project02 wiki - 2022028522

1. Design

> 📌 명세에서 요구하는 조건에 대해서 어떻게 구현할 계획인지, 어떤 자료구조와 알고리즘이 필요한지, 자신만의 디자인을 서술합니다.

어디서부터 시작해야할지 모르겠어서, 명세에서 요구하는 조건을 간단하게 하나하나씩 정리해보았습니다.

---

1. MLFQ를 구현해야한다.
    a. MLFQ는 기본적으로 Round-Robin scheduling을 따르고, 4개의 feedback queue로 구성되어있다.
    b. L0, L1, L2, L3는 각각의 time quantum을 가지고, 처음 실행된 프로세스는 모두 L0에 들어간다. 각각의 큐에서 time quantum을 모두 사용하면 time quantum을 초기화하고 하위 레벨 큐로 이동한다.
    c. 가장 하위 레벨의 큐인 L3에서는 가장 높은 priority의 프로세스가 스케줄링되어야한다. L3에서 실행된 프로세스가 time quantum을 다 쓰면, 해당 프로세스의 priority가 1 감소하고 time quantum이 초기화된다.
    d. starvation을 막기 위해 global tick이 100 ticks가 될 때마다 모든 프로세스를 L0 큐로 옮기고, 모든 프로세스의 time quantum을 초기화한다.
2. MoQ를 구현해야한다.
    a. MoQ는 먼저 큐에 들어온 프로세스가 먼저 스케줄링되는 FCFS 정책을 따른다.
    b. MoQ를 스케줄링 하는 동안 priority boosting은 발생하지 않는다.
    c. MoQ의 모든 프로세스들이 종료되면 unmonopolize 시스템콜이 호출되어 MLFQ 파트로 돌아간다.
    d. unmonopolize 시스템 콜이 호출되면 global tick은 0으로 초기화된다.
3. 다음의 시스템콜들을 구현해야한다.
    a. `void yield(void)`
    b. `int getlev(void)`
    c. `int setpriority(int pid, int priority)`
    d. `int setmonopoly(int pid, int password)`
    e. `void monopolize()`
    f. `void unmonopolize()`

과제 명세를 읽고, 우선 처음으로 `proc.h`에 있는 `proc` 구조체에서 프로세스 새로운 멤버 변수를 추가해야겠다는 생각이 들었습니다.

```
// proc.h

struct proc {
...
  // MLFQ Scheduler를 구현하기 위한 새로운 멤버 변수 선언
  int queueLevel;              // 프로세스가 어떤 큐에 있는지 저장하는 멤버 변수
  int priority;                // 프로세스의 우선순위
  int tick;                    // 프로세스의 tick
```

```
  int idx;                      // 프로세스의 index
};
```

프로세스가 어떤 큐 내에 있는지를 저장할 수 있는 `queueLevel`, 프로세스의 우선순위를 알 수 있는 `priority`, 프로세스가 time quantum을 다 썼는지의 여부를 확인하기 위한 `tick`, 그리고 각각의 프로세스를 구별하기 위한 `idx`를 새로 선언해주었습니다. 프로세스를 구분하기 위해서는 프로세스의 `pid`로 구분해주어도 되지만, 굳이 `idx`를 새로 선언해준 이유는 아래와 같습니다.

```
// proc.c

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

`proc.c`의 `ptable` 구조체를 보면, `struct proc proc[NPROC]`에 모든 프로세스들의 정보를 담고 있음을 알 수 있습니다. 그래서 프로세스의 `idx`에 그 프로세스가 `proc[NPROC]`의 몇 번째 원소인지를 알 수 있으면 구현이 쉬워질 것 같다는 생각에 `idx` 멤버 변수를 따로 선언해주었습니다.

다음으로 든 생각은 L0, L1, L2, L3, MoQ를 구현하기 위해 새로운 `queue` 구조체를 구현해야겠다는 생각이었습니다. 그래서 `proc.c`에 `queue` 구조체를 선언하였습니다.

```
// proc.c

// L0, L1, L2, L3를 구현하기 위한 queue 구조체 선언
struct queue{
  int queue[NPROC + 1];         // 프로세스들의 idx를 저장하는 배열
  int timeQuantum;              // 큐의 time quantum
  int level;                    // 큐의 level(0, 1, 2, 3)
  int size;                     // 큐의 크기
  int head;                     // queue 배열에서의 첫 프로세스 위치
  int tail;                     // queue 배열에서의 마지막 프로세스 위치
};
```

`queue` 구조체 안의 `queue` 배열에서 프로세스의 `idx`를 저장해둘 수 있습니다. `queue` 배열은 혹시 몰라 최대 `NPROC+1`개의 프로세스를 담을 수 있게 해주었으며, 순환 큐의 방식을 채택했습니다. 굳이 순환 큐로 구현한 이유는, 사실 처음에는 큐를 linked list로 구현하려고 `head`와 `tail` 멤버 변수를 만들었다가 잘 안 돼서 배열로 구현하게 된 것인데, 그에 대한 잔재입니다. `timeQuantum`과 `level`은 각각 큐의 time quantum과 level을 저장해둘 수 있는 멤버 변수인데, 필요할 것 같아서 일단은 선언해두었지만 막상 구현 단계에서는 쓰이지 않았던 것 같습니다. 마지막으로 `size`, `head`, `tail`은 각각 큐의 크기, `queue` 배열에서의 첫 번째 프로세스 위치, `queue` 배열에서의 마지막 프로세스 위치를 저장할 수 있게 해주었습니다.

그 다음으로 든 생각은, 추가된 `proc`의 멤버 변수에 맞게 초기값을 설정해주는 것과 `queue` 구조체들을 선언하는 것이었습니다.

`proc`은 처음 생성될 때 `proc.c`의 `allocproc()` 함수를 통해 생성되므로, `allocproc()` 함수를 다음과 같이 수정해주었습니다.

```
// proc.c

static struct proc*
allocproc(void)
{
...
found:
  p->state = EMBRYO;
```

```
  p->pid = nextpid++;

  p->priority = 0;                    // 프로세스의 priority의 초기값을 0으로 설정
  p->queueLevel = 0;                  // 프로세스의 queueLevel의 초기값을 0으로 설정
  p->tick = 0;                        // 프로세스의 tick의 초기값을 0으로 설정
  p->idx = p - ptable.proc;           // 프로세스의 idx의 초기값을 설정

  enqueue(&queues[0], p->idx);        // 프로세스를 L0 큐로 이동
...
}
```

프로세스의 `priority`, `queueLevel`, `tick`은 전부 초기값을 `0`으로 설정합니다. `idx`의 경우에는 `ptable`의 `proc` 배열에서 그 프로세스가 몇 번째 원소인지를 저장해두는 변수라고 했으므로, `p - ptable.proc`에 해당하는 값을 초기값을 지정해주었습니다. 이게 가능한 이유는 C 언어에서, 포인터 간의 빨셈 연산은 두 포인터가 가리키는 메모리 위치 사이의 요소 개수를 반환하기 때문입니다. `ptable.proc`는 프로세스 테이블을 나타내는 배열이며, `p`는 `ptable.proc` 배열 내의 특정 프로세스를 가리키는 포인터입니다. 따라서 `p - ptable.proc` 연산은 `p`가 가리키는 프로세스가 `ptable.proc` 배열의 시작점으로부터 얼마나 떨어져 있는지, 즉 `p`와 `ptable.proc`의 첫 번째 원소 사이에 몇 개의 원소가 있는지를 반환합니다.

마지막으로 `enqueue` 함수를 통해 처음 생성된 프로세스들은 모두 L0 큐에 옮겨주었습니다. L0 큐에 대한 자세한 구현과 `enqueue` 함수에 대해서는 후술하도록 하겠습니다.

그 다음으로는 L0, L1, L2, L3, MoQ에 해당하는 `queue`를 `proc.c`에 선언해주었습니다. 각각의 큐를 선언할 수 있지만, 저는 `queue` 구조체 5개를 저장할 수 있는 배열을 만들었습니다.

```
// proc.c

// 구조체 queue를 저장해두는 배열. 차례대로 L0, L1, L2, L3, moq
struct queue queues[5];
```

위와 같이 `queues`는 `queue` 구조체 5개를 담고 있는 배열이며, 차례대로 L0, L1, L2, L3, MoQ 큐입니다.

그 다음으로는 이 큐들의 초기값을 지정해주는 `initQueues()` 함수를 새로 만들었습니다.

```
// proc.c

void initQueues() {
  // mlfq init
  for(int i = 0; i < 4; i++) {
    queues[i].level = i;
    queues[i].head = 0;
    queues[i].tail = 0;
    queues[i].timeQuantum = (i + 1) * 2;
    queues[i].size = 0;
  }

  // moq init
  queues[4].level = 99;
  queues[4].head = 0;
  queues[4].tail = 0;
  queues[4].size = 0;
}
```

이 함수는 L0, L1, L2, L3와 MoQ에서 동작하는 방식이 다릅니다. 공통적으로는 `head`, `tail`, `size`를 전부 `0`으로 초기화하고 있지만 L0, L1, L2, L3의 `level`은 Li에 해당하는 `i`로 초기화해주고, MoQ의 `level`은 `99`로 초기화해줍니다. L0, L1, L2, L3의 경우 추가로 `timeQuantum`을 `(i + 1) * 2`로 설정해두었습니다.

이 `initQueues()` 함수가 프로그램이 실행되자마자 동작하게 하기 위해 시스템 부팅 시에 실행되는 첫 번째 C 함수인 `main.c` 파일의 `main` 함수에 `initQueues()` 함수를 삽입하였습니다.

```
// main.c

// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
...
  // 추가된 부분: MLFQ 스케줄러 큐 초기화
  initQueues();
...
}
```

다음으로는 프로세스를 큐에 넣고 뺄 수 있는 `enqueue` 함수와 `dequeue` 함수가 필요하겠다는 생각이 들어, `enqueue`와 `dequeue` 함수를 `proc.c` 에서 구현하였습니다.

```
// proc.c

// 큐의 맨 끝에다가 프로세스 집어넣기
void enqueue(struct queue *q, int p){
  if(q->size == NPROC){
    // cprintf("ERROR: cannot enqueue because the queue is full!");
    return;
  }

  q->queue[q->tail] = p;
  q->tail = (q->tail + 1) % (NPROC + 1);
  q->size++;
}

// 해당하는 프로세스 큐에서 지우기
void dequeue(struct queue *q, int p){
  if(q->size == 0){
    // cprintf("ERROR: cannot dequeue because the queue is empty!");
  }

  for(int i = q->head; i != q->tail; i = (i+1) % (NPROC+1)){
    if(q->queue[i] == p){
      for(int j = i; j != q->tail; j = (j+1) % (NPROC+1)){
        q->queue[j] = q->queue[(j+1) % (NPROC+1)];
      }
      q->tail = (q->tail + NPROC) % (NPROC+1);
      q->size--;
      break;
    }
  }
}
```

우선 `enqueue` 함수는 인자로 `struct queue *q`와 `int p`를 받고, `q`의 마지막에 `p`를 집어넣는 함수입니다. 여기서 `int p`의 값에는 프로세스가 `ptable.proc` 배열의 몇 번째 원소인지, 즉 프로세스의 `idx`를 전달해주도록 구현해야 합니다.

`enqueue` 함수의 동작 원리는 다음과 같습니다.

1. 인자로 들어온 `q`가 꽉 차 있는지 검사합니다. `q`가 꽉 차 있으면, `return;`을 호출하여 `enqueue` 함수가 동작하지 않게 합니다.
2. `q`의 `queue` 배열의 `q->tail`번째, 즉 마지막 번째에 인자로 들어온 `p`를 저장합니다.
3. `q->tail` 값을 갱신합니다. `q->tail` 값을 1 증가시켜주되, 그 값이 `NPROC+1`보다 커졌을 때를 대비해서 `NPROC+1`로 나누었을 때의 나머지로 값을 갱신합니다.
4. `q`의 `size`를 1 증가시켜줍니다.

다음으로 `dequeue` 함수는 인자로 `struct queue *q`와 `int p`를 받고, `q`에서 해당하는 `p`를 찾아 없애는 함수입니다. `int p`의 경우 `enqueue`와 마찬가지로 프로세스의 `idx`를 전달해주도록 구현해야 합니다.

`dequeue` 함수의 동작 원리는 다음과 같습니다.

1. 인자로 들어온 `q`가 비어있는지 검사합니다. `q`가 비어 있으면, `return;`을 호출하여 `dequeue` 함수가 동작하지 않게 합니다.
2. `for`문을 사용하여 `q->head`부터 `q->tail`까지 차례대로 돌면서 `q`에서 인자로 들어온 `p`를 찾습니다. 만약 `p`를 찾았을 경우, 그 위치부터 바로 다음에 위치한 프로세스들을 한 칸씩 당겨줍니다.
3. `q->tail` 값을 갱신합니다. `q->tail` 값을 1 감소시키되, 그 값이 음수가 되었을 때를 대비해서 배열의 최대 크기인 `NPROC+1`를 더한 `q->tail + NPROC`을 `NPROC+1`로 나눈 나머지로 갱신해줍니다.
4. `q`의 `size`를 1 감소시켜줍니다.

MLFQ 스케줄링이 활성화된 상태인지, MoQ 스케줄링이 활성화된 상태인지 확인할 수 있게 하는 변수인 `monopolized` 변수를 `proc.c`에 선언하였습니다.

```
// proc.c

int monopolized = 0;      // 1이면 moq, 0이면 MLFQ
```

이 값이 `0`이면 MLFQ가 활성화된 상태이고, `1`이면 MoQ가 활성화된 상태입니다. 초기값은 `0`으로 지정해주어 특별한 일이 없는 이상 MLFQ가 활성화된 상태로 동작하게 만들었습니다. 이 값은 시스템콜인 `monopolize()` 함수와 `unmonopolize()` 함수로 바꿀 수 있습니다.

```
// proc.c

void
monopolize(void)
{
  monopolized = 1;
}

void
unmonopolize(void)
{
  monopolized = 0;
}
```

---

2. Implement

> 📌 실제 구현 과정에서 변경하게 되는 코드영역이나 작성한 자료구조 등에 대한 설명을 구체적으로 서술합니다

우선 스케줄링에서 가장 핵심이 되는 함수인 `scheduler()` 함수를 수정하였습니다. 수정한 내용은 아래와 같습니다.

```
//PAGEBREAK: 42
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{
struct proc *p;
struct cpu *c = mycpu();
c->proc = 0;
int nextpidx;

for(;;){
  sti();

  acquire(&ptable.lock);

  // monopolized
  if(monopolized){
    // cprintf("monopolized\n");
    nextpidx = findNextProcIdx();
    if(nextpidx == -1){
      unmonopolize();
      release(&ptable.lock);
      continue;
    }

    p = &ptable.proc[nextpidx];

    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    swtch(&c->scheduler, p->context);
    switchkvm();

    c->proc = 0;
  }

  // mlfq
  else{
    nextpidx = findNextProcIdx();
    if(nextpidx == -1){
      release(&ptable.lock);
      continue;
    }

    p = &ptable.proc[nextpidx];
```

```
      p->tick = 0;

      c->proc = p;

      switchuvm(p);
      p->state = RUNNING;
      swtch(&c->scheduler, p->context);
      switchkvm();

      c->proc = 0;

      // time out
      if(p->tick >= p->queueLevel * 2 + 2){
        dequeue(&queues[p->queueLevel], p->idx);
        // cprintf("moving queuelevel %d", p->queueLevel);

        if(p->queueLevel == 0 && p->pid % 2 == 1){
          p->queueLevel = 1;
        }
        else if(p->queueLevel == 0 && p->pid % 2 == 0){
          p->queueLevel = 2;
        }
        else if(p->queueLevel == 1 || p->queueLevel == 2){
          p->queueLevel = 3;
        }
        else if(p->queueLevel == 3){
          if(p->priority != 0){
            p->priority--;
          }
        }

        enqueue(&queues[p->queueLevel], p->idx);
        // cprintf("to queuelevel %d\n", p->queueLevel);
      }
      else{
        dequeue(&queues[p->queueLevel], p->idx);
        enqueue(&queues[p->queueLevel], p->idx);
      }
    }
    release(&ptable.lock);
  }
}
```

수정된 부분에 대한 동작원리를 설명하면 다음과 같습니다.

1. `monopolized` 값을 확인합니다. `monopolized` 값이 `1`이면, MoQ 스케줄링을 진행하고, 아니라면 MLFQ 스케줄링을 진행합니다.
2. MoQ 스케줄링을 진행한다면, `findNextProcIdx()` 함수의 반환값을 `nextpidx`에 저장합니다. `findNextProcIdx()` 함수는 후술하겠지만, 다음으로 스케줄링할 프로세스의 `idx`를 반환해주고, 찾지 못했을 경우 `-1`을 반환해주는 함수입니다. 따라서 `findNextProcIdx()` 함수의 반환값인 `nextpidx`를 이용해 다음으로 스케줄링할 프로세스를 지정할 수 있습니다.
3. `nextpidx`가 `-1`이라면 더 이상 MoQ에서 스케줄링할 프로세스가 없다는 뜻이므로 `unmonopolize()` 함수와 `release(&ptable.lock)`, 그리고 `continue;`를 호출하여 다음 스케줄링 주기에서 MLFQ 스케줄링이 진행되도록 합니다.
4. `nextpidx`가 `-1`이 아니라면 MoQ에서 스케줄링할 프로세스가 존재한다는 뜻이므로 `p = &ptable.proc[nextpidx]`로 스케줄링할 프로세스인 `p`를 지정해주고, context switching을 진행합니다.
5. MLFQ 스케줄링을 진행한다면, MoQ 스케줄링과 마찬가지로 `findNextProcIdx()` 함수의 반환값을 `nextpidx`에 저장합니다.
6. MLFQ의 경우에는 웬만해서는 `nextpidx`가 `-1`이 되는 경우가 없겠지만 만일을 대비해서 `nextpidx`가 `-1`이라면 `continue;`를 호출하여 다음 스케줄링 주기로 넘어가게 합니다.

7. `p = &ptable.proc[nextpidx]`로 스케줄링할 프로세스인 `p`를 지정해주고, `p->tick`을 `0`으로 초기화해준 뒤 context switching을 진행합니다.
8. 실행된 프로세스가 설정된 time quantum을 초과하면, 해당 프로세스를 다른 큐로 이동시킵니다. 우선 `p`를 해당하는 `queueLevel`의 큐에서 `dequeue`시킨 뒤, `p->queueLevel`과 `p->pid` 값에 따른 다음으로 이동할 `p->queueLevel`을 지정해주고, `enqueue`로 다른 큐에 이동시킵니다.
9. L3에서 time quantum을 초과하였을 경우에는 `p->priority`를 1 감소시킵니다. 이때 `p->priority`값은 음수가 되면 안 되므로 `p->priority`가 양수일 때, 즉 `0`이 아닐 때만 값을 감소시킵니다.
10. 실행된 프로세스가 time quantum을 초과하지 않았으면 `dequeue`로 우선 큐에서 프로세스를 제거한 뒤, `enqueue`로 같은 큐의 맨 끝으로 프로세스를 위치시킵니다.

다음으로 스케줄링할 프로세스의 `idx`를 반환해주는 `findNextProcIdx()` 함수를 새로 만들었습니다. 이 함수는 스케줄러에 의해 호출되며, 반환된 인덱스는 `ptable.proc` 배열에서 해당 프로세스를 찾는 데 사용됩니다. `findNextProcIdx()` 함수의 구현은 아래와 같습니다.

```
// proc.c
int findNextProcIdx(){
  struct queue *q;
  struct proc *p;

  if(monopolized){
    q = &queues[4];
    int i;

    for(i = q->head; i != q->tail; i = (i+1) % (NPROC+1)){
      p = &ptable.proc[q->queue[i]];
      if(p->state == RUNNABLE){
        break;
      }
    }

    if(i == q->tail){
      return -1;
    }
    else{
      return q->queue[i];
    }
  }


  else{
    // L0, L1, L2
    for(int i=0; i<3; i++){
      q = &queues[i];
      for(int j = q->head; j != q->tail; j = (j+1) % (NPROC+1)){
        p = &ptable.proc[q->queue[j]];
        if(p->state == RUNNABLE){
          return q->queue[j];
        }
      }
    }

    // L3
    q = &queues[3];
    struct proc *next_p;
    int max_priority = -1;
```

```
    for(int i = q->head; i != q->tail; i = (i+1) % (NPROC+1)){
      p = &ptable.proc[q->queue[i]];
      if(p->state == RUNNABLE && max_priority < p->priority){
        max_priority = p->priority;
        next_p = p;
      }
    }

    if (max_priority != -1){
      return next_p->idx;
    }
    else{
      return -1;
    }
  }
}
```

`findNextProcIdx()` 함수의 동작 원리를 설명하면 다음과 같습니다.

1. `monopolized`가 `1`인지 아닌지 검사합니다. `monopolized`가 `1`인 경우, 다음으로 실행할 프로세스는 MoQ인 `queues[4]`에 위치하므로, `q`를 `&queues[4]`로 지정합니다.
2. `q`의 `head`부터 `tail`까지 순회하며, `RUNNABLE` 상태인 프로세스를 찾습니다.
3. 실행 가능한 프로세스를 찾으면, 그 프로세스의 인덱스인 `q->queue[i]`를 반환합니다. 만약 큐에 실행 가능한 프로세스가 없다면, `-1`을 반환하여 호출한 스케줄러에게 실행할 프로세스가 없음을 알립니다.
4. `monopolized`가 `0`인 경우, 시스템은 MLFQ 스케줄링을 합니다.
5. L0부터 L2까지 순차적으로 큐를 순회하며 `RUNNABLE` 상태인 프로세스를 찾습니다. `RUNNABLE` 상태인 프로세스를 발견하면 해당 프로세스의 `idx`를 즉시 반환합니다.
6. L0에서 L2까지 순회했는데도 `RUNNABLE` 상태인 프로세스를 찾지 못했다면, L3 큐에 있는 프로세스를 선택해야 하는데, 이 경우에는 L3 큐에 있는 모든 프로세스를 순회하며 가장 높은 우선순위를 가진 프로세스를 찾고, 우선순위가 가장 높은 프로세스의 `idx`를 반환합니다.
7. 만약 모든 큐에 실행 가능한 프로세스가 없다면, `-1`을 반환하여 스케줄러에게 실행할 프로세스가 없음을 알립니다.

`trap.c`의 `trap` 함수는 타이머 인터럽트를 처리할 수 있는 기능이 있으므로, priority boosting과 프로세스의 `tick`을 처리하기 위해 `trap.c`의 `trap` 함수를 수정하였습니다.

```
// trap.c

...
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
...
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;

            // 새로 추가된 부분
      if(ticks % 100 == 0){
        priority_boosting();
      }

      wakeup(&ticks);
```

```
      release(&tickslock);
    }
    lapiceoi();
    break;

...

  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER){
       // 새로 추가된 부분
      myproc()->tick++;
      if((myproc()->tick >= myproc()->queueLevel*2+2) && myproc()->queueLevel <= 3){
        yield();
      }
     }
...
 }
```

수정된 부분에 대한 동작 원리는 다음과 같습니다.

1. 시스템이 부팅된 이후부터 경과한 타이머 인터럽트의 총 횟수인 `ticks`가 100의 배수가 될 때마다 `priority_boosting()` 함수를 호출합니다. `priority_boosting()` 함수에 대해서는 후술하도록 하겠습니다.
2. 현재 CPU에서 실행 중인 프로세스가 `RUNNING`이고, 현재 처리 중인 인터럽트가 타이머 인터럽트(`T_IRQ0+IRQ_TIMER`)인지 검사합니다. 이 조건들이 모두 만족된다면, 다음 작업을 수행합니다.
3. `myproc()->tick`의 값을 1 증가시킵니다.
4. 현재 프로세스가 속한 MLFQ 큐의 time quantum을 초과했는지 확인합니다. 만약 초과했다면 `yield()` 함수를 호출하여 CPU를 자발적으로 양보합니다.

다음으로 MLFQ에서 일정 시간 간격으로 모든 프로세스의 우선순위를 리셋하여 스케줄링의 공정성을 보장하고 starvation을 방지하는 `priority_boosting()` 함수의 구현입니다.

```
// proc.c

void priority_boosting(){
  // cprintf("priority boosting started.\n");

  if(monopolized){
    // cprintf("Moq mode\n");
    return;
  }

  clearQueues();

  for(int i = 0; i < NPROC; i++){
    if(ptable.proc[i].state == ZOMBIE){
      continue;
    }

    // L1, L2, L3에 있는 process들은 L0에 옮겨주기
    if(ptable.proc[i].queueLevel == 1 ||
       ptable.proc[i].queueLevel == 2 ||
       ptable.proc[i].queueLevel == 3){
      ptable.proc[i].queueLevel = 0;
```

```
      ptable.proc[i].tick = 0;
      enqueue(&queues[0], i);
    }
  }

  // cprintf("priority boosting ended.\n");
}
```

```
// proc.c

// L1, L2, L3 큐 비우기
void clearQueues(){
  for(int i=1; i<4; i++){
    queues[i].size = 0;
    queues[i].head = 0;
    queues[i].tail = 0;
  }
```

`priority_boosting()` 함수의 구현은 다음과 같습니다.

1. `monopolized`가 `1`인지 아닌지 검사합니다. `monopolized`가 `1`이면 MoQ 스케줄링을 진행 중이므로 우선순위 리셋이 불필요하므로, `return;`을 호출하여 우선순위 리셋을 하지 않도록 합니다
2. `monopolized`가 `0`이라면 MLFQ 스케줄링을 진행 중이므로 우선순위 리셋을 진행합니다. 우선 `clearQueues()` 함수를 호출하여 모든 큐를 초기화합니다. `clearQueues()` 함수는 모든 프로세스를 가장 높은 우선순위 큐인 L0으로 이동시키기 전에 기존의 L1, L2, L3의 `size`, `head`, `tail`을 전부 `0`으로 초기화하여 큐에 있던 프로세스 정보를 전부 삭제하는 함수입니다.
3. 시스템의 모든 프로세스를 순회하면서, 각 프로세스의 상태를 검사합니다. 이미 종료된 `ZOMBIE` 프로세스의 경우에는 `continue;`를 호출하여 우선순위 재조정 대상에서 제외되도록 합니다.
4. `ZOMBIE` 프로세스가 아닌 경우에는 프로세스의 `queueLevel`을 확인하여, L1, L2, L3 큐에 있는 프로세스들을 L0으로 이동시킵니다. 이이는 `queueLevel`을 0으로 설정하고, `enqueue` 함수를 호출하는 과정을 통해 이루어집니다. 이 과정에서 각 프로세스의 `tick`도 `0`으로 리셋됩니다.

과제에서 명시한 시스템 콜들을 구현하는 과정입니다.

- `getlev()` 함수는 현재 실행 중인 프로세스가 속한 큐의 레벨을 반환하는 함수입니다.

```
// proc.c

int
getlev(void)
{
  struct proc *p = myproc();
  if(p == 0){
    return -1;
  }

  if(p->queueLevel >= 0 && p->queueLevel < 4){
    return p->queueLevel;
  }
  else{
    return 99;
```

```
  }
}
```

`myproc()` 함수를 호출하여 현재 CPU에서 실행 중인 프로세스의 `proc` 구조체 포인터를 가져옵니다. 반환된 포인터가 `0`이라면, 현재 실행 중인 프로세스가 없다는 것을 의미하므로 이 경우 `-1`을 반환하여 호출자에게 실행 중인 프로세스가 없음을 알립니다. 현재 프로세스의 `queueLevel`을 검사하여 그 값이 `0` 이상 `4` 미만이라면 현재 프로세스가 MLFQ에 속해있음을 의미하므로 그 값을 그대로 반환하고, 그게 아니면 MoQ에 속해있음을 의미하므로 `99`를 반환합니다.

- `setpriority(int pid, int priority)` 함수는 특정 프로세스의 우선순위를 설정하는 함수입니다. 이 함수는 프로세스의 pid인 `int pid`와 새로 설정할 우선순위인 `int priority`를 매개변수로 받아 해당 프로세스의 우선순위를 변경합니다.

```
// proc.c

int setpriority(int pid, int priority)
{
  int i;
  for(i = 0; i < NPROC; i++){
    if(ptable.proc[i].pid == pid){
      break;
    }
  }

  // 못 찾았아서 i가 NPROC이 됨.
  if (i == NPROC){
    return -1;
  }

  if (priority < 0 || priority > 10){
    return -2;
  }

  struct proc *p = &ptable.proc[i];
  p->priority = priority;

  return 0;
}
```

전체 프로세스 테이블인 `ptable.proc`을 순회하면서 매개변수로 받은 `pid`와 일치하는 프로세스 ID를 가진 프로세스를 찾습니다. 만약 `pid`에 해당하는 프로세스를 찾지 못해 `i`가 `NPROC`에 도달하면 함수는 `-1`을 반환하여 호출자에게 해당 ID를 가진 프로세스가 존재하지 않음을 알립니다. 주어진 `priority` 값이 `0` 이상 `10` 이하의 값이 아니라면 함수는 `-2`를 반환하여 잘못된 우선순위 값임을 알립니다. 유효한 프로세스와 우선순위가 주어지면, `ptable.proc[i]`를 통해 해당 프로세스의 `proc` 구조체에 접근하고, `priority` 필드를 새로운 우선순위 값으로 설정한 뒤, `0`을 반환하여 우선순위 변경 작업이 성공적으로 완료되었음을 알립니다.

- `setmonopoly(int pid, int password)` 함수는 프로세스의 pid인 `int pid`와 독점 자격을 증명할 암호인 `int password`를 매개변수로 받아 특정 프로세스를 MoQ로 이동시킵니다.

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

  for(p=ptable.proc; p<&ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      break;
    }
```

```
  }

  // 못 찾아서 p가 맨 마지막 프로세스가 됨.
  if(p == &ptable.proc[NPROC]){
    return -1;
  }

  if(password != 2022028522){
    return -2;
  }

  dequeue(&queues[p->queueLevel], p->idx);
  p->queueLevel = 99;
  enqueue(&queues[4], p->idx);

  return queues[4].size;
}
```

전체 프로세스 테이블인 `ptable.proc`을 순회하면서 매개변수로 받은 `pid`와 일치하는 프로세스 ID를 가진 프로세스를 찾습니다. 만약 `pid`에 해당하는 프로세스를 찾지 못해 `i`가 `NPROC`에 도달하면 함수는 `-1`을 반환하여 호출자에게 해당 ID를 가진 프로세스가 존재하지 않음을 알립니다. `password`가 특정 값, 즉 학번인 `2022028522`와 일치하는지 확인하고, 일치하지 않는다면 `-2`를 반환하여 권한이 없음을 알립니다. 해당하는 프로세스를 찾고 권한이 증명되었다면 `dequeue(&queues[p->queueLevel], p->idx)`를 호출하여 현재 프로세스를 속한 큐에서 제거한 뒤, `p->queueLevel`을 `99`로 설정하고 `enqueue(&queues[4], p->idx)`를 호출하여 프로세스를 MoQ로 이동시킵니다. 이 경우 MoQ에 있는 프로세스의 개수를 나타내는 `queues[4].size`를 반환하여 함수가 성공적으로 실행되었음을 나타냅니다.

- `monopolize()` 함수와 `unmonopolize()` 함수는 시스템의 스케줄링 방식을 조정하는 데 사용되고, 각각 `monopolized` 변수의 값을 `1`이나 `0`으로 설정하여 MoQ 방식 또는 MLFQ 방식으로 전환하는 역할을 합니다.

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

void
unmonopolize(void)
{
  monopolized = 0;
}
```

마지막으로 이 시스템 콜들을 등록하는 과정입니다. 과제에서 명시한 `yield()` 함수도 추가로 시스템 콜로 등록합니다.

- `defs.h` 파일에 다음 함수들을 추가하여 이 헤더 파일을 include하는 다른 파일에서 이 함수를 사용할 수 있도록 합니다.

```
// defs.h

struct buf;
struct context;
struct file;

...

void            yield(void);
```

```
int             getlev(void);
int             setpriority(int, int);
int             setmonopoly(int, int);
void            monopolize(void);
void            unmonopolize(void);
```

- `syscall.h`와 `syscall.c`를 수정합니다. `syscall.h`에 새로운 시스템 콜 번호를 정의하고 새로운 시스템 콜을 등록하고, `syscall.c`에 함수를 등록하여 시스템 콜이 호출될 때 실행되도록 합니다.

```
// syscall.h

...
#define SYS_yield  22
#define SYS_getlev 23
#define SYS_setpriority 24
#define SYS_setmonopoly 25
#define SYS_monopolize  26
#define SYS_unmonopolize 27
```

```
// syscall.c

...

extern int sys_yield(void);
extern int sys_getlev(void);
extern int sys_setpriority(void);
extern int sys_setmonopoly(void);
extern int sys_monopolize(void);
extern int sys_unmonopolize(void);

...

static int (*syscalls[])(void) = {
...
[SYS_yield]   sys_yield,
[SYS_getlev]  sys_getlev,
[SYS_setpriority]   sys_setpriority,
[SYS_setmonopoly]   sys_setmonopoly,
[SYS_monopolize]    sys_monopolize,
[SYS_unmonopolize]  sys_unmonopolize,
};

...
```

- `user.h`에 `getgpid()` 함수를 등록하여 사용자 프로그램에서 이를 호출할 수 있도록 합니다.

```
// user.h

...
void yield(void);
int getlev(void);
int setpriority(int, int);
int setmonopoly(int, int);
void monopolize(void);
```

```
void unmonopolize(void);
...
```

- 사용자 프로그램에서 시스템 콜을 호출할 수 있도록 `usys.S`에 매크로를 추가합니다.

```
// usys.S

...
SYSCALL(yield)
SYSCALL(getlev)
SYSCALL(setpriority)
SYSCALL(setmonopoly)
SYSCALL(monopolize)
SYSCALL(unmonopolize)
```

3. Result

📌 컴파일 및 실행 과정과, 해당 명세에서 요구한 부분이 정상적으로 동작하는 실행 결과를 첨부하고, 동작 과정에 대해 설명합니다.

- xv6-public 디렉토리 안에서 다음과 같은 명령어를 차례대로 입력합니다.

```
$ make clean
$ make
$ make fs.img
…
$ ./bootxv6.sh
```

```
$ mlfq_test
```

테스트 결과는 다음과 같습니다.

Test1

```
$ mlfq_test
MLFQ test start
[Test 1] default
Process 5
L0: 16897
L1: 27511
L2: 0
L3: 55592
MoQ: 0
Process 7
L0: 16424
L1: 27212
L2: 0
L3: 56364
MoQ: 0
Process 11
L0: 17952
L1: 30185
L2: 0
L3: 51863
MoQ: 0
Process 9
L0: 17689
L1: 31359
L2: 0
L3: 50952
MoQ: 0
```

```
Process 4
L0: 18783
L1: 0
L2: 44840
L3: 36377
MoQ: 0
Process 6
L0: 15078
L1: 0
L2: 43488
L3: 41434
MoQ: 0
Process 8
L0: 13979
L1: 0
L2: 48183
L3: 37838
MoQ: 0
Process 10
L0: 16274
L1: 0
L2: 49229
L3: 34497
MoQ: 0
[Test 1] finished
```

L1 큐가 L2 큐보다 우선 순위가 높으므로, pid가 홀수인 프로세스가 대체로 먼저 끝나게 됩니다.

Test2

```
[Test 2] priorities
Process 13
L0: 4959
L1: 13434
L2: 0
L3: 81607
MoQ: 0
Process 19
L0: 11447
L1: 26405
L2: 0
L3: 62148
MoQ: 0
Process 18
L0: 11762
L1: 0
L2: 43284
L3: 44954
MoQ: 0
Process 16
L0: 11565
L1: 0
L2: 41661
L3: 46774
MoQ: 0
```

```
Process 17
L0: 15856
L1: 29253
L2: 0
L3: 54891
MoQ: 0
Process 15
L0: 15560
L1: 36258
L2: 0
L3: 48182
MoQ: 0
Process 14
L0: 17222
L1: 0
L2: 51282
L3: 31496
MoQ: 0
Process 12
L0: 7736
L1: 0
L2: 28978
L3: 63286
MoQ: 0
[Test 2] finished
```

pid가 큰 프로세스에게 더 높은 우선순위를 부여하여, 전체적인 시간 사용량은 결국 비슷해지기 때문에 끝나는 시간도 비슷하지만, pid가 큰 프로세스가 조금 더 먼저 끝나는 경향성이 있습니다.

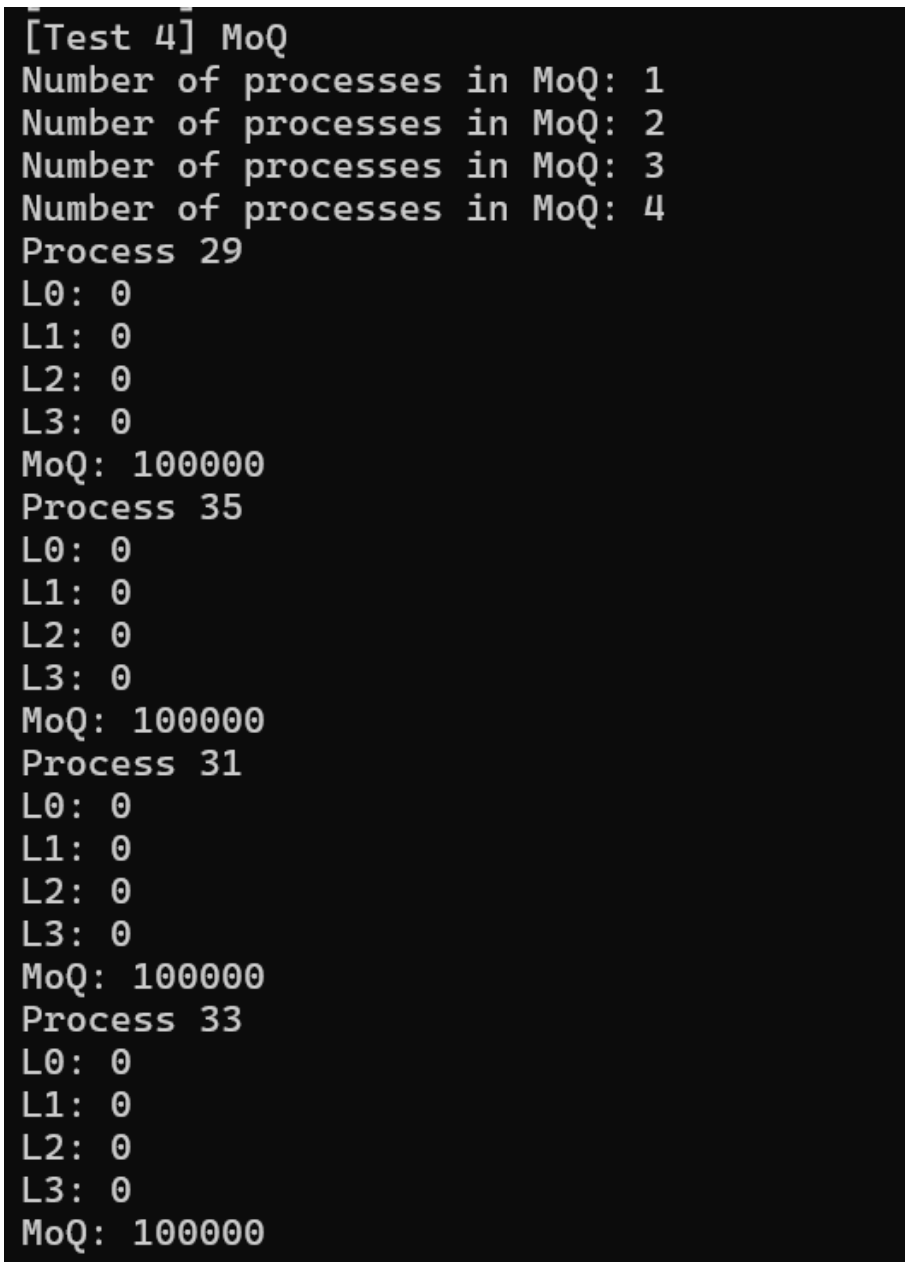
Test3

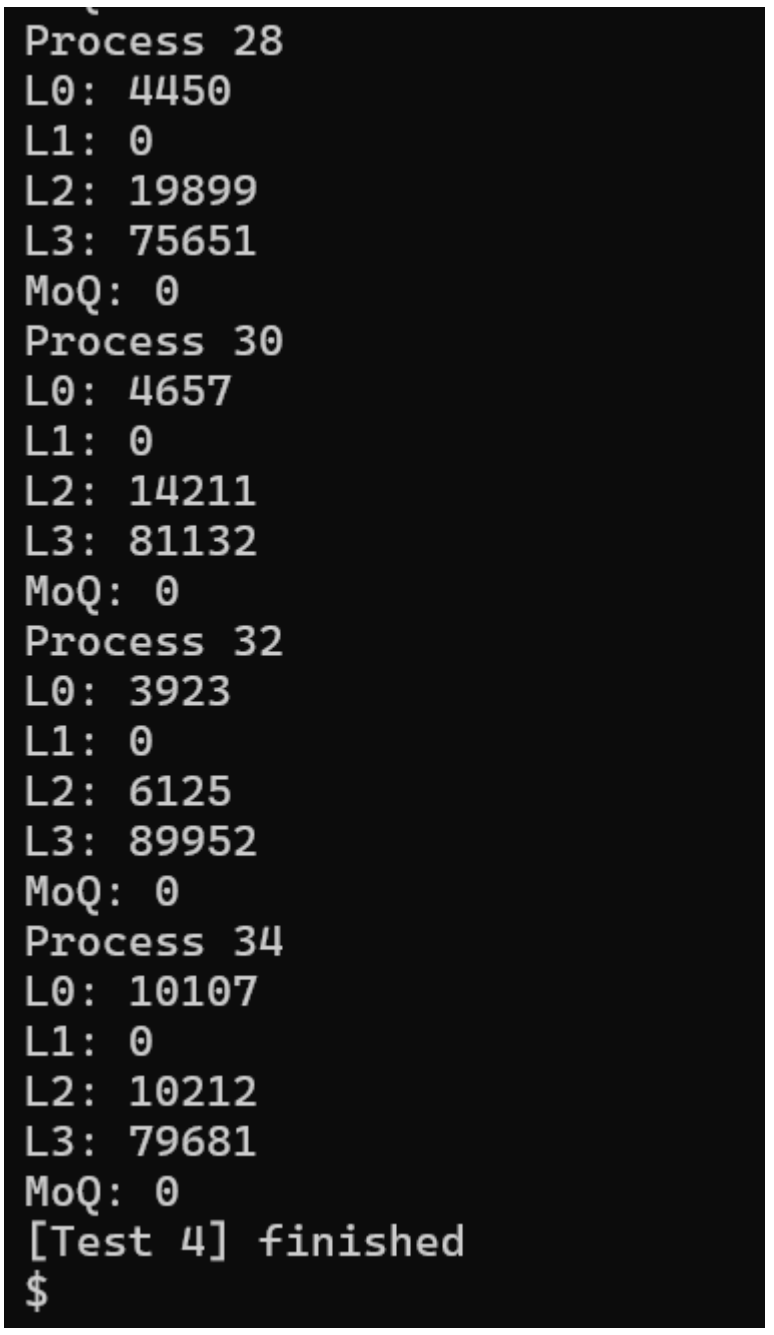
```
[Test 3] sleep
Process 20
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 21
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 27
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 22
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
```

```
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
[Test 3] finished
```

pid가 큰 프로세스에게 더 높은 우선순위를 부여하고, 각 프로세스는 루프를 돌 때마다 바로 sleep 시스템 콜을 호출합니다. sleep 상태에 있는 동안에는 스케줄링 되지 않고 다른 프로세스가 실행될 수 있기 때문에 거의 동시에 작업을 마무리하게 됩니다. 대부분 L0에 머무르기 때문에, pid가 작은 프로세스가 먼저 끝나는 경향성이 있습니다.

Test4





pid가 홀수인 프로세스를 MoQ에 저장합니다. 새로운 프로세스가 추가될 때마다 MoQ에 존재하는 프로세스의 개수를 출력합니다.
마지막 자식 프로세스(pid 36)는 monopolize를 호출하고 exit()를 호출합니다. exit()이 호출되어 마지막에 $가 뜨는 것을 확인할 수 있습니다.
MoQ에 존재하는 프로세스들은 FCFS 방식으로 스케줄링 되므로 pid가 작은 프로세스가 먼저 종료됩니다.
이후 기존 MLFQ 스케줄링으로 돌아와 pid가 짝수인 나머지 프로세스들을 실행합니다.

4. Trouble Shooting

📌 과제 수행 과정에서 겪은 문제가 있다면, 해당 문제와 해결 과정을 서술합니다. 해결하지 못했다면 어떤 문제였고 어떻게 해결하려 했는지 서술합니다.

- 처음에는 큐의 구현을 배열이 아닌 linked list로 했었습니다. 우선순위 재조정 과정을 구현하기 전에는, 모든 프로세스가 L3 큐로 넘어가는 것을 확인하면서 우선순위 재조정 과정 구현하기 단계로 넘어가면서 `lapicid 0: panic` 에러, 무한 루프 등의 다양한 문제가 생겼습니다. 디버깅을 하기 위해
  `priority_boosting()` 함수의 시작과 끝에 `cprintf` 문을 넣고, `enqueue` 와 `dequeue` 함수에서 예외 상황이 발생하였을 때 `cprintf` 문으로 오류 메시지를 출력하게 하였습니다.

```
process queuelevel: 1, but queuelevel: 0
ERROR: The process should be dequeued!!
ERROR: you are trying to dequeue from the wrong queue.
process queuelevel: 1, but queuelevel: 0
ERROR: The process should be dequeued!!
ERROR: you are trying to dequeue from the wrong queue.
process queuelevel: 1, but queuelevel: 0
ERROR: The process should be dequeued!!
ERROR: you are trying to dequeue from the wrong queue.
process queuelevel: 1, but queuelevel: 0
ERROR: The process should be dequeued!!
ERROR: you are trying to dequeue from the wrong queue.
process queuelevel: 1, but queuelevel: 0
ERROR: The process should be dequeued!!
ERROR: you are trying to dequeue from the wrong queue.
process queuelevel: 1, but queuelevel: 0
ERROR: The process should be dequeued!!
ERROR: you are trying to dequeue from the wrong queue.
process queuelevel: 1, but queuelevel: 0
ERROR: The process should be dequeued!!
ERROR: you are trying to dequeue from the wrong queue.
process queuelevel: 1, but queuelevel: 0
ERROR: The process should be dequeued!!
Process 7
L0: 5778
L1: 94222
L2: 0
ERROR: you are trying to dequeue from the wrong queue.
process queuelevel: 1, but queuelevel: 0
ERROR: The process should be dequeued!!
```

그랬더니 위와 같이 자꾸 `dequeue` 가 실패하여, 결국에는 프로세스들이 L3 큐로 이동을 못하는 상황이 발생하였습니다. 이상한 점은 첫 번째 `priority_boosting()` 은 `dequeue` 와 `enqueue` 가 성공적으로 이루어지는데, 두 번째 `priority_boosting()` 부터 `dequeue` 가 제대로 이루어지지 않아 `enqueue` 도 되지 않는다는 점이었습니다. `dequeue` 의 오류 메시지를 분석해보니, 프로세스의 `queueLevel` 값은 `1` 또는 `2` 인데 그 프로세스가 L1 나 L2 큐가 아닌 L0 큐에 있다고 인식되어서 생기는 문제점인 것 같았습니다.

그러나 도저히 `enqueue` 와 `dequeue` 함수의 문제점을 찾지 못했고, 또 이상한 점은 `priority_boosting()` 을 호출하는 줄을 주석 처리하면 `enqueue` 또는 `dequeue` 오류 없이 모든 프로세스가 큐 이동을 잘 한다는 점이었습니다. `priority_boosting()` 에서도 딱히 프로세스의 `queueLevel` 을 잘못 설정하거나 `enqueue` 와 `dequeue` 를 잘못 호출하는 경우도 없었고, 첫 번째 `priority_boosting()` 을 호출했을 때만 제대로 된 뒤에 두 번째 이후 `priority_boosting()` 을 호출하였을 때 오류가 나는 이유를 끝까지 찾지 못하였습니다.

결국 수시로 프로세스의 `queueLevel` 을 재조정해주는 함수 만들어서 호출하기, 프로세스의 `queueLevel` 과 큐의 레벨이 다르면 일단 넘어가기 등등 정말 많은 방법을 시도해보았지만 해결을 할 수 없어 큐를 linked list로 구현하는 방식을 갈아엎고 circuit queue로 구현하는 방식을 새로 채택하여 구현을 하였습니다.

- 다음으로 `trap.c` 에서 `priority_boosting()` 을 호출하려고 할 때 다음과 같은 문제가 발생한 적이 있었습니다.

```
trap.c: In function ‘trap’:
trap.c:56:9: error: implicit declaration of function ‘priority_boosting’ [-Werror=implicit-function-declaration]
   56 |         priority_boosting();
      |         ^~~~~~~~~~~~~~~~~
cc1: all warnings being treated as errors
make: *** [<builtin>: trap.o] Error 1
```

알고 보니 `priority_boosting()` 함수를 사용하기 위해서는 `defs.h` 에 또 따로 정의를 해주어야 했다는 것을 알게 되어 `defs.h` 에 `priority_boosting()` 함수를 정의하였더니, 문제 없이 잘 작동하게 되었습니다.

- `lapicid 0: panic: release` 또는 `lapicid 0: panic: acquire` 오류도 정말 많이 떴었는데, 이에 대해 찾아보니까 이미 `acquire` 을 했는데 또 `acquire` 을 할 때나 이미 `release` 를 했는데 또 `release` 를 하려는 경우 이 에러가 발생한다는 것을 알았습니다. 그래서 `acquire` 함수와 `release` 함수를 항상 세트로 오게 코드를 짜려고 노력했는데, 함수의 리턴값이 `if` `else` 문에 의해 분기가 되는 경우나 `&ptable.lock` 에 접근하는 함수 안에서 또 `&ptable.lock` 에 접근하는 함수를 호출하는 경우 이 문제를 해결하기가 상당히 어려웠습니다. 그래서 어차피 과제

명세에서 cpu는 하나라고 명시되어 있고 cpu가 하나면 공유자원에 동시에 접근하는 문제가 생기지가 않으니 새로 짠 함수에서 `acquire` 랑 `release` 부분을 그냥 다 지워버렸습니다.
