[ proc.h ]

```
struct proc {
  // PRIORITY(Circular linked list)
  struct proc *next;           // Next process in queue
  struct proc *prev;           // Prev process in queue
  struct queue_proc *now;       // Process 's current queue.
  // PRIORITY
  int priority;                 // Process 's priority
  // MLFQ
  int level;                   // Process 's level.
  int ticks;                   // Process 's current tick.

  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)
};
```

우선순위 스케줄링(이하 PR)을 위해서 proc 구조체 안에 원형 연결리스트를 위한 이전, 다음 프로세스 포인터, 프로세스의 현재 큐를 나타내는 현재 포인터 추가했고. PR에서 사용 할 프로세스 우선순위 변수 추가함.

MLFQ 스케줄링(이하 MLFQ)을 위해 프로세스의 레벨, 현재 틱을 나타낼 변수들을 추가.

[ proc.c ]

```
#define PRIORITY_SCHED


struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;

struct {
#ifdef PRIORITY_SCHED
  struct queue_proc queue_arr[NPROC];
#else
  struct queue_proc queue_arr[3];
#endif
  struct queue_proc * head_q;
  struct queue_proc * now_q;
  struct queue_proc * tail;
  int num_q;
} pqueue;
```

pqueue 구조체에서 PR할 경우 64 크기의 큐, MLFQ 또는 기본 RR 스케줄링 사용 시 3 크기의 큐를 생성하도록 전처리함.

```
//Initiate queue and return that queue.
struct queue_proc *
q_init(int priority)
{
  struct queue_proc * q;
#ifdef PRIORITY_SCHED
  for(q = pqueue.queue_arr; q < &pqueue.queue_arr[NPROC]; q++)
#else
  for(q = pqueue.queue_arr; q < &pqueue.queue_arr[3]; q++)
#endif
    if(q->state == UNUSED_Q) {
      q->state = DELETABLE;
      q->next = q->prev = 0; q->head = q->tail = 0; q->num_proc = q->num_runnable = 0;
      q->priority = priority;
      pqueue.num_q++;
      return q;
    }
  panic("QUEUE is over allocated\n");
  return 0;
}
```

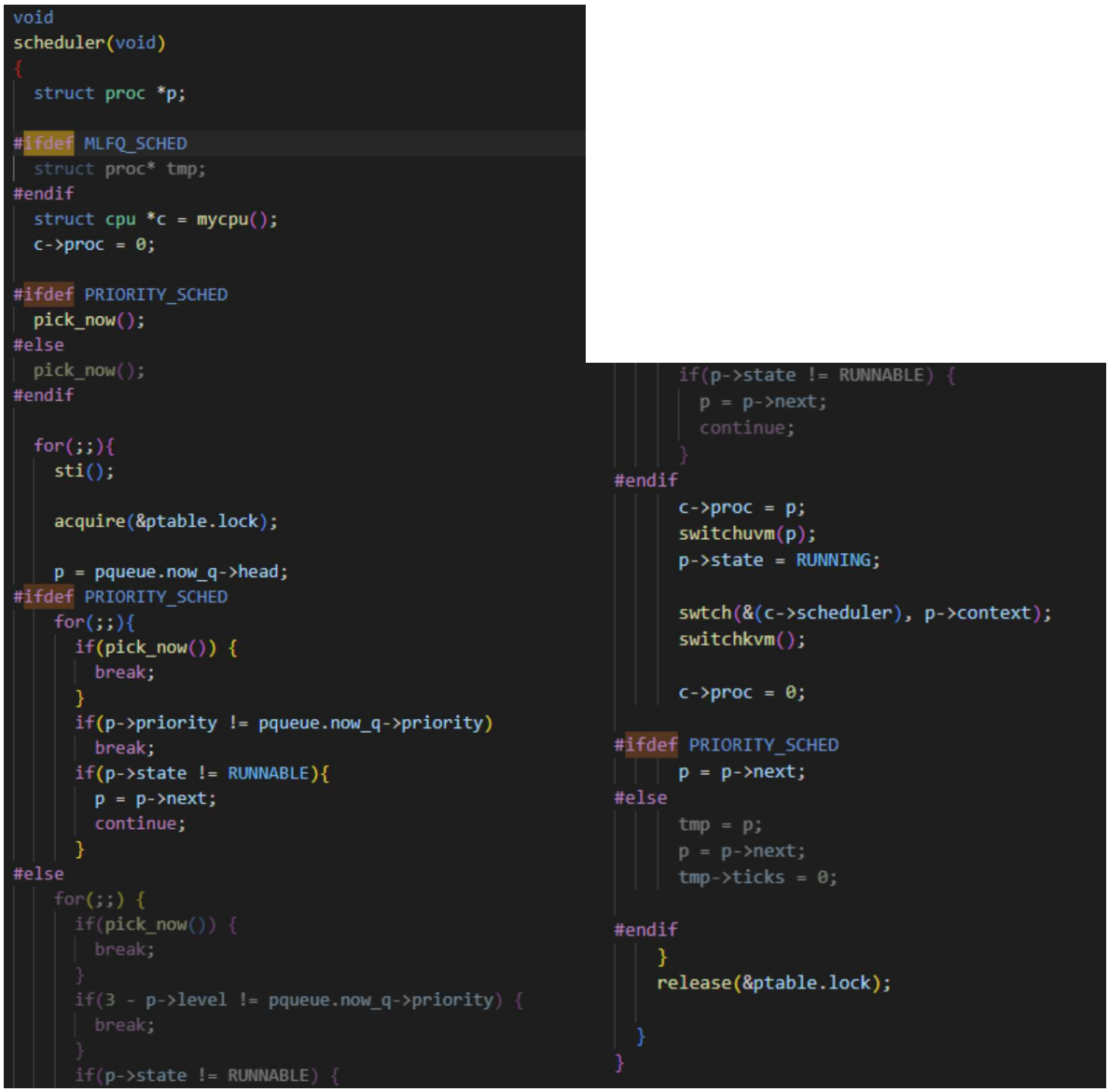
초기화 함수 역시 전처리 선언에 따라 크기에 알맞게 전처리하도록 수정.

```
int
insert(struct proc * p)
{
#ifdef MLFQ_SCHED
  if(p->level > 2 || p->level < 0)
    panic("MLFQ Unknown queue level\n");

  insert_proc(&pqueue.queue_arr[p->level], p);
  p->now = &pqueue.queue_arr[p->level];

  return 0;
#else
  struct queue_proc *q;
  if(!(q = find_q(p->priority))) {
    q = q_init(p->priority);
    insert_queue(q);
  }
  insert_proc(q, p);
  p->now = q;
  return 0;
#endif
}

int
delete(struct proc * p)
{
#ifdef MLFQ_SCHED
  if(p->level > 2 || p->level < 0)
    panic("MLFQ Unkown queue level\n");

  delete_proc(&pqueue.queue_arr[p->level], p);
  p->now = 0;
  return 0;
#else
  struct queue_proc *q = p->now;
  delete_proc(q, p);
  p->now = 0;
  if(q->state == DELETABLE) {
    delete_queue(q);
  }
  return 0;
#endif
}
```

삽입 및 삭제 함수에서 MLFQ의 경우 레벨을 체크하여 삽입 및 현재 큐를 갱신하고, PR의 경우 우선순위를 찾아 갱신 여부 확인 후, 갱신 및 삽입 또는 그냥 삽입이 이루어짐.

```
int
setpriority(int pid, int n)
{
#ifdef PRIORITY_SCHED
  struct proc * p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->pid == pid)
      break;

  acquire(&ptable.lock);
  delete(p);
  p->priority = n;
  insert(p);
  release(&ptable.lock);
  return p->priority;
#endif
  return 0;
}
```

PR을 위한 setpriority 함수를 따로 추가하여 lock을 이용하여 우선순위를 지정해줌.



```
void
scheduler(void)
{
  struct proc *p;

#ifdef MLFQ_SCHED
  struct proc* tmp;
#endif
  struct cpu *c = mycpu();
  c->proc = 0;

#ifdef PRIORITY_SCHED
  pick_now();
#else
  pick_now();
#endif

  for(;;){
    sti();

    acquire(&ptable.lock);

    p = pqueue.now_q->head;
#ifdef PRIORITY_SCHED
    for(;;){
      if(pick_now()) {
        break;
      }
      if(p->priority != pqueue.now_q->priority)
        break;
      if(p->state != RUNNABLE){
        p = p->next;
        continue;
      }
#else
    for(;;) {
      if(pick_now()) {
        break;
      }
      if(3 - p->level != pqueue.now_q->priority) {
        break;
      }
      if(p->state != RUNNABLE) {
        p = p->next;
        continue;
      }
#endif
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

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

      c->proc = 0;

#ifdef PRIORITY_SCHED
      p = p->next;
#else
      tmp = p;
      p = p->next;
      tmp->ticks = 0;

#endif
    }
    release(&ptable.lock);

  }
}
```

scheduler 함수에서 역시 MLFQ, PR 각각에 맞는 방식으로 우선순위 비교 및 레벨 비교, RUNNABLE한 상태인지에 따라서 순회하고 스위칭함.

[ trap.h ]

```
#ifdef MLFQ_SCHED
  curticks++;
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER){
    int tmp = (myproc()->ticks)++;
    if(pick_now())
      yield();
    switch(myproc()->level) {
    case 0:
      if(tmp >= 1) {
        levelup(myproc());
        yield();
      }
      break;
    case 1:
      if(tmp >= 3) {
        levelup(myproc());
        yield();
      }
      break;
    case 2:
      if(tmp >= 7)
        yield();
      break;
    }
  }
  if(curticks >= 100) {
    curticks = 0;
    boost();
    if(myproc())
      yield();
  }
#else
  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
  // For interrupt timer interrupt.
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
#endif
```

MLFQ일 경우, 현재 틱을 증가시켜 사면서 트랩 프레임의 트랩 넘버가 IRQ0 + IRQ TIMER와 같은 경우에만 프로세스의 틱 증가 및 yield 함수를 호출하고, 레벨에 따라 레벨 증가 또는 yield만 시킴. 현재 틱이 100을 넘은 순간에 현재 틱을 0으로 초기화 시켜 부스팅한다. PR의 경우 같은 조건 충족시 yield만 호출.

해당 과제에서 두 가지 스케줄링 기법에 대해서 각각 컴파일을 해야했는데, 원활한 컴파일을 위해 전처리

기능을 사용하여 매크로를 사용해 편하게 컴파일을 하였음.