사용할 수 있도록 extern으로 선언했다. monopoly가 0이면 mlfq에서 스케줄링을 하는 상태이고, 1이면 moq가 독점한 상태이다. 두번째 코드에서 보이듯이 proc.c에서 처음 초기화하며 사용한 것을 볼 수 있다.

```
// Per-process state
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)
//edited for project2
  int priority;
  int level;                    //queue level
  int timeQuantum;
};
```

proc.h 헤더파일에 정의되어있는 proc 구조체이다. 여기에 L3에서 필요한 우선순위를 나타내는 priority, 큐의 레벨을 나타내는 level, 그리고 사용 시간을 할당해주는 timeQuantum 필드를 추가해줬다. CPU 구조체에도 moq가 독점한 상황인지 알려주는 필드를 추가할까 하였지만 그보단 monopoly 변수를 글로벌하게 사용하기로 결정했다.

B. Queue.h 구현

```
//defining queue structure for project02
struct queue{
        struct proc* buf[NPROC];
        int front; //index of start
        int rear; //index of end
        int size; //number of elements in queue
        int level; //level of queue
};
```

Multilevel queue를 구현하기 위해 먼저 큐 구조체를 만들어줬다. 해당 큐에 속해있는 프로세스들을 저장하기 위한 배열과 앞, 뒤 프로세스들의 인덱스를 저장하는 front와 rear, 큐에 있는 프로세스의 개수인 size, 그리고 해당 큐의 레벨을 나타내는 level이 있다.

해당 헤더 파일 내에는 initQ(), sizeOfQ(), isTerminate(), isFull(), isEmpty(), enqueue(), dequeue(), deleteProcess() 함수들이 정의 되어있다. 큐의 배열은 순환 구조를 따르기 때문에 아래의 함수에서 쓰인 것과 같이 최대 capapcity인 NPROC을 기준으로 배열을 관리해주었다. Enqueue, dequeue 모두 큐의 기본 FIFO 방식을 따른다.

```
void enqueue(struct queue *q, struct proc *p){
        if(isFull(q)){
                return;
        }
        q->buf[q->rear] = p;
        q->rear = (q->rear + 1) % NPROC;
        q->size++;
}
```

```
struct proc *dequeue(struct queue *q){
        if(isEmpty(q)){
                return NULL;
        }
        struct proc *p = q->buf[q->front];
        q->front = (q->front + 1) % NPROC;
        q->size--;
        return p; //return the head
}
```

```
void deleteProcess(struct queue* q, struct proc* p){
        if(isEmpty(q)){
                return;
        }
        for(int i = q->front; i != q->rear; i = (i+1) % NPROC){
                if(q->buf[i] == p){
                        for(int j = i; j != q->rear; j = (j+1) % NPROC){
                                q->buf[j] = q->buf[(j+1) % NPROC];
                        }
                        q->rear = (q->rear - 1 + NPROC) % NPROC;
                        q->size--;
                        return;
                }
        }
}
```

C. System Calls 구현

필요한 시스템 콜을 구현하기 위해 proc.c에서 함수들을 정의해줬다. 모두 스케줄링에 관한 함수이기에 해당 파일에 작성하기로 하였다.

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

→ 여기서 level은 L0 = 0, L1 = 1, L2 = 2, L3 = 3, moq = 99로 나타내주었다.

```
int
setpriority(int pid, int priority){
        if(priority < 0 || priority > 10)
                return -2; //not a valid priority
        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; //successfully set the priority
                }
        }
        release(&ptable.lock);
        return -1; //couldn't find process with the given pid
}
```

→ 과제 명세에 따라 파라미터로 받은 pid와 같은 pid를 가지는 프로세스의 우선순위를 배정해준다. 해당 프로세스를 찾기 위해 ptable을 통해 모든 프로세스들을 for문을 통해 읽었고, 공유되는 ptable을 사용하기에 동기화 문제를 해결하기 위해 acquire()과 release()를 앞뒤에 붙여주었다.

```
int setmonopoly(int pid, int password){
        if(password != 2022074057)
                return -2;
        struct proc *p;
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                if(p->pid == pid){
                        deleteProcess(&ptable.mlfq_queues[p->level], p);
                        p->level = 99;
                        enqueue(&ptable.moq_queue, p);
                        release(&ptable.lock);
                        return sizeOfQ(&ptable.moq_queue);
                }
        }
        release(&ptable.lock);
        return -1;
}
```

→ 파라미터로 받은 pid와 같은 프로세스를 monopoly queue에 이동시키는 함수이다. setpriority()와 같이 ptable을 사용하여 acquire()과 release() 함수를 붙여주고, 큐의 레벨을

moq를 뜻하는 99로 옮겨준다. 동시에 해당 프로세스가 소속되어있는 mlfq에서는 지워주고 moq에 넣어준다.

```
void monopolize(){
        monopoly = 99;
}
```

→ moq가 cpu를 독점했다는 뜻으로 monopoly 값을 99를 설정해준다.

```
void unmonopolize(){
        monopoly = 0;
}
```

→ moq가 cpu 독점을 중지했다는 뜻으로 monopoly 값을 0으로 설정해준다.

```
//edited for project02
int sys_yield(void){
        yield();
        return 0; //not reached
}

int sys_getlev(void){
        return getlev();
}

int sys_setpriority(){
        int pid, priority;

        if(argint(0, &pid) < 0 || argint(1, &priority) < 0)
                return -1;
        return setpriority(pid, priority);
}

int sys_setmonopoly(void){
        int pid, password;

        if(argint(0, &pid) < 0 || argint(1, &password) < 0)
                return -1;
        return setmonopoly(pid, password);
}

int sys_monopolize(void){
        monopolize();
        return 0;
}

int sys_unmonopolize(void){
        unmonopolize();
        return 0;
}
```

→ proc.c 함수들이 속해있는 sysproc.c에서 위의 모든 함수들에 대해 wrapper 함수들을 정의해주었다.

함수들을 사용할 수 있도록 Makefile과 defs.h를 수정해주고, 유저프로그램에서도 사용할 수 있도록 user.h와 usys.S에도 함수들을 등록해주었다. 또한, 메인 함수가 있는 유저 프로그램인 "mlfq_test.c" 파일도 Makefile에 추가해주었다.

D. Proc.c 수정

스케줄링을 구현하며 스케줄링에 직접적으로 영향을 가하는만큼 proc.c 소스파일을 수정하는 것이 가장 중요한 일이라고 느꼈다. 먼저 앞서 말했듯이 queue.h 헤더파일을 이용하여 ptable 구조체에 mlfq와 moq에 관한 큐를 필드에 추가해줬다. 또한 design 단계에서 설명하였듯 **pinit()** 함수에서 큐들을 초기화 해주었다.

```
#include "queue.h"

#define NMLFQ 4

struct {
  struct spinlock lock;
  struct proc proc[NPROC];
  struct queue mlfq_queues[NMLFQ];
  struct queue moq_queue;
} ptable;
```

```
void
pinit(void)
{
  initlock(&ptable.lock, "ptable");

  for(int i = 0; i < NMLFQ; i++){
      initQ(&ptable.mlfq_queues[i], i);
  }
  initQ(&ptable.moq_queue, 99);
}
```

다음으로는 **allocproc()** 함수를 수정해주었다. Ptable을 통해 아직 사용되지 않고 있는 프로세스들을 찾아 커널에서 사용되기 위해 초기화 해주는 함수이다. 여기에서 아까 proc 구조체에 추가해준 필드에 대한 초기값을 할당해준다. 과제 명세대로 처음에는 가장 높은 우선순위를 가지는 L0 큐에 넣어준다.

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  //initializing struct proc's fields
  p->priority = 0;
  p->level = 0;
  p->timeQuantum = 0;
  enqueue(&ptable.mlfq_queues[0], p);
```

가장 중요한 scheduler() 함수를 아래와 같이 과제 명세에 충족하도록 작성해주었다. 먼저, moq에 CPU가 독점 당한 상황이 아닐때, 즉 mlfq에서 스케줄링을 관리할때, L0부터 각 레벨의 큐에 RUNNABLE한 프로세스가 있는지 확인해주었다. 만약 레벨 3이라면, 여기서는 우선순위를 이용하여 스케줄링을 진행한다. 해당 큐의 front부터 rear까지 접근을 하여 가장 높은 priority를 갖고 있는 프로세스를 찾아 그에 대한 context switch가 일어나도록 해준다.

```
  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);

    if(monopoly != 99){
      p = 0;
      for(int i = 0; i < NMLFQ; i++){
        struct queue* q = &ptable.mlfq_queues[i];
        if(isTerminate(q) != 0){
          if(i == 3){
            struct proc * highestP = 0;
            for(int j = q->front; j != q->rear; j = (j+1) % NPROC){
              struct proc *temp = q->buf[j];
              if(temp && temp->state == RUNNABLE && (highestP == 0 ||
temp->priority > highestP->priority)){
                highestP = temp;
              }
            }
            if(highestP != 0){
              p = highestP;
              deleteProcess(&ptable.mlfq_queues[3], highestP);
            }
            else{
              break;
            }
            c->proc = p;
            switchuvm(p);
            p->state = RUNNING;
            swtch(&(c->scheduler), p->context);
            switchkvm();
            c->proc = 0;
          }
```

다음 아래 코드는 위 코드에 이어 레벨이 0~2인 경우를 보여준다. 먼저 큐의 가장 앞에 있는 프로세스를 저장한 후 그에 대한 context switch를 시켜준다. 만약 실행가능한 프로세스가 아니라면 다시 큐에 넣어주어 다음 프로세스에 대한 context switch를 하도록 해준다. 위 과정을 마친 후, round-robin 정책을 따르기 위해 시간 제한 범위를 넘진 않았는지 확인해준다. Time quantum을 다 사용한 L3의 경우 priority가 하나 감소한다. 만약 레벨 1, 2에 있는 프로세스라면 레벨 3으로 옮겨주고, 레벨 0이라면 홀수 pid인 경우 레벨 1로, 짝수인 경우 레벨 2인 큐로 옮겨준다. 해당 프로세스의 time quantum은 0으로 초기화된다. 그 다음 모든 설정을 마친 후 해당 프로세스를 설정된 레벨에 맞는 큐에 들어가게된다.

```
      else{
        p = dequeue(q);
        if(p->state != RUNNABLE){
          enqueue(&ptable.mlfq_queues[p->level], p);
          break;
        }
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;
        swtch(&(c->scheduler), p->context);
        switchkvm();
        c->proc = 0;
      }
      if(p->timeQuantum >= (2 * p->level + 2)){
        if(p->level == 3){
          if(p->priority > 0){
            p->priority--;
          }
        }
        else{
          if(p->pid % 2 == 1 && p->level == 0){
            p->level = 1;
          }
          else if(p->pid % 2 == 0 && p->level == 0){
            p->level = 2;
          }
          else if(p->level == 1 || p->level == 2){
            p->level = 3;
          }
        }
        p->timeQuantum = 0;
        enqueue(&ptable.mlfq_queues[p->level], p);
      }
      else{
        enqueue(&ptable.mlfq_queues[p->level], p);
      }
      break;
    }
```

아래 코드는 moq가 CPU를 독점한 상황이다. 이때에는 moq 안에 RUNNABLE 한 상태인 프로세스가 있는지 확인해준다. 만약 실행가능한 프로세스가 남아있지 않은 경우, unmonopolize() 함수를 통해 moq에 독점된 상황에서 벗어나게 된다. 그 후에는 다시 mlfq 스케줄링을 따르게 된다. 반면에 만약 다음 프로세스가 실행가능한 상태일 경우, context switch가 일어난다.

```
    else if(monopoly == 99){
      struct queue* q = &ptable.moq_queue;
      p = 0;
      for(int i = q->front; i != q->rear; i = (i+1) % NPROC){
        p = q->buf[i];
        if(p->state == RUNNABLE){
          break;
        }
        p = 0;
      }
      if(p == 0){
        unmonopolize();
        release(&ptable.lock);
        continue;
      }
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&(c->scheduler), p->context);
      switchkvm();
      c->proc = 0;
    }
    release(&ptable.lock);
  }
}
```

E. Trap.c 수정

Trap.c 소스 파일에서는 매 tick이 지날때마다 timer interrupt로써 trap() 함수에 신호가 올때 yield()를 불러주는 조건에 대하여 수정해주었다. 그 전에 global ticks가 1씩 증가할때마다 ticks가 100 ticks가 됐는지 확인해준다. 만약 100 ticks가 되었다면 moq의 프로세스를 제외한 모든 프로세스들은 priorityBoosting() 함수를 통해 L0로 재조정된다. priorityBoosting() 또한 스케줄링에 관한 함수이므로 proc.c에서 정의해주었고, 다른 파일에서도 사용 가능하게 하기위해 defs.h에 등록을 해주었다. 아래에 보이는 바와 같이 priorityBoosting은 먼저 moq가 독점한 상황인지 확인한 후 mlfq에서 스케줄링이 일어나고 있는 상황이라면 ptable을 이용하여 moq에 해당되지 않는 동시에 사용가능한 프로세스들을 L0 큐에 넣어준다. 이때 이들의 timeQuantum은 초기화 되어 다시 스케줄링이 된다. 재조정을 한 후 mlfq_queues의 L1~L3 큐들은 초기화시켜준다.

```
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;

    //priority boosting
    if(ticks % 100 == 0){
      priorityBoosting();
      //ticks = 0;
    }

    wakeup(&ticks);
    release(&tickslock);
  }
```

```
void priorityBoosting(void){
        if(monopoly == 99){ //if moq, don't do priority boosting
                return;
        }
        struct proc *p;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                if(p->state == ZOMBIE || p->state == UNUSED || p->level == 0 ||
p->level == 99){
                        continue;
                }
                p->timeQuantum = 0;
                p->level = 0;
                p->priority = 0;
                enqueue(&ptable.mlfq_queues[0], p);
        }
        for(int i = 1; i < NMLFQ; i++){
                initQ(&ptable.mlfq_queues[i], i);
        }
}
```

Time quantum을 다 사용한 경우에 대해서 아래 코드를 수정해주었다. 만약 해당 프로세스가 mlfq에 속해 있는 경우, 즉 mlfq의 스케줄링을 따르고 있는 경우에는 starvation을 해결해주기위해 각 레벨마다 시간 기한이 있다. $L_i = 2i + 2$ ticks로 제한되므로 과제 명세에 따라 아래 조건문을 작성해주었다. Time quantum을 모두 사용했을때에는 context switch가 필요하므로 yield() 함수를 불러 프로세스가 점유한 CPU를 포기할 수 있도록 해주었다.

```
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER){

    //edited for project2
    myproc()->timeQuantum++;
    //when mlfq exceeds time quantum, give up cpu
    if(myproc()->level >= 0 && myproc()->level <= 3 && myproc()->timeQuantum >=
(2 * myproc()->level + 2)){
      yield();
    }
  }
```

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

proc.c에 정의 되어있는 yield() 함수는 프로세스의 상태를 RUNNABLE한 상태로 바꾸어 더이상 RUNNING하고 있지 않게 해주고, sched()을 통해 scheduler()를 사용하여 context switch가 일어나도록 해주었다.

III. Result

A. 컴파일 및 실행

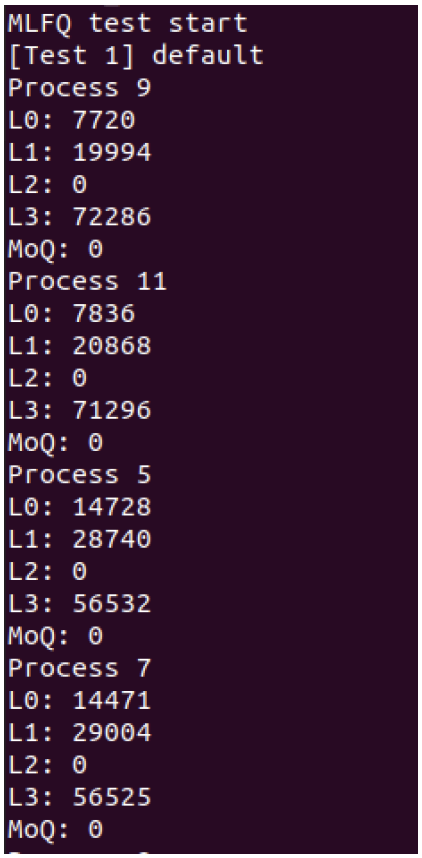
Compile:

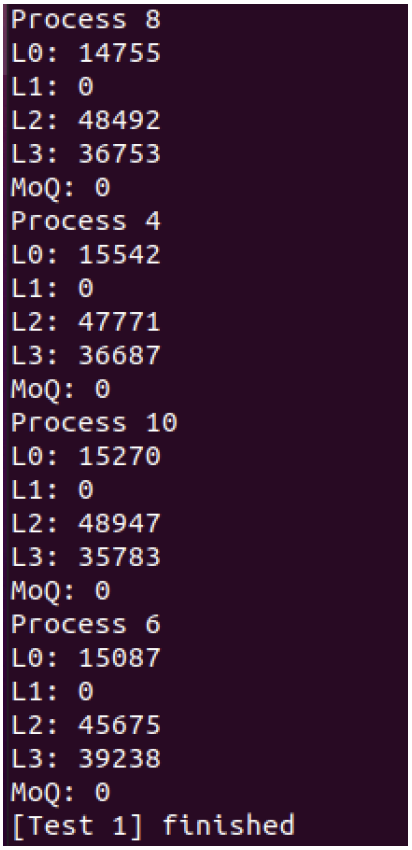
make CPUS=1 → make fs.img → ./bootxv6.sh 순으로 명령어를 작성하여 컴파일하면 아래와 같이 실행 되었다. 여기서 CPUS=1은 CPU가 하나라고 가정한다는 전제 조건을 주게 된다.

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$ mlfq_test
MLFQ test start
[Test 1] default
```

여러번 실행을 하였을때, 과제 명세에 명시된 것처럼 간혹 출력문이 꼬여 조금 뒤죽박죽 나올때가 있었다. 하지만 논리에 따라 그대로 실행되니 문제는 없었다.

Test 1. Default

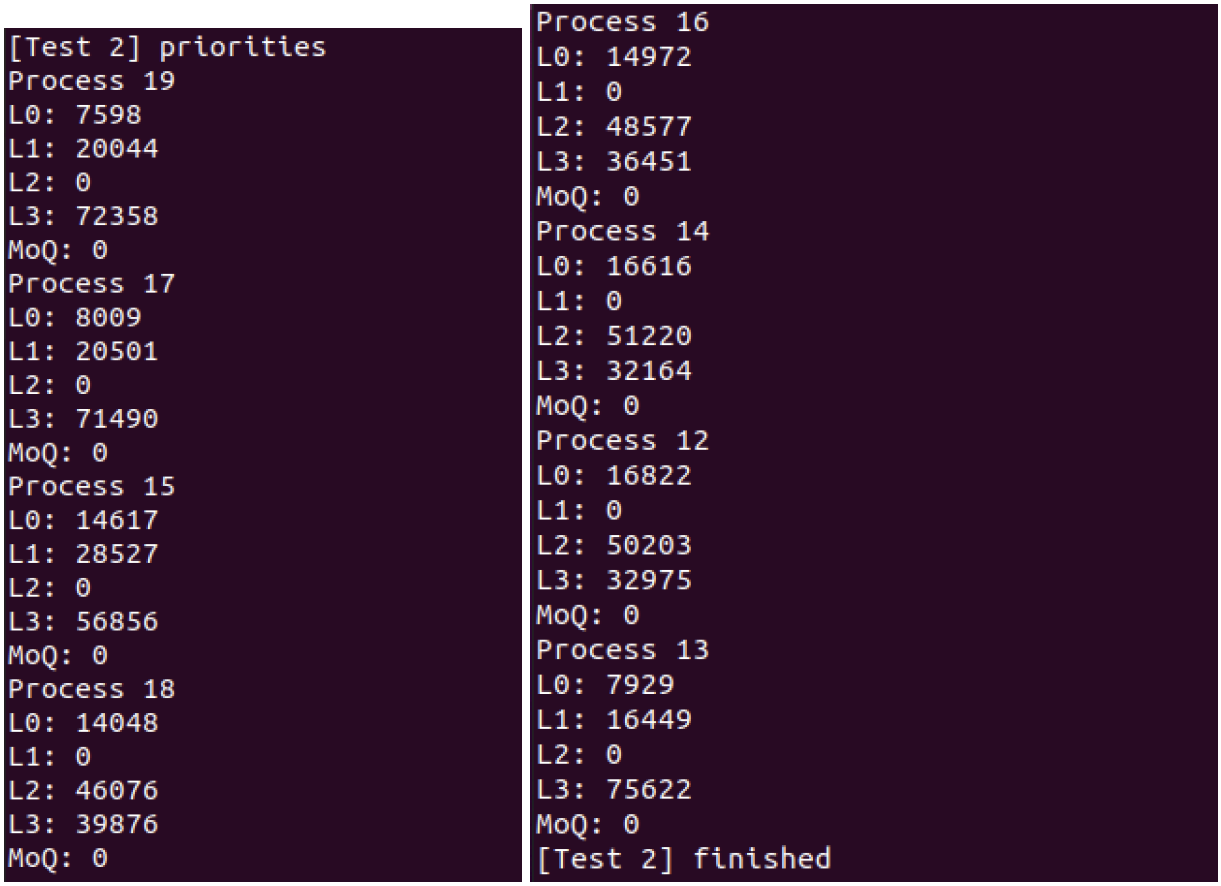




위 결과를 보면, 우선순위가 더 높은 L2 큐에 해당하는 pid가 홀수인 프로세스들이 대체로 짝수 pid인 프로세스들보다 먼저 끝나는 것을 볼 수 있다. 짝수 pid는 L2에, 홀수 pid는 L1으로 큐를 옮긴 것을 확인 할 수 있다. 이는 스케줄러 함수에서 명시했듯이 각자 레벨에서 주어진 time quantum을 다 사용하였을 경우 조건에 따라 큐 사이를 옮겨간 것이다. 또한 해당 테스트 케이스는 mlfq 스케줄링을 따른 것으로 moq에는 비어있는 것을 확인 할 수 있다. 프로세스들 간의 각 큐의 숫자를 살펴보면, L0의 숫자가 다시 올라가는 것을 보아 priority boosting도 제대로 일어난 것을 볼 수 있다.

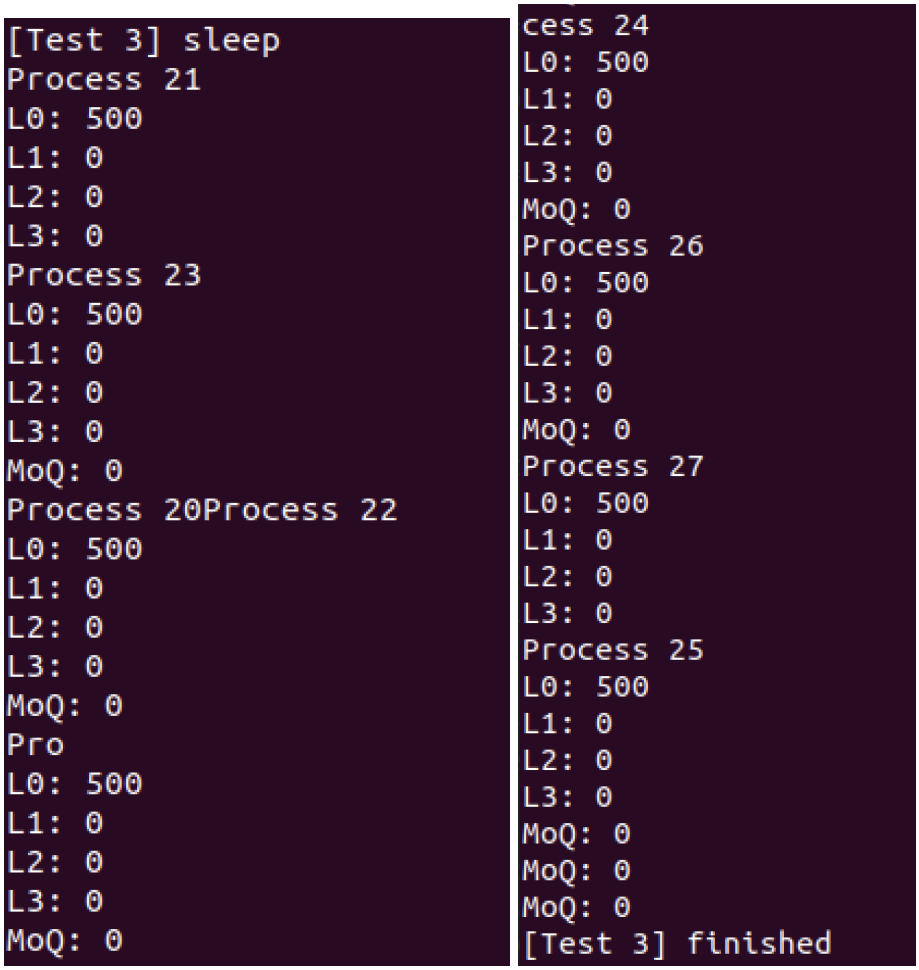
간혹 결과가 꼬이는 것을 확인 할 수 있는데, 이는 round-robin 정책을 따르기 때문에 각 프로세스가 공평하게 시간을 비슷하게 쓰기 때문이다. 이로써 종료 시간이 매우 비슷하여 출력하는 과정에서 예상대로 차례로 나오지 않는 경우가 있다.

Test 2. Priorities



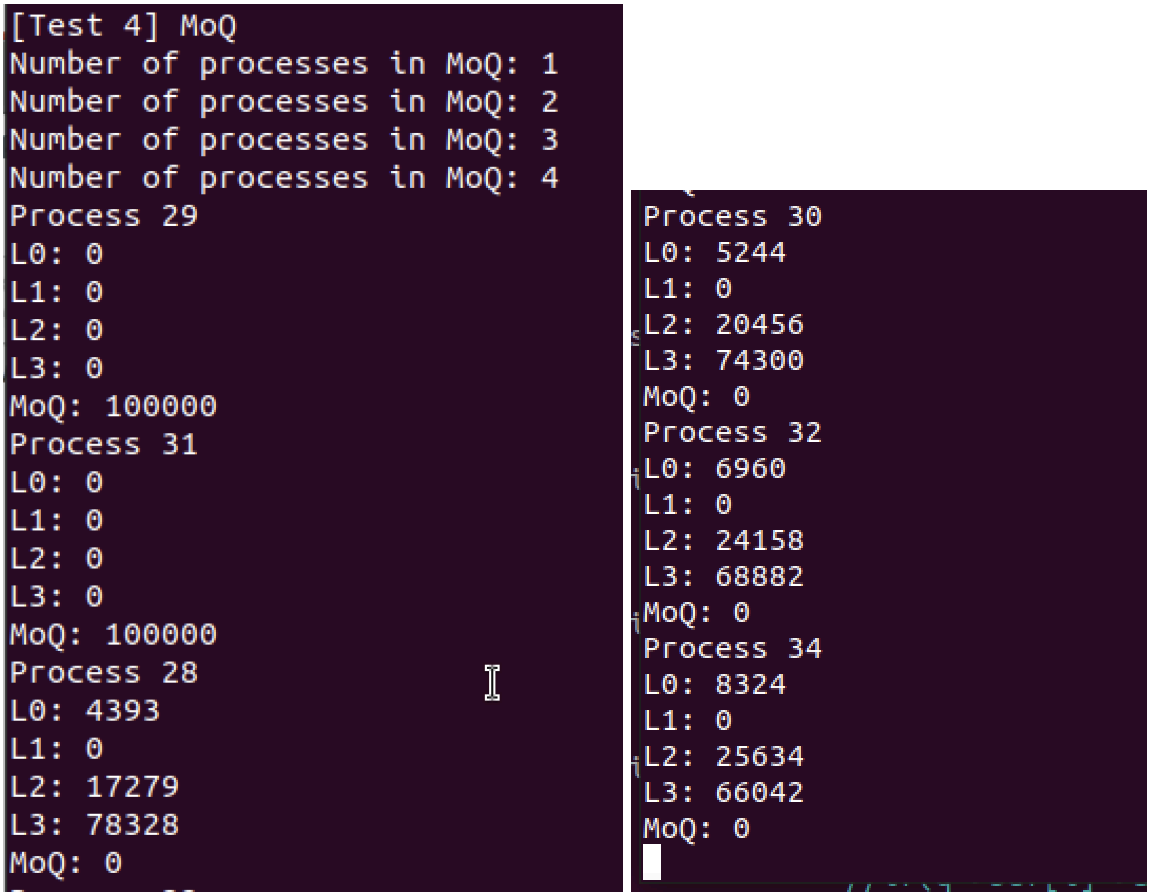
이 테스트 케이스에서는 pid가 큰 프로세스에 더 높은 우선순위를 부여한다. 그렇기에 결과에서 볼 수 있듯이 대체로 19와 같이 pid가 큰 프로세스가 먼저 끝난 것을 볼 수 있다. Test1과 같이 round-robin 방식을 따르며 priority boosting도 잘 일어났으며 큐 사이 간의 이동이 잘 된것을 확인 할 수 있다.

Test 3. Sleep

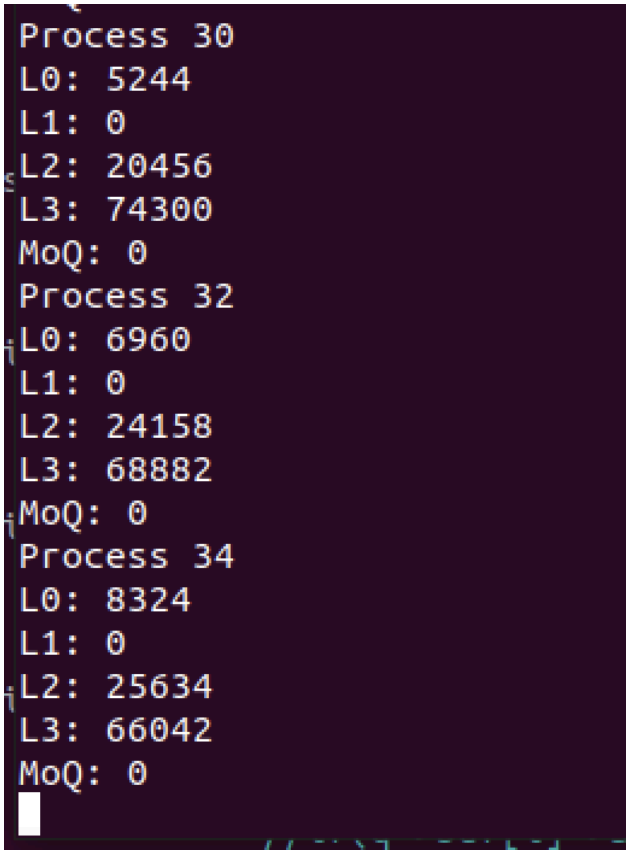


각 프로세스가 루프를 돌때마다 sleep 시스템 콜이 호출되어 스케줄링을 하지 않게 된다. 그 때 다른 프로세스가 실행될 수 있기 때문에 비슷한 시간에 종료가 되어 위와 같이 출력이 뒤섞이는 현상이 일어날 수 있다. 하지만 대부분 L0 큐에 머무는 것을 관찰 할 수 있고, 그로 인해 pid가 작은 프로세스들이 대체로 먼저 끝난 것을 볼 수 있다.

Test 4. MoQ



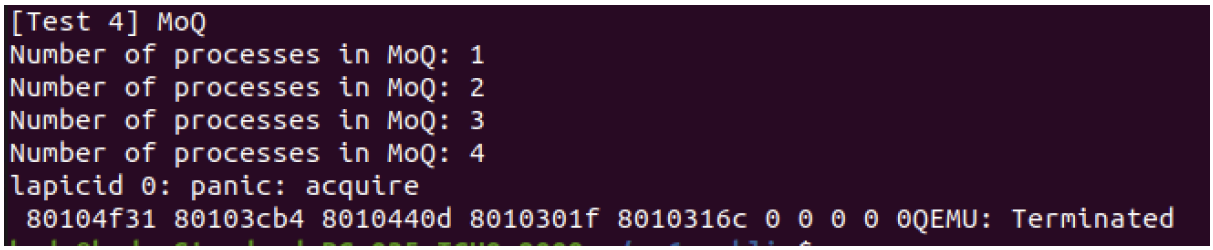
해당 결과는 보이는 것과 같이 마지막에 무한 루프를 돈다. 이는 출력과정에서 문제가 생긴 것으로 보인다. 위에 스케줄러 함수의 로직을 따라가면 moq에 속해 있던 프로세스들에 대한 출력과정이 먼저 나온 후 mlfq로 돌아갔을때의 프로세스에 대한 결과 값이 나와야 한다. 하지만 테스트 파일에 있던 pid 36을 만나면 프로그램이 끝나니 pid 33과 35가 아직 출력되지 않은 상태에서 프로그램이 끝나버려 출력과정에 문제가 생긴 것으로 보인다.

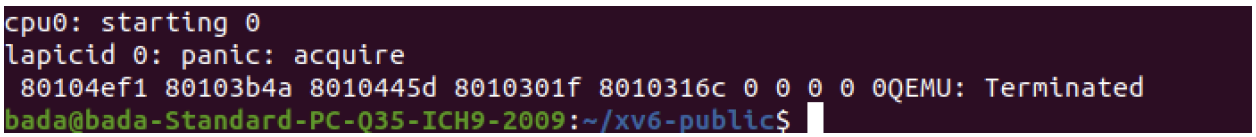


IV. Trouble Shooting

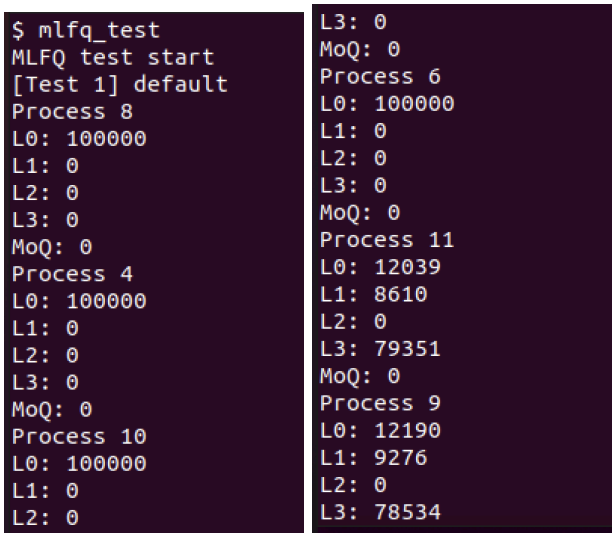
A. Lock으로 인한 문제들

공유되는 데이터에 접근하기 위해서는 lock 사용이 필수이다. 그래야만 동기화 문제를 해결 할 수 있기 때문이다. 그래서 ptable을 사용하는 지점마다 acquire와 release를 붙여줬다. 하지만 아래의 두 사진과 같이 "panic: acquire" 에러가 자주 발생했다. 어차피 이번 과제에서는 CPU가 하나라는 가정 하에 수행되는 것이니 동기화 문제를 걱정하지 않고 꼭 필요한 곳이 아니라면 지워주기로 했다. 그리하여 대부분의 acquire 또는 release 문제가 해결되었다.

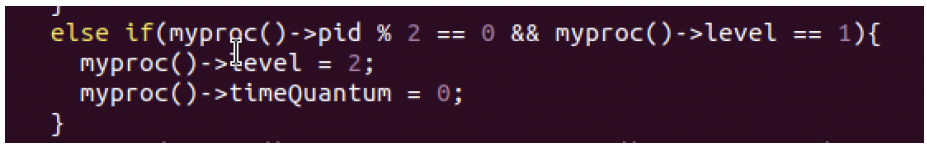




B. 간단한 로직 문제로 인한 시간 소요



```
$ mlfq_test
MLFQ test start
[Test 1] default
Process 8
L0: 100000
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 4
L0: 100000
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 10
L0: 100000
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 6
L0: 100000
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 11
L0: 12039
L1: 8610
L2: 0
L3: 79351
MoQ: 0
Process 9
L0: 12190
L1: 9276
L2: 0
L3: 78534
```



```
  else if(myproc()->pid % 2 == 0 && myproc()->level == 1){
    myproc()->level = 2;
    myproc()->timeQuantum = 0;
  }
```

→

위 사진에 나와있다싶이 Test 1번에서 L0 큐에서 더이상 내려오지 않는걸 알 수 있다. 원인을 찾기 위해 scheduler()와 priority boosting을 한참 분석했다. 특히 자꾸 L0에 머무는 것을 보아 priority boosting 문제인줄 알았다. 하지만 그냥 level이 0일때만 옮겨줘야하는데 1일때 옮겨주는 걸로 실수를 하여 생긴 일이었다.

C. panic: trap

해결하는데 가장 많은 시간을 소요했다. 이것으로 인해 아래 사진들과 같이 부팅을 하다가 멈추고, 부팅은 완료 되었지만 다시 멈추는 등 trap 14에 관한 오류가 굉장히 많았다. grep -rn "unexpected trap"을 찾아보니 trap.c에서 생긴 오류인 것을 알 수 있었고, 더 찾아보니 page fault로 인한 오류임을 알 수 있었다. 이때에는 time quantum과 priority boosting에 관한 문제를 scheduler()에서 처리했었다. Ticks 관리와 priority boosting을 따로 해결하니 복잡하고 문제가 많이 생기는 듯 하였다. 그래서 myproc()->timeQuantum++;을 trap.c에서 해주고 프로세스의 timeQuantum이 다 차면 yield()를 부르는 동작도 같이 해주니 문제가 해결되어 부팅이 안전하게 되었다.

```
Booting from Hard Disk...
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
unexpected trap 14 from cpu 0 eip 80104597 (cr2=0xc)
lapicid 0: panic: trap
 801067f1 801063f8 8010301f 8010316c 0 0 0 0 0 0QEMU: Terminated
```

```
Booting from Hard Disk...
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$ mlfq_test
MLFQ test start
[Test 1] default
unexpected trap 14 from cpu 0 eip 80104597 (cr2=0xc)
lapicid 0: panic: trap
 801067d1 801063f8 8010301f 8010316c 0 0 0 0 0 0QEMU: Terminated
```

D. panic: zombie exit

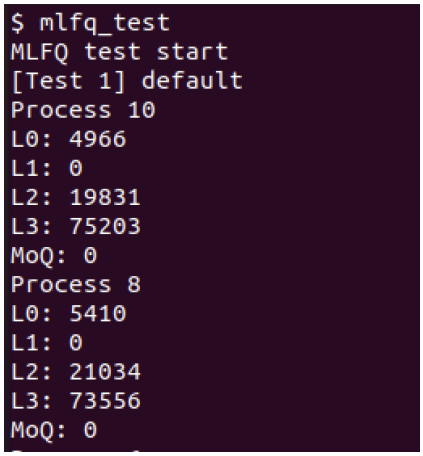
```
Process 12
L0: 9266
L1: 0
L2: 30616
L3: 60118
MoQ: 0
lapicid 0: panic: zombie exit
 80104691 80105f8f 8010533d 8010656d 80106258 0 0 0 0 0QEMU: Terminated
```

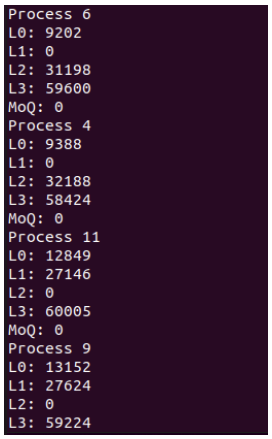
위 문제를 해결하니 자꾸 Test 2번 실행 중간에 zombie exit에 대한 경고가 떴다. Zombie 상태란 프로세스가 일을 끝낸 후에도 ptable에 남아있는 상태이다. 이 뜻을 정확히 파악한 후 priorityBoosting()을 다시보니 ptable을 사용할때 UNUSED만 고려한 조건문을 볼 수 있었다. 그래서 아래 코드와 같이 해당 조건문에 p->state == ZOMBIE라는 조건을 덧붙여주니 문제가 해결되었다.

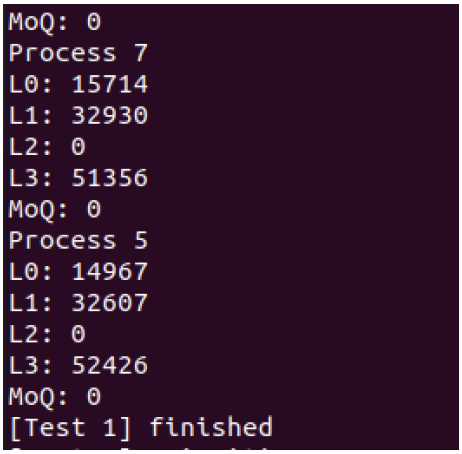
```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state == ZOMBIE || p->state == UNUSED || p->level == 0 ||
l == 99){
                continue;
        }
        p->timeQuantum = 0;
        p->level = 0;
        p->priority = 0;
        enqueue(&ptable.mlfq_queues[0], p);
}
```

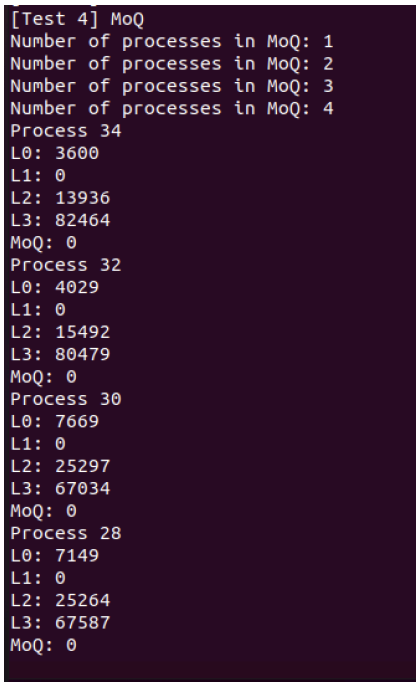
E. Test 예시와 다른 결과로 인한 **scheduler()** 코드 변경

아래 Test 1번 결과를 보면 제시된 test 결과 예시와는 다르게 짝수 pid를 가진 프로세스부터 차례로 나오는 것을 볼 수 있다. 또한, Test 4번 결과를 보면 moq인 프로세스에 대한 결과가 출력이 되지 않은 것을 확인했다. 처음에는 interrupt 문제인지 다른 시스템 콜 함수들에 의한 문제인지 파악이 잘 되지 않았지만, 결국 scheduler() 함수 내에서의 문제라는 것을 깨달았다. 그리하여 전체 코드를 바꾸었다. 처음에는 각각의 큐에 RUNNABLE하거나 RUNNING하고 있지 않은 프로세스들은 모두 없애주었다. 그 상태에서 스케줄링을 하다보니 출력문이 흐트러지는 등 문제가 많았다. 그래서 큐에 들어온 모든 프로세스들을 스케줄링 시킨 후 dequeue 후 다시 q의 rear에 enqueue 시켜주어 다시 큐에 들어오게끔 하였다. 이를 기반으로 코드를 변경하니 위의 결과에서 보이듯이 잘 나오는것을 확인했다.









하지만 여전히 풀리지 않는 문제가 있었다. Test 4번에서 디버깅을 통해 확인한 결과 pid 35까지 나온 것을 보아 moq 프로세스는 잘 됐지만 출력 문제인 것으로 보인다.

```
[Test 4] MoQ
Number of processes in MoQ: 1
Number of processes in MoQ: 2
Number of processes in MoQ: 3
Number of processes in MoQ: 4
size: 4
pid: 29
size: 3
pid: 31
Process 31
L0: 0
L1: 0
L2: 0
L3: 0
MoQ: 100000
size: 2
pid: 33
size: 1
pid: 35
size: 0
Process 28
L0: 2572
L1: 0
L2: 13019
L3: 84409
MoQ: 0
Process 32
L0: 6704
L1: 0
L2: 21987
```

→ 뭔가 나온걸 볼 수 있음.