2️⃣

# Project2_CPU Scheduler

## 1. Design

**`MultiLevel Feedback Queue`**란 프로세스에 고정된 우선순위를 부여하는 것이 아니라 각 작업의 특성에 따라 동적으로 우선순위를 부여하는 scheduling algorithm을 의미합니다. 동적으로 우선순위를 부여한 이유는 **many short CPU burst time**을 갖는 I/O-bound process와 **few very long CPU burst time**을 갖는 CPU-bound process를 구분하기 위해서는 하나의 queue로는 priority scheduling에 의한 **starvation** 문제와 같이 한계점이 분명하기 때문에 여러 개의 queue를 사용하여 각각의 queue에 적절한 scheduling algorithm을 적용해서 다른 특성을 가진 process를 분리해서 CPU utilization을 최대화하고 user에게 process가 실행중이라는 것을 바로 체감되게 하기 위함입니다.

그렇기 때문에 먼저 process를 적재시킬 queue를 만들어야 하고 Multi level queue이므로 level에 대응 하는 여러 queue를 선언해서 각각의 queue에 알맞은 scheduling algorithm을 적용해야 합니다.

저 같은 경우는 **LinkedList**를 이용하여 MLFQ와 MoQ를 구현하려고 설계하였고, 이에 맞는 변수와 함수들을 추가하였습니다.

## 2. Implementation

```c
// 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)

  int queueLevel;              // Queue level
  int priority;                // Priority
  int timeQuantum;             // Time quantum
  struct proc* next;           // Next process in the queue

  int original_ql;             // MLFQ queuelevel before entering MoQ
  int inMoQ;                   // In the monopoly queue
};
```

`proc.h` 파일의 proc 구조체를 보면, process에 대한 정보들이 선언되어 있는 것을 알 수 있습니다. 과제 명세를 통해 MLFQ와 MoQ에 대한 정보를 확인해보면 다음과 같습니다.

### MLFQ

| | |
|---|---|
| Scheduling Algorithm | Round-Robin Scheduling<br>Priority Scheduling |
| #Queue | 4 ($L_0$ ~ $L_3$) |
| Time Quantum | $L_i = 2i + 2$ |

### MoQ

| | |
|---|---|
| Scheduling Algorithm | FCFS |
| #Queue | 1 (Monopoly Queue) |

### 추가한 변수

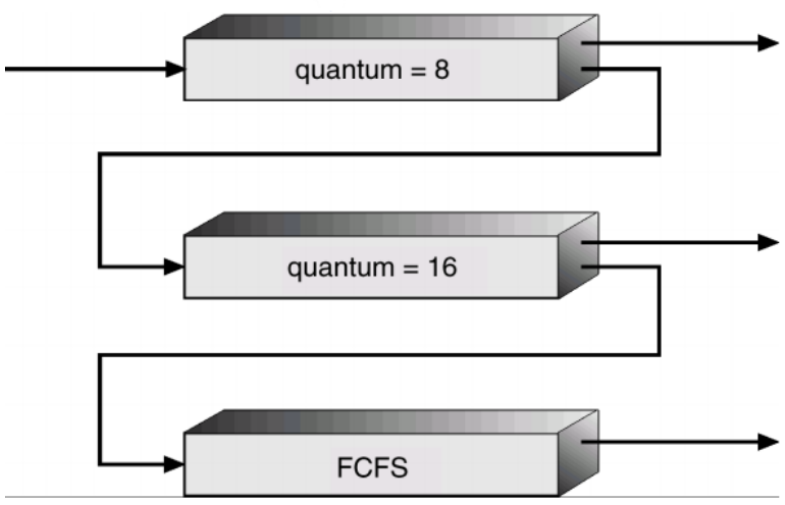
- queue의 경우 $L_0$부터 $L_3$까지 총 4개를 구현해야 하고 **숫자가 작을 수록 더 높은 우선순위를 갖는 queue** 이므로 queue의 level을 나타내면서 동시에 서로 다른 queue들을 구분할 변수가 필요하다고 생각을 해서 int형 변수 `queueLevel`을 선언하였습니다. 또한 `priority`와 `timeQuantum`도 각각의 queue마다 다른 값을 가지기 때문에 구조체 안에 변수로 선언해주었습니다.
- 마지막으로 주어진 timeQuantum 안에 process가 다 끝나지 못할 경우 현재 실행중이던 process를 다음 level의 queue로 보내고 다음 process를 불러와야 합니다. 이 때 process의 priority와 같은 기준에 따라 LinkedList 처럼 미리 pointer로 현재 process와 다음에 실행할 process를 연결해 놓으면 pointer로 빠르게 다음 process를 RUNNABLE 상태로 바꿀 수 있다고 생각해서 queue에서 현재 실행중인 프로세스의 바로 뒤에 있는 process를 가리키는 `next` 구조체 pointer를 선언하였습니다. 구조체 안에 구조체 포인터 변수를 선언하는 것이 곧 linked list를 구현한 것과 일맥상통하기 때문입니다.
- `original_ql`을 선언해서 `setmonopoly()` system call이 호출되어 MLFQ에 있던 process가 MoQ로 들어가기 전에 이 변수에게 process가 있던 queuelevel 값을 복사하여 `unmonopolize()`가 호출되었을 때 이 값을 불러와서 다시 원래 위치하고 있던

queuelevel로 이동할 수 있도록 하였습니다.

- `getlev()`와 `setmonopoly()`를 구현할 때 process가 MoQ에 있는지 확인하는 부분이 존재하는데, 이를 확인하기 위해 `inMoQ` 변수를 선언해서 inMoQ=1이면 MoQ에 존재하는 process이고 0이면 MoQ가 아닌 MLFQ part에 존재하는 process로 구분해 주었습니다.

그 다음으로 monopoly 여부를 구분하기 위한 전역변수 monopoly와 queue 구조체를 만들어서 $L_0$ ~ $L_3$ queue 4개의 경우 level과 배열의 index가 같기 때문에 MLFQ 배열을 만들어서 그 배열에 저장할 것이고, 여기에 추가로 구현해야 하는 MoQ 까지 선언을 해주었습니다. queue 구조체에는 총 6개의 변수를 선언해 주었습니다.

```
int monopoly = 0;
...
// queue 구조체
struct queue
{
  struct spinlock lock;
  struct proc *front;
  struct proc *rear;
  int timequantum; // 2*i+2
  int level; // queuelevel
  int size;
};

struct queue SchedQ[5]; // SchedQ[0]~SchedQ[3] = L_0 ~ L_3 , SchedQ[4] = MoQ
```




이론 pdf에 있는 위의 그림을 보면 3개의 level을 갖는 MultiLevel Queue가 구현되어 있고, 각각 다른 **timeQuantum**과 **scheduling algorithm**이 적용되어 있는 것을 확인할 수 있습니다. 하나의 queue에는 queue로 들어가는 방향(`Enqueue`)의 화살표 1개와 queue에서 나가는 방향(`Dequeue`)의 화살표 2개로 구성되어 있는데, 먼저 이 부분에 대해 설명을 하겠습니다.

| **화살표** | 의미 |
|---|---|
| **Enqueue** | - Process가 fork 되어 실행을 위해 queue로 들어가는 경우<br>- Process가 SLEEPING 상태에 있다가 `wakeup()`에 의해 깨워져서 다시 queue로 들어가는 경우<br>- Process의 timequantum이 queue에 할당된 timequantum보다 커서 다음 level로 이동하게 되었을 때의 **다음** 레벨의 큐 관점 |
| **Dequeue** | - Process가 실행을 마치고 terminate 되는 경우<br>- Process의 timequantum이 queue에 할당된 timequantum보다 작아서 해당 queue에서 process 실행이 끝난 경우<br>- Process의 timequantum이 queue에 할당된 timequantum보다 커서 다음 level로 이동하게 되었을 때의 **이전** 레벨의 큐 관점 |

`Enqueue()`와 `Dequeue()`를 구현할 때 이 부분을 고려하여 구현하여야 합니다. 더하여 MLFQ 뿐만 아니라 MoQ도 존재하기 때문에 system call의 호출과도 연계가 되어야 합니다.

각각의 queue의 경우 저는 MLFQ와 MoQ를 하나로 묶어서 크기가 5인 배열이지만 그 배열에 들어있는 $L_0$부터 $L_3$까지는 서로 다른 level과 timequantum을 갖기 때문에 `queueLevel`과 `timeQuantum`을 변수로 선언하여 각각의 queue에서 값을 설정하도록 하였습니다. 그리고 MoQ는 MLFQ와 별개이기 때문에 `SchedQ[4]`만 따로 빼서 값을 설정해주었습니다.

```
...
acquire(&ptable.lock);
p->state = RUNNABLE;
release(&ptable.lock);
...
```

`proc.c` 파일의 코드를 보다보면 위와 같이 ptable에 대한 lock을 걸어준 뒤 process의 상태를 RUNNABLE로 바꿔주기만 하는 함수들이 존재합니다. (ex, userinit, fork .. )

## Process

### allocproc()

```
//  Look in the process table for an UNUSED proc.
//  If found, change state to EMBRYO and initialize
...
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  if (p->state == UNUSED)
    goto found;
...
found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  p->queueLevel = -1; // enqueue되기 전까지는 queueLevel은 -1
  p->priority = 0;    // 처음 process가 실행될 때 priority는 0
  p->timeQuantum = 0; // 처음 process가 실행될 때 timeQuantum은 0
  p->inMoQ = 0;       // 처음엔 MoQ에 들어있지 않음
...
```

`allocproc()`의 역할은 아직 새 process로 할당되지 않은 **UNUSED** 상태의 process를 ptable에서 찾아서 **EMBRYO**(new) 상태로 바꿔주고 pid 값을 할당해주는 함수입니다. **EMBRYO** 상태의 경우 아직 enqueue되지는 않은 상태이기 때문에 아직 아무 queue에 속하지 않았다는 의미로 process의 `queuelevel`을 -1로 설정하고 나머지는 모두 0으로 초기화 해주었습니다.

### Userinit()

`userinit()`을 보면 위에 주석으로 ***"Set up first user process."*** 라는 문구가 적혀있습니다. 그러므로 `Userinit()`은 user program이 실행되게 되어 process가 생성될 때 process에 대한 정보를 초기화해주는 부분일 것이라고 생각을 했습니다.

```
...
acquire(&ptable.lock);
// MLFQ 초기화
for (int i = 0; i < 4; i++){
  SchedQ[i].front = SchedQ[i].rear = NULL;
  SchedQ[i].timequantum = 2 * i + 2;
  SchedQ[i].level = i;
  SchedQ[i].size = 0;
}
// MoQ 초기화 (SchedQ[4] = MoQ)
SchedQ[4].front = NULL;
SchedQ[4].rear = NULL;
SchedQ[4].level = 99;
SchedQ[4].size = 0;

p->state = RUNNABLE;

// User process가 생성될 때, 새로운 process를 SchedQ[0]에 넣어줌
p->timeQuantum = 0;
p->queueLevel = 0;
p->original_ql = 0;
p->inMoQ = 0;
Enqueue(&SchedQ[0], p);

release(&ptable.lock);
...
```

따라서 이 부분에서 초기화 작업을 진행해주었습니다. 먼저 MLFQ의 경우 현재 process가 없는 상태이므로 `size`는 0, 포인터는 전부 아무것도 가리키고 있지 않다는 의미로 NULL로 설정하고 `timeQuantum`은 $2 * i + 2$, `level`은 index와 같기 때문에 index로 설정을 해주었습니다.
그 다음 MoQ도 초기화를 해주었습니다. MoQ도 마찬가지로 포인터와
`size`는 process가 없기 때문에 0으로 설정해주는데, MoQ는 **FCFS algorithm**이 적용되어 있으므로 먼저 들어오게 되면 process가 끝날때까지 CPU를 점유합니다. 따라서 `timeQuantum`은 의미가 없기 때문에 지정해줄 필요가 없습니다. MoQ의 `level`의 경우, 뒤에서 나오겠지만

`getlev()`와 연관이 매우 깊습니다. `getlev()`를 보면 process가 MoQ에 있는 경우 99를 return하도록 되어 있습니다. 이를 통해 MoQ의 `level`은 99임을 알 수 있습니다.

MLFQ와 MoQ에 대한 초기화를 진행하였다면, first user process에 대한 초기화를 해주었습니다. process의 경우 처음엔 모두 $L_0$로 들어가기 때문에 `timeQuantum`과 process의 `level`을 0으로 초기화 한 뒤, $L_0$로 enqueue해주었습니다.

## fork()

`fork()`의 경우 호출이 되었을 때, 자식 process를 생성합니다. 여기서 저는 fork를 했을 때도 process에 대한 정보를 초기화해주어야 한다고 생각을 했습니다.

```c
int fork_children2()
{
  int i, p;
  for (i = 0; i < NUM_THREAD; i++)
  {
    if ((p = fork()) == 0)
    {
      sleep(10);
      return getpid();
    }
    else
    {
      int r = setpriority(p, i + 1);
      if (r < 0)
      {
        printf(1, "setpriority returned %d\n", r);
        exit();
      }
    }
  }
  return parent;
}
```

```c
int fork_children3()
{
  int i,p;
  for(i=0;i<=NUM_THREAD;i++){
    if((p = fork()) == 0)
    {
      sleep(10);
      return getpid();
    }else{
      int r = 0;
      if(p % 2 == 1)
      {
        r = setmonopoly(p, 2021088304);
        printf(1, "Number of processes in MoQ: %d\n",r);
      }
      if(r < 0)
      {
        printf(1, "setmonopoly returned %d\n", r);
        exit();
      }
    }
  }
  return parent;
}
```

Test 파일에서 유추할 수 있지만 main에서 호출하는 fork_ 함수의 경우 **NUM_THREAD** 만큼 반복하면서 `fork()`로 process를 생성한 뒤 자식 process의 경우 `getpid()`를 호출하여 process의 pid 값을 return하게 하고 부모 process의 경우 system call을 호출하거나 부모 process 자체를 return하게 합니다.
fork를 하게 되면 pid를 제외하고 부모 process와 동일한 정보를 갖지만, 여기서는 fork를 하게 되었을 때 이러한 정보들과 상관없이 모두 $L_0$ queue로 enqueue되기 때문에 달라지는 부분은 $L_0$에서 다음 level의 queue로 갈 때 pid 값에 의해 다음 queue가 결정되기 때문에 행선지가 달라질 수 있다는 점 밖에 없습니다. 따라서 `fork()`에서도 `userinit()`에서와 동일하게 process를 초기화한 뒤 $L_0$로 enqueue해주었습니다.

## wakeup1()

```
static void
wakeup1(void *chan)
{
  struct proc *p;

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p->state == SLEEPING && p->chan == chan){
      p->state = RUNNABLE;
      if (p->inMoQ || monopoly){
        p->queueLevel = 99;
        Enqueue(&SchedQ[4], p); // wakeup한 process가 MoQ에 속한 경우 MoQ에 넣어줌
      }else{
        Enqueue(&SchedQ[p->queueLevel], p); // wakeup한 process를 queue에 넣어줌
      }
    }
  }
}

void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
```

```
  release(&ptable.lock);
}
```

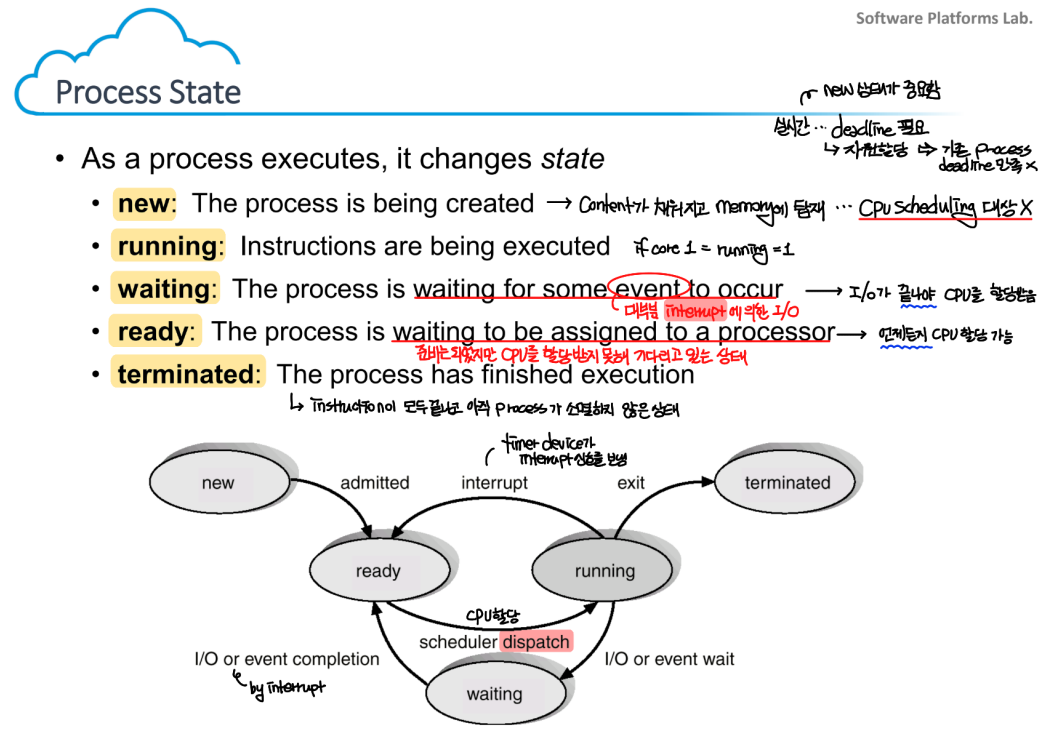
process가 **SLEEPING** 상태가 되었을 때 process를 깨우는 system call이 필요합니다. 이 system call은 이미 구현이 되어있습니다. 원래는 code에서 **빨강색 형광펜**을 칠한 부분은 존재하지 않았습니다.

```
case T_IRQ0 + IRQ_TIMER:
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    if(ticks % 100 == 0){
      priorityBoosting();
    }
    wakeup(&ticks);
    release(&tickslock);
  }
  lapiceoi();
  break;
```

`trap.c`의 timer interrupt 부분을 보면, tickslock을 걸고 1 tick씩 올리다가 일정 tick이 되면 `wakeup()`을 호출해서 **SLEEPING** 이던 process의 상태를 **RUNNABLE**로 바꿔줍니다. 여기서는 process의 상태를 **RUNNABLE**로 바꿔주기만 하는데, 뒤에 제가 구현한 `scheduler()`에서 보면은 **SLEEPING** 상태일 경우 process의 queuelevel을 바꾸지 않고 dequeue만 해줍니다. Dequeue 된 뒤, `wakeup()`에 의해 **RUNNABLE** 상태가 된다면 다시 실행할 수 있기 때문에 원래 있던 queue로 enqueue되어야 합니다. 따라서 이 부분에서 process가 **RUNNABLE** 상태로 바뀐 뒤 enqueue 될 수 있도록 수정해주었습니다.

## kill

kill은 pid를 인자로 받아서 pid가 일치하는 process를 말 그대로 죽이는 함수 입니다.



Basic State Transition Diagram of a Process

Kill에 대해 설명하기에 앞서 Process state에 대한 강의 자료를 보면, 5가지의 상태가 있습니다. xv6에서는 `proc.h` 파일에 상태 정보가 주어져 있습니다.

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

**EMBRYO** - new / **SLEEPING** - waiting / **RUNNABLE** - ready / **RUNNING** - running을 매칭시키고 나면 **UNUSED**와 **ZOMBIE** 2가지 state와 terminated가 남습니다. 이 둘의 경우 유사한 의미를 지니는 것 처럼 보이지만, 약간의 차이가 존재합니다.

- **ZOMBIE:** 현재 process가 다 끝났는데 부모 process에 의해 회수되지 않아서 여전히 ptable에 존재하는 상태
- **UNUSED:** 프로세스를 사용 조차 하지 않았기 때문에 새 프로세스에 할당되기를 기다리고 있는 상태
- terminated: 프로세스가 완전히 끝나고 ptable에 있는 자원을 회수당한 뒤 완전히 시스템에서 제거된 상태

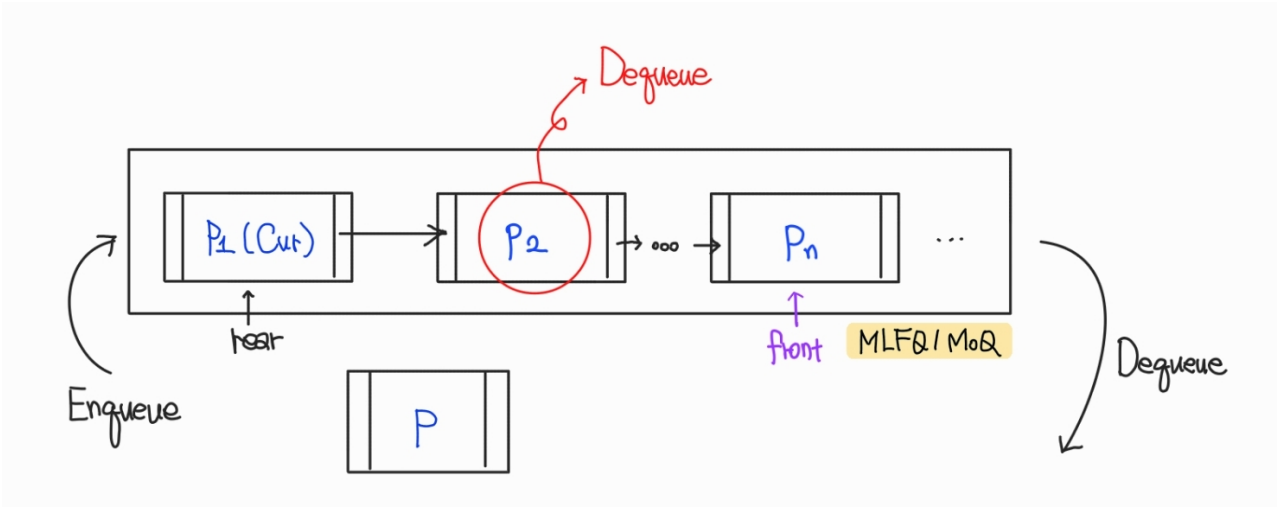
그러므로
**ZOMBIE** 상태는 terminated 상태로 가기 전 단계라고 볼 수 있습니다. 여기서 process를 terminated로 보내는 역할을 하는 함수가 `kill()`이라고 생각을 했습니다. 따라서 Kill에서도 `wakeup()`과 같이 process의 state에 따라 process를 설정해주었습니다.

## Enqueue / Dequeue

```
#include <stddef.h>
```

Enqueue, Dequeue를 구현하기에 앞서서 **stddef** library를 include 해서 process가 없는 경우를 0 대신 `NULL`을 사용해서 가독성을 높였습니다.

Enqueue와 Dequeue에 대해 간단히 그림으로 표현을 하면 아래와 같습니다.




### void Enqueue(struct queue *q, struct proc *p)

```
void
Enqueue(struct queue *q, struct proc *p)
{
  acquire(&q->lock);
  // queue가 가득 차있거나 queuelevel이 다른 경우
  if (q->size >= NPROC || p->queueLevel != q->level){
    release(&q->lock);
    return;
  }
  p->next = NULL;
  if (q->rear != NULL){ // queue에 process가 존재하는 경우
    q->rear->next = p; // queue의 rear에 process 추가
    q->rear = p; // rear를 추가한 process로 변경
  }else{ // queue가 비어있는 경우
    q->front = q->rear = p; // queue에는 process 1개만 존재
  }
  q->size++; // q에 들어있는 process개수 증가
  release(&q->lock);
}
```

Queue에 process들이 들어가고 이전 level의 queue에서 다음 level의 queue로 이동하기 위해서는 `Enqueue()`를 구현해야 합니다.
queue가 가득 차있거나(**NPROC**) q의 level과 process가 있는 queue의 level이 다른 경우 아무 동작 없이 반환하도록 하였고, queue에 대한 lock을 얻어서 여러 thread가 queue에 동시에 접근하는 것을 방지한 뒤 조건문을 탐색합니다. 첫 번째 조건은 queue가 비어있지 않은 상태에서 process가 enqueue된 경우이고, 두 번째 조건은 enqueue를 했을 때 queue에 들어있는 process가 enqueue한 process 1개만 존재하게 되는 경우입니다.

먼저 queue에 process가 존재하는 상태 (`q→rear != 0`)에서 enqueue된 경우, rear 포인터는 제일 뒤에 있는 process를 가리키고 있습니다. 이 때 새로운 process가 들어오게 된다면 rear가 가리키고 있는 process를 process의 next 포인터로 연결을 한 뒤, 추가한 process를 rear 포인터가 가리키도록 update를 하는 방식으로 구현하였습니다.
그 다음 조건을 보면, queue가 비어있는 상태에서 새로 process가 들어온 상태이기 때문에 queue의 front와 rear 포인터 모두 NULL인 상태이다가 새로 추가된 process가 queue의 첫번째이면서 동시에 마지막 process이기 때문에 두 포인터 모두 enqueue된 process를 가리키도록 설정해주었습이다.

### void Dequeue(struct queue *q, struct proc *p)

```
void
Dequeue(struct queue *q, struct proc *p)
{
  acquire(&q->lock);
  if (q->front == NULL){ // queue가 비어있는 경우
    release(&q->lock);
    return;
```

```
  }

  struct proc *cur = q->front;
  struct proc *prev = NULL;

  while(cur){
    if (cur == p){ // 빼낼 process를 찾은 경우
      if (prev == NULL){
        q->front = cur->next; // front를 다음 process로 변경
        if (q->rear == cur){ // process가 첫번째 원소인 경우
          q->rear = NULL; // rear를 0으로 변경
        }
      }else{
        prev->next = cur->next; // 이전 process의 next를 다음 process로 변경
        if (q->rear == cur){ // p가 rear인 경우
          q->rear = prev; // rear를 이전 process로 변경
        }
      }
      q->size--;     // q에 들어있는 process개수 감소
      cur->next = NULL; // queue에서 빼낸 process
        break;
      }
      prev = cur;      // cur를 prev로 설정
      cur = cur->next; // cur를 다음으로 이동
    }
    release(&q->lock);
  }
```

Dequeue는 원래 같은 경우 front가 가리키고 있는 process를 제거하기만 하면 되지만, 이번 구현에서는 한 가지 조건을 추가해야 합니다. 그 이유는 **priority scheduling**에 있습니다. Priority scheduling에서는 priority 값이 가장 큰 process를 우선적으로 scheduling합니다. 그러나 저는 linked list로 구현을 하였기 때문에 priority 순으로 정렬을 하고자 하면 비효율적일 뿐더러 시간도 오래걸리고 구현이 복잡해질 것이라고 생각을 했습니다. 따라서 뒤에서 나오는 `find_m_proc()`으로 max priority를 갖는 process만 찾아서 그 process를 dequeue하는 방향으로 구현하였습니다. 이 때, 이 process가 항상 queue의 제일 앞에 있다는 보장이 없어서 중간에 위치할 수도 있기 때문에 process가 제일 앞에 위치하지 않는 경우에 dequeue하는 code를 추가해야 한다고 생각했습니다.

코드를 설명하자면, 먼저 front 포인터가 NULL인 경우는 queue에 process가 존재하지 않는 상태이기 때문에 바로 return을 하였습니다. 그 다음 queue의 front가 가리키고 있는 process에 대하여 반복문을 돌면서 dequeue할 process를 찾았다는 가정하에 두 가지 경우의 수로 나누었습니다.

- 첫번째 조건은 `prev==NULL` 로 제거해야할 process가 제일 앞에 있는 경우를 의미합니다. 이 경우 front 포인터를 다음 process를 가리키도록 만듭니다. 이 때 한 가지 조건을 더 추가해야하는데, queue에 process가 1개만 있었던 경우 그 때는 front와 rear 모두 같은 process를 가리키고 있었기 때문에 rear 포인터를 NULL로 만들어 줘야 합니다.
- 두번째 조건은 priority scheduling에서 사용할 조건으로 process가 맨앞이 아니라 중간이나 끝에 있는 경우입니다. 이 경우 이전 process의 next 포인터를 현재 process의 next를 가리키게 하여 현재 process를 제거할 수 있도록 하였습니다. 이 때도 process가 1개였을 때의 조건을 추가하였습니다.

Pointer 조정을 다 끝냈다면 아직 다음 process를 가리키고 있던 현재 process의 next 포인터를 NULL로 만들어 외딴 node로 만든 뒤 반복문을 탈출하고 현재 process와 이전 process를 update하는 code로 구성하였습니다.

## [Scheduler]

MLFQ Scheduler를 구현하기 위한 함수들과 System call 구현이 끝났다면, `proc.c`에 구현 되어 있는 `scheduler()`에 scheduling algorithm을 구현해야 합니다.

그 전에 기존에 구현되어 있던 scheduler 함수를 보면 다음과 같습니다.

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

  // 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)
      continue;
      // 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); // switch to user mode
      p->state = RUNNING;
      swtch(&(c->scheduler), p->context); // context switching
      switchkvm(); // switch to kernel mode
      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
      release(&ptable.lock);
  }
}
```

`scheduler()`를 보면 ptable에 있는 모든 process들을 순회하면서 RUNNABLE 상태인 process에 대해 scheduling을 실행하는 code로 구현되어 있습니다. 이 경우 scheduling algorithm 없이 단순히 context switching을 통해 process를 실행만 하는 code와 마찬가지이기 때문에 이 함수에 MLFQ와 MoQ에서 필요한 알고리즘들을 추가해야 합니다.

MLFQ의 경우 **Round-Robin Scheduling**과 **Priority-Scheduling**을 구현해야 하고, MoQ의 경우 **FCFS Scheduling**을 구현해야 하기 때문에 scheduler 함수에서 3 part로 나누어 각각의 part에 알고리즘을 구현하고 $L_0$부터 $L_2$까지는 Round-Robin, $L_3$에는 Priority-Scheduling, MoQ에는 FCFS를 적용하는 방향으로 구현하였습니다.

먼저 scheduling을 하는 부분을 따로 `scheduling()`으로 빼서 `scheduler()`의 길이를 줄이고 가독성을 높이고자 하였습니다.
`scheduling()`에서는 process의 state를 RUNNING으로 바꿔서 context switch를 통해 scheduling하는 scheduler에서의 핵심적인 부분이 포함되어 있습니다.

- scheduling()

```
void
scheduling(struct cpu *c, struct proc *p)
{
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING;
  swtch(&(c->scheduler), p->context);
  switchkvm();
  c->proc = 0;
}
```

- scheduler()

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

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

    if(monopoly){ // 독점상태가 되면
      scheduling_MoQ(c);
    }
    scheduling_RR(c);
    scheduling_PQ(c);

    release(&ptable.lock);
  }
}
```

`scheduler()`의 경우 3가지로 구분되어 있습니다 (Round-Robin, Priority Scheduling, Monopoly Scheduling(FCFS)). 이 중에서 Monopoly scheduling을 가장 먼저 구현하였습니다. 그 이유는 전역변수로 선언된 `monopoly`가 `monopolize()` 호출에 의해 1로 변경이 되게 될 경우 즉시 MoQ를 scheduling 해야하기 때문에 제일 처음 구현을 했습니다.

- MoQ (`scheduling_MoQ()`)

```
int monopoly = 0; // CPU 독점 여부 - 전역변수
...
void
scheduling_MoQ(struct cpu *c)
{
  struct proc* p = NULL;
  int size = SchedQ[4].size;
  while (size){
    p = SchedQ[4].front;
    if (p->state == RUNNABLE){
      if (p->queueLevel == SchedQ[4].level) // RUNNABLE이고 MoQ에 있는 proc이면
        break;                              // 반복문 탈출
      else
        continue;
    }else{
      if (p->state == SLEEPING){
        p->queueLevel = -1;
        Dequeue(&SchedQ[4], p);
      }
    }
  }
  if (SchedQ[4].size){
    p = SchedQ[4].front;
    if (p->state == RUNNABLE){
      scheduling(c, p); // RUNNABLE인 process이면 scheduling
    }
  }else{
    unmonopolize(); // 실행이 다 끝나면 unmonopolize
  }
}
```

`monopoly=1` 이 될 경우 MoQ에 들어가 있는 process의 개수만큼 while문을 돌면서 process의 state가 **RUNNABLE**하고 level이 99인 process를 찾았을 때 반복문을 탈출하여 한번에 하나의 process를 scheduling하도록 하였습니다. 그 이유는 FCFS는 하나의 process를 실행하게 되면 process가 끝날 때까지 실행을 하기 때문에 process가 실행을 끝마치게 되면 **RUNNABLE**한 process의 개수는 1개 줄게되고 **RUNNABLE**한 개수가 0개가 되었을 때 `unmonopolize()`를 호출하도록 할 수 있다고 생각했기 때문입니다. 여기서는 다른 부분과 달리 UNUSED와 ZOMBIE 상태의 process를 다루는 부분을 dequeue하지 않습니다. 그 이유는 `unmonopolize()`를 호출하게 되면 어차피 MoQ에 있는 모든 process들이 MLFQ part로 넘어가기 때문입니다.

- Round-Robin

```
void
scheduling_RR(struct cpu *c)
{
  struct proc *p, *nxt;

  for (int i = 0; i < 3; i++){
    p = SchedQ[i].front;
    while (p){
      nxt = p->next;

      if (p->state == RUNNABLE){
        scheduling(c, p);

        if (p->state == ZOMBIE){
          p->queueLevel = -1;
          Dequeue(&SchedQ[i], p);
        }
      }else if (p->state == SLEEPING){
        p->original_ql = p->queueLevel;
```

```
        Dequeue(&SchedQ[i], p);
      }else if (p->state == UNUSED){
        p->queueLevel = -1;
        Dequeue(&SchedQ[i], p);
      }
      p = nxt;
    }
  }
}
```

$L_0$부터 $L_2$까지는 Round-Robin algorithm이 적용되어 있습니다. 이 부분은 다음 레벨의 queue로 넘어갈 때를 제외하면 동일한 메커니즘으로 scheduling 작업이 진행되기 때문에 `for(int i=0; i<3; i++)` 반복문으로 묶어서 한꺼번에 처리를 해주었습니다. 각각의 queue에 들어있는 process들을 순회하면서 process의 state를 확인하고 **RUNNABLE**한 process와 **RUNNABLE**하지 않은 process로 나누어 처리를 해주었습니다. 먼저 RUNNABLE하지 않은 경우, **UNUSED**와 **SLEEPING**으로 나누어 dequeue를 해주는데 차이점은 **SLEEPING**의 경우 `wakeup()`에 의해 **RUNNABLE**한 상태로 바뀌게 될 경우 queue의 뒤로 들어가 있다가 자신의 차례가 되었을 때 다시 실행이 가능하기 때문에 `queuelevel`을 변경하지 않았지만 **UNUSED**의 경우는 아직 새 process에 할당조차 되지 않은 상태이기 때문에 queue에 들어간다해도 실행을 하지 못하기 때문에 `queuelevel`을 -1로 바꾸어 어느 queue에도 속하지 않은 상태가 되도록 한 뒤 dequeue를 해주었습니다. **RUNNABLE**한 process의 경우 scheduling을 한 뒤 **ZOMBIE** 상태가 된 process가 생기면 dequeue 해주도록 하였습니다.

- Priority Scheduling

```
void
scheduling_PQ(struct cpu *c)
{
  struct proc* p = NULL;
  while (SchedQ[3].size > 0){
    p = find_m_proc(); // max priority process = h_proc
    if (p->state != RUNNABLE){
      if (p->state == SLEEPING){
        p->original_ql = p->queueLevel;
        Dequeue(&SchedQ[3], p);
      }else if (p->state == UNUSED){
        p->queueLevel = -1;
        Dequeue(&SchedQ[3], p);
      }
    }else{
      scheduling(c, p);
      if (p->state == ZOMBIE){
        p->queueLevel = -1;
        Dequeue(&SchedQ[3], p);
      }
    }
  }
}
```

`SchedQ[3]` ($L_3$)의 경우 이전 queue들과는 달리 Priority Scheduling을 적용해야 합니다. Priority Scheduling의 경우 priority 값이 큰 순서대로 CPU를 할당해주기 때문에 가장 먼저 해야할 일은 priority 값이 가장 큰 process를 찾는 작업입니다. 저는 그 과정을 find_m_proc()을 구현하여 queue에 있는 process들을 탐색하면서 max priority를 갖는 process를 return하도록 해주었습니다.

```
// SchedQ[3]에서 최대 priority를 갖는 process return
struct proc *
find_m_proc(void)
{
  struct proc *p = 0;
  struct proc *max_p = 0;
  int max_priority = -1;

  for (p = SchedQ[3].front; p != 0; p = p->next){
    if (p->priority > max_priority)
    {
      max_priority = p->priority;
      max_p = p;
    }
```

```
  }
  return max_p;
}
```

Priority Scheduling은 단순히 scheduling algorithm에 적용이 되는 정책일 뿐 알고리즘의 관점으로 보면 SJF와 FCFS에 가깝다고 볼 수 있습니다. 따라서 FCFS 처럼 구현을 하였습니다. `Setpriority()`에 의해 설정된 priority 중 가장 큰 priority를 갖는 process를 `find_m_proc()`으로 찾아서 state에 맞게 처리를 해주었습니다.

# MLFQ (MultiLevel Feedback Queue)

## Function

### [System Call]

1. `void yield(void)`

> **< 명세 >**
> - 자신이 점유한 CPU를 양보합니다.

```
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

`yield()`의 경우 이미 `proc.c` 파일에 구현이 되어있고, 실습 시간을 통해 system call 등록까지 완료한 상태입니다. 간단히 설명을 하자면, `acquire()`를 통해 process의 상태를 관리하는 ptable의 lock을 얻어서 synchronization을 보장하고 그 다음 process의 상태를 **RUNNABLE**로 바꿉니다. **RUNNABLE** 상태가 되고 나면 scheduler를 호출하여 context switch를 수행하는 `sched()`를 통해 일련의 작업을 수행하고 작업이 끝나면 `release()`로 `acquire()`로 얻었던 lock을 해제하여 다시 ptable에 접근할 수 있도록 합니다. `yield()`를 통해 process가 CPU 점유를 원하지 않는 경우 다른 process에게 CPU 점유를 양도할 수 있게 됩니다.

<참고 - sched 함수>

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler); // context switching
  mycpu()->intena = intena;
}
```

2. `int getlev(void)`

> **< 명세 >**
> - 프로세스에 속한 큐의 레벨을 반환합니다.
> - MoQ에 속한 프로세스인 경우 99를 반환합니다.

```
int
getlev(void)
{
  acquire(&ptable.lock);
  struct proc *p = myproc();
  int level=0;

  if (p->inMoQ){ // MoQ에 속한 프로세스인 경우 99 return
    level = 99;
  }else{
    level = p->queueLevel;
  }
  release(&ptable.lock);
  return level;
}
```

먼저 함수에 현재 process (`myproc()`)를 참조하는 proc 구조체 포인터 `p` 변수를 선언하였습니다. 그 다음 조건문을 통해 현재 process가 MoQ에 속한 경우 99를 return하여 MoQ에 들어있는 process의 queuelevel을 99로 설정하고, 그렇지 않은 경우 proc 구조체에 선언한 queueLevel 변수를 이용하여 `p→queueLevel` 로 process가 속한 queue의 level을 return하게 하였습니다. `getlev()`가 호출되게 된다면 level이 return되기 때문에 return된 값을 통해 process가 들어있는 queue의 위치를 알 수 있습니다.

3. `int setpriority(int pid, int priority)`

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

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

  acquire(&ptable.lock);
  struct proc *p;

  int found = 0;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p->pid == pid){ // pid가 일치하는 process를 찾은 경우
      p->priority = priority; // priority 설정
      found = 1;              // 찾았다는 flag
      break;
    }
  }
  release(&ptable.lock);
  return found ? 0 : -1; // 찾았으면 0 return 못찾으면 -1 return
}
```

`setpriority()`의 경우 똑같이 구조체 포인터 p를 선언하였습니다. 가장 먼저 parameter로 받은 `priority`가 명세에 주어진 유효범위 밖인 경우 -2를 return하도록 선언하여 주었습니다.
그 다음 process의 priority를 지정해주어야 하는데, 이 때 process table을 안전하게 탐색하고 **synchronization**을 보장하기 위해 `yield()`에서와 동일하게 `acquire()`, `release()`로 ptable의 lock을 선언해 주었습니다. 이를 통해 priority를 선언해주고자 하는 process만 접근을 할 수 있습니다.
또한 flag 변수 용도로
`found`를 선언해서 인자로 받은 pid와 일치하는 process를 찾은 경우 `found`를 1로 변경해서 priority를 설정한 process의 경우 0을 return하고 process를 찾지 못한 경우 -1을 return 하도록 하였습니다. lock을 지정한 다음 process table 에 있는 모든 process들을 탐색하면서 pid가 일치하는 process를 찾아서 priority를 설정하고 found 값에 따라 0 또는 -1을 return 합니다.

## [Priority Boosting]

```
void
priorityBoosting(void)
{
  acquire(&ptable.lock);
  if (monopoly) // MoQ를 scheduling 중인 경우 priorityBoosting X
    return;
  // queue의 모든 process 탐색 -> 모든 process를 L0로 재조정
  for (int i = 0; i < 4; i++){
    for (struct proc *p = SchedQ[i].front; p != 0; p = p->next){
      p->queueLevel = -1;
      Dequeue(&SchedQ[i], p);

      p->timeQuantum = 0; // time quantum 초기화

      p->queueLevel = 0;
      Enqueue(&SchedQ[0], p);
    }
  }
  release(&ptable.lock); // Release the lock
}
```

Priority가 낮은 process의 경우 $L_2$ queue에서 $L_3$ queue로 넘어 오는 과정에서 더 높은 priority를 갖는 process가 priority scheduling이 구현되어 있는 $L_3$ queue로 들어가게 될 경우 queue에는 들어가 있는데, priority를 기준으로 자신의 차례가 올 때까지 무기한 대기만 하는 상태인 **Starvation problem**이 발생할 가능성이 매우 높아집니다. 이를 해결하는 방법 중 하나로 시간이 지날 때마다 priority를 높여주는 **Aging** 기법이 있지만, 명세에서는 $L_3$에서 time quantum을 다 사용한 process의 priority를 1씩 낮추는 방법과 `priorityBoosting()`을 구현하여 global tick이 100이 될 때마다 모든 process들의 time quantum을 초기화 한 뒤 $L_0$ queue로 재조정하여 다시 처음부터 scheduling을 시작하도록 하는 방법을 사용하라고 되어 있습니다. 여기서 모든 process의 경우 $L_3$에 들어있는 process 뿐만 아니라 다른 queue에 들어 있는 process도 해당되기 때문에 전부 다 dequeue해서 $L_0$로 enqueue해야 합니다.

priority를 1씩 낮추는 code는 `scheduling()`에 구현 되어 있기 때문에 `priorityBoosting()`에서는 global tick(`ticks`)이 100이 되었을 때 모든 process들을 dequeue해서 그 process들의 time quantum을 초기화하고 $L_0$에 enqueue하는 역할만 하면 됩니다.

code에 대해 설명을 하면, ptable에 대한 lock을 획득한 뒤 ptable에 있는 모든 process들을 탐색하면서 $L_0$부터 $L_3$까지의 queue에 들어 있는 모든 process들을 dequeue합니다. 그 다음 $L_0$에 enqueue한 뒤, process가 들어있는 queuelevel과 process의 timequantum을 0으로 설정하고 ptable에 대한 lock을 해제하는 흐름으로 구성되어 있습니다.

`priorityBoosting()` 구현이 끝났다면, 이 함수가 동작할 조건을 구현해야 합니다. 명세에서는 global tick이 100 ticks가 될 때마다 이 함수를 호출하도록 명시했기 때문에 **timer interrupt**에 대해 다루는 부분을 찾아야 합니다.

```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      if(ticks % 100 == 0)
        priorityBoosting();
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE+1:
```

`trap.c`를 보면 **`T_IRQ0 + IRQ_TIMER`** 부분을 찾을 수 있습니다. 이 case의 의미는 일정 tick 간격 마다 timer interrupt를 발생시키는 부분입니다. 따라서 이 부분에 `priorityBoosting()`을 사용하면 되겠다고 생각하여 ticks가 100이 되었을 때 `priorityBoosting()`을 호출하도록 구현하였습니다.

```
103   // Force process to give up CPU on clock tick.
104   // If interrupts were on while locks held, would need to check nlock.
105   if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
106     yield();
107
```

Trap.c의 아래쪽을 보면 priorityBoosting()을 호출했던 **T_IRQ0 + IRQ_TIMER** 를 호출하는 부분이 있습니다. Process가 현재 실행 중인 process (myproc())이고, 그 process가 **RUNNING** 상태이며 trapframe에서의 trapno가 timer interrrupt인 경우 yield() system call을 호출하는 code 입니다. 여기서 yield()에 대해 되짚어 보면, process가 실행할 필요가 없을 때 (교수님 표현으로는 "*실행 의지가 없을 때*") process가 자발적으로 CPU 제어를 포기해서 Scheduler가 다른 process에게 CPU 제어권을 할당해주는 함수입니다. 제가 생각한 바로는 단순히 timer interrupt가 발생했을 때 계속 yield()를 호출하기만 하면 안된다고 생각했습니다. 그 이유는 $L_0$부터 $L_3$까지 적용된 scheduling algorithm도 다를 뿐더러 yield()를 호출해야 하는 조건도 다르기 때문입니다. 예를 들어, Round Robin이 적용된 queue의 경우는 queue의 timequantum이 다 끝나면 넘겨주면 되지만, FCFS가 적용된 MoQ의 경우 한번 CPU를 잡으면 burst time이 끝날때까지 CPU를 사용하기 때문에 조건문을 통해 yield()를 호출하는 timing을 변경해야 한다고 생각해서 새로운 함수 yield1()과 proc_yield()를 구현하여 trap.c에서 yield() 대신 proc_yield()를 호출하도록 하였습니다.

```
105   // Force process to give up CPU on clock tick.
106   // If interrupts were on while locks held, would need to check nlock.
107   if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
108     proc_yield();
109
```

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

void
yield1(void)
{
  struct proc *p = myproc();

  p->state = RUNNABLE;

  int level = p->queueLevel;
  p->queueLevel = -1;
  Dequeue(&SchedQ[level], p);

  if (level == 0){
    if (p->pid % 2 == 0){
      p->queueLevel = 2;
      Enqueue(&SchedQ[2], p); // L2 queue로 이동
    }else{
      p->queueLevel = 1;
      Enqueue(&SchedQ[1], p); // L1 queue로 이동
    }
  }else if (level == 1 || level == 2){
    p->queueLevel = 3;
    Enqueue(&SchedQ[3], p);
  }else if (level == 3){
    if (p->priority > 0)
      p->priority--;

    p->queueLevel = 3;
    Enqueue(&SchedQ[3], p);
  }
  p->timeQuantum = 0;
  sched();
}
```

`yield()`의 경우, 기존에 구현되어 있는 그대로입니다. 새로 추가한 `yield1()`의 경우, 현재 process를 불러와서 **RUNNABLE** state로 바꿔준 뒤 MLFQ에서 dequeue해줍니다. `yield1()`의 역할은 process가 queue에 할당되어 있는 timequantum 내에 실행을 끝마치지 못할 경우 다음 process에게 CPU를 넘기고 queue를 빠져나와 같은 level의 queue로 enqueue할지 다음 level로 enqueue할지 결정해주는 역할입니다. `yield()`에서 state를 바꿔주는 부분과 `sched()` 사이에 queue 이동 조건들을 추가해준 code입니다.
자세히 설명을 하자면, queue에서 dequeue한 뒤 level이 0일때, 1&2일 때, 3일때 3가지로 나눠서 0일때는 pid값에 따라, 1과 2일때는 $L_3$로, 3일 때는 priority 값을 1만큼 낮춰서 다시 같은 queue로 enqueue하도록 하였습니다. 이를 통해 round-robin을 보장할 수 있고, 더 나아가 한 process가 너무 오래 CPU를 잡고 있어서 뒤에 있는 process가 오랜 기간 동안 대기만 하는 **starvation** 문제도 일부 해결할 수 있습니다.

```
void
proc_yield(void)
{
  acquire(&ptable.lock);
  struct proc* p = myproc();
  if(monopoly){ // monopolize가 호출된 경우
    yield2(); // 먼저 들어온 순서대로 scheduling
  }else{
    p->timeQuantum++; // 1tick 마다 process의 timequantum을 늘려줌
    if(p->timeQuantum >= MLFQ[p->queueLevel].timequantum){
      //process가 timequantum 내에 끝나지 않은 경우
      yield1(); // 0queue 이동 + yield
    }
  }
  release(&ptable.lock);
}
```

`proc_yield()`에서는 `yield1()`과 `yield2()`를 호출하는 조건을 구현하였습니다. 현재 process를 불러와서 이 process가 독점상태인 경우 yield를 해주게 하였고, 독점상태가 아니여서 MLFQ에 존재하는 경우 1 tick마다 process의 timequantum 값을 늘려주면서 만약 이 process가 queue의 timequantum보다 커졌는데도 process가 끝나지 않은 경우 `yield1()`으로 가서 어느 queue로 보내고 CPU 점유를 yield할지 결정하도록 하였습니다.

# MoQ (Monopoly Queue)

## Function

1. `int setmonopoly(int pid, int password)`

> **< 명세 >**
> - 특정 pid를 가진 프로세스를 MoQ로 이동합니다. 인자로 독점 자격을 증명할 암호(자신의 학번)을 받습니다.
> - 암호가 일치할 경우, MoQ의 크기를 반환합니다.
> - 존재하지 않는 프로세스의 pid인 경우 -1을 반환합니다.
> - 암호가 일치하지 않는 경우 -2를 반환합니다.
> - 이미 MoQ에 존재하는 프로세스인 경우 -3을 반환합니다.
> - 자기 자신을 MoQ로 이동시키려 하는 경우 -4를 반환합니다.

```
int
setmonopoly(int pid, int password)
{
  if (password != 2021088304){
    return -2;
  }

  acquire(&ptable.lock);

  struct proc *p;

  int found = 0;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p->pid == pid){
      if (p->inMoQ){ // 이미 MoQ에 속한 프로세스인 경우
        release(&ptable.lock);
        return -3;
```

```
      }
      if(p->pid == myproc()->pid){
        release(&ptable.lock);
        return -4;
      }

      p->original_ql = p->queueLevel; // unmonopolize 호출시 돌아오기 위해 복사해놓음
      p->queueLevel = -1;
      Dequeue(&SchedQ[p->original_ql], p); // 현재 프로세스가 속한 queue에서 제거

      p->inMoQ = 1;
      p->queueLevel = 99;
      Enqueue(&SchedQ[4], p); // MoQ로 이동

      found = 1;
      break;
    }
  }

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

`setmonopoly()`의 경우 명세만 봐도 알 수 있듯이 조금 복잡합니다. 먼저 인자로 받은 `password`가 학번 (**2021088304**)과 일치하지 않는 경우 -2를 return하게 합니다. 그 뒤 ptable을 순회하면서 ptable에 들어 있는 process 중 인자로 받은 pid 값과 일치하는 process를 찾아서 그 process를 if문 안으로 들어가 처리를 해줍니다.

pid가 일치하는 process를 찾았다면 명세에 맞게 알맞은 값을 return하도록 해주어야 합니다. 먼저 process가 이미 MoQ에 들어가 있는 경우는 lock을 해제하고 -3을 return하게 하였고, 자기 자신을 MoQ로 이동시키려는 경우는 -4, MoQ에 없는 경우는 MLFQ에 있는 process의 `queuelevel`을 `original_ql`에 복사해서 `unmonopolize()`가 호출되었을 때 다시 지금 있는 queue로 돌아올 수 있게 한 뒤 dequeue해서 MoQ에 enqueue한 다음 found 변수를 1로 설정하였습니다. 그 이유는 MoQ에 넣는 process가 사용자의 요청에 따라 CPU 자원을 길게 사용해야 하는 process이므로 요청을 받게 되면 이 system call을 통해 MoQ로 이동해야 하기 때문입니다. 그 뒤, 반복문을 빠져나와서 MoQ에 process가 없으면 -1을 return하게 하고 process가 1개라도 존재한다면 MoQSize를 return하게 하였습니다..

2. **`void monopolize()`**

> **< 명세 >**
> - MoQ의 프로세스가 CPU를 독점하여 사용하도록 설정합니다.

```
void
monopolize(void)
{
  acquire(&ptable.lock);
  if (!monopoly){ // 독점 상태가 아닌 경우
    monopoly = 1; // CPU 독점
  }
  release(&ptable.lock);
}
```

`monopolize()`는 process가 Monopoly Queue에 존재하는데 MoQ에 존재하는 현재 process가 독점 상태가 아닌 경우 CPU를 독점하도록 하는 함수 입니다. 현재 독점 상태가 아닌 경우 (`monopoly=0`) process가 독점 상태가 되었다는 의미로 monopoly 값을 1로 바꿔주고 lock을 해제 합니다.

3. **`void unmonopolize()`**

> **< 명세 >**
> - 독점적 스케줄링을 중지하고 기존의 MLFQ part로 돌아갑니다.

```
extern uint ticks;

// Stop monopolizing the CPU and return to the original MLFQ part.
void
unmonopolize(void)
{
```

```
  acquire(&ptable.lock);
  monopoly = 0;
  // struct proc *p = SchedQ[4].front;
  for(struct proc *p=SchedQ[4].front; p!=0; p=p->next){
    // 모든 process를 MLFQ part로 옮김
    p->queueLevel = -1;
    Dequeue(&SchedQ[4], p);

    p->inMoQ = 0;
    p->queueLevel = p->original_ql;
    Enqueue(&SchedQ[p->original_ql], p); // 복사해놓았던 MLFQ level로 이동
  }
  ticks = 0; // reset global ticks
  release(&ptable.lock);
}
```

`unmonopolize()`는 monopoly 상태인 경우만 함수를 돌도록 하였습니다. 먼저 `unmonopolize()`가 호출 되고 나서 global ticks(`ticks`)를 0으로 초기화하기 위해 `trap.c`에서 사용한 `uint` type 전역변수 `ticks`를 불러와서 `extern uint ticks;` 선언을 해주었습니다. 그 다음 `unmonopolize()`에서는 임시 변수 `tmp`에 MoQ의 front 위치에 있는 process를 복사한 뒤 MoQ를 순회하면서 `tmp`를 dequeue한 뒤, 원래 있던 MLFQ part로 enqueue하는 과정을 반복하고 나서 제일 마지막에 독점 여부를 나타내는 `monopoly`를 0으로 설정하고 global ticks를 0으로 선언하여 독점상태가 해제되었음을 나타내는 system call 함수입니다.

# System Call

`sysproc.c` 파일에 system call 등록을 위한 커널 함수를 작성했습니다.

```
int
sys_yield(void)
{
  yield();
  return 0;
}

int
sys_yield1(void)
{
  yield1();
  return 0;
}

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

int
sys_setpriority(void)
{
  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) != 2021088304)
    return -1;
  return setmonopoly(pid);
```

```
}

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

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

그 다음 system call을 등록하는 과정을 Lab과 첫 번째 project에서 진행했던 방법과 동일하게 진행하면 됩니다.

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _my_userapp\
    _project01\
    _user_int\
    _test_cprintf\
    _mlfq_test\
```

마지막으로 `MakeFile`에 `test file`을 추가해줍니다.

---

# 3. Result

🔥 **Booting**

```
[1]
./makexv6.sh
./bootxv6.sh
----------------------
make clean
make
make fs.img
qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512

[2]
$mlfq_test
```

Booting의 경우는 제가 만들어 놓은 `makexv6.sh`와 `bootxv6.sh` 파일을 컴파일 하거나 xv6를 부팅하는 명령어로 시작하면 됩니다. 참고로 `makexv6.sh`는 make와 관련된 명령어로 구성되어 있고 `bootxv6.sh`의 경우 xv6를 부팅하는 명령어가 들어 있습니다.
부팅이 되고 shell 입력창이 뜨면
`mlfq_test` 를 입력하면 program이 실행됩니다.

✅ **Printed Result**

- Test1

```
$ mlfq_test
MLFQ test start
[Test 1] default
Process 5
L0: 13990
L1: 28469
L2: 0
L3: 57541
MoQ: 0
Process 7
L0: 14277
L1: 29091
L2: 0
L3: 56632
MoQ: 0
Process 9
L0: 16435
L1: 30283
L2: 0
L3: 53282
MoQ: 0
Process 11
L0: 14509
L1: 29958
L2: 0
L3: 55533
MoQ: 0
```

```
Process 10
L0: 14563
L1: 0
L2: 49626
L3: 35811
ProcMoQ: 0
ess 8
L0: 15675
L1: 0
L2: 50146
L3: 34179
MoQ: 0
Process 4
L0: 16678
L1: 0
L2: 50632
L3: 32690
MoQ: 0
Process 6
L0: 15700
L1: 0
L2: 51622
L3: 32678
MoQ: 0
[Test 1] finished
```

- Test2

```
[Test 2] priorities
Process 19
L0: 14685
L1: 29102
L2: 0
L3: 56213
MoQ: 0
Process 17
L0: 13950
L1: 30104
L2: 0
L3: 55946
MoQ: 0
Process 18
L0: 13265
L1: 0
L2: 46389
L3: 40346
MoQ: 0
Process 15
L0: 15287
L1: 31000
L2: 0
L3: 53713
MoQ: 0
```

```
Process 16
L0: 15300
L1: 0
L2: 49057
L3: 35643
MoQ: 0
Process 13
L0: 16258
L1: 34369
L2: 0
L3: 49373
MoQ: 0
Process 14
L0: 14192
L1: 0
L2: 50544
L3: 35264
MoQ: 0
Process 12
L0: 7997
L1: 0
L2: 30048
L3: 61955
MoQ: 0
[Test 2] finished
```

- Test3

```
[Test 3] sleep
Process 20
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 21
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 22
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 23
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
```

```
Process 24
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 25
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 26
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 27
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
[Test 3] finished
```

- Test4

```
[Test 4] MoQ
Number of processes in MoQ: -1
setmonopoly returned -1
$ Process 29
L0: 3141
L1: 7970
L2: 0
L3: 88889
MoQ: 0
Process 28
L0: 5518
L1: 0
L2: 14128
L3: 80354
MoQ: 0
zombie!
zombie!
```

test4를 보면 MoQ에 process가 없다는 의미인 -1이 출력됩니다.

*”Test 4 의 경우 구현에 따라 결과값이 나오지 않을 수 있습니다”.*
이 문구의 의미가 이런 경우를 의미하는지에 대한 기준을 정확히 모르겠지만 제 구현방식으로는 부분부분 code를 수정해도 MoQ에 있는 process의 개수가 -1개라고만 뜹니다.

이 부분은 해결하지 못하였습니다.

또한 매번 이렇게 출력되는 것이 아니라 Test1에서는 process끼리 겹쳐서 출력이 되거나 Test2, Test3에서 무한루프 때문인지 다른 이유에서인지 출력이 더 이상 안되는 경우도 존재합니다.

# 4. Trouble Shooting

```c
// 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)

  int queueLevel;       // The level of the queue to which the process belongs
  int priority;         // Process priority
  int timeQuantum;      // Process time quantum
  int scheduler_locked; // Variable to keep track of scheduler lock
  struct proc *next;    // Next process in the queue
};
```

```c
// queue 구조체
struct queue
{
  struct spinlock lock;
  struct proc *front;
  struct proc *rear;
  int timequantum; // 2*i+2
  int level; // queuelevel
  int size;
};

struct queue MLFQ[4];
struct queue MoQ;
```

- 원래는 queueLevel부터 next 포인터까지 5개만 추가로 선언을 했었는데, `getlev()`와 `setmonopoly()`에서 MoQ에 존재하는 프로세스인지에 대한 여부가 조건이 되는 부분이 있어서 추가적으로 `inMoQ` 변수를 추가해서 inMoQ가 1이면 MoQ에 존재하는 process이고 MoQ의 모든 프로세스가 종료되어 `unmonopolize()` 시스템콜이 호출되고 MLFQ로 돌아가게 될 때 0으로 설정해서 MoQ에 존재하지 않는 프로세스임을 나타내주었습니다. 그리고 `monopolize()`에 의해 process가 CPU를 독점하고 `unmonopolize()`에 의해 독점을 해제할 때 독점 여부를 표시할 변수가 필요할 것 같아서 `monopoly` 변수도 추가하였습니다.
- `queue 구조체`의 경우 원래 5개의 포인터와 변수로 설정을 했었는데, `isFull()`을 구현하는 과정에서 `size` 변수가 필요하다고 판단해서 `size` 변수를 추가하였습니다.

---

```
trap.c: In function ‘trap’:
trap.c:60:9: error: implicit declaration of function ‘priorityBoosting’ [-Werror=implicit-function-declaration]
   60 |         priorityBoosting(); // 100 ticks 마다 PriorityBoosting을 수행
      |         ^~~~~~~~~~~~~~~~~
cc1: all warnings being treated as errors
make: *** [<builtin>: trap.o] Error 1
```

priorityBoosting을 구현할 때, 구현한 뒤 바로 build를 하면 위 이미지와 같이 구현한 함수를 인식하지 못하는 문제가 발생합니다. 그 이유는 `proc.c`에서 구현한 함수를 `trap.c`에서 불러오려고 했기 때문입니다. 이를 해결하기 위해서는 header 파일에 `priorityBoosting()`을 선언해두어야 합니다. 따라서 `defs.h`와 `user.h` 두 파일에 선언을 하였습니다.

```
문제  출력  디버그 콘솔  터미널  포트

objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0898654 s, 57.0 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.000573827 s, 892 kB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
451+1 records in
451+1 records out
231408 bytes (231 kB, 226 KiB) copied, 0.00455766 s, 50.8 MB/s
hjpark@ubuntu:~/Desktop/xv6-public$
```

선언 후 build에 문제가 없는 것을 확인할 수 있었습니다.

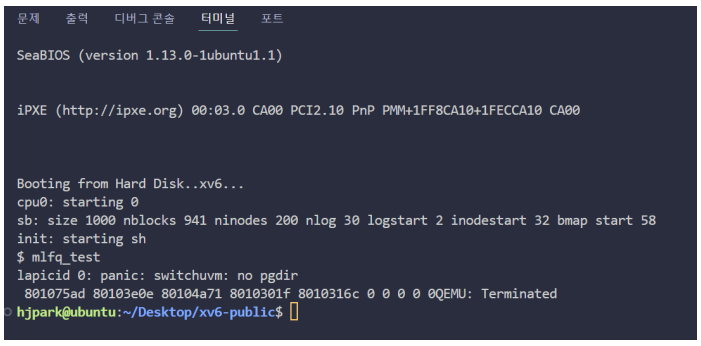
```
문제  1  출력  디버그 콘솔  터미널  포트

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
lapicid 0: panic: acquire
 80104f31 80103948 801048b4 8010661c 801062a8 8010301f 8010316c 0 0 0
```

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ mlfq_test
MLFQ test start
[Test 1] default
Process 7
L0: 100000
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 6
L0: 100000
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 10
L0: 100000
L1: 0
L2: 0
L3Process 11
L0: 100000
L1: 0
L2: 0
L3: 0
Process 4
L0: 100000
L1: 0
L: 0
MoQ: 0
MoQ: 0
2: 0
L3: 0
MoQ: 0
Process 8
L0: 100000
```

xv6를 부팅하였을 때 `lapicid 0 : panic : acquire` 라는 문구가 뜨면서 부팅이 제대로 이뤄지지 않은 것을 확인할 수 있었습니다. 찾아보니, lock을 걸을 때 acquire / release 함수로 선언을 하는데 이 부분에서 문제가 발생한 것 같았습니다. 그래서 scheduler()와 trap.c에

서 priorityBoosting을 호출한 부분을 백업해두고, 디버깅을 위해 L0 queue 부분만 구현해서 부팅을 해보았습니다.
부팅이 잘 된 것 같은 모습을 확인할 수 있었습니다.

---

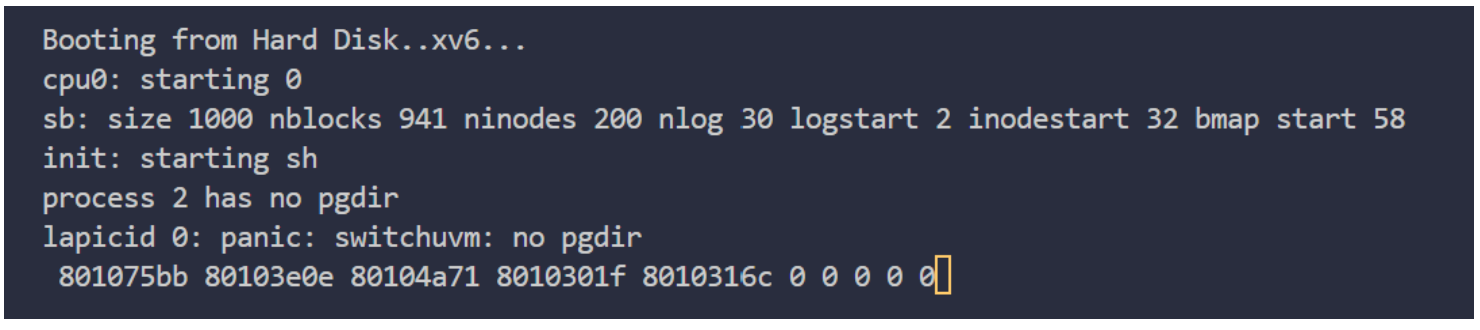



queue 간의 이동을 구현하기 위해 C언어의 goto 문법으로 구현을 하였는데, 구현 뒤 xv6 부팅까지는 되었지만, mlfq_test 파일을 실행하였을 때 **no pgdir** 이라는 error가 떴습니다.

```
// Switch TSS and h/w page table to correspond to process p.
void
switchuvm(struct proc *p)
{
  if(p == 0)
    panic("switchuvm: no process");
  if(p->kstack == 0)
    panic("switchuvm: no kstack");
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");

  pushcli();
  mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
                                sizeof(mycpu()->ts)-1, 0);
  mycpu()->gdt[SEG_TSS].s = 0;
  mycpu()->ts.ss0 = SEG_KDATA << 3;
  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
  // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
  // forbids I/O instructions (e.g., inb and outb) from user space
  mycpu()->ts.iomb = (ushort) 0xFFFF;
  ltr(SEG_TSS << 3);
  lcr3(V2P(p->pgdir));  // switch to process's address space
  popcli();
}
```

switchuvm을 보니 `p->pgdir==0` 이 부분에서 문제가 생겼다는 것을 알 수 있었습니다.

```
if(p->pgdir == 0){
        cprintf("process %d has no pgdir\n", p->pid);
    panic("switchuvm: no pgdir");
}
```




디버깅을 해보았더니 process 2가 pgdir를 갖고 있지 않다는 것을 확인할 수 있었습니다.

---

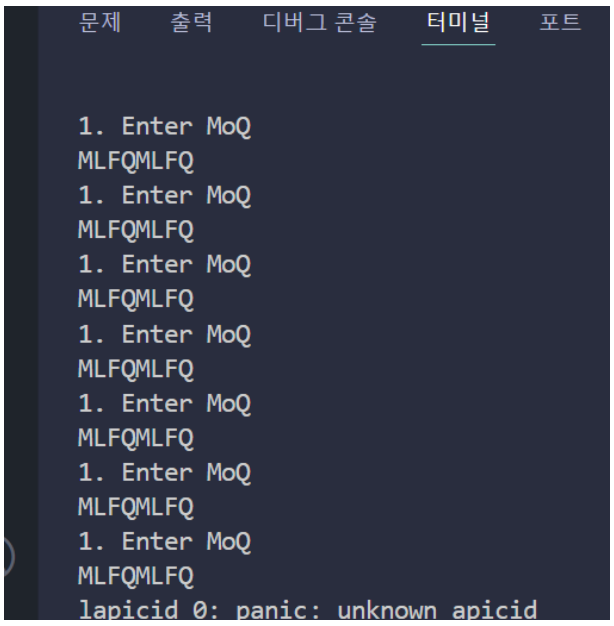
```
void scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
```

```
  for (int i = 0; i < 4; i++){
    MLFQ[i].front = MLFQ[i].rear = 0;
    MLFQ[i].timequantum = 2 * i + 2;
    MLFQ[i].level = i;
  }

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

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

    cprintf("1. Enter MoQ\n");
    if (MoQ.front != 0 && monopoly){
      while(MoQ.front->state != RUNNABLE){
        p = MoQ.front;
        if (p->state == RUNNABLE){
          cprintf("2. before MoQ Sched\n");
          scheduling(c, p);
          cprintf("3. after MoQ Sched\n");

        if (p->state == ZOMBIE)
          unmonopolize();
      }
    }
  }
  cprintf("MLFQMLFQ\n");
  for(int i=0; i<3; i++){
    while(MLFQ[i].size > 0){
      cprintf("4. Enter MLFQ\n");
      cprintf("Process name = %s\n", &MLFQ[i].front->name);
      p = MLFQ[i].front;
      if(p->state != RUNNABLE){
        cprintf("5. MLFQ start\n");
        if(p->state==UNUSED || p->queueLevel != MLFQ[i].level){
          Dequeue(&MLFQ[i], p);
        } else if(p->state==SLEEPING){
          Dequeue(&MLFQ[i], p);
          Enqueue(&MLFQ[i], p);
        }
      }else if(p->state == RUNNABLE){
        cprintf("6. before MLFQ Sched\n");
        scheduling(c, p);
        cprintf("7. after MLFQ Sched\n");
...
```

Scheduler 함수에서 동작을 할 때 어느 조건문도 통과를 하지 못하기에 debugging을 위한 cprinft 함수 몇줄을 추가하여 어디서 막히는지 확인을 해보았습니다. 그랬더니 scheduler 함수 처음에 MLFQ 배열에 대한 초기화를 진행하였는데, 어느 조건문도 통과하지 못하는 현상이 나타났습니다.

```
Booting from Hard Disk..xv6...
1, initcode
cpu0: starting 0
1. Enter MoQ
MLFQMLFQ
4. Enter MLFQ
Process name = lapicid 0: panic: acquire
 80104fa1 801007c2 80106685 80106348 80103f67 8010301f 8010316c 0 0 0
```

원인을 찾기 위해 queue의 front와 rear 포인터를 초기화한 부분을 주석처리 후 부팅을 해보았습니다.

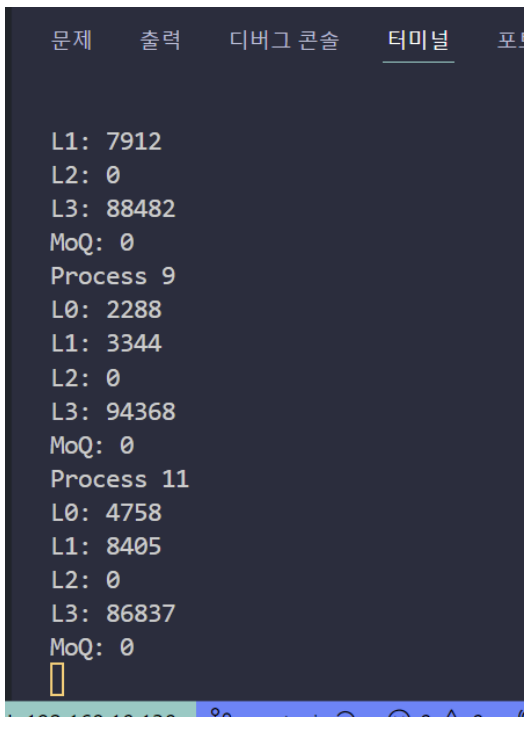



그랬더니 무한 루프를 돌다가 멈추는 현상이 나타났습니다. 이 때 원인을 생각한 점은 `initproc()`과 `fork()`에서 생성된 process를 L0 큐에 enqueue를 하면서 `Enqueue()`에서 front와 rear 포인터를 update 해주었는데 여기서 다시 0으로 초기화를 하였기 때문에 queue에는 process가 들어있음에도 process를 가리키는 포인터가 엉뚱한 위치를 가리키고 있어서 이러한 문제가 생겼다고 생각을 해서 초기화 하는 부분을 `initproc()`에서 enqueue하기 전 위치로 이동시켰습니다.

```
  p->queueLevel = -1;  // enqueue되기 전까지는 queueLevel은 -1
  p->priority = 0; // 처음 process가 실행될 때 priority는 0
  p->timeQuantum = 0; // 처음 process가 실행될 때 timeQuantum은 0
  p->inMoQ = 0; // 처음엔 MoQ에 들어있지 않음
```

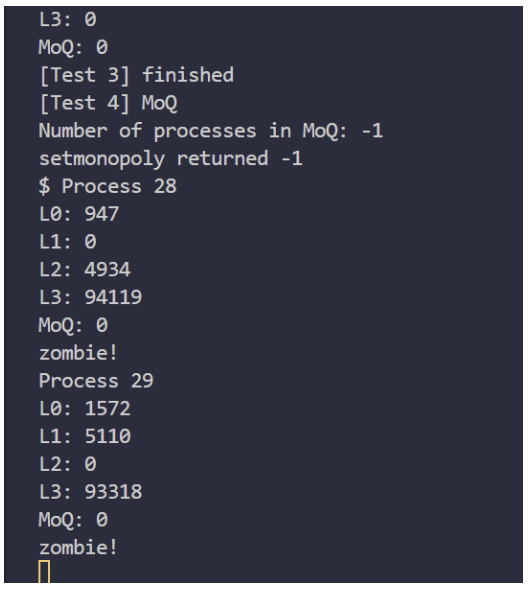
처음 initialize를 할 때 process의 queuelevel을 0으로 초기화했었는데, 그렇게 되면 L0 queue와 중복이 되기 때문에 -1로 변경을 하였습니다. 그러나 이렇게 되면 queue의 level은 index와 같기 때문에 index로 초기화한 반면 process는 -1로 초기화 되었기 때문에 enqueue와 dequeue를 하면서 queue에 들어있는 process의 개수(`q→size`)를 조절할 때 두 level이 달라서 (`p→queuelevel != q→level`) size 값이 늘어나지 않는 문제가 생겼습니다. 문제해결을 위해 enqueue와 dequeue 함수에서 `q→size` 를 늘리고 줄이고 하지 않고 `proc.c`의 여러 함수들에서 `Enqueue()`와 `Dequeue()`를 호출할 때마다 `p→queuelevel` 을 아래의 code 일부분과 같이 정수값으로 선언하여 변수로 인해 꼬이는 일이 발생할 여지를 없앴습니다. (`q→level` 은 선언된 뒤 변하지 않기 때문)

```
...
if(level==0){
  if (p->pid % 2 == 0){
    p->queueLevel=2; // process의 queuelevel을 2로 선언한 뒤
    Enqueue(&MLFQ[2], p); // L2 queue로 이동
  }else{
    p->queueLevel=1; // process의 queuelevel을 1로 선언한 뒤
    Enqueue(&MLFQ[1], p); // L1 queue로 이동
  }
...
```



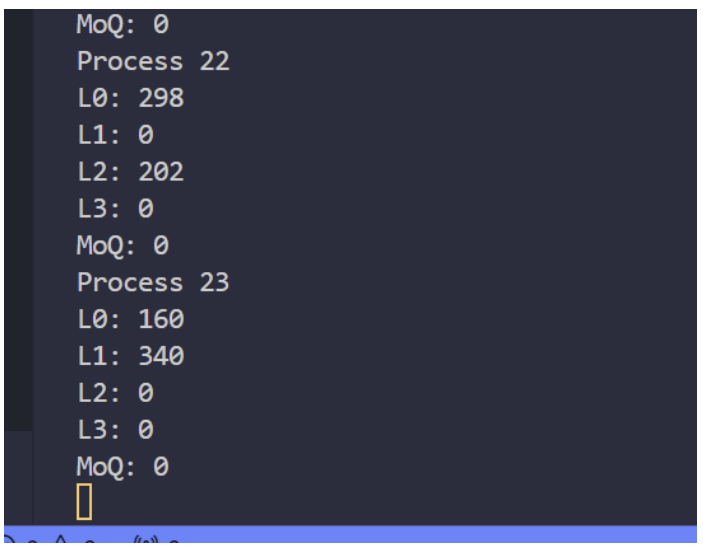
Queue간 이동하는 부분은 잘 구현된 것 처럼 보이지만, test file에서 지정한 **NUM_THREAD 8**개 중에서 7개만 통과되고 무한루프를 돌고 있는 것을 확인할 수 있었습니다.

---




Test 4에서 zombie가 뜨는 현상이 계속 발생했습니다. 어떠한 이유에서인지는 잘 모르겠지만 MoQ를 따로 선언했을 때와 MoQ를 MLFQ 배열에 한꺼번에 추가했을 때 모두 MoQ에 enqueue가 되지 않는 모습을 보입니다. 제가 의문인점은 MLFQ에 enqueue와 dequeue는 제대로 되는데 MoQ만은 제대로 되지 않는다는 것입니다. `Setmonopoly()`에 문제가 있는 듯 싶었지만, 어떻게 수정을 하던지 해결을 못해서 이 부분은 시간이 더 필요할 것 같습니다. Test4 출력부분에서도 언급했지만 매번 Test4까지 출력되지 않고 멈추는 경우도 존재해서 여러번 `mlfq_test` program을 실행시켜봐야 확인이 가능합니다.

---




mlfq_test 파일 실행 도중 test3에서 L0가 아니라 다른 queue에 porcess가 enqueue되는 경우에도 무한루프를 돌고 있는 것으로 추정이 됩니다. 이 부분에 대한 내용도 원인을 찾지 못해서 해결하지 못하였습니다.