Project02

2020055350 강현중

1. Design

큐 자료구조를 사용하지 않고 구현하기 위해 proc.h의 proc 구조체를 수정하여, 레벨, monopolize 상태 여부, 실행된 틱, 우선순위를 저장할 것이다. 이러면 큐를 사용하지 않고, ptable을 순회하며, 레벨에 맞는 프로세스를 선택할 수 있다.

만약 프로세스의 수가 너무 많으면 안 되겠지만 xv6의 프로세스는 NPROC=64개기 때문에 L3 큐까지 전부 순회 한다고 해도 300회가 넘지 않기 때문에 가능할 것이다.

큐 간의 이동은 트랩에서 각 큐에 할당된 시간을 넘기면 프로세스의 레벨을 상승시키는 형태로 구현 할 것이다.

MoQ도 큐를 사용하지 않고, MoQ에 프로세스를 넣을 때마다 레벨 99부터 순서대로 증가하도록 구현할 것이다. 이러면 레벨이 99 이상인 프로세스 중 레벨이 가장 낮은 프로세스를 선택하면 되기에 큐를 사용하지 않고 FCFS를 만들 수 있다. 또 지금이 MoQ를 사용하는 상태인지를 알려주는 변수를 프로세스와 proc.c에 하나씩 두어, MoQ를 사용 중일 때는 트랩에서 타이머도 흐르지 않고, cpu를 넘기지도 않아 명세의 조건에 만족하도록 한다.

Priority boosting은 Starvation을 방지하기 위해 트랩에서 100틱이 지나면 proc.c에 정의된 priorityboost 함수를 호출하며, MoQ를 제외한 즉 99레벨 미만의 모든 프로세스의 레벨을 0으로 초기화 한다.

스케줄러에서는 MoQ를 사용중일 때가 아니면 항상 L0부터 시작하여 Runnable한 프로세스를 찾으면 실행한다. 다만 L3에서는 우선순위가 가장 높은 프로세스를 실행해야하기 때문에, ptable을 한바퀴 돌고 프로세스를 선택하여 실행한다.

MoQ를 사용 중일 때에는 처음에 MoQ의 크기가 0인지 확인하여 0이면 즉시 unmonopolize 해 실행흐름이 MLFQ로 넘어가도록 한다.

2. Implement

1) proc 구조체 수정

```
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
  enum procstate state;
  int pid;
  struct proc *parent;
  struct trapframe *tf;
  struct context *context;
  void *chan;
  int killed;
  struct file *ofile[NOFILE];
  struct inode *cwd;
  char name[16];
  int priority;
  int level;
  int ticks;
  int ismo;
```

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->level = 0;
  p->priority = 0;
  p->ticks=0;
  p->ismo=0;
```

proc 구조체와 allocproc

proc.h의 proc 구조체를 수정하여 프로세스가 있는 큐의 레벨, L3에서 사용할 우선순위, 실행한 tick, 현재 프로세서가 MoQ를 사용하는지에 대한 변수를 만들고 proc.c의 allocproc 함수에서 초기 값을 설정해 주었다.

2) 작성한 시스템 콜과 함수

시스템 콜은 전부 userSys.c에 작성하였다. 또한 시스템콜 등록을 위해 defs.h Makefile,

syscall.h, syscall.c user.h usys.S에 필요한 정보를 작성하였다.

①yield

```
10 void
11 sys_yield(void)
12 {
13   yield();
14 }
15
```

yield 시스템 콜

proc.c에 있는 yield를 호출하여, 현재 프로세스에서 cpu 사용을 포기하도록 한다.

②getlev

```
16 int
17 sys_getlev(void)
18 {
19  if((myproc()->level) >=99)
20  return 99;
21  return myproc()->level;
22 }
```

getlev 시스템 콜

내 프로세스가 속해 있는 큐의 레벨을 반환한다. 레벨이 99 이상이면 MoQ의 멤버이므로 99를 반환하고, 나머지 경우에는 자신의 레벨을 반환한다.

③setpriority

```
int
sys_setpriority(void)
{
  int a,b;
  if((argint(0,&a)<0)||(argint(1,&b)<0))
    return -1;
  return setpriority(a,b);
}
```

```
int
setpriority(int pid, int priority)
{
  if(priority<0 || 10<priority)
    return -2;
  acquire(&ptable.lock);
  struct proc* p;
  for(p=ptable.proc; p<&ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->priority = priority;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock); // didn't match pid
  return -1;
}
```

setpriority 시스템 콜

ptable에 접근하는 함수들은 전부 proc.c에 작성하였다. 시스템 콜에서는 pid와 우선순위를 인자로 받아 setpriority로 넘겨주고 setpriority에서는 pid에 해당하는 프로세스를 찾아 우선순위를 설정한다. 만약 우선순위의 범위가 잘못되면 -2, pid에 해당하는 프로세스가 없으면 -1을 반환한다.

④setmonopoly

```
int
sys_setmonopoly(void)
{
  int a,b;
  if((argint(0,&a)<0)||(argint(1,&b)<0))
    return -1;
  return setmonopoly(a,b);
}
```

```
int
setmonopoly(int pid, int password)
{
  int som;
  if(password != 2020055350){
    return -2;
  }
  struct proc* p;
  acquire(&ptable.lock);
  for(p=ptable.proc; p< &ptable.proc[NPROC]; p++){
    if((p->pid) == pid){
      (p->level) = nextMoQ;
      nextMoQ++;
      release(&ptable.lock);
      som = SofMoQ();
      return som;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

setmonopoly 시스템 콜

```
int SofMoQ()
{
  int i = 0;
  struct proc* p;
  acquire(&ptable.lock);
  for(p=ptable.proc; p< &ptable.proc[NPROC]; p++){
        if(p->level >= 99 && ((p->state == RUNNABLE) || (p->state == RUNNING)||(p->state == SLEEPING)))
        i++;
  }
  release(&ptable.lock);
  return i;
}
```

MoQ의 크기 반환

특정 프로세스를 MoQ에 넣는다. FCFS 제공을 위해 들어올 때 마다, 레벨이 99에서 높아지게 설정하였다. 학번을 암호로 하여 암호가 틀렸을 경우 -2, 해당 pid를 가진 프로세스가 없으면 -1을 반환한다. 만약 MoQ에 넣는데 성공했을 경우 ptable을 순회해 레벨이 99이상인 종료되지 않은 프로세스의 개수를 세어 MoQ의 크기를 반환한다.

⑤monopolize 와 unmonopolize()

```
43 void
44 sys_monopolize()
45 {
46   monopolize();
47 }
48
49 void
50 sys_unmonopolize()
51 {
52   unmonopolize();
53 }
```

```
void monopolize()
{
  acquire(&ptable.lock);
  struct proc* p;
   for(p=ptable.proc; p<&ptable.proc[NPROC]; p++){
      p->ismo = 1;
    }
 release(&ptable.lock);
  ismono=1;
}

void unmonopolize()
{
  acquire(&ptable.lock);
  struct proc* p;
  for(p=ptable.proc; p<&ptable.proc[NPROC]; p++){
      p->ismo = 0;
    }
 release(&ptable.lock);
  ismono=0;
}
```

mono/unmonopolize

userSys.c에서는 proc의 (un)monopolize 함수를 호출한다. 각각의 함수에서는 전역변수인 ismono와 프로세스의 ismono를 변경해 준다. 프로세스나 전역변수 중 하나만 쓰지 않는 이유는 스케줄러에서 MoQ를 사용하는지 여부를 ptable을 순회 하기 전에 알고 싶어서 전역변수를 사용했고, 현재 돌고 있는 프로세스를 MoQ에 넣고, 바로 monopolize 할 때, 지금 상태가 MoQ를 사용한다는 것을 trap에 알려주기 위해 둘 다 사용했다.

⑥priority boost

```
411 void priority_boost()
412 {
413   struct proc *p;
414   acquire(&ptable.lock);
415   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
416     if(p->level < 99)
417     {
418          p->level = 0;
419     }
420   }
421   release(&ptable.lock);
422 }
```

priority boost

priority_boost 함수는 레벨이 99미만인 MLFQ에 있는 프로세스의 레벨을 0으로 초기화 해준다. 모든 레벨의 프로세스를 L0으로 낮춰주기에 Starvation을 방지하는 용도로 사용한다.

2) 스케줄러 구현

① MoQ 사용시

```
for(;;){
  sti();
  select = 0;
  lev = nextMoQ;
  pri = -1;
  if(ismono && (SRaMoQ() == 0)){
    unmonopolize();
    }
  acquire(&ptable.lock);
  q = ptable.proc;
  if(ismono){
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(((p->level) >= 99)&&((p->level)<lev)&&(p->state ==RUNNABLE)){
        q = p;
        select = 1;
        lev = q->level;
      }
    }
  }
```

MoQ 사용 시 스케줄러

먼저 monopolized가 호출되어 MoQ를 사용하는 경우이다. ismono는 p->ismo를 확인하지 않고 프로세서의 상태를 알기 위해 proc.c 내에 선언된 전역변수이다. 먼저 SRaMoQ 함수를 호출하여 현재 MoQ를 사용하고 있는데, 현재 MoQ의 프로세스가 없으면 unmonopolize 함수를 호출해 MLFQ로 돌아간다. 만약 MoQ에 프로세스가 있으면 ptable을 한바퀴 순회하며, 99레벨 이상 중 가장 레벨이 낮은 프로세스를 선택하는 방식으로 FCFS를 구현하였다. MoQ에 넣을 때 마다 레벨을 99부터 올려나가기 때문에 레벨이 가장 낮은 프로세스가 먼저 MoQ에 들어온 프로세스이다.

select는 조건에 맞는 프로세스가 있으면 이후 바깥 반복문을 빠져나가 context switch 전에 프로세스가 선택되었는지 확인하는 용도로 사용하는 flag이다.

② MLFQ 사용시

항상 0레벨부터 시작해 레벨이 가장 낮고, RUNNABLE한 프로세스가 있으면, 바로 선택하여 반복문을 빠져나온다. select는 조건에 맞는 프로세스가 있을 때, 바깥 반복문을 무시하고 빠져 나와 바로 context-switch로 넘어가는 데도 사용한다. 만약 레벨이 3일 경우에는 ptable을 한바퀴 순회하고 나서 우선순위가 가장 높은 프로세스를 선택한다.

```
for(int i =0; i<=3; i++) {
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(i <= 2){
      if((p->level == i) && (p->state == RUNNABLE)){
        q = p;
        select = 1;
        break;
      }
    }
    if(i == 3){
      if((p->level == i) && (p->state == RUNNABLE)&&(p->priority > pri)){
        pri = p->priority;
        q = p;
        select = 1;
      }
    }
  }
  if(select == 1){
   break;
   }
}
```

MLFQ 사용 시 스케줄러

③ context switch

```
  if(select == 1){
    c->proc = q;
    switchuvm(q);
    q->state = RUNNING;
    q->ticks = 0;
    swtch(&(c->scheduler), q->context);
    switchkvm();
    c->proc = 0;
  }
  release(&ptable.lock);
}
```

context switch

기존의 스케줄러와 마찬가지로 context switch를 통해 프로세스를 넘겨준다. 다만 기존의 스케줄러에서는 context switch가 ptable을 순회하는 반복문 안에 있었는데, 이는 다음에 스케줄러가 호출되었을 때, 스케줄러의 context가 유지되어 기존에 순회하던 곳부터 시작하여 round robin을 제공하기 위해 그랬던 것이고, 이번 프로젝트에서는 스케줄러가 호출되었을때 S항상 L0의 처음이나 MoQ의 처음부터 순회해야 하기 때문에 무한반복문 안쪽에서 context switch를 시행했다.

2) trap.c

① MoQ 처리

```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER){
   if(globalismo != (myproc()->ismo)){
     if(globalismo == 1 && (myproc()-> ismo)==0){
       globalticks = 0; // unmono
       if(myproc()->level>=99){
         yield();
         )
     }
     globalismo = (myproc()->ismo);
   }
   if(((globalismo==0) || (myproc()->level < 99))){
     myproc()->ticks++;
     globalticks++;
     }
```

Trap.c에서의 MoQ 처리

globalismo는 trap.c에서 정의되어 지금 어떤 큐를 쓰고 있는지 알기 위해 사용하는 변수이다. 만약 현재 프로세스와 globalismo가 바뀌었다는 것은 상황이 바뀌었다는 것을 의미하기에 globalismono를 최신화 해준다.

또 globalismo가 0으로 바뀌었다는 것은 unmonopolize가 발생했다는 뜻이기에 globaltick을 0으로 초기화 해준다. 만약 이 때, MoQ의 프로세스가 돌고 있다면, 즉시 yield 하도록 했다.

현재 globalismo가 1이고 내 프로세스가 MoQ의 프로세스일 때는 틱이 멈추어 yield 되지 않고 cpu를 계속 점유할 수 있도록 한다.

②priority boosting

```
if(globalticks >=100)
{
  priority_boost();
  globalticks = 0;
}
```

priority boosting

global tick은 trap.c 내에 정의 되어 있고 mlfq를 사용 중일 때, timer interrupt가 발생될 때마다 증가한다. 100이상일 때 priority_boost를 호출하여 MoQ를 제외한 프로세스를 L0으로 초기화 하고 global tick을 초기화 한다.

③ process level 조정과 yield

```
if((myproc()->ticks)>=((myproc()->level)*2+2)){
  if((myproc()->level) == 3){
    if(myproc()->priority>0){
      myproc()->priority--;
      }
      myproc()->ticks = 0;
  }
  if( (myproc()->level) ==0){
      myproc()->level = 1+(myproc()->pid+1)%2;
      myproc()->ticks = 0;
  }
  else{
    myproc()->level = 3;
    myproc()->ticks = 0;
  }
    yield();
}
```

process level 조정과 yield

만약 프로세스가 진행된 시간이 (레벨*2+2)틱이 지났다면 그에 맞는 재조정을 해준다. 프로세스 레벨이 0이면 pid에 따라 홀수는 L1로 짝수는 L2로 옮겨준다.

프로세스 레벨이 1,2라면 L3로 옮겨주고, 레벨이 3이라면, 우선순위가 0이 되기 전까지 1씩 줄여 준다. 이 때, 지금까지 프로세스가 실행 된 시간을 초기화 해 준다. 이 레벨과 우선순위 조정을 통해 MLFQ와 L3에서의 priority Queue를 제공 할 수 있다.

3.Result

```
vectors.o\
vm.o\
prac_syscall.o\
getgpid.o\
userSys.o\
_wc\
_zombie\
_my_userapp\
_project01\
_project02\
_mlfq_test\
```

```
EXTRA=\
        mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
        ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
        printf.c umalloc.c my_userapp.c project01.c project02.c mlfq_test.c\
        README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
        .gdbinit.tmpl gdbutil\
```

Makefile

Makefile에 테스트를 위한 mlfq_test 유저 프로그램과 시스템 콜을 작성한 userSys.o를 추가하였다. mlfq_test.c는 제공된 파일을 사용하였다.

```
make clean
make CPUS=1
make fs.img CPUS=1
./bootxv6.sh
mlfq_test
```

컴파일 및 실행

1)Test 1

```
[Test 1] default
Process 4
L0: 2155
L1: 0
L2: 13612
L3: 84233
MoQ: 0
Process 5
L0: 4538
L1: 19114
L2: 0
L3: 76348
MoQ: 0
Process 6
L0: 9366
L1: 0
L2: 33169
L3: 57465
MoQ: 0
Process 7
L0: 13673
L1: 23771
L2: 0
L3: 62556
Process 8
L0: 13185
L1: 0
L2: 41173
L3: 45642
MoQ: 0
Process 9
L0: 12617
L1: 31612
L2: 0
L3: 55771
MoQ: 0
Process 10
L0: 15187
L1: 0
L2: 47825
L3: 36988
MoQ: 0
Process 11
L0: 15389
L1: 35176
L2: 0
L3: 49435
```

Test 1 수행 결과

```
[Test 1] default
Process 5
L0: 1907
L1: 1593
L2: 0
L3: 0
MoQ: 0
Process 7
L0: 2091
L1: 1409
L2: 0
L3: 0
MoQ: 0
Process 9
L0: 2298
L1: 1202
L2: 0
L3: 0
MoQ: 0
Process 11
L0: 2192
L1: 1308
L2: 0
L3: 0
Process 4
L0: 1879
L1: 0
L2: 1621
L3: 0
MoQ: 0
Process 6
L0: 2260
L1: 0
L2: 1240
L3: 0
MoQ: 0
Process 8
L0: 2168
L1: 0
L2: 1332
L3: 0
MoQ: 0
Process 10
L0: 2248
L1: 0
L2: 1252
L3: 0
```

Test 1 (loop 3500)

Test 1은 프로세스를 전부 fork 하고 mlfq로 스케줄링 하는 과정을 테스트 한다. 테스트 결과 pid가 낮은 프로세스부터 먼저 종료 되었다.

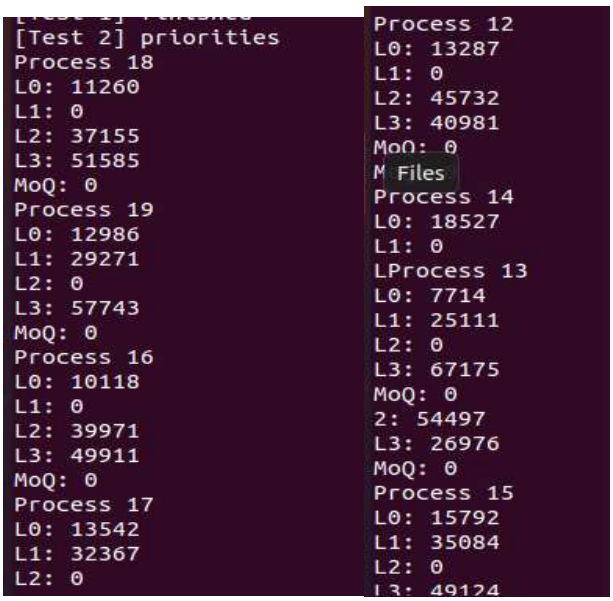
이는 L1과 L2에서 L3큐로 넘어가고 나면 L3 큐에서는 우선순위가 0으로 같은데, 명세의 조건에서 우선순위가 같은 프로세스 중에서는 어느 것을 스케줄링 해도 되기에 pid가 낮은 프로세스가 먼저 스케줄링 되도록 했기 때문이다.

또 global tick이 100틱이 되면 priority boost가 되는데, 프로그램을 처음 실행하고 L3큐에 모든 프로세스가 모이게 되면 60틱 정도인데, 이러면 L3에서 pid가 높은 프로세스가 스케줄링 되기 전에 priority boost가 되어 L0로 돌아가 Round-Robin 과정으로 수행되기 때문에, pid가 낮은 프로세스가 먼저 종료되었다.

만약 큐를 사용해서 L3에 먼저 들어온 프로세스가 먼저 스케줄링 되도록 설계 했다면, pid가 홀수인 프로세스가 먼저 종료 되었을 것이다.

반복 횟수를 3500으로 줄여 L1, L2 큐에서만 스케줄링 되게 했을 때는 pid가 홀수인 L1큐의 프로세스가 먼저 종료되는 것을 확인 할 수 있다.

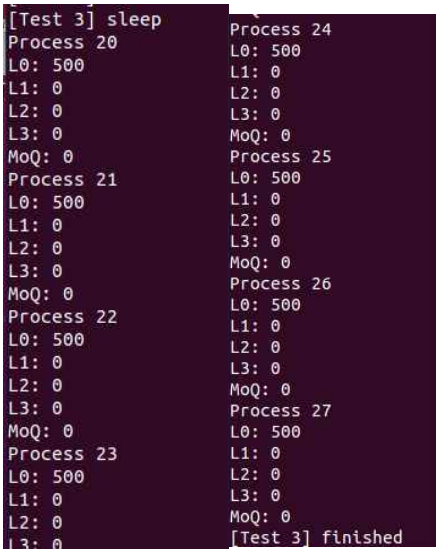
2) Test 2



Test2 수행 결과

pid가 큰 프로세스가 L3 큐에서 우선 순위가 높다. 실행 결과에서도 대체로 pid가 높은 프로세스가 먼저 종료 됨을 확인 할 수 있다. 대신 L0, L1, L2 큐에서 pid가 낮은 프로세스가 먼저 종료되는 경우가 있기에 정확히 pid가 큰 프로세스부터 먼저 종료되지는 않았다.

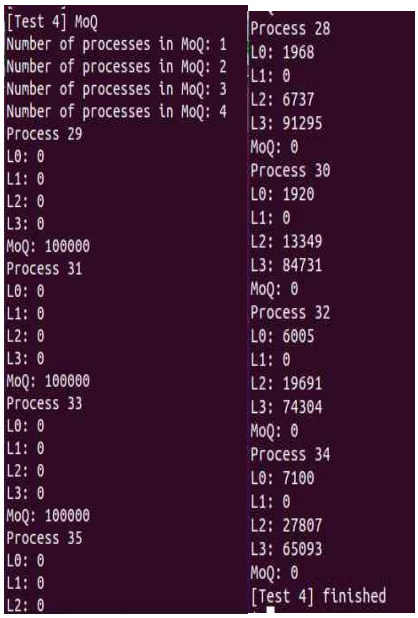
3) Test 3



Test 3 수행 결과

Test 3에서는 루프를 돌 때마다 sleep을 호출해, 프로세스는 보통 L0 큐에서 수행된다. 실행 결과에서도 모두 L0 큐에서 프로세스가 수행되었음을 알 수 있다. L0 큐에서는 pid가 낮은 프로세스가 먼저 스케줄링 되기에 pid가 낮은 프로세스가 먼저 종료되었다.

4) test 4



Test 4 수행 결과

Test 4에서는 pid가 홀수인 프로세스를 MoQ에 넣고 monopolize() 시스템 콜을 호출한다. MoQ에서는 FCFS를 제공하기에 MoQ에 먼저 들어온, p->level이 가장 낮은 프로세스가 먼저 스케줄링 된다.

따라서 프로세스가 홀수인 프로세스들부터 pid가 낮은 순서대로 스케줄링 되고, 이후 MoQ의 프로세스가 전부 종료되면 unmonopolize 되어 MLFQ로 돌아가 pid가 짝수인 프로세스들이 pid가 작은 순서대로 스케줄링 되어 종료된다.

4.Trouble shooting

1) 시스템 콜 만들 때, ptable에 접근하려고 했는데, userSys에서 바로 접근하기도 무섭고, 접근이 안 되어, ptable을 Extern으로 만들까 고민했었는데, proc.c에 필요한 함수를 정의해 두고 이를 호출하는 방식으로 안전하게 처리했습니다.

2) 메일로 질문도 보냈었는데 현재 스케줄러의 상태를 CPU구조체에 저장을 하고 싶었는데, 어디서 초기화 되는지도 모르고, 안전하게 하려고 proc.c와 proc구조체에 각각 현재 상태를 저장하는 변수(ismo, ismono)를 만들어 처리했습니다.

3) 항상 ptable에 접근할 때, Lock을 중복해서 걸기도 하고, release하지 않았는데 lock을 다시 걸고 하는 경우가 있어 에러를 많이 경험했습니다. 이후 ptable에 접근하는 함수에서는 항상 처음과 끝에 lock을 걸고 풀었으며, 이를 호출하기 직전에 lock을 풀고, 끝나면 다시 거는 형식으로 코딩 했습니다.

4) 과제를 수행하다가 VM를 켜놓고 이틀 정도 방치했는데, xv6를 켜니 알 수 없는 오류가 생겨 시스템 콜에서 잘못 건드린줄 알고 해결하는데 시간이 걸렸습니다. VM을 껐다 켜니 해결된 것으로 보아, 너무 오래 켜둬서 할당된 메모리에 문제가 생긴 것 같습니다.

5) 큐를 만들지 않고 명세를 해결하기 위해, 많은 시행착오가 있었습니다. MoQ에서 FCFS를 제공할 때, 생성된 순으로 하는줄 알고 pid순으로 하려는 안일한 생각으로 과제를 작성하였는데, 명세를 다시 읽고, 큐에 들어온 순서대로 실행해야 한다는 사실을 알게 되었습니다. 결국 큐를 만들어야 하나 고민하였는데, 99부터 큐에 들어온 순서대로 레벨을 높여나가는 방식으로 해결했습니다.

6) MLFQ를 만들 때, 스케줄러가 기존의 context를 유지한다는 생각을 하지 못하고, 당연히 처음부터 시행 할 것이라는 생각을 했습니다. 이를 처리하지 않으면, 다음에 스케줄러가 호출되었을 때, 기존에 순회하던 큐부터 시작하기 때문에, 항상 L0나 MoQ의 처음부터 시작해야 한다는 조건에 맞지 않아, 코드를 수정했습니다.

7) mlfq_test를 수행할 때, fork 후에 sleep(10)을 호출하는데, MoQ에 있는 프로세스가 스케줄링 되고 바로 sleep을 호출하니 MoQ에 있는 프로세스가 없다고 파악해서 바로 unmonopolize 되어 MoQ의 프로세스가 실행되지 않는 결과가 있었습니다. 따라서 RUNNABLE과 SLEEPING 상태인 프로세스의 수를 세는 함수를 만들어 해결했습니다.

8) test1에서 왜 pid가 작은 프로세스가 먼저 종료될까 고민했는데, MoQ에서 우선순위가 같을 때, pid가 작은 프로세스가 먼저 스케줄링 되고, global tick이 100이 될 때마다 priority boosting이 되어, pid가 작은 프로세스가 먼저 종료됨을 알 수 있었습니다. 명세의 조건에서 우선순위가 같을 때의 스케줄링은 상관없었기 때문에, 큐를 사용하지 않고 정상적으로 스케줄링 했습니다.