[ELE3021] Project02 Wiki

한양대학교 컴퓨터소프트웨어학부 2020089007 김형민

목차

Introduction

디자인과 구현, 두 측면으로 나누어 서술하기에는 기본적이고 간단한 것들이라 이곳에 서술한다.

MLFQ와 MoQ , 각각에서 활용될 수 있는 요소들을 담고 있다.

movemlfq나 popmlfq 같이 아직 설명하지 않은 함수들도 포함하고 있지만 그 이름에서 뜻을 알 수 있다.

흐름이 이해되지 않는다면 밑으로 내려가 디자인 및 구현 설명에서 관련 함수들을 찾아보길 권한다.

또, 앞으로 나올 코드들 사이에서 개인적으로 추가한 Lock들이 사용된다.

우리 과제의 조건은 싱글 cpu이고 xv6는 멀티 스레드를 지원하지 않는다.

따라서 ptable에 대한 Lock을 가지고 있는 이상 인터럽트는 비활성화 되므로 동기화는 문제 되지 않는다.

acquire와 release만 적절히 이루어진다면 Project02 내에서 중요한 문제는 아니다.

따라서 추가적으로 작성한 Lock들에 대해서 깊게 들어가지 않을 계획이니 이해해주길 바란다.

또 작성한 코드 내에 주석이 적절하지 않은 경우가 꽤 있다.

이는 디자인을 수정하면서 시간 관계상 미처 바로잡지 못한 것들이니 위키를 기준으로 참고하길 바란다.

Process

Struct of Process

```
// proc.h
sturct proc {
  ...
  int level;
  int priority;
  int pticks;
  int order;
  int uyield;
}
```

1. **int level**

프로세스가 속한 큐의 레벨을 나타낸다. 0 에서 3, 그리고 MoQ에 속할 경우 99를 할당받는다.

처음 프로세스가 생성될 때 0으로 초기화된다.

2. **int priority**

프로세스가 가진 우선순위를 나타낸다. 0에서 10 사이의 값을 가진다.

프로세스가 생성될 때 0으로 초기화된다. L3 큐에서 priority 스케줄링을 위해 사용된다.

3. **int pticks**

프로세스가 타임 퀀텀을 얼마나 소모했는지를 나타낸다. cpu를 할당받아 실행될 때마다 1씩 증가한다.

프로세스가 생성될 때 0으로 초기화된다. MLFQ에서 사용된다.

1. **int order**

프로세스가 MoQ에 도착한 순서를 나타낸다. 0에서 NRROC(64) 사이의 값을 가진다.

값이 작을수록 MoQ의 FCFS 스케줄링에서의 우선순위가 높다. 일종의 우선순위처럼 사용된다.

프로세스가 생성될 때 NPROC(64)으로 초기화된다. 64는 MoQ에 없다는 것을 나타내기도 한다.

1. **int uyield**

유저 프로세스가 yield를 호출했는지를 나타낸다. 유저가 yield했을 경우 sys_yield에서 1로 설정된다.

프로세스가 생성될 때 0으로 초기화된다.

Check usable process

```
// proc.c
static int
isusableproc(struct proc *p)
{
  if(p != 0 && p->state == RUNNABLE)
    return 1;
  return 0;
}
```

프로세스가 NULL은 아닌지, RUNNABLE한 상태인지 검사한다.

사용가능하면 1, 사용불가능하면 0을 반환한다. proc.c 파일 내에서만 사용되기에 static으로 선언한다.

Required system calls

Common

아래 나열한 6개의 시스템 콜들은 유저와 커널 모두 사용가능해야 한다.

따라서 구현 세부사항에 있어서는 조금씩 차이가 있으나 큰 틀에서 해야할 일들은 모두 같다.

1. **proc.c**: 함수들에 대한 대한 명세의 요구 조건들을 구현한다.
2. **sysproc.c**: 함수들을 유효하지 않은 인자들로부터 보호하는 wrapper 함수들을 구현한다.
3. **defs.h**: 명세에서 요구하는 함수들을 선언한다.
4. **syscall.c**: wrapper 함수들을 extern으로 선언한다.
5. **syscall.c**: void syscall()에서 호출가능하도록 함수들을 등록한다.
6. **syscall.h**: wrapper 함수들에 대한 매크로를 정의한다.
7. **usys.S**: 유저 프로그램에서 함수들을 사용할 수 있도록 만드는 매크로들을 정의한다.
8. **user.h**: 유저 프로그램에서 사용가능하도록 함수들을 선언한다.

void yield(void)

Requirements

1. 자신이 점유한 cpu를 양보합니다.

```
// proc.c
void
yield(void)
{
  ...
  if(p->uyield == 1 && p->level != 99)
    movemlfq(p, 0);
  else
    p->uyield = 0;
  ...
}

// sysproc.c
int
sys_yield(void)
{
  myproc()->uyield = 1;
  yield();
  return 0;
}
```

시스템 콜 함수

기존에 구현되어 있던 yield 함수의 기능들은 유지한다.

만약 프로세스의 uyield 속성 값이 1이라면 프로세스를 L0 큐로 이동시키고 타임 퀀텀을 0으로 만든다.

wrapper 함수

wrapper 함수는 인자 전처리가 필요하지 않다. 추가로 프로세스의 uyield 변수를 1로 만들어준다.

이는 사용자가 만든 프로세스가 yield를 호출할 경우 L0로 이동시키고 타임 퀀텀을 초기화하는데 사용된다.

프로세스가 애매하게 남은 타임 퀀텀을 소진하기 위해 다시 스케줄링 되는 것은 비효율적인 작업이다.

또, 사용자 레벨에서 yield를 호출한 프로세스는 다른 프로세스보다 상대적으로 cpu를 적게 사용한다.

따라서 Time sharing의 공평성을 생각해 이런 식으로 구현할 것이다.

만약 MoQ라면 이동시켜서는 안되므로 if 분기를 통해 해결한다.

int getlev(void)

Requirements

1. 프로세스가 속한 큐의 레벨을 반환합니다.
2. MoQ에 속한 프로세스의 경우 99를 반환합니다.

```
// proc.c
int
getlev(void)
{
  return myproc()->level;
}
```

```
// sysproc.c
int
sys_getlev(void)
{
  return getlev();
}
```

시스템 콜 함수

getlev 함수는 현재 프로세스의 level 속성 값을 반환한다.

wrapper 함수

wrapper 함수는 인자 전처리가 필요하지 않으므로 getlev() 함수에서 반환한 값 그대로 반환한다.

int setpriority(int pid, int priority)

Requirements

1. 특정 pid를 가지는 프로세스의 priority를 설정합니다.
2. priority 설정에 성공한 경우 0을 반환합니다.
3. 주어진 pid를 가진 프로세스가 존재하지 않는 경우 -1을 반환합니다.
4. priority가 0이상 10 이하의 정수가 아닌 경우 -2를 반환합니다.

```
// proc.c
int
setpriority(int pid, int priority)
{
  if(priority < 0 || priority > 10)
    return -2;

  int rv = -1;
  struct proc *p = myproc();
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(pid == p->pid){
      p->priority = priority;
      rv = 0;
      break;
    }
  }
  return rv;
}

// sysproc.c
int
sys_setpriority(void)
{
  int pid, priority;
  if(argint(0, &pid) < 0)
```

```
    return -3;
  if(argint(1, &priority) < 0)
    return -3;
  return setpriority(pid, priority);
}
```

시스템 콜 함수

1번 요구조건을 충족하기 위해 ptable을 순회하면서 같은 pid를 가진 프로세스가 있는지 확인한다.

존재한다면 프로세스의 priority 속성 값을 인자로 받은 priority 값으로 설정한다.

priority 설정에 성공하면 0을 반환한다.

3, 4번 요구조건의 경우 예외처리에 해당하므로 각각의 요구조건에 맞게 적절히 구현한다.

wrapper 함수

wrapper 함수는 pid와 priority, 두 번의 인자 전처리가 요구되므로, argint 함수를 이용해서 구현해준다.

또, 인자 전처리에 실패했을 때 -1을 반환하면 명세와 충돌할 수 있으므로 -3을 반환하도록 한다.

인자들이 유효함을 확인했다면 이들을 setpriority 함수에 인자로 넘겨준다.

최종적으로 setpriority에서 반환한 값을 그대로 반환한다.

int setmonopoly(int pid, int password)

Requirements

1. 특정 pid를 가진 프로세스를 MoQ로 이동합니다. 독점 자격을 증명할 암호(학번)을 받습니다.
2. 암호가 일치할 경우, MoQ의 크기를 반환합니다.
3. pid가 존재하지 않는 프로세스인 경우 -1을 반환합니다.
4. 암호가 일치하지 않는 경우 -2를 반환합니다.

```
// proc.c
setmonopoly(int pid, int password)
{
  if(password != 2020089007)
    return -2;

  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(pid == p->pid){
        if(popmlfq(p) == -1)
          return -1;
        return pushmoq(p);
      }
  }
  return -1;
}

// sysproc.c
int
```

```
sys_setmonopoly(void)
{
  int pid, password;
  if(argint(0, &pid) < 0)
    return -3;
  if(argint(1, &password) < 0)
    return -3;
  return setmonopoly(pid, password);
}
```

시스템 콜 함수

1번 요구조건을 충족하기 위해 ptable을 순회하면서 같은 pid를 가진 프로세스가 있는지 확인한다.

같은 pid를 가진 프로세스가 존재한다면 프로세스를 MLFQ에서 MoQ로 이동시킨다.

프로세스 이동에 성공하면 MoQ의 size 속성 값을 반환한다.

2, 3, 4번 요구조건의 경우 단순히 값을 반환하는 것이므로 각각의 요구조건에 맞게 적절히 구현한다.

wrapper 함수

wrapper 함수는 pid와 password, 두 번의 인자 전처리가 요구된다.

argint 함수를 이용해서 적절히 구현해준다.

또, 인자 전처리에 실패했을 때 -1을 반환하면 명세의 요구조건과 충돌할 수 있으므로 -3을 반환하도록 한다.

인자들이 유효함을 확인했다면 이들을 setmonopoly 함수에 인자로 넘겨준다.

최종적으로 setmonopoly 에서 반환한 값을 그대로 반환한다.

void monopolize(void)

Requirements

1. MoQ의 프로세스가 cpu를 독점하여 사용하도록 설정합니다.

```
// proc.c
void
monopolize(void)
{
  acquire(&modelock);
  schedule_mode = MoQ;
  release(&modelock);
  yield();
}

// sysproc.c
int
sys_monopolize(void)
{
  monopolize();
  return 0;
}
```

시스템 콜 함수

스케줄링 모드 변수를 MoQ로 변경하고, yield를 호출하여 현재 프로세스의 cpu사용을 반납한다.

wrapper 함수

인자 전처리가 요구되지 않으므로 단순히 monopolize를 호출한다.

void unmonopolize(void)

Requirements

1. 독점적 스케줄링을 중지하고 기존의 MLFQ part로 돌아갑니다.
2. Global tick을 0으로 초기화 합니다.

```
// proc.c
void
unmonopolize(void)
{
  acquire(&modelock);
  schedule_mode = MLFQ;
  release(&modelock);

  acquire(&tickslock);
  ticks = 0;
  release(&tickslock);

  acquire(&moqlock);
  moq.size = 0;
  release(&moqlock);
}

// sysproc.c
int
sys_unmonopolize(void)
{
  unmonopolize();
  return 0;
}
```

시스템 콜 함수

글로벌 틱과 MoQ의 size를 0으로 초기화하고, 스케줄링 모드 변수를 MLFQ로 변경한다.

wrapper 함수

인자 전처리가 요구되지 않으므로 단순히 unmonopolize를 호출한다.

Design

MLFQ(Multi-Level Feedback Queue)

Specification

1. L0-L3, 총 4개의 큐로 이루어져 있고, 수가 작을수록 우선순위가 높은 큐이다.
2. 처음 실행된 프로세스는 모두 가장 높은 레벨의 큐(L0)에 들어간다.
3. Li 큐는 2i + 2ticks의 타임 퀀텀을 가진다.
4. L0, L1, L2큐는 기본 RR 정책을 따른다.
5. L0→L1→L2 순서로 앞의 큐에 RUNNABLE한 프로세스가 없을 경우 다음 큐로 내려간다.
6. pid가 홀수인 프로세스들은 L1, 짝수인 프로세스들은 L2로 내려가고 타임 퀀텀을 초기화한다.
7. L1, L2 큐에서 타임 퀀텀을 모두 사용하면 L3로 내려가고 타임 퀀텀을 초기화한다.
8. L3 큐는 priority 스케줄링을 한다. 같은 큐 내에서는 priority가 높은 프로세스가 먼저 스케줄링 된다.
9. Priority가 같은 프로세스끼리는 어느 것을 먼저 스케줄링해도 된다.
10. Priority 값은 L3 큐에서만 유효하다.
11. L3 큐의 프로세스가 time quantum을 모두 소모하면 priority가 1 감소하고, 타임 퀀텀을 초기화한다.
12. Global tick이 100ticks가 될때마다 priority boosting(타임 퀀텀 초기화, L0 이동)을 진행한다 .
13. 스케줄러에게 실행 흐름이 넘어올 때마다, RUNNABLE한 프로세스가 존재하는 큐 중 가장 레벨이 높은 큐를 선택한다.

Implementation Plan

MLFQ는 그 이름에서 알 수 있듯이 큐의 집합으로 이루어져 있다.

따라서 큐 구조체와 그에 맞는 push, pop, empty 등의 함수들이 필요하다.

```
// proc.h
struct mlfq { ... };

// proc.c
struct mlfq mlfq[4];
struct spinlock mlfqlock;

static void priorityboosting(void);
static int isemptymlfq(struct proc *p);
static int popmlfq(struct proc *p);
static int pushmlfq(struct proc *p, int level);
static int movemlfq(struct proc *p, int level);
```

mlfq는 총 4개의 큐로 이루어져 있으므로, 크기 4의 mlfq 구조체 배열을 선언한다.

mlfq 구조체는 이름은 mlfq지만 L0, L1, L2, L3와 같은 하나의 큐를 의미한다.

또, 프로세스들을 이동시키는 작업이 필요하므로 pushmlfq, popmlfq, movemlfq 함수가 요구된다.

popmlfq를 위해선 empty check가 필요하므로 isemptymlfq 함수도 필요하다.

starvation을 막기 위한 priority boosting도 필요하므로 priorityboosting 함수도 만들 것이다.

이 함수들은 proc.c 파일 내에서 스케줄링 목적으로만 사용될 것이기에 static으로 선언해준다.

1번 명세: mlfq 구조체 배열을 활용한다.

2번 명세: allocproc 함수에서 갓 생성된 프로세스를 pushmlfq로 L0 큐에 넣어준다.

3번 명세: mlfq 구조체의 속성으로 만들어 level 별로 조건에 맞는 값을 가지도록 한다.

4,5, 10, 13번 명세: 스케줄링 라운드 마다 제일 먼저 가장 높은 레벨의 RUNNABLE한 프로세스를 찾는다.

L0, L1, L2 큐는 RR 정책을 채용하므로, 만약 이전에 스케줄링한 프로세스가

1. RUNNABLE하고
2. 타임 퀀텀을 다 소모하지 않았으며
3. 앞서 선택한 가장 높은 레벨의 프로세스와 같은 레벨이라면

우리는 이전에 스케줄링 했던 그 프로세스를 다시 스케줄링 해줄 필요가 있다. priority는 L3에서만 의미를 가지므로 여기서는 고려하지 않는다.

6,7, 11번 명세: 프로세스가 한 라운드 동안 실행을 마치고 스케줄러에게 다시 실행 흐름이 넘어오면 pticks를 1 증가시키면서 이 프로세스가 타임 퀀텀을 다 소모하지는 않았는지 확인한다. 만약 L0라면 pid가 짝수면 L2, 홀수면 L1으로 이동시킨다. 만약 L1, L2라면 L3로 이동시킨다. L3라면 priority > 0일 때, priority를 1 감소시킨다. 앞선 과정 모두 공통적으로 타임 퀀텀을 초기화한다.

8, 9번 명세: 앞서 가장 높은 레벨의 프로세스를 찾은 뒤, prev 프로세스가 조건을 충족하는지 확인하고 실행할 프로세스를 결정했었다. 만약 가장 높은 레벨이3이라면 앞선 두 가지 절차 뒤에 priority 스케줄링을 진행해야 한다. ptable을 처음부터 순회하면서 각각의 priority를 비교하고 그 중 가장 priority가 높은 프로세스가 스케줄링 될 수 있도록 한다. priority가 같은 프로세스 사이의 스케줄링 순서는 상관 없다.

12번 명세: 스케줄러에게 실행 흐름이 넘어올 때마다 global tick(ticks 변수)의 값을 확인하면서 100으로 나누어 떨어진다면 priority boosting을 진행한다.

앞선 내용 중 스케줄러 관련 핵심 흐름을 정리하면 다음과 같다.

1. ptable을 순회하면서 가장 레벨이 높은 큐에 속한 프로세스를 고른다.
2. prev 프로세스의 유효(위의 조건 참조) 여부를 확인하면서 유효하다면 prev로 선택을 바꾼다.
3. 1, 2에서고른 프로세스의 레벨이 3이라면 ptable을 다시순회하면서 priority에 의한 선택을 진행한다.
4. 위의 과정을 거쳐 선택된 프로세스를 스케줄링한다.
5. 실행이 끝나고 실행 흐름이 스케줄러에게 넘어오면 프로세스가 타임 퀀텀을 소모했는지 확인한다.
6. 소모했으면 프로세스의 상태에 따라 적절하게 큐 이동과 상태 변경을 해주고 타임 퀀텀을 초기화한다.
7. 만약 ticks가 0보다 크면서 100으로 나누어 떨어진다면 priority boosting을 진행한다.

MLFQ 스케줄러는 위 과정을 계속해서 반복한다.

MoQ(Monopoly Queue)

Specification

1. MoQ는 FCFS 정책을 따른다.
2. 평소에는 스케줄링 되지 않다가 monopolize 시스템콜이 호출되면 즉시 스케줄링 된다.
3. priority boosting은 MoQ 스케줄링 동안 발생하지 않는다.
4. 프로세스는 setmonopoly 시스템 콜을 통해 MoQ로 이동한다.

5. MoQ의 모든 프로세스들이 종료되면 자동으로 unmonopolize 시스템 콜이 호출되고 MLFQ로 돌아간다.
6. unmonopolize 시스템 콜이 호출되면, Global tick은 0으로 초기화된다.

Implementation Plan

MoQ는 단순히 하나의 큐로 이루어져 있다.

mlfq는 큐와 큐 사이의 이동을 고려해야 했지만, MoQ의 경우 push하는 경우만 생각하면 된다.

```
//proc.h
struct moq { ... };

//proc.c
struct moq moq;
struct spinlock moqlock;

static int isemptymoq(void);
static int pushmoq(struct proc *p);
```

MoQ는 setmonopoly 시스템 콜이 호출될 때마다 단순히 MLFQ에 있던 프로세스를 MoQ로 옮기는 작업만 요구된다. 따라서 우리는 pop과 move 등에 대해서는 생각할 필요가 없다.

그럼에도 isemptymoq가 필요한 것은 모든 프로세스들이 실행을 마치고 종료 되었을 때 unmonopolize를 호출해줘야 하기 때문이다.

1,2,4,5번 명세: FCFS 스케줄링 역시 일종의 priority 스케줄링이라고 볼 수 있다. 도착 순서가 priority가 되는 것이다. setmonopoly에 의해 MoQ에 도착한 순서에 따라 프로세스의 order 속성의 값이 결정되고, 이것이 곧 priority가 되는 것이다. 따라서 이 경우 역시 ptable을 순회하면서 가장 order 값이 작은, 즉 RUNNABL한 프로세스들 중 가장 먼저 도착한 프로세스를 선정한 다음, 그 프로세스를 스케줄링 한다. 선점형 스케줄링이므로 먼저 도착한 프로세스가 스케줄링을 종료 때까지 cpu를 독점한다. 평소에는 MLFQ 스케줄링을 하다가 monopolize 시스템 콜이 호출되면 MoQ 스케줄링으로 전환된다. isemptymoq 검사가 스케줄러에게 실행 흐름이 넘어올 때마다 진행되므로 MoQ 내의 모든 프로세스가 종료되었다면unmonopolize 시스템 콜에 의해 다시 MLFQ로 돌아간다.

3번 명세: MLFQ와 MoQ는 디자인 상 각각 독립된 코드 영역에서 실행될 것이므로 MoQ 동안 priority boosting은 이루어지지 않는다.

6번 명세: unmonopolize 시스템 콜 내에서 tickslock을 걸고 ticks를 0으로 초기화 하도록 한다.

앞선 내용 중 스케줄러 관련 핵심 흐름을 정리하면 다음과 같다.

1. ptable을 순회하면서 가장 도착 순서가 빠른(order 값이 작은) 프로세스를 선택한다.
2. 선택한 프로세스를 스케줄링한다.
3. 실행을 마치고 스케줄러에게 실행 흐름이 넘어오면 MoQ가 비었는지 확인한다.
4. 만약 비었다면 unmonopolize를 호출하고 MLFQ로 돌아간다.

MoQ 스케줄러는 위 과정을 계속해서 반복한다.

Switching Between MLFQ and MoQ

Implementation Plan

C언어에서 제공하는 goto와 Label문을 이용해서 구현한다.

```
// proc.h
enum schedulemode { MLFQ, MoQ };

// proc.c
uint schedule_mode;
struct spinlock modelock;
```

schedule_mode 변수를 이용한다. enum을 이용해서 가독성을 높이고, 실수의 가능성을 낮춘다.

기본 값은 MLFQ(0)이다.

monopolize 시스템 콜이 호출되면 이 변수의 값이 MoQ(1)로 바뀌고 MLFQ 스케줄러 코드의 말미에서 그 값을 확인한다음 goto를 이용해 MoQ 스케줄러 코드로 이동한다.

unmonopolize 시스템 콜이 호출되면 이 변수의 값이 MLFQ(0)로 바뀌고 MoQ 스케줄러 코드의 말미에서 그 값을 확인한다음 goto를 이용해 MLFQ 스케줄러 코드로 이동한다.

Implement

Common

Process initialization during allocation

```
// proc.c
static struct proc*
allocproc(void)
{
    ...
    found:
    ...
      p->level = 0;
      p->priority = 0;
      p->pticks = 0;
      p->order = NPROC;
      p->uyield = 0;
    ...

    pushmlfq(p, 0);
    return p;
}
```

프로세스가 새로 생성되면 우리가 추가한 proc 구조체 내의 속성들을 모두 초기화 해준 뒤 명세를 따라 mlfq의 L0큐에 push한다.

Move process to L0 when SLEEP calls

```
// proc.c
void
sleep(void *chan, struct spinlock *lk)
{
  ...
  p->chan = 0;
  if(p->level != 99)
    movemlfq(p, 0);
  ...
}
```

sleep을 마치고 돌아오면 movemlfq를 통해 L0 큐로 옮긴 다음 타임 퀀텀을 초기화해준다. 이때 MoQ에 있다면 move해주면 안된다.

Scheduler initialization

```
void
scheduler(void)
{
  uint i;
  struct proc *p, *pp, *hp;
  struct cpu *c = mycpu();
  c->proc = 0;

  acquire(&modelock);
  schedule_mode = MLFQ;
  release(&modelock);

  acquire(&mlfqlock);
  for(i = 0; i < 4; i++){
    mlfq[i].tq = 2 + (i << 1);
    mlfq[i].prev = 0;
  }
  release(&mlfqlock);

  acquire(&moqlock);
  moq.size = 0;
  release(&moqlock);
  ...
}
```

스케줄러는 never return하므로 여기서 진행하는 초기화는 두 번 반복되지는 않는다.

먼저 기본 schedule_mode를 MLFQ로 설정해준다. 그리고 mlfq 구조체 배열의 속성들을 각각 초기화해준다.

또 moq의 size 속성을 초기화해준다.

MLFQ(Multi-Level Feedback Queue)

Struct of MLFQ

```
// proc.h
struct mlfq {
  int tq;
  struct proc *prev;
  struct proc *proc[NPROC];
};

// proc.c
struct mlfq mlfq[4];
struct spinlock mlfqlock;
```

struct mlfq는 Li큐를 추상화한 구조체이다. 따라서 우리는 길이 4의 struct mlfq 배열이 필요하다.

1. int tq는 mlfq의 Li 큐가 가진 한계 타임퀀텀을 나타낸다. Li 큐의 경우 2 + 2*i의 값을 가진다.
2. struct proc *prev는 앞서 디자인에서 말했듯이 이전에 스케줄링한 프로세스를 저장하기 위한 속성이다.
3. struct proc *proc[NPROC]은 Li 큐들에 들어있는 프로세스들의 배열이다. xv6는 최대 NPROC만큼 프로세스를 만들 수 있으므로 여기도 마찬가지로 최대 NPROC의 프로세스를 가질 수 있도록 해준다.

Check is empty

```
// proc.c
static int
isemptymlfq(struct proc *p)
{
  int i;
  for(i = 0; i < NPROC; i++){
    if(mlfq[p->level].proc[i] != 0 &&
    (mlfq[p->level].proc[i]->state != UNUSED
    && mlfq[p->level].proc[i]->state != ZOMBIE))
      return 0;
  }
  return 1;
}
```

이 함수는 L(p→level) 큐가 비어있는지 확인할 때 사용된다.

1. struct proc *p를 인자로 받는다.
2. 이 프로세스가 속한 L(p→level)큐가 비어있으면 1, 비어있지 않으면 0을 반환한다.
3. mlfq[p→level].proc[i]가 존재하더라도 그것이 UNUSED나 ZOMBIE라면 비어있는 것으로 간주한다.

Pop process

```
// proc.c
static int
popmlfq(struct proc *p)
```

```
{
  if(isemptymlfq(p))
    return -1;

  int i;
  acquire(&mlfqlock);
  for(i = 0; i < NPROC; i++){
    if(mlfq[p->level].proc[i] == 0)
      continue;
    if(mlfq[p->level].proc[i]->pid == p->pid){
      mlfq[p->level].proc[i] = 0;
      release(&mlfqlock);
      return 0;
    }
  }
  release(&mlfqlock);
  return -1;
}
```

이 함수는 L(p→level) 큐에서 프로세스를 제거할 때 사용한다.

1. L(p→level) 큐가 비어있는지 확인하고, 비어있으면 -1을 반환한다.
2. 큐를 순회하면서 인자로 받은 p의 pid와 일치하는 pid를 가진 프로세스가 큐 내에 있는지 확인하고, 있으면 그 프로세스를 pop한 뒤 0을 반환한다. 순회가 끝났다면 p가 큐 내에 없다는 뜻이므로 -1을 반환한다.

Push process

```
// proc.c
static int
pushmlfq(struct proc *p, int level)
{
  int i;
  acquire(&mlfqlock);
  for(i = 0; i < NPROC; i++){
    if(mlfq[level].proc[i] == 0 || mlfq[level].proc[i]->state == UNUSED){
      p->level = level;
      p->pticks = 0;
      p->uyield = 0;
      mlfq[level].proc[i] = p;
      release(&mlfqlock);
      return 0;
    }
  }
  release(&mlfqlock);
  return -1;
}
```

이 함수는 L(level) 큐에 프로세스를 push할 때 사용한다.

1. 큐를 순회하면서 자리가 비었거나 자리를 차지한 프로세스가 UNUSED인지 확인한다.
2. 자리가 있다면 인자로 받은 p의 level을 바꾸고, 타임 퀀텀(pticks)를 초기화하고, uyield를 0으로 한다. 왜냐면 우리는 사용자가 yield 했을 시 L0 큐로 옮겨주고 타임 퀀텀을 초기화 해주는데 그 과정이 끝나면 uyield를 0으로 바꿔줘야 두 번 옮겨주는 일을 방지할 수 있다. p의 속성 설정이 끝났다면 L(level) 큐의 빈자리에 p를 할당해준다. 그 후 0을 반환한다.
3. 만약 빈 자리가 없다면 순회가 끝나게 되고, -1을 반환한다.

Move process

```
static int
movemlfq(struct proc *p, int level)
{
  if(p->level == 99)
    return -1;
  if(p->level == 3 && level == 3 && p->priority > 0)
    p->priority--;
  if(popmlfq(p) == -1)
    return -1;
  return pushmlfq(p, level);
}
```

이 함수는 프로세스를 다른 레벨의 큐로 옮겨줄 때 사용한다.

1. 먼저 p의 레벨이 99라면 -1을 반환한다. 레벨 99는 MoQ에 속한다는 것을 뜻한다. 우리가 monopolize 시스템 콜을 호출한 뒤 MoQ에 진입하기 전까지 계속해서 타이머 인터럽트가 발생하기 때문에 setmonopoly 시스템 콜에 의해 MoQ로 이동한 함수들이들이 movemlfq에 의해서 의도치 않게 MLFQ로 이동하는 것을 막아줄 필요가 있다. 인터럽트 비활성화, lock 등 여러가지 해결 방법이 있겠지만 가장 간단한 방법으로 결정했다.
2. 만약 이동하려는 p가 L3큐에서 타임 퀀텀을 다 소모한 경우라면 p→priority가 0보다 크다는 조건하에 p→priority를 1 감소시켜준다.
3. 그 이후엔 차례대로 popmlfq를 통해 p를 원래 속한 L(p→level)큐에서 pop하고 이동 목표인 L(level)큐로 pushmlfq를 통해 push하는 식으로 이동시킨다.

Priorityboosting

```
// proc.c
static void
priorityboosting(void)
{
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == UNUSED || p->state == ZOMBIE)
      continue;
    movemlfq(p, 0);
  }
}
```

global tick(ticks)가 100ticks가 되면서 priorityboosting을 해야할 때 사용한다.

ptable을 순회하면서 UNUSED나 ZOMBIE 상태가 아닌 모든 프로세스들을 L0 큐로 이동시킨다.

MLFQ Scheduler

```
// proc.c
void
scheduler(void)
{
  ...
  for(;;){
    ...
    MultiLevelFeedbackQueue:
    // 1번
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(!isusableproc(p) || p->level == 99)
        continue;

      hp = p;
      // 2번
      // Select highest level process for RR scheduling of L0, L1, L2.
      for(pp = ptable.proc; pp < &ptable.proc[NPROC]; pp++){
        if(!isusableproc(pp) || p->level == 99)
          continue;

        if(pp->level < hp->level)
          hp = pp;
      }
      // 3번
      pp = mlfq[hp->level].prev;
      if(
          isusableproc(pp) &&
          pp->pticks > 0 &&
          (pp->level <= hp->level)
        )
        hp = pp;
      // 4번
      // Select process for priority scheduling of L3.
      if(hp->level == 3){
        for(pp = ptable.proc; pp < &ptable.proc[NPROC]; pp++){
          if(!isusableproc(pp) || p->level == 99)
            continue;

          // Process higher than level 3 already out
          if(pp->priority > hp->priority)
            hp = pp;
        }
```

```
      }
      p = hp;
      ...
```

1번은 MLFQ 스케줄러 코드 마지막에 나올 p = ptable.proc과 함께 무한루프를 형성한다. 이것은 MLFQ와 MoQ의 영역을 구분하기 위해 사용되고, 스케줄링할 프로세스를 선택하는 과정에 있어 구조적인 편의를 제공한다. MLFQ 스케줄링 자체에서 별다른 역할은 없다.

2번은 ptable을 순회하면서 가장 레벨이 높은 프로세스를 선택한다. 이 때, 이 프로세스는 usable($p \neq 0$ && RUNNABLE) 해야하고, MLFQ에 속해있어야 한다.

3번은 앞서 디자인에서 언급한 조건들을 prev 프로세스가 충족할 경우 prev 프로세스를 선택하는 코드이다. 그 이유로 L0, L1, L2는 RR 스케줄링 채택하기 때문에 프로세스는 ABABAB가 아닌 AAABBB와 같이 연속적인 스케줄링으로 타임 퀀텀을 보장받아야 하기 때문이다.

4번은 앞선 과정에서 선택한 가장 높은 레벨의 프로세스가 L3에 속할 경우 앞의 다른 조건들은 잊어버리고 ptable을 처음부터 다시 순회하면서 priority를 기준으로 다시 실행할 프로세스를 선택하는 코드이다. 이때 우선순위가 같은 프로세스 사이에선 순서 제약이 없으므로 상대적으로 먼저 생성된 프로세스가 cpu 사용에 있어 상대적인 혜택을 보게 된다. 그러나 starvation을 막기 위한 priority boosting을 사용하므로 문제는 아니다.

이렇게 2, 3, 4번 세 과정을 거쳐 실행할 프로세스가 선택된다.

```
      ...
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

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

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;

      // Increase pticks by one scheduling round.
      p->pticks++;

      // Save previous process for RR scheduling.
      mlfq[p->level].prev = p;

      // Running out of time quantum.
      if(mlfq[p->level].tq - p->pticks <= 0)
        movemlfq(p, p->level == 0 ? ((p->pid % 2 == 0) ? 2 : 1) : 3);

      // Priority boosting.
      if(ticks > 0 && ticks % 100 == 0)
        priorityboosting();
```

```
      // if user calls monopolize(), goto MonopolyQueue.
      if(schedule_mode == MoQ)
        goto MonopolyQueue;

      p = ptable.proc;
    }
    ...
```

실행할 프로세스의 결정이 끝났으므로 그 프로세스에게 실행 흐름을 넘겨준다.

이후 타이머 인터럽트, sleep, yield 등에 의해 스케줄러에게 실행 흐름이 다시 넘어오면 실행한 프로세스의 pticks를 1 증가시켜주고, L(p→level)의 이전 프로세스를 방금 실행한 p로 설정해준다.

만약 실행을 마친 프로세스가 타임 퀀텀을 다 소모하였다면 명세에 나온 조건에 맞게 프로세스를 MLFQ의 큐들 내에서 적절히 움직여준다. 구체적으로 0이고 pid가 짝수면 2로 0이고 pid가 홀수면 1로, 1, 2, 3이면 3으로 이동시켜준다. 삼항연산자가 중첩되어 한 눈에 파악하기 헷갈릴 수 있지만 명세대로 구현하였다.

그리고 100ticks마다 priorityboosting이 요구되므로 만약 global tick(ticks)가 0보다 크고 100으로 나누어 떨어진다면 priorityboosting을 진행해준다.

스케줄링 코드 말미에 스케줄링 모드가 바뀌었는지 확인한다음 바뀌었으면 goto로 이동하고 아니라면 p를 다시 ptable.proc으로 초기화 시켜 MLFQ 스케줄러 내에서 무한 루프를 돌도록 만든다.

MoQ(Monopoly Queue)

Struct of MoQ

```
// proc.h
struct moq {
  int size;
  struct proc *proc[NPROC];
};

// proc.c
struct moq moq;
struct spinlock moqlock;
```

1. MoQ는 프로세스의 배열인 struct proc *proc[NPROC]을 속성(멤버 변수)으로 가진다.
2. int size는 MoQ의 크기를 나타내는데, 이는 FCFS 스케줄링에서 사용된다. 도착한 순서 당시 size 대로 우선순위가 정해진다.
3. MoQ는 MLFQ와 다르게 단일 큐이므로 배열로 선언해줄 필요가 없으므로 하나의 객체로 선언해준다.

Check is empty

```
// proc.c
static int
isemptymoq(void)
{
```

```
  int i;
  for(i = 0; i < NPROC; i++){
    if(moq.proc[i] != 0 &&
    (moq.proc[i]->state != UNUSED && moq.proc[i]->state != ZOMBIE))
      return 0;
  }
  return 1;
}
```

앞서 설명한 isemptymlfq와 똑같은 로직이다.

MoQ를 순회하면서 비어있는지 확인하고, 비어있지 않으면 0, 비어있으면 1을 반환한다.

Push process

```
// proc.c
static int
pushmoq(struct proc *p)
{
  int i;
  acquire(&moqlock);
  for(i = 0; i < NPROC; i++){
    if(moq.proc[i] == 0 || moq.proc[i]->state == UNUSED){
      p->order = moq.size;
      p->level = 99;
      moq.size++;
      moq.proc[i] = p;
      release(&moqlock);
      return moq.size;
    }
  }
  release(&moqlock);
  return -1;
}
```

앞서 설명한 pushmlfq와 같은 로직이다.

MoQ를 순회하면서 push할 자리가 있는지 확인하고 자리가 있다면 p의 order(도착에 따른 우선순위, 낮을수록 우선순위 높음)를 도착 당시 MoQ의 size로 설정해주고 level을 99로 바꿔줌으로써 MoQ에 속해있음을 나타낸다. 그리고 최종적으로 MoQ의 빈 자리에 할당한다.

성공하면 push 후 MoQ의 크기, moq.size를 반환한다. 만약 자리가 없다면 -1을 반환한다.

MoQ Scheduler

```
// proc.c
    ...
    MonopolyQueue:
    // 1번
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
```

```
      if(!isusableproc(p) || p->level != 99)
        continue;

      hp = p;
      // 2번
      // It works because the order of arrival is the priority.
      for(pp = ptable.proc; pp < &ptable.proc[NPROC]; pp++){
        if(!isusableproc(pp) || p->level != 99)
          continue;
        if(pp->order < hp->order)
          hp = pp;
      }
      p = hp;
      ...
```

1번의 경우 앞서 MLFQ 스케줄러에서 보았던 1번과 같은 개념이다. 영역을 구분하고, 프로세스 선택의 용이성을 높이고, MoQ 스케줄링 동안 무한루프를 돌면서 빠져나가지 못하게 만든다.

2번의 경우는 FCFS 스케줄링을 나타낸다. 사실상 또 다른 priority 스케줄링이다. 먼저 도착한 프로세스는 더 작은 order 값을 갖고있기 때문에 가장 order가 작은 프로세스는 가장 우선순위가 높은 프로세스이다. 따라서 order를 priority처럼 생각하여서 실행할 프로세스를 선택한다. 여기서는 타임 퀀텀, RR 같은 개념이 없으므로 preemption이 일어난다.

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

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

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;

      // If moq is empty, unmonopolize.
      if(isemptymoq())
          unmonopolize();

      // If scedule_mode is MLFQ, goto MLFQ.
      if(schedule_mode == MLFQ)
        goto MultiLevelFeedbackQueue;

      p = ptable.proc;
    }
    ...
  }
}
```

앞에서 선택한 프로세스를 실행한다. 이후에 다시 실행 흐름이 스케줄러에게 넘어오면 MoQ가 비었는지 확인하고 비었으면 unmonopolize를 호출한다. 거기서 schedule_mode가 MLFQ로 바뀌게 되고 goto를 통해 다시 MLFQ 스케줄러 영역으로 넘어가게 된다. 만약 아직 실행할 프로세스가 큐에 남아있다면 MoQ 스케줄링이 계속 반복된다.

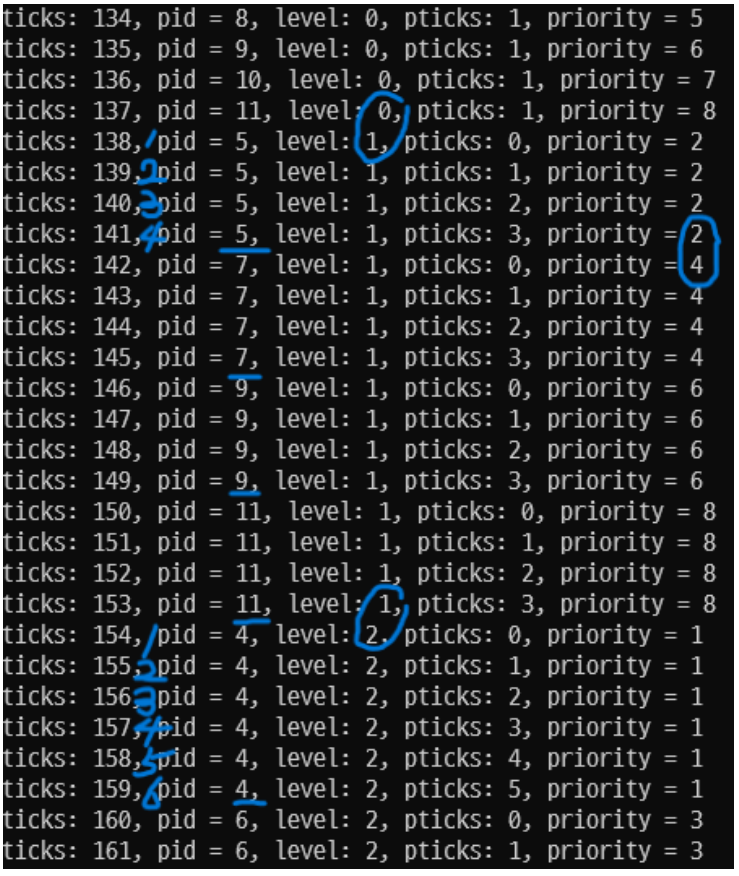
Result

How to Test

1. 다운 받은 xv6-public 폴더를 압축해제 한다.
2. 우분투 터미널에서 cd를 이용하여 xv6-public 폴더 내부로 이동한다.
3. 폴더 내부에서 “sudo chmod 777 -R ./” 명령어를 통해 해당 폴더에 대한 모든 권한을 받는다.
4. 터미널에 “make CPUS=1”, “make fs.img”를 입력하여 xv6를 컴파일한다.
5. 터미널에 “./bootxv6.sh”를 입력하여 QEMU 위의 xv6로 진입한다.
6. 테스트를 위해 xv6 터미널에 “mlfq_test”를 입력한다.
7. 출력 결과를 확인한다.

MLFQ(Multi-Level Feedback Queue)

Round Robin Scheduling



여기서는 RR 스케줄링에 관해 얘기하겠다.

위 사진은 Test 2를 실행한 결과를 나타낸다. pid가 홀수인 큐는 L1, 짝수인 큐는 L2로 넘어간다.

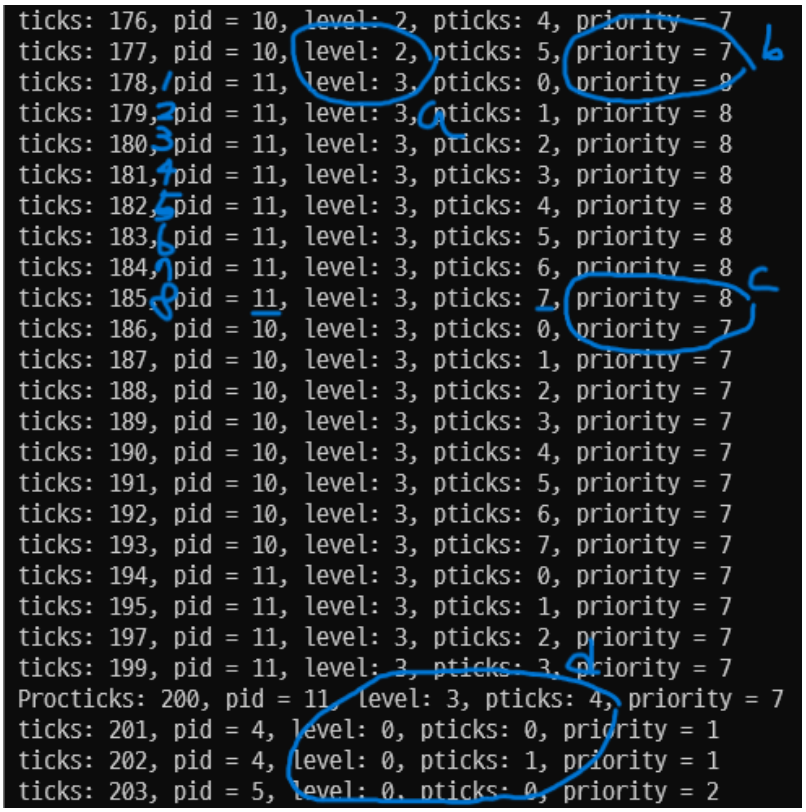
L0부터 L2까지, 각각의 큐가 비었을 경우 다음 레벨에 큐로 넘어가 순차적으로 실행된다.

또 각각에 할당된 타임 퀀텀에 맞게 L1큐는 총 4번, L2큐는 총 6번의 tick동안 cpu를 사용한다.

RR 명세에 맞게 프로세스가 57575757… 이런식이 아니라 55557777 이런식으로 실행된다.

또 pticks가 0부터 시작하는 것으로 보아 큐를 이동했을 때 0으로 초기화가 된 것도 확인할 수 있다.

Priority Scheduling



여기서는 priority 스케줄링 및 priority boosting이 이뤄지는 것을 설명하겠다.

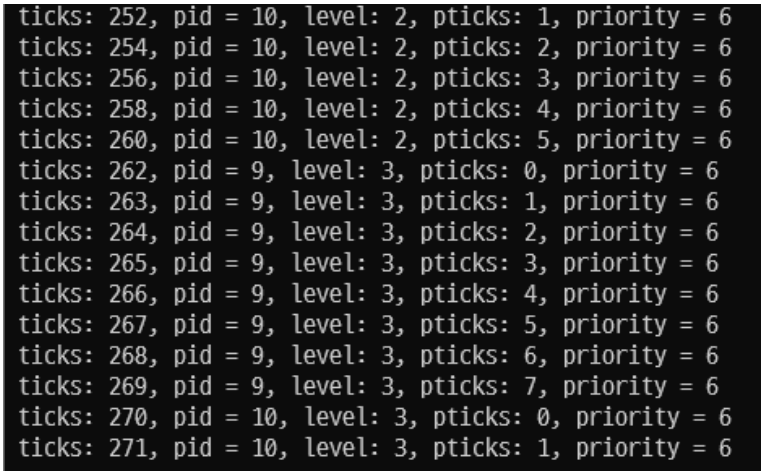
위 사진은 앞서 보여준 Test 2의 연장선이다. 더 높은 레벨의 큐들이 비었다면 L3 스케줄링이 시작된다.

오른쪽의 c 영역을 보면 priority가 더 높은 프로세스가 먼저 실행되는 것을 알 수 있다.

또 L3의 타임 퀀텀에 맞게 총 8번의 tick 동안만 cpu를 사용한다.

priority 감소 및 타임 퀀텀의 초기화는 다음에 나올 예시에서 보여줄 것이다.

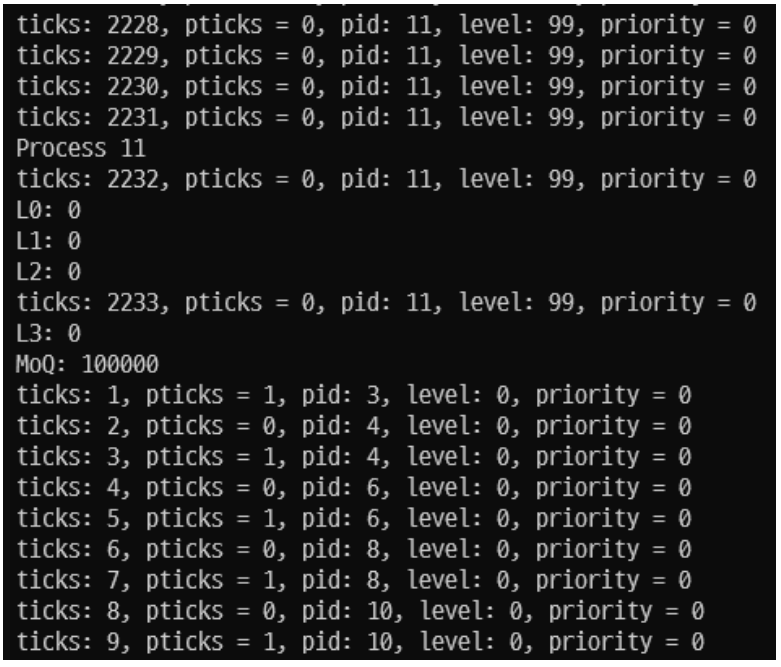
제일 밑에 d 영역은 priority boosting이 이뤄지는 것을 보여준다.



이제 L3 큐의 프로세스가 타임 퀀텀을 다 소모했을 경우 priority가 감소하고 pticks가 0이 된 것을 설명한다. 위 사진의 오른쪽을 보면 이전에 priority 7을 갖고 있던 pid 10의 프로세스가 타임 퀀텀을 다 소모한 후, priority가 6으로 감소한 것을 볼 수 있다. 중간에 priority boosting의 영향을 받아 살짝 헷갈릴 수는 있지만 명세에서 요구한대로 정상 동작 하였다.

MoQ(Monopoly Queue)

FCFS Scheduling



테스트 케이스 1-3 까지를 주석처리하고 진행하였다. 그에 따라 테스트 코드 내에서 monopolize를 진행하는 프로세스의 pid도 12로 적절하게 변경해줬다.

마지막 홀수 pid 11을 가진 프로세스를 기준으로 MoQ 스케줄링은 종료되고, unmonopolize가 호출되면서 global tick은 0으로 초기화된다. 이후 MLFQ 스케줄러로 돌아와서 남은 짝수 pid 프로세스들을 스케줄링 한다.

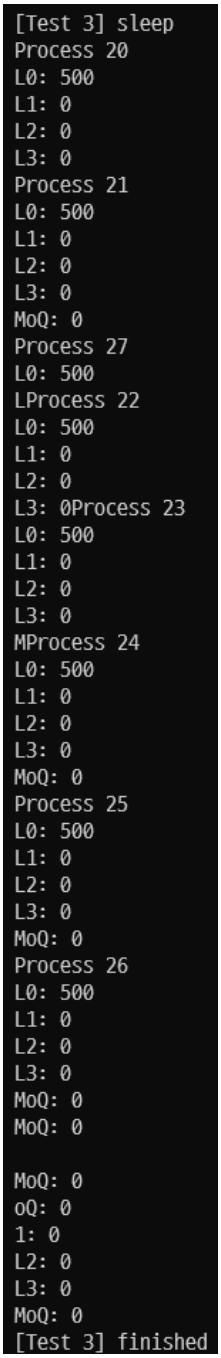
Explanation about test result

```
[Test 1] default
Process 4
L0: 5762
L1: 0
L2: 10304
L3: 83934
MoQ: 0
Process 5
L0: 9111
L1: 17046
L2: 0
L3: 73843
MoQ: 0
Process 6
L0: 15269
L1: 0
L2: 37123
L3: 47608
MoQ: 0
Process 7
L0: 15941
L1: 21986
L2: 0
L3: 62073
MoQ: 0
Process 8
L0: 19788
L1: 0
L2: 46645
L3: 33567
MoQ: 0
Process 9
L0: 20095
L1: 32497
L2: 0
L3: 47408
MoQ: 0
Process 10
L0: 18850
L1: 0
L2: 46095
L3: 35055
MoQ: 0
Process 11
L0: 21707
L1: 36697
L2: 0
L3: 41596
MoQ: 0
[Test 1] finished
```

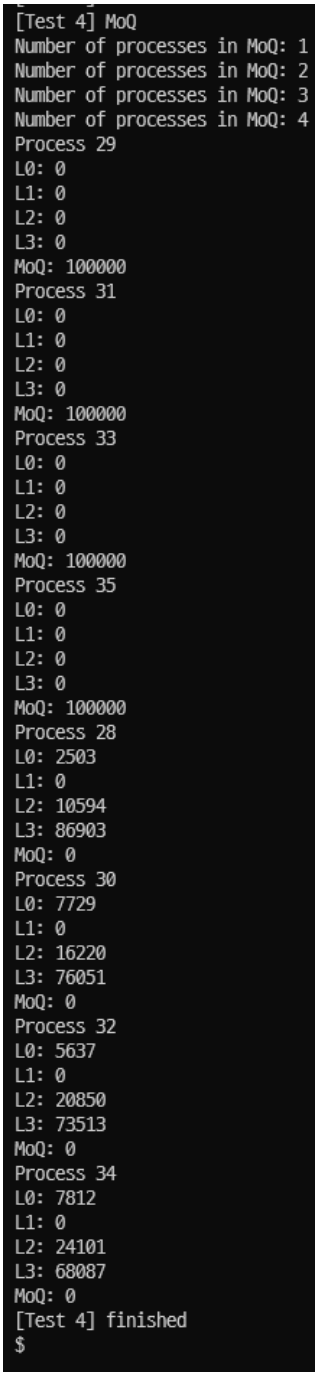
priority가 같을 때에는 프로세스 선택에 제약이 없으므로 모두가 priority 0이면 L3를 기준으로 FCFS 스케줄링이 된다. 하지만 명세를 만족하는 정상 결과이다. 이것을 극복하기 위해 priority boosting이 존재하는 것이고, 8개의 프로세스가 아닌 더 많은 프로세스가 동시에 동작하는 환경이라면 더더욱 문제는 아니다. 프로세스가 적다면 어차피 개개인에게 가는 자원이 많을 것이기에 앞선 프로세스가 좀 더 많은 자원을 차지해도 괜찮다.

```
[Test 2] priorities
Process 19
L0: 13048
L1: 20905
L2: 0
L3: 66047
MoQ: 0
Process 18
L0: 13750
L1: 0
L2: 37435
L3: 48815
MoQ: 0
Process 16
L0: 17025
L1: 0
L2: 45994
LProcess 17
L0: 16625
L1: 34421
L2: 0
L3: 48954
MoQ: 0
3: 36981
MoQ: 0
Process 14
L0: 18679
L1: 0
L2: 51414
L3: 29907
MoQ: 0
Process 15
L0: 18281
L1: 40099
L2: 0
L3: 41620
MoQ: 0
Process 12
L0: 16896
L1: 0
L2: 55021
L3: 28083
MoQ: 0
Process 13
L0: 11425
L1: 29706
L2: 0
L3: 58869
MoQ: 0
[Test 2] finished
```

priority 스케줄링이 정상적으로 구현되었기에 제시된 테스트 케이스 예시와 비슷한 결과가 나왔다. pid가 클수록 우선순위가 높으므로 pid가 큰 프로세스가 대체로 먼저 끝난다.



대부분의 시간을 L0에 머무르므로 작은 프로세스가 먼저 끝나게 되고, 거의 동시에 작업이 끝난다.



MoQ에서 홀수 pid 프로세스의 FCFS 스케줄링이 이루어지므로 모든 스케줄링은 MoQ에서만 이루어지고, pid가 작은 프로세스들이 먼저 끝난다. MoQ스케줄링이 끝나면 MLFQ 스케줄링으로 돌아와서 짝수 pid 프로세스들의 스케줄링이 실행된다. 결과가 정상적으로 잘 나온 모습이다.

Trouble shooting

setmonopoly 시스템 콜을 호출해 MoQ로 이동시켜도 monopolize가 호출되기 전까지는 MLFQ 스케줄링이 진행된다. 그러나 MoQ와 MLFQ는 같은 ptable을 공유하기 때문에 MLFQ 스케줄링 과정 중에 의도치 않게 MoQ의 프로세스들을 건들거나 영역을 침범하는 문제를 겪었었다. 이 때문에 MoQ의 프로세스가 자꾸 MLFQ로 이동하는 문제가 있었다.

이를 해결하기 위해 단순히 MLFQ에서는 level 99인 프로세스들을 무시하는 방향으로 접근하였고, 다행히도 문제를 해결할 수 있었다.

프로세스가 종료되었다고 해도 그것을 mlfq나 moq로부터 명시적으로 pop해주지 않는다면 여전히 자리를 차지하고 있게 된다. 이때문에 unmonopolize가 호출되지 않는 문제를 겪었었다.

pushmlfq와 pushmoq에서 단순히 큐의 자리가 빈 것 뿐만 아니라 그 상태가 UNUSED인지도 확인하면서 새로운 값을 할당하게 되면서 죽은 프로세스가 큐를 차지하고 있는 문제를 해결할 수 있었다.

또, isemptymoq 함수에서 단순히 비어있는 것 뿐만 아니라 UNUSED와 ZOMBIE 상태까지 같이 조건에 포함해줬더니 unmonopolized가 정상적으로 호출되면서 문제가 해결되었다.

sleep시 프로세스들이 L0에 머무르지 않는 문제가 있었다. 이를 해결하기 위해 sleep시 프로세스들을 L0로 옮겨주는 코드를 추가하였다. sleep을 한다면 타임 퀀텀을 온전히 소모하지 못한 것이므로 L0로 옮겨주는 것이 개념적으로 옳다고 생각해서 이런 식으로 문제를 해결하였다.

MLFQ와 MoQ 스케줄링을 하는데 MLFQ의 가장 바깥쪽 for문이 끝나서 자꾸 MoQ의 영역을 침범하는 문제가 있었다. 33번 까지는 잘 실행되다가 35번부터 갑자기 mlfq와 moq가 번걸아가면서 실행됐었다.

이를 해결하기 위해 각 for문의 마지막에 p를 다시 ptable.proc으로 초기화해주면서 무한 루프를 만들어 각자의 영역에서 벗어나지 못하게 하여 문제를 해결할 수 있었다.