Operating System Project 2 Wiki

2019073309 윤무원

Design

MLFQ 구현 방식 결정

가장 먼저 고민한 것은 MLFQ를 어떠한 형태로 구현할 것인지에 대한 방향성에 대해 고민하였다. 기존 xv6가 작동되는 구조를 최대한 적게 수정하면서 기능을 구현하는 것이 효율적이라고 생각, 이러한 방향으로 방식을 결정하였다.

기존 xv6의 작동방식은 전체 프로세스가 `ptable.proc` 이라는 배열에 저장되어 생성을 할때는 배열 내에서 상태가 UNUSED인 프로세스를 하나 골라서 초기화하여 프로세스를 생성. 종료할때도 배열 내에서 ZOMBIE 프로세스를 찾아서 부모프로세스가 이를 확인하면 데이터를 삭제하고 종료시키는 방식. 나는 이러한 과정을 최대한 유지하면서 MLFQ를 구현하기 위해 아래와 같은 단계를 수행하였다.

> 💡 **MLFQ 구현 단계**
>
> 1. 프로세스 구조체의 포인터 변수를 담는 MLFQ를 구현한다.
> 2. `ptable.proc`에서 프로세스의 생성과 종료가 발생하면 그 내용을 즉시 MLFQ에 반영한다.
> 3. 스케쥴링은 오로지 MLFQ에서만 이루어지게 한다.

위의 과정을 통해 MLFQ를 구현할 시 기존 xv6의 프로세스 생성, 종료 방식에 전혀 개입하지 않고 그저 ptable에서 생성된 프로세스를 MLFQ에 저장하고 종료되기 직전에 MLFQ에서 pop하는 방식으로 프로세스 관리가 용이해지는 장점이 있기에 이러한 방법으로 구현하기로 결정하였다.

proc 구조체 변경

MLFQ 기능을 구현하기 위해 몇가지 정보가 프로세스에 담겨 있어야 하므로 proc 구조체를 수정하였다.

```
//proc.h
struct proc{
    uint sz;
    ...
    int priority;
    int tq;
    int q_level;
    int moq_flag;
};
```

L3큐의 우선순위 스케줄링을 위해 해당 프로세스의 우선순위를 나타내는 `int priority`, 프로세스가 속한 큐의 time quantum를 다 소비하면 하위 큐로 이동해야 하기에 프로세스가 현재 속한 큐에서 얼만큼의 시간동안 수행되었는지를 나타내는 `int tq`, 현재 프로세스가 속한 큐 레벨을 나타내는 `int q_level`, 프로세스가 MoQ에 속해있는 상태인지 나타내는 `int moq_flag`를 추가하였다.

MLFQ 구조체 구성

실제로 프로세스들이 해당하는 레벨의 큐에 저장되어야 할 공간이 필요하기에 mlfq 라는 구조체를 생성해주었다.

```
//proc.c
struct {
    struct proc *queue[NPROC];
    int index, amount;
} mlfq[5];

struct spinlock mlfq_lock;
```

앞서 MLFQ 구현 방식 결정 파트에서 설명했다시피 큐에 프로세스 구조체의 포인터를 담았다. 그 이유는 나는 앞으로 MLFQ에서는 스케쥴링만 이루어지게 하고 프로세스의 정보를 수정하거나 생성, 종료를 하는 동작은 기존의 방식대로 ptable에서 이루어지게 할 것인데 ptable에서 정보 수정이 일어나면 그 사실이 즉각 mlfq에 있는 똑같은 프로세스에도 반영이 되어야 하므로 프로세스의 메모리 주소값을 큐에 저장하는 식으로 이를 해결하였다.

구성으로는 프로세스들이 저장될 배열인 `proc *queue`를 만들고 스케쥴링 구현의 편의성을 위해 두 가지 정보를 추가하였다. `int index`는 해당 레벨의 큐에서 마지막으로 스케쥴링의 대상이 되었던(**실제로 스케쥴링이 되었는지 여부를 떠나서**) 프로세스의 다음 index를 나타

낸다. `int amount`는 큐에 저장되어있는 프로세스의 갯수를 나타낸다. L0~L3, MoQ 까지 총 5개의 큐가 필요하기에 `mlfq[5]`로 선언하였다.

추가로 ptable과 마찬가지로 여러 프로세스가 동시에 mlfq에 접근할 시 잘못된 정보를 받아올 수 있으므로 동기화를 위한 `struct spinlock mlfq_lock`을 따로 만들어주었다.

process 스케쥴링 방식 결정

프로세스 스케쥴링은 총 3가지의 방식을 구현해야했다. L0~L2에서 사용될 round robin 스케줄링, L3에서 사용될 priority 스케줄링, MoQ에서 사용될 cpu 독점 후 FCFS 스케줄링. 나는 이 3가지의 방식을 구현하기 위해 아래와 같은 단계를 수행하였다.

> 💡 **스케쥴링 단계**
>
> 1. 현재 존재하는 프로세스 중 RUNNABLE한 프로세스가 하나 이상 존재하면 스케쥴링을 시작한다. 여기서 사용자가 monopolize를 실행하지 않았는데 RUNNABEL한 프로세스가 MoQ에만 존재한다면 스케쥴링을 하지않고 새로운 RUNNABLE한 프로세스가 생길때까지 대기한다.
> 2. L0~L2에서 RUNNABLE한 프로세스가 존재하는 가장 상위의 큐를 선택하고 앞서 서술한 index 정보를 활용하여 해당 큐에서 스케줄링 될 프로세스를 선택한다.
> 3. L2까지 RUNNABLE한 프로세스가 없다면 L3에서 탐색한다. 큐 전체를 한 번 순회하면서 우선순위가 가장 높은 프로세스 중 맨 앞에 위치한 프로세스를 선택한다.
> 4. 사용자가 monopolize를 호출하였을시 MoQ에서만 스케줄링을 한다.

큰 줄기는 이러한 방향성을 잡고 구현하였으며 이를 위해 몇 가지 변수를 추가로 선언하였다. 이는 이후 Implement에서 서술하겠다.

time interrupt 수정

time interrupt가 발생되면 trap.c의 interrupt handler가 실행되어 관련 처리를 해준다. time interrupt handler의 주요 동작은 xv6 자체의 global ticks인 ticks를 1 증가시켜주고 yield함수를 호출한다. yield 함수는 현재 프로세스의 상태를 RUNNABLE로 바꾸고 스케쥴링을 호출, 즉 현재 프로세스가 잡고 있는 CPU를 놓게 만드는 함수이다. 각각의 프로세스는 스케쥴링이 되고 1틱동안 수행을 하면 CPU를 내려놓고 time quantum과 큐 레벨이 수정되어야 하므로 이와 관련된 동작을 yield 함수가 실행되기 직전에 해주었다.

MoQ 구현 방식 구체화

MoQ에 속한 프로세스는 평상시엔 스케쥴링이 되지 않다가 사용자가 monopolize를 호출할시 cpu를 독점하여 MoQ에 속한 프로세스가 모두 끝날때까지 MoQ 내에서만 스케쥴링 되어야한다. 또한 FCFS 정책을 사용하기에 프로세스는 자신이 종료될때까지 계속 일을 수행해야한다. 이 말은 즉 time interrupt를 허용하지 않겠다는 말이다. 나는 이를 어떻게 구현해야되나 고민하던 중 한 가지 방법을 생각해냈다. 일단 monopolize가 실행되면 time interrupt는 기존처럼 계속 발생하게 유지하되 **다시 스케쥴링이 일어날때 동일한 프로세스가 스케쥴링 되도록 로직을 구현했다**. 이를 해당 프로세스가 끝날때까지 반복, 이후엔 MoQ 내의 다음 프로세스가 스케쥴링. 이 동작을 MoQ내의 프로세스가 모두 종료될때까지 반복하였다. 이후 MoQ 내의 프로세스가 존재하지 않으면 자동으로 unmonopolize를 호출하게 하여 MoQ part에서 MLFQ part로 넘어오게 구현하였다.

Priority Boosting

priority boosting을 구현하기 위해선 global ticks가 100틱일때마다 0으로 수정을 해주어야한다. 이러한 동작을 xv6 자체의 global ticks인 ticks를 이용해서 하기에는 살짝 위험할 수 있다고 판단, 새로운 global ticks를 만들어주었다. 이 글로벌 틱은 프로세스 관리를 위한 정보이므로 ptable안에 추가해주었다.

```
//proc.c
struct {
    struct spinlock lock;
    struct proc proc[NPROC];
    int global_ticks;
} ptable;
```

스케쥴러 함수 내에서 context switching이 끝나면 global_ticks를 1 증가 시켜준 뒤 global_ticks가 100 이상이 되면 priority boosting을 실시하고 global_ticks를 0으로 설정해주었다. 이때 monopolize가 실행된 상태라면 global_ticks가 100이상이더라도 priority boosting을 실행하지 않게 구현하였다.

Implement

디자인 파트에서 내가 설계한 MLFQ, MoQ 및 스케쥴러 로직에 대해 설명하였다. 이를 xv6 기존 함수를 수정하고 새로운 함수를 구현함으로써 이 기능들을 완성시켰다. 차근차근 살펴보자.

proc.h

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

  int priority;
  int tq;
  int q_level;
  int moq_flag;
};
```

기존의 proc 구조체에 4개의 변수를 추가하였다.

`int priority` : L3 내에서 스케쥴링 할때 필요한 정보인 프로세스의 우선순위이다.

`int tq` : 현재 프로세스가 속한 큐에서 얼마만큼의 시간동안 수행되었는지에 대한 정보이다.

`int q_level` : 현재 프로세스가 속한 큐의 레벨을 나타내는 정보이다.

`int moq_flag` : 현재 프로세스가 MoQ에 속해 있는지에 대한 정보를 나타낸다. 0이면 MLFQ, 1이면 MoQ에 속해있다는 뜻이다.

proc.c

struct ptable

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
  int global_ticks;
} ptable;
```

`int global_ticks` : priority boosting을 위한 정보로 time interrupt가 발생할때마다 1씩 증가한다.

struct mlfq & struct spinlock mlfq_lock & int mono_mode

```
struct {
  struct proc *queue[NPROC];
  int index, amount;
} mlfq[5];

struct spinlock mlfq_lock;

int mono_mode;
```

`struct proc *queue[NPROC]` : 각 레벨의 큐에 존재하는 프로세스가 저장된 공간이다. proc 구조체의 포인터 변수를 저장한다.

`int index` : 해당 큐에서 마지막으로 스케쥴링된 프로세스의 다음 인덱스를 가리킨다.

`int amount` : 해당 큐에 속해 있는 프로세스의 갯수를 나타낸다.

`struct spinlock mlfq_lock` : 프로세스가 mlfq에 접근할때 동기화를 위해 사용하는 spinlock 변수이다.

`int mono_mode` : 현재 monopolize가 실행된 상태인지 나타내는 변수이다. 1이면 monopolize가 실행되서 MoQ 내의 프로세스가 스케쥴링 되고 있는 상태. 0이면 mlfq에서 프로세스가 스케쥴링 되고있는 상태이다.

isEmpty()

```
int isEmpty(int level){
  return (mlfq[level].amount == 0);
}
```

인자로 받은 level에 해당되는 큐가 비어있는지 확인하는 함수이다. 비어있으면 1을, 하나라도 있으면 0을 리턴한다.

enmlfq()

```
int enmlfq(int level, struct proc *p){
  if(level < 0 || level > 4){
    if(level != 4){
      cprintf("level range is 0~3.\n");
      return -1;
    }
  }

  p->tq = 0;
  p->q_level = level;
  if (level == 4)
    p->moq_flag = 1;
  mlfq[level].queue[mlfq[level].amount++] = p;
  return 0;

}
```

특정 level의 큐에 인자로 받은 프로세스를 저장하는 함수이다. 레벨의 범위에 맞지 않는 입력을 받으면 -1을 리턴한다. 특정 큐에 프로세스가 저장되는 경우는 time quantum을 모두 다 쓰고 하위 큐로 내려가거나 priority boosting이 일어난 경우에 발생하므로 해당 프로세스의 time quantum을 0으로 설정한다. 프로세스의 q_level 또한 바꾸어주고 만약 MoQ로 이동하는 경우 moq_flag를 0으로 설정한다.

demlfq()

```
int demlfq(struct proc *p){
  int level = p->q_level;

  if(isEmpty(level)){
    cprintf("Queue is Empty.\n");
    return -1;
  }

  int index; // dequeue position
```

```
  for(index = 0; index < mlfq[level].amount; index++){ // fin
    if(mlfq[level].queue[index] == p){
      if(index < mlfq[level].index)
        mlfq[level].index--;
      if(index == (mlfq[level].amount - 1) && index == mlfq[l
        mlfq[level].index = 0;
      mlfq[level].amount--;
      mlfq[level].queue[index] = 0;
      break;
    }
  }

  for(int i = index; i < mlfq[level].amount; i++){ // move el
    mlfq[level].queue[i] = mlfq[level].queue[i + 1];
  }

  mlfq[level].queue[mlfq[level].amount] = 0; // remove last e

  return 0;
}
```

특정 프로세스가 속한 큐에서 해당 프로세스를 pop하는 함수이다. 여기서 신경을 썼던 점이 바로 큐 중간에서 프로세스가 빠져나가는 경우를 어떻게 처리하냐는 것이었다. 내가 구현한 스케쥴링 방식은 mlfq 내의 index부터 시작해서 runnable한 프로세스를 찾을때까지 배열을 한 바퀴 도는 방식이었는데 만약 demlfq가 일어나 큐 중간에 위치한 프로세스가 빠져나가면 배열 중간에 빈공간이 발생해 프로세스를 찾는 과정에 문제가 발생한다.

나는 이를 방지하기 위해 큐에서 프로세스가 빠져나간 위치에서 오른쪽에 있는 모든 요소들을 한칸씩 왼쪽으로 당겨왔다. 이렇게 하면 어느 위치에서 dequeue가 일어나더라도 큐 중간에 빈공간이 발생하지 않으므로 스케쥴링에 문제가 생기지 않아 이러한 방식으로 함수를 구현하였다.

또한 기존 프로세스의 스케쥴링 순서를 보존하기 위해 index를 상황에 맞게 수정해주었다. 만약 지워진 프로세스의 위치가 index 보다 왼쪽에 위치해있으면 지워진 위치에서 오른쪽에 있는 요소들이 모두 왼쪽으로 이동하므로 index를 1 감소시켰다. 만약 index가 큐의 맨 끝 프로세스를 가르키고 있을때 demlfq를 통해 맨 끝 프로세스가 지워지면 index는 맨앞, 0을 가르키도록 수정해주었다.

모든 요소들의 위치이동이 끝나면 mlfq의 amount를 1 감소시키고 기존의 큐 맨끝에 담겨있는 공간은 0으로 비워주었다.

find_mlfq()

```
struct proc* find_mlfq(int level){ //find process to execute
  struct proc *p;
  int cnt = 0;

  for(;;){
    p = mlfq[level].queue[mlfq[level].index];

    if(cnt > mlfq[level].amount) // if cnt > amount, there is
      break;

    if(mlfq[level].index == mlfq[level].amount - 1){
      mlfq[level].index = 0;
      cnt++;
    }
    else{
      mlfq[level].index++;
      cnt++;
    }
    if(p->state != RUNNABLE)
      continue;

    return p;
  }

  return p;
}
```

특정 레벨의 큐에서 실행될 프로세스를 찾는 함수이다. mlfq 내의 index 변수가 다음에 스케쥴링 되어야 할 프로세스의 위치를 담고 있기에 해당 프로세스를 뽑아서 runnable이면 바로 리턴해주고 아니면 index를 1 증가시켜 그 다음 프로세스를 뽑아서 확인하는 방식으로 구현했다. 여기서 프로세스를 한번 검사할때마다 cnt 변수를 1 증가 시켜줬는데 이 cnt가 mlfq의 amount 보다 크면 큐를 한바퀴 다 돌아도 runnable한 프로세스가 존재하지 않은 것이므로 마지막으로 검사한 프로세스를 리턴해서 스케쥴러 함수 내에서 따로 처리를 해주었다.

priority_boosting()

```
int priority_boosting(void){
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->moq_flag == 1)
      continue;
    if(p->pid != 0){
      demlfq(p);
      enmlfq(0, p);
    }
  }
  for(int level = 0; level < 4; level++){
    mlfq[level].index = 0;
  }
  return 0;
}
```

global_ticks가 100 이상이 되었을때 priority를 boosting 해주는 함수이다. 여기선 굳이 mlfq에서 순회할 필요 없이 ptable에서 돌면서 pid가 0이 아닌(프로세스가 종료되면 pid가 0으로 초기화되므로) 프로세스들을 자신이 속한 큐에서 pop하고 L0 큐에 push 해주었다. 여기서 MoQ에 속한 프로세스는 부스팅이 일어나면 안 되므로 moq_flag가 1인 프로세스는 부스팅에서 제외하였다.

setpriority()

```
int setpriority(int pid, int priority){
  struct proc *p;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->tq=0;
      if(priority < 0 || priority > 10){
        release(&ptable.lock);
        return -2;
      }
      p->priority = priority;
```

```
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

특정 프로세스의 우선순위를 수정하는 함수이다. priority_boosting과 비슷하게 ptable을 순회하면서 해당 프로세스를 찾아 priority를 수정해주었다. 추가로 time quantum 또한 0으로 수정해주었는데, 이는 L3 내의 프로세스가 time quantum을 모두 사용한 경우 setpriority 함수를 통해 우선순위를 1 감소시켜주어야 하는데 이때 time quantum도 같이 0으로 초기화 해주기 위해서 해당 코드를 작성하였다. 리턴값은 과제 명세에 맞게 상황에 따라 다르게 리턴한다.

getlev()

```
int getlev(void){
  if(myproc()->moq_flag == 1)
    return 99;
  return myproc()->q_level;
}
```

프로세스가 속한 큐의 레벨을 반환하는 함수이다. 과제 명세에 맞게 MoQ에 속한 프로세스는 99를 반환한다.

monopolize()

```
void monopolize(void){
  mono_mode = 1;
}
```

MoQ 내의 프로세스가 cpu를 독점하여 사용하도록 하는 함수이다. cpu 독점은 이후 서술할 scheduler 함수 내에 구현되어있으므로 이 함수는 그저 MoQ part인지 나타내는 mono_mode 변수를 1로 설정해주는 동작만 한다.

unmonopolize()

```
void unmonopolize(void){
  mono_mode = 0;
  for(int i = 0; i < mlfq[4].amount; i++){
    mlfq[4].queue[i] = 0;
  }
  mlfq[4].amount = 0;
  mlfq[4].index = 0;
}
```

MoQ 내의 프로세스가 모두 종료되면 cpu 독점을 중지하고 mlfq part로 돌아가는 함수이다. mono_mode를 0으로 설정하고 MoQ 내의 큐를 모두 비워준다.

setmonopoly()

```
int setmonopoly(int pid, int password){
  struct proc *p;

  if(password == 2019073309){

    if(pid == myproc()->pid){
      return -4;
    }

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->pid == pid){
        if(p->moq_flag == 1){
          return -3;
        }
        else{
          acquire(&mlfq_lock);
          demlfq(p);
          enmlfq(4, p);
          release(&mlfq_lock);
          return mlfq[4].amount;
        }
      }
    }
    return -1;
```

```
  }

  return -2;
}
```

특정 프로세스를 MoQ로 이동하는 함수이다. 인자로 받은 password가 암호, 내 학번과 일치하면 해당 프로세스를 MoQ에 이동하고 MoQ의 크기를 반환한다. 과제의 명세에 맞게 존재하지 않는 pid일 경우, 자기자신을 MoQ로 옮기는 경우, 이미 MoQ에 존재하는 경우, 암호가 맞지 않은 경우에 대해 각각의 리턴값을 설정하였다.

clear_zombie()

```
void clear_zombie(void){

  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == ZOMBIE){

      if(p->moq_flag == 1){ // if p is in moq
        p->moq_flag = 0;
        mlfq[4].index++;
      }
      else{
        //remove from mlfq
        acquire(&mlfq_lock);

        demlfq(p);

        release(&mlfq_lock);
      }

      // Found one.
      kfree(p->kstack);
      p->kstack = 0;
      freevm(p->pgdir);
      p->pid = 0;
      p->parent = 0;
      p->name[0] = 0;
```

```
      p->killed = 0;
      p->state = UNUSED;
      p->priority = 0;
      p->q_level = 0;
      p->tq = 0;

    }
  }
}
```

현재 존재하는 모든 좀비 프로세스를 종료시키는 함수이다. 원래 좀비 프로세스는 wait 상태에 있는 부모 프로세스가 이를 확인하여 회수하는 과정을 통해 종료해야하지만 MoQ 스케줄링 이전에 임의로 좀비 프로세스를 처리할 필요가 있어 따로 함수를 구현하였다. 코드 자체는 wait() 함수 내에서 좀비 프로세스를 종료시키는 코드를 따와서 완성하였다.

allocproc()

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  //init mlfq information
  p->tq = 0;
  p->priority = 0;
  p->q_level = 0;
```

```
  p->moq_flag = 0;

  release(&ptable.lock);

  ...

  // enqueue to mlfq
  acquire(&mlfq_lock);

  enmlfq(0, p);

  release(&mlfq_lock);

  return p;
}
```

allocproc()을 통해 프로세스가 하나 생성될때는 mlfq 관련 변수들을 모두 0으로 초기화해주고 함수가 끝나기 전 enmlfq()를 통해 L0 큐에 생성된 프로세스를 저장하였다.

exit()

```
void
exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  if(curproc == initproc)
    panic("init exiting");

  ...

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  if(curproc->moq_flag == 1){ // if process is in moq
    curproc->moq_flag = 0;
    mlfq[4].index++;
  }
```

```
  sched();
  panic("zombie exit");
}
```

기존 exit()에서 exit를 호출한 프로세스가 MoQ에 속한 프로세스일시 moq_flag를 0으로 수정하고 MoQ의 index를 1 증가시켰다. 이 부분에 대한 자세한 설명은 이후에도 하겠지만 MoQ를 구현할때 문제점이 MoQ에 속한 프로세스가 일을 다 마치고 exit를 호출할시 좀비로 상태가 변하는데 이를 회수할 부모프로세스는 MoQ에 속해있지 않아 MoQ part가 끝날 때까지 스케쥴링이 대상이 되지 못한다. 그렇기에 좀비가 된 MoQ 프로세스를 제때 회수해주지 못해 panic : zombie exit가 발생한다. 이를 방지하기 위해 좀비 프로세스가 된 MoQ 내의 프로세스는 일단 좀비 상태로 두되 스케쥴링이 되지 않도록 로직을 만들어 이를 해결하였다.

wait()

```
int
wait(void)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();


  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){

        if(p->q_level != 4){
          //remove from mlfq
          acquire(&mlfq_lock);

          demlfq(p);
```

```
          release(&mlfq_lock);
        }


        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        p->priority = 0;
        p->q_level = 0;
        p->tq = 0;

        release(&ptable.lock);


        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

프로세스가 exit를 호출하여 좀비로 상태가 바뀌고 이를 wait 함수 내에서 sleep 중인 부모가 깨어나 이를 회수하여 최종적으로 프로세스를 종료시킨다. 종료시킬때 해당 프로세스가

속한 큐에서 pop하고 mlfq 관련 정보를 0으로 수정하였다.

scheduler()

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

  int l3_trigger = 1; // L3 flag
  int mono_flag = 0; // monopoly flag

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

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE || (p->moq_flag == 1 && mono_mo
        continue;

      if(mono_mode == 1){ // if monopolize mode, find process
        mono_flag++;
        if(mlfq[4].amount == mlfq[4].index){ // if monopolize
          unmonopolize();
          ptable.global_ticks = 0;
          mono_flag = 0;
        }
        else
          p = mlfq[4].queue[mlfq[4].index];
      }

      if(mono_mode == 0){
        acquire(&mlfq_lock);

        for(int level = 0; level < 3; level++){ //find proces
```

```
          if(isEmpty(level))
            continue;
          p = find_mlfq(level);
          if(p->state != RUNNABLE) // there is no runnable pr
            continue;
          l3_trigger = 0;
          break;
        }

        if(l3_trigger){ // if there is no runnable process in
          if(mlfq[3].amount > 0)
            p = mlfq[3].queue[0];
          for(int i = 0; i < mlfq[3].amount; i++){
            if(mlfq[3].queue[i]->state == RUNNABLE && mlfq[3]
          }
        }

        l3_trigger = 1;

        release(&mlfq_lock);
      }

      if(mono_flag == 1) // if first scheduling in monopoly,
        clear_zombie();




      //selection done

      // 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 ba
        c->proc = 0;

        // increase process time quantum, global ticks
        ptable.global_ticks++;


        acquire(&mlfq_lock);

        if(ptable.global_ticks >= 100 && mono_mode == 0){
          priority_boosting();
          ptable.global_ticks = 0;
        }

        release(&mlfq_lock);

      }
      release(&ptable.lock);

    }
  }
```

지금까지 구현한 함수를 사용하여 mlfq 스케쥴링을 하는 새로운 scheduler를 구현하였다.

`int l3_trigger` : L3에서 프로세스를 탐색해야하는지 나타내는 변수. 초기값은 1로 설정하고 L0~L2에서 runnable한 프로세스를 찾으면 0으로 설정하여 L3에서 탐색하지 않도록 구현하였다.

`int mono_flag` : monopolize가 실행되고 처음 스케쥴링 되는지 판단하는 변수이다. 초기값을 0으로 설정하고 monopolize가 실행된후에 스케쥴링이 한 번 실행될때마다 1씩 증가하였다.

scheduler는 크게 세부분으로 나뉜다. MoQ part, MLFQ part, priority boosting part. 나눠서 따로따로 살펴보자

```
//MoQ part

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

  int l3_trigger = 1; // L3 flag
  int mono_flag = 0; // monopoly flag

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

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE || (p->moq_flag == 1 && mono_mo
        continue;

      if(mono_mode == 1){ // if monopolize mode, find process
        mono_flag++;
        if(mlfq[4].amount == mlfq[4].index){ // if monopolize
          unmonopolize();
          ptable.global_ticks = 0;
          mono_flag = 0;
        }
        else
          p = mlfq[4].queue[mlfq[4].index];
      }

...
```

기존에 구현된 ptable을 도는 for문은 그대로 두고 for문 안에서 mlfq와 moq를 탐색해 새로운 p를 찾아 context switching을 해주었다. 일단 먼저 ptable을 순회하면서 runnable한 프로세스가 존재하면 for문 안으로 접근해야하는데 여기서 monopolize가 실행되지 않

았는데 MoQ에 속해있는 프로세스의 상태를 보고 runnable인 프로세스가 존재한다고 판단해 for 문 내로 접근하면 오류가 발생하기에 mono_mode가 0일때는 MoQ에 속한 프로세스의 상태는 무시하였다.

monopolize가 사용자에 의해 호출되면 mono_mode가 1로 설정되어 MoQ part가 시작된다. mono_flag를 1 증가시키고 MoQ의 index를 활용해 스케쥴링을 하는데 index는 초기에 0으로 설정되므로 MoQ의 맨 앞에서부터 스케쥴링이 시작된다. 이후 프로세스가 종료되면 앞서 서술했듯이 exit 함수 내에서 MoQ의 index를 1 증가해주었기에 큐의 다음칸에 위치한 프로세스가 또 다시 종료될때까지 스케쥴링 된다. 이렇게 함으로써 FCFS 정책을 갖는 MoQ를 완성하였다. 이 과정을 반복하다가 MoQ의 index가 amount와 같아지면 MoQ 내의 모든 프로세스가 종료된 것이므로 unmonopolize를 호출하고 global_ticks, mono_flag를 0으로 설정한다. 이렇게 모든 프로세스가 종료되면 자동으로 MLFQ part로 넘어오게 구현하였다.

```
//MLFQ part
if(mono_mode == 0){
        acquire(&mlfq_lock);

        for(int level = 0; level < 3; level++){ //find proces
          if(isEmpty(level))
            continue;
          p = find_mlfq(level);
          if(p->state != RUNNABLE) // there is no runnable pr
            continue;
          l3_trigger = 0;
          break;
        }

        if(l3_trigger){ // if there is no runnable process in
          if(mlfq[3].amount > 0)
            p = mlfq[3].queue[0];
          for(int i = 0; i < mlfq[3].amount; i++){
            if(mlfq[3].queue[i]->state == RUNNABLE && mlfq[3]
          }
        }

        l3_trigger = 1;
```

```
        release(&mlfq_lock);
      }

      if(mono_flag == 1) // if first scheduling in monopoly,
        clear_zombie();

...
```

mono_mode가 0이면, 즉 monopolize가 실행되지 않은 상태에선 mlfq에서 스케쥴링이 일어난다. mlfq에 접근하기 전 acquire을 통해 mlfq_lock을 걸어주고 L0부터 탐색한다. find_mlfq()를 통해 L0~L2에서 runnable한 프로세스를 하나 찾으면 l3_trigger를 0으로 바꿔준다. 찾지 못했으면 l3_trigger가 1이므로 if문을 만족, L3에서 프로세스를 탐색한다. L3에서 탐색할때는 큐의 맨 처음부터 순회하면서 우선순위가 높은 프로세스 중 가장 앞에 위치한 프로세스를 선택하였다. 이후 l3_trigger를 다시 1로 설정하고 mlfq_lock을 release 한다.

추가로 mono_flag가 1이면, 즉 monopolize가 실행되고 첫 스케쥴링이 일어날때는 clear_zombie()를 호출해 현재 존재하는 모든 좀비 프로세스를 회수하였다. 이러한 동작을 한 이유는 일단 MoQ가 실행되고 나서는 mlfq 내의 프로세스는 스케쥴링이 될 수 없는데 monopolize를 호출하자마자 exit를 호출하는 프로세스는 moq가 끝날때까지 회수가 되지 않으므로 이게 어떤 부작용을 불러올지 몰라 초기에 처리해주기 위함이다.

```
//priority boosting part

        //selection done

      // 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 ba
```

```
      c->proc = 0;

      // increase process time quantum, global ticks
      ptable.global_ticks++;


      acquire(&mlfq_lock);

      if(ptable.global_ticks >= 100 && mono_mode == 0){
        priority_boosting();
        ptable.global_ticks = 0;
      }

      release(&mlfq_lock);

    }
    release(&ptable.lock);

  }
}
```

위의 MoQ, MLFQ part를 통해 프로세스를 선택 후 context switching이 끝나면 global_ticks를 1 증가시켜주고 global_ticks가 100 이상이고 mono_mode가 0이면 priority_boosting()을 호출하였다. 부스팅이 끝나면 global_ticks를 0으로 초기화한다.

trap.c

```
if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER){
      if(myproc()->moq_flag == 0){
        myproc()->tq++;

        if(myproc()->tq >= myproc()->q_level * 2 + 2){
          if(myproc()->q_level == 0){
            demlfq(myproc());
            if(myproc()->pid % 2 == 1)
              enmlfq(1, myproc());
            else
```

```
            enmlfq(2, myproc());
          }
          else if(myproc()->q_level == 3){
            setpriority(myproc()->pid, myproc()->priority - 1
          }
          else{
            demlfq(myproc());
            enmlfq(3, myproc());
          }
        }
       }
       yield();
      }
```

time interrupt가 발생했을때 trap.c에서 yield()를 호출하기 전에 현재 프로세스의 time quantum을 증가시키고 해당 프로세스가 모든 time quantum을 사용하였으면 하위 레벨의 큐로 이동시켰다. L0에서 하위 레벨로 이동할땐 pid가 홀수인 프로세스는 L1, 짝수인 프로세스는 L2로 이동시키고 L1, L2에 있는 프로세스는 L3로 이동시켰다. L3에 있는 프로세스는 큐의 이동 없이 현재 우선수위를 1 감소시켰다.

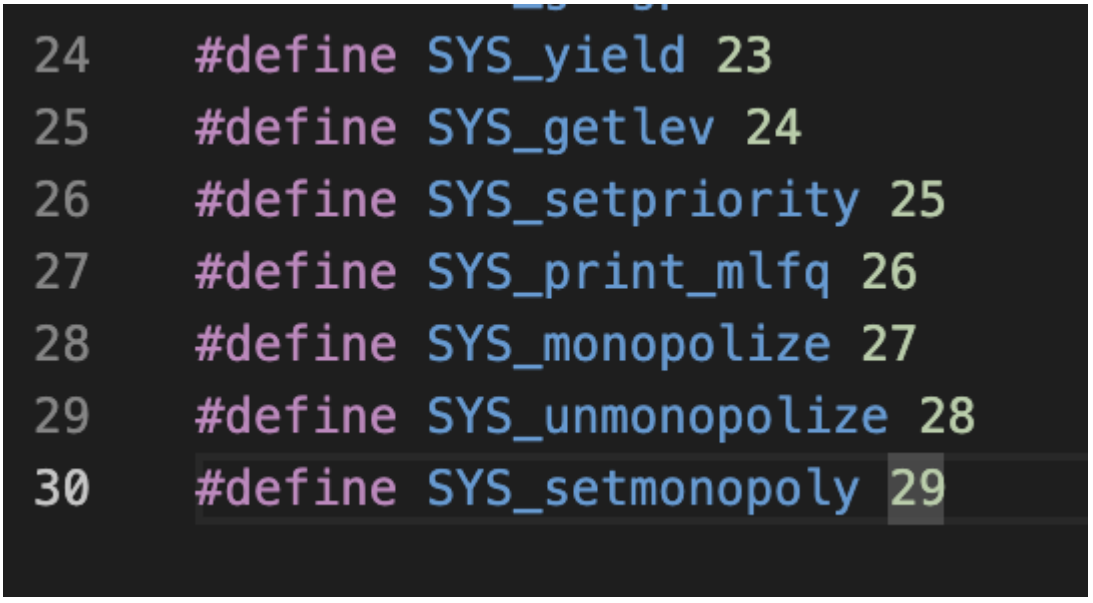
sysproc.c

```
void
sys_yield(void){
    yield();
    return;
}

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

int
sys_setpriority(void){
  int pid, priority;
  argint(0, &pid);
  argint(1, &priority);
```

```
  return setpriority(pid, priority);
}

void
sys_monopolize(void){
  monopolize();
}

void
sys_unmonopolize(void){
  unmonopolize();
}

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

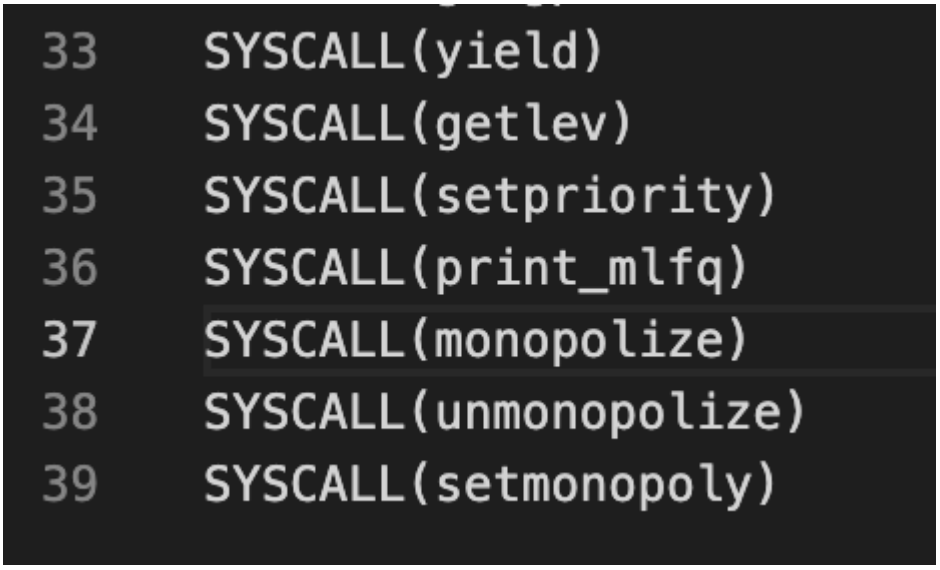
구현한 시스템 콜들을 유저프로그램에서 사용할 수 있게 sysproc.c에 wrapper function을 구현하였다.

system call settings



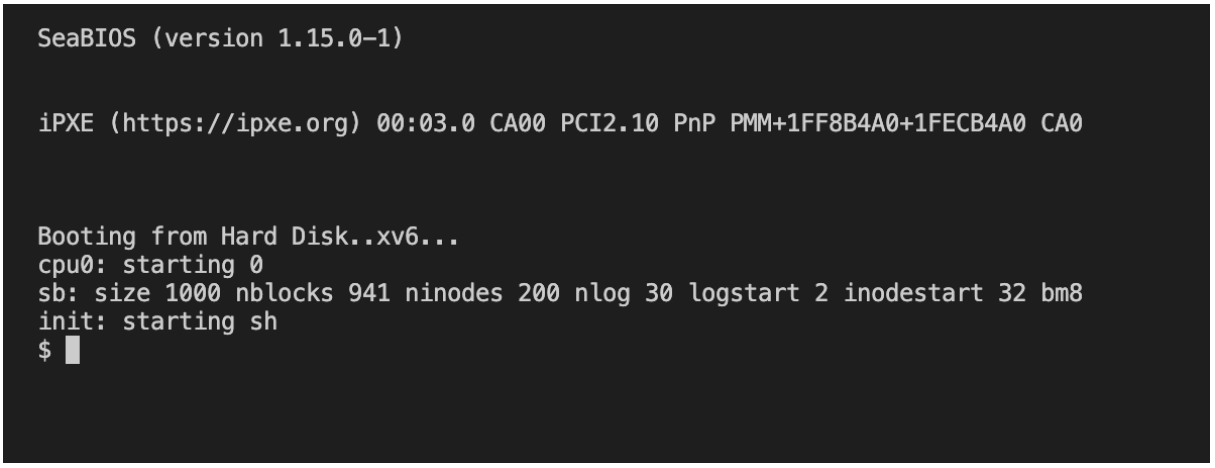
```
[SYS_yield] sys_yield,
[SYS_getlev] sys_getlev,
[SYS_setpriority] sys_setpriority,
[SYS_print_mlfq] sys_print_mlfq,
[SYS_monopolize] sys_monopolize,
[SYS_unmonopolize] sys_unmonopolize,
[SYS_setmonopoly] sys_setmonopoly,
};
```

```
27    void yield(void);
28    int getlev(void);
29    int setpriority(int, int);
30    void print_mlfq(void);
31    void monopolize(void);
32    void unmonopolize(void);
33    int setmonopoly(int, int);
```



시스템 콜 관련 세팅들을 모두 다 해줌으로써 유저프로그램에서 이 함수들을 사용할 수 있게 세팅을 완료하였다.

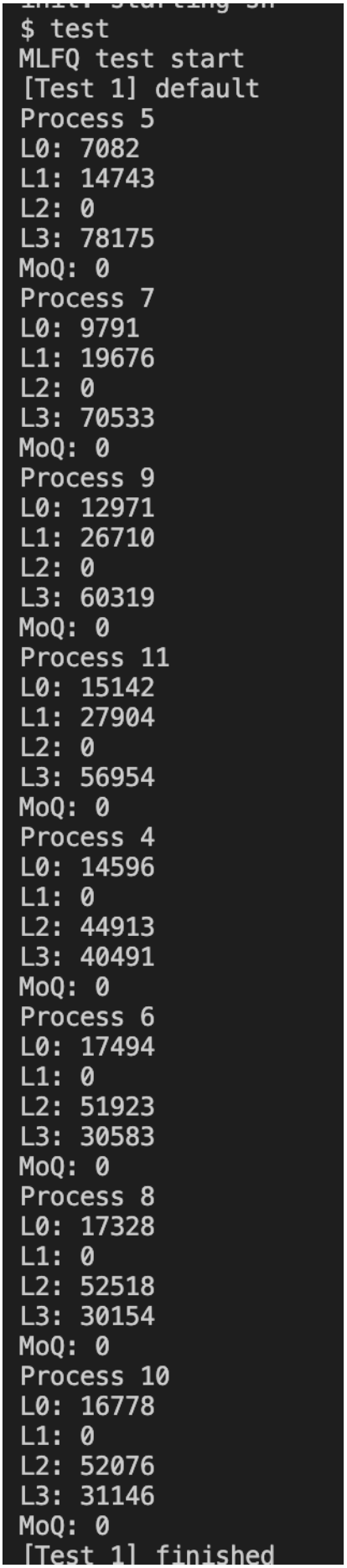
Result



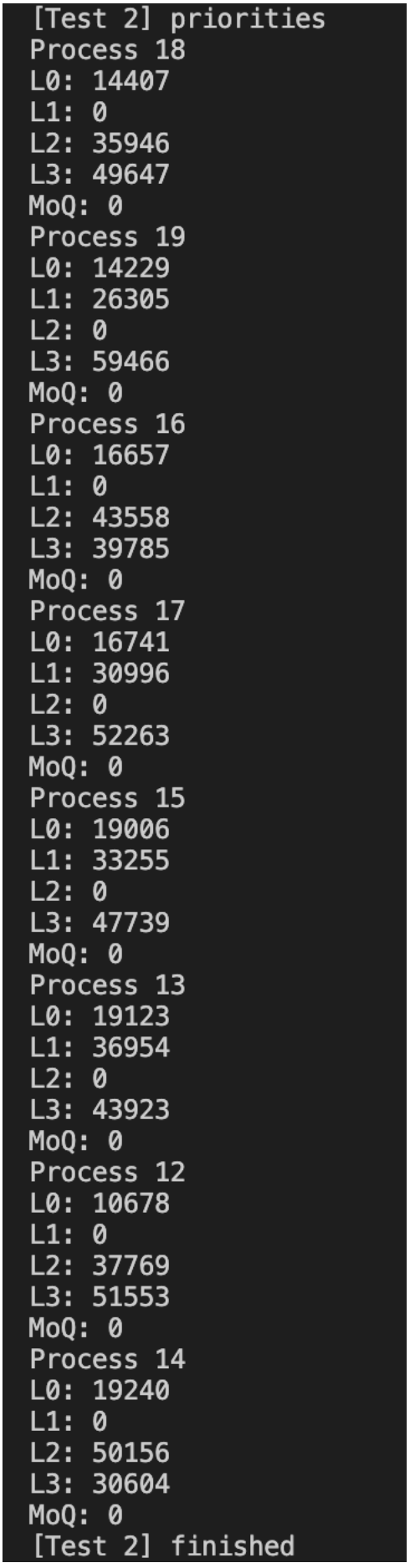
xv6가 성공적으로 부팅되었다.

미리 테스트코드를 유저프로그램 test로 만들어 등록을 하였다.

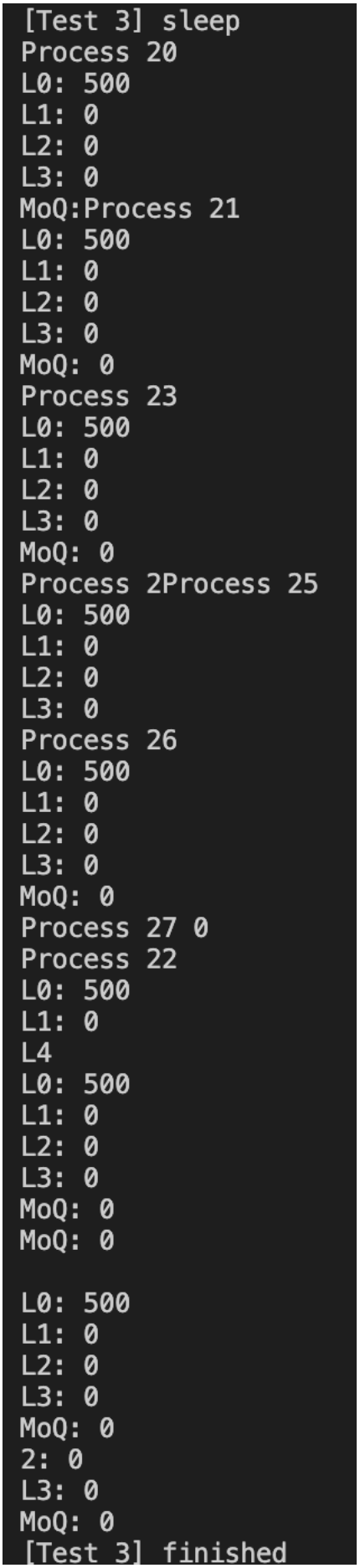
test를 실행해보자.


$ test
MLFQ test start
[Test 1] default
Process 5
L0: 7082
L1: 14743
L2: 0
L3: 78175
MoQ: 0
Process 7
L0: 9791
L1: 19676
L2: 0
L3: 70533
MoQ: 0
Process 9
L0: 12971
L1: 26710
L2: 0
L3: 60319
MoQ: 0
Process 11
L0: 15142
L1: 27904
L2: 0
L3: 56954
MoQ: 0
Process 4
L0: 14596
L1: 0
L2: 44913
L3: 40491
MoQ: 0
Process 6
L0: 17494
L1: 0
L2: 51923
L3: 30583
MoQ: 0
Process 8
L0: 17328
L1: 0
L2: 52518
L3: 30154
MoQ: 0
Process 10
L0: 16778
L1: 0
L2: 52076
L3: 31146
MoQ: 0
[Test 1] finished

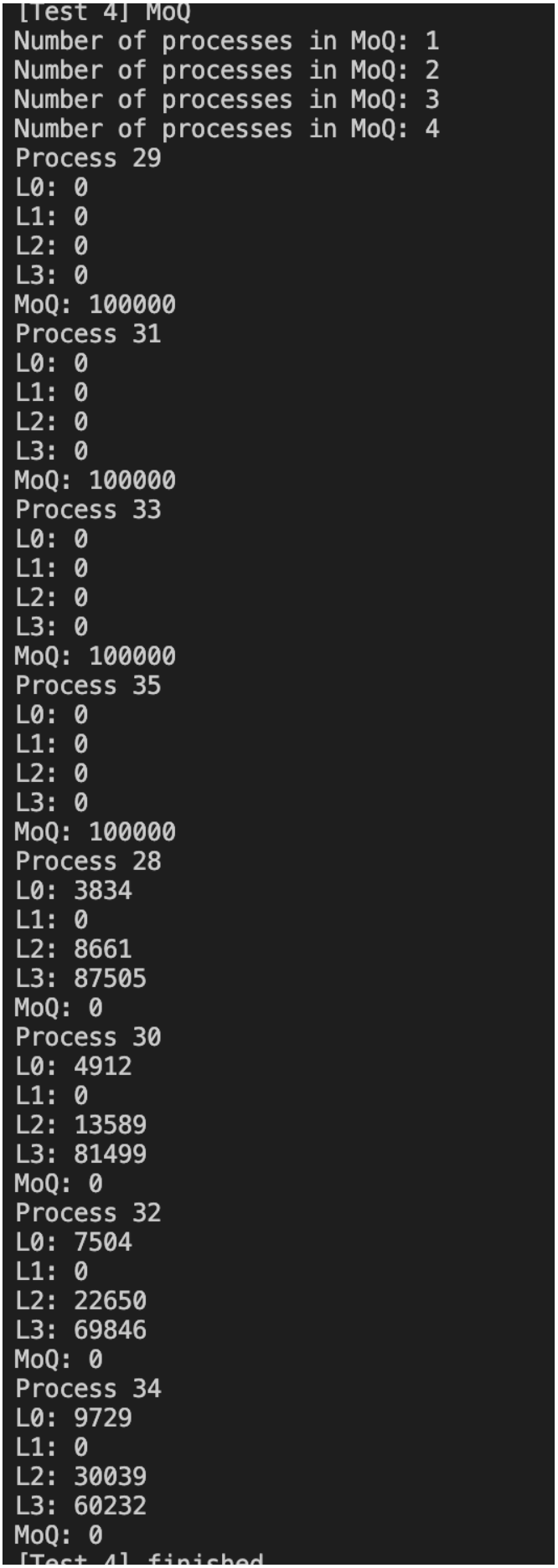
test 1 의 출력 결과. 테스트 코드의 예제대로 홀수 pid가 짝수 pid보다 우선순위가 높으므로 먼저 종료가 되는 모습이다. 여기서 각 큐에서 동작되는 시간을 보면 pid가 클수록 L0에서 동작되는 시간이 많은걸 볼수가 있는데 이는 예상된 결과이다. pid가 작을수록 먼저 스케쥴링 되므로 time quantum을 먼저 소비해 L3에 일찍 들어가게 되는데 내가 구현한 L3 스케쥴링은 우선순위가 같은 경우 가장 앞에 있는 프로세스가 먼저 스케쥴링 된다. 그렇기에 pid가 작은 프로세스는 L3에서 많은 시간동안 수행하고 pid가 클수록 L3 큐 내에서는 대기하고 있는 시간이 길어지고 priority boosting은 계속해서 발생하므로 L0, L1, L2에서 실행되는 시간이 길어진다. 결론적으로 pid가 작은 프로세스는 L0보다 L3에서 수행되는 시간이 많고 pid가 커질수록 L0 및 L1, L2에서 수행되는 시간이 많아진다. 각 레벨별로 수행되는 시간또한 L0의 time quantum은 2, L1는 4, L2는 6 이므로 대부분 이 비율에 맞게 수행된걸 확인할 수 있다.


[Test 2] priorities
Process 18
L0: 14407
L1: 0
L2: 35946
L3: 49647
MoQ: 0
Process 19
L0: 14229
L1: 26305
L2: 0
L3: 59466
MoQ: 0
Process 16
L0: 16657
L1: 0
L2: 43558
L3: 39785
MoQ: 0
Process 17
L0: 16741
L1: 30996
L2: 0
L3: 52263
MoQ: 0
Process 15
L0: 19006
L1: 33255
L2: 0
L3: 47739
MoQ: 0
Process 13
L0: 19123
L1: 36954
L2: 0
L3: 43923
MoQ: 0
Process 12
L0: 10678
L1: 0
L2: 37769
L3: 51553
MoQ: 0
Process 14
L0: 19240
L1: 0
L2: 50156
L3: 30604
MoQ: 0
[Test 2] finished

test 2의 출력내용. 출력예시와 동일하게 대부분 비슷한 시기에 종료되지만 pid가 클수록 높은 우선순위를 가지기에 대부분 pid가 큰 프로세스가 먼저 끝이 난다.


[Test 3] sleep
Process 20
L0: 500
L1: 0
L2: 0
L3: 0
MoQ:Process 21
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 23
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 2Process 25
L0: 500
L1: 0
L2: 0
L3: 0
Process 26
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 27 0
Process 22
L0: 500
L1: 0
L4
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
MoQ: 0
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
2: 0
L3: 0
MoQ: 0
[Test 3] finished

test3의 출력결과. 출력이 꼬이긴 했지만 출력예시와 비슷하게 pid가 작은 프로세스가 대부분 먼저 종료되고 모두 L0에서만 500이 출력되는 것을 확인할 수 있다.


[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
Process 29
L0: 0
L1: 0
L2: 0
L3: 0
MoQ: 100000
Process 31
L0: 0
L1: 0
L2: 0
L3: 0
MoQ: 100000
Process 33
L0: 0
L1: 0
L2: 0
L3: 0
MoQ: 100000
Process 35
L0: 0
L1: 0
L2: 0
L3: 0
MoQ: 100000
Process 28
L0: 3834
L1: 0
L2: 8661
L3: 87505
MoQ: 0
Process 30
L0: 4912
L1: 0
L2: 13589
L3: 81499
MoQ: 0
Process 32
L0: 7504
L1: 0
L2: 22650
L3: 69846
MoQ: 0
Process 34
L0: 9729
L1: 0
L2: 30039
L3: 60232
MoQ: 0
[Test 4] finished

test 4의 출력결과. 출력예시와 동일하게 MoQ의 크기를 잘 출력하고 있고 홀수 pid는 모두 MoQ에 속해 먼저 수행되는 것을 확인할 수 있고 짝수 pid는 제대로 MLFQ part에서 수행된 것을 확인할 수 있다.

Trouble Shooting

MoQ 내의 좀비 프로세스 처리

xv6 내의 모든 프로세스는 자식 프로세스가 죽으면 부모가 wait() 내에서 sleep 하고 있다가 깨어나 좀비가 된 자식을 확인 후 이를 회수하면서 최종적으로 종료가 된다. 그러나 MoQ를 구현함에 있어서 문제가 발생하였다. monopolize가 실행돼 MoQ 내의 프로세스만 스케쥴링 되는 중에 프로세스가 할 일을 끝내고 exit()를 호출하면 좀비로 상태가 바뀌면서 부모를 깨우고 스케쥴러를 호출한다. 하지만 부모는 MoQ 내에 존재하지 않아 스케쥴링 되지 못해 제때제때 좀비가 회수 되지 않는 문제가 발생하게된다. 나는 이 문제를 좀비 프로세스를 회수할 다른 방법을 구하는게 아닌 아예 스케쥴링에서 제외되도록 구현하였다. 좀비 프로세스가 계속 남아있더라도 논리적으로 좀비 프로세스가 스케쥴링이 되지만 않는다면 문제가 없을거라고 판단하여 exit()에서 구현한 것처럼 MoQ 내의 프로세스가 좀비로 변하면 MoQ 내의 index를 1 증가시켜 스케쥴러 함수 내에선 좀비 프로세스가 있는 공간에는 절대 접근하지 못하도록 하였다. 이후 모든 MoQ 프로세스의 동작이 끝나고 unmonopolize가 실행되면 그때 부모프로세스가 좀비를 회수하도록 유도하는 방식으로 이 문제를 해결하였다.

MoQ 내의 프로세스가 fork를 하는 상황

위의 문제를 고민하다가 파생된 또 다른 문제 상황이 바로 MoQ 내의 프로세스가 fork를 하는 상황이었다. 만약 monopolize가 실행된 상태에서 MoQ에 있는 프로세스가 프로그램을 수행하는 중 fork를 하게된다면 어떻게 되는가? 나는 이를 두가지로 나뉘어 생각해보았다.

1. MoQ 내 프로세스가 만들어낸 자식 프로세스는 mlfq에 들어간다
2. MoQ 내 프로세스가 만들어낸 자식 프로세스를 MoQ의 맨 앞에 저장한다.

일단 첫번째 방식은 무조건 문제가 발생할 수 밖에 없다. 부모인 MoQ 내의 프로세스는 기본적으로 자식이 생겼으니 자식이 종료될때까지 wait을 해야하는데 그러면 monopolize가 끝나지 않는다. monopolize는 MoQ내의 모든 프로세스가 종료되야 끝나는데 부모 프로세스가 스케쥴링 대상이 되지도 않는 mlfq내의 자식프로세스를 계속 기다리기 때문이다. 그렇다고 자식이 끝나지도 않았는데 먼저 종료를 할 수도 없기에 모순이 생긴다.

두번째 방식 또한 부모가 먼저 종료되는 문제는 사라졌지만 또다른 문제점이 존재한다. MoQ는 기본적으로 FCFS 정책을 지켜야한다. 그런데 첫번째 방식의 문제점을 해결하기 위해 자식 프로세스를 부모 프로세스 앞으로 끌고와 큐에 저장한다는 건 FCFS 정책에 위배가 되는 수행이다.

나는 위 내용을 종합했을때 MoQ 내의 프로세스가 자식 프로세스를 만들어내는 상황은 애초에 허용되지 않는 상황이라고 판단, 이에 관련된 예외처리는 하지 않았다.