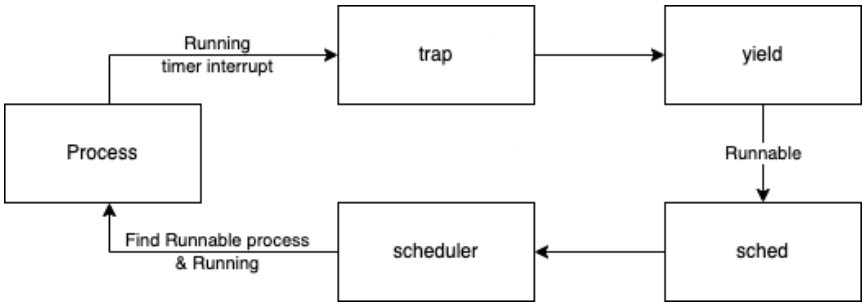
Project #1 : MLFQ scheduler

1. Design For MLFQ Scheduler

1-1. 기존 xv6 scheduler 분석

간단하게 동작방식을 파악해 보자면 다음과 같다.



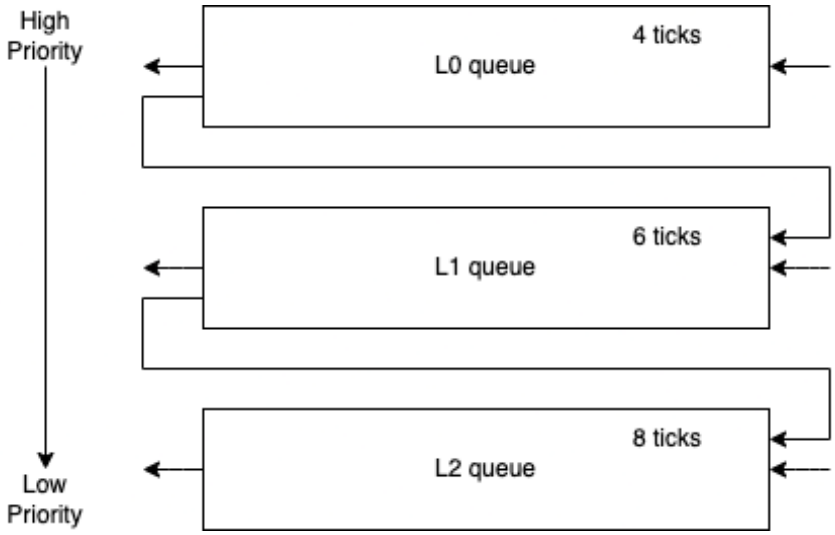
1. `Scheduler`
    a. Runnable한 process를 탐색한다.
    b. Runnable한 process를 발견했으면 해당 Process에게 cpu를 할당 및 user virtual memory를 넘겨주고 Process를 Running으로 만들어 준 후 Scheduler에서 해당 Process으로 Context switching을 발생시킨다.
2. `Process`
    a. Process가 작동하다가 1 tick마다 Timer interrupt가 발생한다.

3. `Trap`
    a. timer interrupt에 의해 도착하면 tick을 올려준다.
    b. 현재 cpu를 가지고 있는 process가 있고 process의 상태가 Running이였으면 위와 같이 timer interrupt에 의해 trap에 온 것이라면 다른 프로세스에게 점유권을 넘겨주기 위해 `yield`를 호출한다.
4. `Yield`
    a. 현재 process의 상태를 Runnable로 바꿔주고 `sched` 를 호출한다.
5. `Sched`
    a. 현재 process에서 scheduler로 context switch를 발생시킨다.
6. 1로 돌아간다.

- 중간중간에 생략한 부분도 많지만 크게 보면 저렇게 순환하는 방식으로 Process가 작동함을 알 수 있었다.

1-2. Design for MLFQ in xv6

과제 명세에 따르면 우리가 구현해야하는 MLFQ는 3-Level이며 각 Queue의 기본 작동방식은 **Round-Robin** 방식이므로 다음과 같이 그려볼 수 있다.



- Multi Level Feedback Queue (MLFQ) Design

A. Design For MLFQ

1. **MLFQ 구현**
    a. 따로 MLFQ 구현 없이 Ptable을 이용하는 방안
        - 장점 : 기존 xv6에서 많은 부분을 수정안해도 됨
        - 단점 : 탐색의 문제 / 각 Queue에 어느게 먼저 들어온지 파악하기 힘듦
    b. Linked List vs Circular Queue
        - Linked List : 공간을 더 효율적으로 사용할 수 있음, 삽입 및 삭제가 편리함 / Process를 감싸는 Node를 만들어야한다.
        - Circular Queue : 기존의 배열 공간내에서 Process를 저장하는 방식 / index를 통한 탐색이 가능함

⇒ 기존의 Ptable은 이용하면서 Ptable에서의 Process Pointer를 이용하는 *Circular Queue*를 이용하기로 하였다.

- Process의 level이 변화가 자주 일어나므로 Ptable의 Process의 Pointer를 이용한 circular queue방식을 이용하기로 하였다.

2. **SchedulerLock / SchedulerUnlock의 구현**

- Process가 Lock되었는지 체크할 방법과 Lock된 Process가 있는지 없는지 탐색을 빠르게 할 방법에 대해서 고민했다.

⇒ MLFQ 및 Queue, Proc 구조체에 `int is_locked` 를 추가하여 해결하였다.

3. **MLFQ 공간 문제**

- 기존 ptable이 어떻게 공간을 확보하는지 탐색하였다. `wait` 함수에서 `ZOMBIE` state인 자식을 `UNUSED` state로 변경하는 것을 확인하였다.

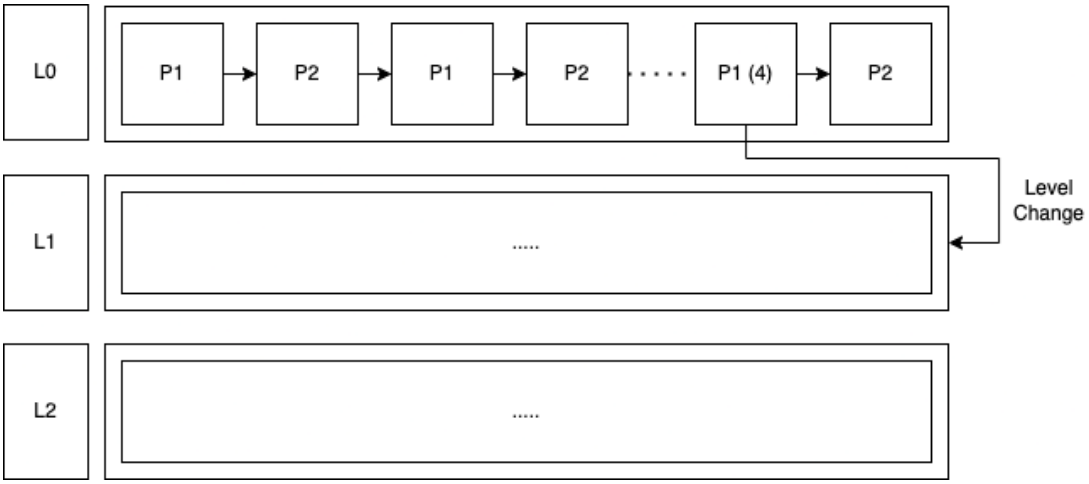
```
// wait(void) function in proc.c
void wait(void) {
  ...
  havekids = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent != curproc)
      continue;
    havekids = 1;
    if(p->state == ZOMBIE){
      // Found one.
      pid = p->pid;
      kfree(p->kstack);
      p->kstack = 0;
      freevm(p->pgdir);
      p->pid = 0;
      p->parent = 0;
      p->name[0] = 0;
      p->killed = 0;
      p->state = UNUSED;
      release(&ptable.lock);
      return pid;
    }
  }
  ...
}
```

- `wait` 함수는 자식이 있는 process이면 자식이 종료 될 때까지 기다리고 자식이 `ZOMBIE` 이 되면 이 process의 공간을 다시 사용할 수 있게 `UNUSED`로 변경하여 ptable의 공간을 확보해 주는 역할을 한다.

4. Process

- MLFQ에서 Process를 1tick마다 변경할 것인지 아니면 한 Process가 Level에 맞는 time quantum을 다 사용하고 다음 Process로 넘겨줘야할지 고민하였다.

⇒ 각 Process가 공평하게 CPU를 나눠갖는게 맞다고 판단하여 1tick마다 다음 Process로 넘겨주고 지정된 time quantum을 다 사용하면 Level을 변경하도록 하였다.



⇒ 결과적으로 디자인은 다음과 같다.

1. ptable에 있는 process의 pointer를 이용한 3-level feedback queue (based on circular queue)
2. 모든 프로세스는 1 tick 사용 후 scheduler에게 cpu를 넘겨준다.
3. lock이 되기 전까지는 1 tick 사용 후 프로세스를 뒤로 넘겨준다.
4. time quantum을 다 사용할 경우 level이 올라가거나 priority가 작아진다.
5. MLFQ에서의 공간확보를 위해 wait에서 ZOMBIE처리시 MLFQ에서도 처리 되도록 한다.
6. Lock을 위해 is_locked 변수를 추가하며 빠른 접근을 위해 process 뿐만 아니라 MLFQ, 각 QUEUE에도 해당 변수를 추가한다.

B. MLFQ sturcture

- MLFQ를 구현하기 위해 만들었던 구조체이며 3개의 queue를 담도록 디자인하였다.

```
#define MLFQ_LEVEL 3
...

struct {
  struct proc_queue_t queue[MLFQ_LEVEL];

  // is_locked for schedulerLock, Unlock
  int is_locked;

  // global_ticks is tick for MLFQ scheduler
  uint global_ticks;
} MLFQ;
```

C. Queue & Proc structure (for MLFQ)

- Process를 담는 queue type이 필요하게 되어 `proc_queue_t` 타입을 만들었다.
- `ptable` 의 process들의 pointer를 `NPROC` size만큼 담는 queue 배열을 3개를 담았으며 탐색을 위해 front, rear, size를 담았다.

```
struct proc_queue_t {
  struct proc* data[NPROC];

  // for traversal
  int front;
```

```
  int rear;
  int size;

  // for schedulerLock/Unlock
  int is_locked;
};
```

- Process를 담는 `proc` 구조체는 MLFQ로 바꾸면서 다음과 같은 변수들을 추가하였다.

```
// struct proc in proc.h
// for mlfq
int level;           // Level in mlfq for this process
int time_quantum;    // Time of how much this process has been used
int priority;        // Priority in L2 queue
int is_locked;       // Variable indicating whether this process is locked or not
```

D. Scheduler

- 기존 Scheduler의 코드는 다음과 같다.

```
// void scheduler
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);
  p->state = RUNNING;

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

  // Process is done running for now.
  // It should have changed its p->state before coming back.
  c->proc = 0;
}
```

- ptable을 순회하면서 RUNNABLE한 process를 찾고 그 process에게 넘겨준다. (1 tick마다 Round-Robin이 일어나는 방식이다.)
- 이를 명세에 맞게 다음과 같은 부분으로 수정하였다. (자세한 코드는 Implemetation에서 다루도록 하겠다.)

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

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

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

    // MLFQ Scheduling
```

```
    // proc initialization
    p = 0;

    // MLFQ has locked process
    if(MLFQ.is_locked == 1){
      // Scheduler TODO 1 : find Locked Process
      ...
    }
    // MLFQ has no locked process.
    else {
      // Scheduler TODO 2 : find runnable Process
      ...
    }

    if (p != 0) {
      // 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();

      // Scheduler TODO 3 : Round-Robin and Boosting
      ...

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);
  }
}
```

- 크게 3가지 부분으로 나뉘는데 각각은 다음과 같다.

1. MLFQ에 Locked Process가 존재할 때
   a. lock된 process를 탐색한다.
      - 만약 lock된 process가 Zombie process이면 Lock을 풀고 다시 runnable한 프로세스를 탐색한다.
2. MLFQ에 Locked된 Process가 존재하지 않을 때
   a. MLFQ의 방식에 따라서 RUNNABLE한 process를 탐색한다.
3. Round-Robin 및 Boosting
   a. global tick 및 time quantum을 올려준다.
   b. global tick이 100이상일 경우 boosting을 발생시킨다.
   c. 아닐경우 MLFQ의 방식에 따라 Level 및 순서를 재조정한다.

2. Implementation for MLFQ

2-1. MLFQ

- MLFQ를 위해서 만든 함수들에 대해서 설명하겠다.

1. `mlfq_initialization`

```
// in proc.c
// MLFQ initialization
void
mlfq_initalization(void) {

  // For traversal
  struct proc_queue_t *temp;

  // Initialize queue
  for (int l = 0; l < MLFQ_LEVEL; ++l){
    temp = &MLFQ.queue[l];

    memset(temp->data, 0, sizeof(struct proc *) * NPROC);

    temp->front = 1;
    temp->rear = 0;
    temp->size = 0;
    temp->is_locked = 0;
  }

  // For schedulerLock,Unlock
  MLFQ.is_locked = 0;

  // For MLFQ ticks
  MLFQ.global_ticks = 0;
}
```

- mlfq에서 사용하는 변수들을 초기화해주는 함수이다.
  - 각 queue들을 0으로 초기화 해준다.
  - front는 queue가 들어있는 첫번째 데이터를, rear은 마지막 데이터를 가리킨다.
  - size는 현재 queue에 들어있는 process의 갯수를 가리킨다.
  - is_locked는 현재 queue에 locked process의 유무를 나타낸다.
  - MLFQ의 is_locked와 global_ticks를 초기화해준다.
- 이 함수는 `void pinit(void)` 에서 호출하여 ptable.lock과 함께 초기화하였다.

```
void
pinit(void)
{
  initlock(&ptable.lock, "ptable");

  // for mlfq
  // mlfq initalization part
  mlfq_initalization();
}
```

2. `mlfq_enqueue`

```
// in proc.c
// Enqueue function for MLFQ
// This function takes process struct pointer and level
int
mlfq_enqueue(struct proc* proc, int level){

  struct proc_queue_t *target = &MLFQ.queue[level];
```

```
  // maximum number of possible processes in the queue (of the level)
  if (target->size == NPROC)
    return -1;

  // if queue[level] has empty space
  int target_idx = (target->rear + 1) % NPROC;
  target->data[target_idx] = proc;
  target->rear = target_idx;
  ++target->size;
  return 0;
}
```

- 특정 process를 특정 level의 queue에 넣는 함수이다.
  - 우선 level을 이용해서 대상 queue의 Pointer를 가져온다.
  - 만약 현재 queue가 가득 찼으면 -1을 반환한다.
  - queue에 공간이 있을 경우 rear의 한칸 뒤에 프로세스를 집어넣는 함수이다.
  - circular queue 이므로 `(target→rear+1) % NPROC`을 대상 index로 지정하였다.

3. `mlfq_dequeue`

```
// in proc.c
struct proc*
mlfq_dequeue(int level){

  // target queue를 가져온다.
  struct proc_queue_t *target = &MLFQ.queue[level];

  // target이 비어있을 경우
  if (target->size == 0)
    return 0;

  // 해당 queue에 1개 이상의 process가 존재하는 경우
  struct proc* ret = target->data[target->front];
  target->data[target->front] = 0;
  target->front = (target->front + 1) % NPROC;
  --target->size;

  return ret;
}
```

- 특정 level의 맨 앞의 프로세스를 빼면서 반환하는 함수이다.
  - 작동방식은 거의 enqueue와 동일하다.

4. `mlfq_locking` `mlfq_unlocking`

```
// in proc.c
// MLFQ lock
// Turn on is_locked variables
void
mlfq_locking(struct proc* p){
  MLFQ.is_locked=1;
  MLFQ.queue[p->level].is_locked=1;
  p->is_locked=1;
}
```

```
// MLFQ unlock
// Turn off is_locked variables
void
mlfq_unlocking(struct proc* p){
  MLFQ.is_locked=0;
  MLFQ.queue[p->level].is_locked=0;
  p->is_locked=0;
}
```

- 특정 process의 lock을 올리고 내리는 함수이다.
  - 대상 process와 그 process가 들어있는 queue와 MLFQ의 is_locked를 올리거나 내린다.

5. `mlfq_push`

```
// in proc.c
void mlfq_push(struct proc* proc, int level){
  struct proc_queue_t *target = &MLFQ.queue[level];

  int target_idx = (target->front-1+NPROC)%NPROC;
  target->data[target_idx] = proc;
  target->front = (target->front-1+NPROC)%NPROC;
  ++target->size;
}
```

- 특정 level의 맨 앞에 process를 넣는 함수이다. (Like stack)
  - schedulerLock이 풀릴 때 lock이 되었던 프로세스가 L0 queue의 맨 앞으로 이동하기 때문에 이 부분을 따로 함수로 구현하였다.
  - front 위치의 한 칸 앞에 process를 넣고, front를 바꾸는게 효과적이므로 그러한 방식을 선택하였다.
  - front의 한 칸 앞에 process를 넣고 front와 size를 변경한다.

6. `mlfq_kill`

- `wait` 함수를 보면 ZOMBIE인 자식 process를 다시 이용할 수 있도록 UNUSED 상태로 바꿔주는 부분이 있다.

```
// int wait(void)
if(p->state == ZOMBIE){
  ...
  p->state = UNUSED;
  ...
  return pid;
}
```

- UNUSED인 process가 MLFQ에 남아있지 않도록 하기 위해서 제거될 process를 MLFQ에서 지워주는 함수이다. 코드는 다음과 같다.

```
// in proc.c
void
mlfq_kill(struct proc *proc){
  struct proc_queue_t *target = &MLFQ.queue[proc->level];

  int idx;
  for(idx = 0; idx < NPROC; ++idx){
    if (target->data[idx] == proc){
      break;
```

```
    }
  }

  if (idx == NPROC) return;

  int nxt;
  while(idx != target->front){
    nxt = idx - 1;
    if (nxt < 0)
      nxt += NPROC;

    target->data[idx]=target->data[nxt];
    idx=nxt;
  }

  target->data[target->front] = 0;
  target->front = (target->front + 1) % NPROC;
  --target->size;
}
```

- 우선 제거해야할 process가 있는 곳을 탐색하고 해당 process를 기준으로 왼쪽의 process를 한 칸씩 오른쪽으로 옮기고 queue의 size와 front를 재조정한다.
- 수정한 `wait`는 다음과 같다.

```
// int wait(void)
if(p->state == ZOMBIE){
  mlfq_kill(p);
  ...
  p->state = UNUSED;
  ...
  return pid;
}
```

- ZOMBIE 상태인 자식 프로세스가 있다면 MLFQ에서 제거한다.

7. `mlfq_priority_boosting`

```
// in proc.c
void
mlfq_priority_boosting(void){
  struct proc_queue_t *target = &MLFQ.queue[0];
  struct proc *proc;
  MLFQ.global_ticks = 0;

  // L0 queue의 모든 것들을 모두 초기화한다.
  for(int i = 0; i < target->size; ++i){
    proc = mlfq_dequeue(0);
    proc->priority = 3;
    proc->time_quantum = 0;

    mlfq_enqueue(proc, 0);
  }

  // L1, L2의 모든 process를 L0로 옮긴다.
  for(int lev = 1; lev <= 2; ++lev){
    target = &MLFQ.queue[lev];

    for(int i = 0; i < target->size; ++i){
      proc = mlfq_dequeue(lev);
      proc->priority = 3;
      proc->time_quantum = 0;
      proc->level = 0;

      mlfq_enqueue(proc, 0);
```

```
    }
  }
}
```

- L0부터 L2부터 차례대로 L0 queue에 집어 넣는 함수이다.
- MLFQ의 global tick을 0으로 초기화 해주고 각 process의 priority, level, time quantum을 조정해준다.

2-2. Scheduler

1. `scheduler`

- 수정사항이 많아 위의 디자인에서 TODO 부분을 순서대로 설명하겠다.
- scheduler TODO 1 : find Locked Process

```
if(MLFQ.is_locked == 1){
  int lev;
  for(lev = 0; lev < MLFQ_LEVEL; ++lev){
    traversal = &MLFQ.queue[lev];
    if(traversal->is_locked == 1) break;
  }

  // is_locked가 켜져있는 process를 탐색할 때까지 회전시킨다.
  for(int i = 0; i < traversal->size; i++){
    p = traversal->data[traversal->front];
    if(p->is_locked == 1)
      break;

    mlfq_dequeue(lev);
    mlfq_enqueue(p, lev);
  }

  // locked process가 zombie process인 경우
  // 해당 process에 lock을 해제한다.
  if (p->state == ZOMBIE){
    mlfq_unlocking(p);
    release(&ptable.lock);
    continue;
  }
}
```

  - MLFQ가 잠겨 있으면 is_locked가 올라가있는 queue를 탐색하고, 그 queue안에서 locked process를 탐색하는 함수이다.
  - locked process를 찾을 때까지 queue를 돌린다.
  - 만약 locked process가 `ZOMBIE` process라면 lock을 풀고 `ptable.lock`을 풀고 올바른 process를 탐색하도록 한다. (이 경우는 schedulerLock이 된 process가 lock을 풀기 전에 `exit()`으로 종료했을 경우에 발생한다.

- scheduler TODO 2 : find RUNNABLE process (lock이 잠기지 않았을 경우)

```
else {
  p = find_runnable_proc();
}
```

  - `find_runnable_proc` 함수를 이용하여 runnable한 process를 MLFQ에서 가져온다.

- scheduler TODO 3 : Round-Robin and Boosting

```
int lev = p->level;

// process가 정상적으로 끝났으면 time quantum, global_ticks을 상승시킨다.
if (p->state == RUNNABLE){
  p->time_quantum++;
  MLFQ.global_ticks++;
}
else if (p->is_locked == 1){
  MLFQ.global_ticks++;
}
// whether current process is locked or not and ticks is 100 then run priority_boosting.
if(MLFQ.global_ticks >= 100){
  int flag = 0;

  // if current process is locked, then the flag value sets 1
  if(p->is_locked == 1) {
    flag = 1;
    // MLFQ Unlocking
    mlfq_unlocking(p);
  }

  // Remove the current process from MLFQ and Enqueue this process in L0 queue
  if (flag == 1){
    mlfq_dequeue(lev);

    p->level=0;
    p->priority=3;
    p->time_quantum=0;
    mlfq_push(p, 0);
  }
  // myproc이 time quantum을 다 썼으면 뒤로 보내고 boosting한다.
  else if (p->time_quantum >= 2 * lev + 4){
    if(lev < 2){
        ++p->level;
    }
    else{
      int priority = p->priority ? p->priority - 1 : 0;
      p->priority = priority;
    }
    mlfq_dequeue(lev);
    mlfq_enqueue(p, p->level);
  }
  // Run priority boosting function.

  mlfq_priority_boosting();
}
// MLFQ is not locked and current process runs out of time quantums
// if level of current process is 0 or 1 then raises the level
// else process's priority down
else if (MLFQ.is_locked == 0){

  // time quantum을 다 사용하였는지 check
  if (p->time_quantum >= 2 * lev + 4){
    p->time_quantum = 0;

    // L0, L1인경우 level을 상승시킨다.
    if(lev < 2){
        ++p->level;
    }

    // L2인경우 priority를 낮춘다. (단, 최소 0)
    else{
      int priority = p->priority ? p->priority - 1 : 0;
      p->priority = priority;
    }
  }
```

```
  // For round robin
  // 현재 process를 뒤로 보낸다.
  mlfq_dequeue(lev);
  mlfq_enqueue(p, p->level);
}
```

- process가 올바르게 작동했으면 time quantum, global tick을 올려준다. 다만, lock이 되어있을 경우에는 global tick을 계속 올리도록 하였다.

  ```
  // example test
  ...
  schedulerLock(PASSWORD);
  int pid = fork();
  if (pid != 0)
    while(wait() != -1);
  ...
  ```

  이와 같은 경우 계속 멈춰 있을 수 있으므로 lock이 되었을 경우에는 `gloabl_ticks`는 계속 올리도록 하였다, 그리고 sleep이나 wait로 너무 오랫동안 locked process만 볼 경우 성능이 올바르게 나오지 않는다고 생각해서 위와 같이 design 하였다.

- global_ticks가 100이상일 경우
  - locked process일경우 : lock을 해제하고 해당 process를 빼고 L0 queue의 맨 앞에 넣는다.
  - 아닐경우 : time quantum을 다 사용하였을 경우 MLFQ의 기준에 따라 level 변경 또는 priority 변경을 하고 조건에 맞는 queue의 맨뒤로 보낸다.

  그 후 `mlfq_priority_boosting` 을 실행한다.

- 아닐 경우
  - time quantum을 전부 사용하였으면 L0, L1은 level을 올리고 time quantum을 초기화한다. L2는 priority를 하나 감소시키는데 priority가 이미 0일경우 0으로 나둔다.

  현재 queue에서 빼고 target queue의 맨뒤에 넣는다.

2. `find_runnable_proc`

```
// in proc.c
struct proc*
find_runnable_proc(void)
{
  struct proc *r_proc = 0;
  int lev = 0, pid = -1, priority = 4;

  struct proc_queue_t *traversal;

  while(lev < MLFQ_LEVEL) {
    traversal = &MLFQ.queue[lev];

    // 해당 level을 가진 queue에 process가 없을 경우
    if(traversal->size == 0){
      ++lev;
      continue;
    }

    // circular queue이므로 L0, L1인 경우 한바퀴를 돌면서 runnable한 process를 탐색한다.
    if(lev < 2){
      for(int i = 0; i < traversal->size; ++i){
        r_proc = traversal->data[traversal->front];
        if(r_proc->state == RUNNABLE){
```

```
          return r_proc;
        }
        mlfq_dequeue(lev);
        mlfq_enqueue(r_proc, lev);
      }
    }

    // level 2 queue인경우
    // priority를 max_priority + 1로 잡고 순회하면서
    // FCFS, priority 기준으로 적절한 process를 탐색한다.
    else{
      struct proc * temp;
      pid = -1, priority = 4;
      for(int i = 0; i < traversal->size; ++i){
        int idx = (traversal->front+i)%NPROC;
        temp = traversal->data[idx];
        if(temp->state != RUNNABLE) continue;
        if(priority > temp->priority){
          priority = temp->priority;
          pid = temp->pid;
        }
      }

      // 발견하지 못했을 경우
      if (pid == -1)
        break;

      // 발견한 process가 맨 앞에 오도록 circular queue를 회전시킨다.
      r_proc = traversal->data[traversal->front];
      while(pid != r_proc->pid){
        mlfq_dequeue(lev);
        mlfq_enqueue(r_proc, lev);
        r_proc = traversal->data[traversal->front];
      }
    }

    // runnable한 process를 발견했으면 return 한다.
    if (r_proc->state == RUNNABLE) {
      return r_proc;
    }

    // runnable한 process가 없을 경우
    ++lev;
  }

  return 0;
}
```

- 이 함수는 MLFQ를 탐색하면서 RUNNABLE한 process를 탐색하는 함수이다. 다음과 같이 실행된다.
  1. level 변수가 3이상이면 0을 반환한다.
  2. 해당 level queue가 비어있으면 level을 올린고 1번으로 간다.
  3. L0, L1인경우 runnable한 process를 한바퀴를 돌리면서 runnable한 process를 탐색한다.
  4. L2인경우 runnable이면서 priority가 가장 작은 process를 찾는다. (priority가 같으면 맨 앞에 있는 것을 우선으로 한다.) 그 후 그 process가 맨 앞으로 오도록 queue를 돌린다.
  5. 찾은 process가 runnable이면 그 process를 반환한다.
  6. 못 찾은 경우 다음 level을 올려 다음 level queue를 탐색하도록 한다.

2-3. SYSTEM CALL

system call을 등록하기 위해서 다음과 같이 기존 파일들을 수정하였다.

- `usys.S`

```
// in usys.S
...
SYSCALL(yield)
SYSCALL(getLevel)
SYSCALL(setPriority)
SYSCALL(schedulerLock)
SYSCALL(schedulerUnlock)
```

- `syscall.h` : system call number를 등록한다.

```
// in syscall.h
...
#define SYS_myfunction 22
#define SYS_yield 23
#define SYS_getLevel 24
#define SYS_setPriority 25
#define SYS_schedulerLock 26
#define SYS_schedulerUnlock 27
```

- `syscall.c` : 등록할 system call의 wrapper function을 선언하고 syscall에 번호와 함수를 mapping한다.

```
// in syscall.c
...
extern int sys_yield(void);
extern int sys_getLevel(void);
extern int sys_setPriority(void);
extern int sys_schedulerLock(void);
extern int sys_schedulerUnlock(void);

static int (*syscalls[])(void) = {
...
[SYS_yield]   sys_yield,
[SYS_getLevel]   sys_getLevel,
[SYS_setPriority]   sys_setPriority,
[SYS_schedulerLock]   sys_schedulerLock,
[SYS_schedulerUnlock]   sys_schedulerUnlock,
};
```

- `sysproc.c` : 등록할 system call의 wrapper function을 정의하는 부분이다.
  - parameter가 필요한 system call일경우 argint와 그의 return value를 이용해 정상적으로 실행할 수 있을지 판단하고 paramter의 갯수가 부족한 경우 -1을 정상적으로 수행하였으면 0을 리턴한다.

```
// in sysproc.c

...

// yield
int
sys_yield(void)
{
  yield(); // yield를 호출한다.
  return 0;
}

// getLevel
```

```
int
sys_getLevel(void)
{
  return getLevel(); // GetLevel function을 호출
}

// setPriority
int
sys_setPriority(void)
{
  int pid, priority;
  if(argint(0, &pid) < 0 || argint(1, &priority) < 0) // pid, priority가 정상적으로 들어오지 않았을 경우 -1을 반환한다.
    return -1;

  setPriority(pid, priority); // 받은 parameter을 기반으로 함수를 호출
  return 0;
}

// schedulerLock
int
sys_schedulerLock(void)
{
  int password;
  if(argint(0, &password) < 0) // parameter가 제대로 들어오지 않은경우 -1을 반환.
    return -1;

  schedulerLock(password); // 받은 parameter을 기반으로 함수를 호출
  return 0;
}

// schedulerUnlock
int
sys_schedulerUnlock(void)
{
  int password;
  if(argint(0, &password) < 0) // parameter가 제대로 들어오지 않은경우 -1을 반환.
    return -1;

  schedulerUnlock(password); // 받은 parameter을 기반으로 함수를 호출
  return 0;
}
```

1. `yield`

- 기존 xv6코드 그대로 활용하였다.

2. `getLevel`

```
int
getLevel(void){
  int cur_level = myproc()->level;

  return cur_level;
}
```

- proc 구조체의 level을 return 하는 방향으로 구현하였다.

3. `setPriority`

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

  // 지정할 수 없는 priority면 setPriority를 끝낸다.
  if (priority < 0 || priority > 3) {
    cprintf("This priority %d is not allowed in MLFQ\n", priority);
    return;
  }

  acquire(&ptable.lock);

  // 해당 pid를 가진 process를 탐색후 변경
  // 없는 경우 그냥 종료한다.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if (p->pid == pid){
      p->priority = priority;
      break;
    }

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

- 우선 입력받은 priority가 지정할 수 없는 priority면 그냥 return한다.
- 아닐경우 ptable을 순회하면서 pid에 해당하는 process를 찾고 priority를 변경한다.

4. `schedulerLock`

```
void
schedulerLock(int password){
  acquire(&ptable.lock);
  // MLFQ has already locked process
  if (MLFQ.is_locked == 1) {
    cprintf("Duplicate Lock\n");
    release(&ptable.lock);
    kill(myproc()->pid);
    return;
  }

  // password is not correct
  if(password != PASSWORD){
    cprintf("%d %d %d\n", myproc()->pid, myproc()->time_quantum, myproc()->level);
    release(&ptable.lock);
    kill(myproc()->pid);
    return;
  }

  // password is correct!
  MLFQ.global_ticks = 0;
  mlfq_locking(myproc());
  release(&ptable.lock);
}
```

- 이미 잠겨있는 process가 한 번 더 잡는 경우 해당 프로세스를 종료하도록 `kill`을 호출한다.
- 만약 password를 틀렸을 경우 명세에 따라 pid, time_quantum, level을 출력하고 종료한다.
- 아닐경우 global tick을 0으로 초기화해주고 `mlfq_locking` 을 통해 해당 process를 잠근다.

5. `schedulerUnlock`

```
void
schedulerUnlock(int password){

  acquire(&ptable.lock);

  // MLFQ has already locked process
  if(MLFQ.is_locked == 0) {
    cprintf("Duplicate Unlock\n");
    release(&ptable.lock);
    return;
  }

  // password is not correct
  if(password != PASSWORD){
    cprintf("%d %d %d\n", myproc()->pid, myproc()->time_quantum, myproc()->level);
    mlfq_unlocking(myproc());
    release(&ptable.lock);
    kill(myproc()->pid);
    return;
  }

  // password is correct!
  mlfq_unlocking(myproc());
  mlfq_dequeue(myproc()->level);
  myproc()->level = 0;
  myproc()->time_quantum = 0;
  myproc()->priority = 3;
  mlfq_push(myproc(), 0);

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

- 이미 풀려있는 process인 경우 이미 풀려있다는 사실을 알려주고 return시킨다.
- 만약 password를 틀렸을 경우 명세에 따라 pid, time_quantum, level을 출력하고 종료한다.
- 아닐경우 `mlfq_unlocking` 을 통해 해당 process를 풀고, 해당 process의 level, time_quantum, priority를 초기화하고, 기존 queue에서 빼서 L0 queue의 맨 앞에 넣는다.

2-4. TRAP

- `schedulerLock`, `schedulerUnlock`을 interrupt로 등록하기 위해 다음과 같은 과정을 거쳤다.
- traps.h

```
// traps.h
...
#define T_PROCLOCK     129
#define T_PROCULOCK    130
...
```

- trap.c
  - 우선 schedulerLock, schedulerUnlock을 USER에서도 사용할 수 있도록 tvinit에 등록하였다.

- 그 후 trapno가 schedulerLock, Unlock인경우 proc.c의 `schedulerLock` `schedulerUnlock`을 호출 하였고 yield를 하고 trap을 종료하도록 하였다.

```
// in trap.c
void
tvinit(void)
{
  ...
  // LOCK UNLOCK을 USER에서도 사용할 수 있도록 권한을 변경한다.
  SETGATE(idt[T_PROCLOCK], 1, SEG_KCODE<<3, vectors[T_PROCLOCK], DPL_USER);
  SETGATE(idt[T_PROCULOCK], 1, SEG_KCODE<<3, vectors[T_PROCULOCK], DPL_USER);
  ...
}

...
void
trap(struct trapframe *tf)
{
  ...

  // scheduler lock interrupt check
  if(tf->trapno == T_PROCLOCK){
    schedulerLock(2019044711);
    yield();
    return;
  }

  // scheduler unlock interrupt check
  if(tf->trapno == T_PROCULOCK){
    schedulerUnlock(2019044711);
    yield();
    return;
  }
  ...
}
```

2-5. ETC

1. `allocproc`

- proc 구조체에서 mlfq를 위해서 추가된 변수들이 있고 할당하면 MLFQ에도 들어가야하므로 다음과 같이 allocproc을 수정하게 되었다.

```
// in proc.c
static struct proc*
allocproc(void)
{
  ...
  // for MLFQ
  p->level = 0;
  p->is_locked = 0;
  p->priority = 3;
  p->time_quantum = 0;

  // mlfq에 자리가 없으면 종료한다.
  if(mlfq_enqueue(p, p->level) != 0){
    release(&ptable.lock);
    return 0;
  }

  ...
}
```

- level, is_locked, time_quantum을 0으로 초기화 해주고 priority를 초기값인 3으로 설정하였다.
- 그 후 mlfq_enqueue를 통해서 할당된 proc을 mlfq에 넣어주었다.
  - 예외처리를 위해 혹시 자리가 없으면 ptable.lock을 release시키고 종료하도록 하였다.

3. Result

- 다음은 진행했던 테스트 및 결과들이다.

⇒ mlfq_test

- 기존 제공된 코드를 조금 수정하여서 실행하였고 L0와 L1이 1:1.5로 적절히 잘 실행되고 있음을 확인하였다.

```
$ my_userapp 1
MLFQ test start
[Test 1] default
Process 26
Process 26 , L0: 5935
Process 26 , L1: 8793
Process 26 , L2: 85272
Process 25
Process 25 , L0: 7449
Process 25 , L1: 11065
Process 25 , L2: 81486
Process 24
Process 24 , L0: 10401
Process 24 , L1: 15374
Process 24 , L2: 74225
Process 27
Process 27 , L0: 9546
Process 27 , L1: 14112
Process 27 , L2: 76342
[Test 1] finished
done
```

⇒ system call 및 interrupt를 이용한 lock/unlock test (in my_locktest.c)

- system call을 이용한 locktest
  - 결과적으로 child process에 lock이 잘 잡혀서 돌아가고 있음을 확인 할 수 있었다.

```
...
printf(1, "[Test 1] scheduler Lock/Unlock Test with system call\n");
pid = fork();
if (pid == 0) {
    schedulerLock(2019044711);
}
for(int i = 0; i < 10; i++) {
    if(pid == 0) {
        printf(1, "[child] %d\n", i);
    }
    else{
        printf(1, "[parent] %d\n", i);
    }
}
if(pid == 0){
    schedulerUnlock(2019044711);
    exit();
}
else{
    while(wait() != -1);
}
```

```
printf(1, "[Test 1] finished\n");
...
```

```
$ my_locktest 1
[Test 1] scheduler Lock/Unlock Test with system call
[child] 0
[child] 1
[child] 2
[child] 3
[child] 4
[child] 5
[child] 6
[child] 7
[child] 8
[child] 9
[parent] 0
[parent] 1
[parent] 2
[parent] 3
[parent] 4
[parent] 5
[parent] 6
[parent] 7
[parent] 8
[parent] 9
[Test 1] finished
done
```

- interrupt를 이용한 locktest
    - 또한 child process에 lock이 잘 잡혀서 돌아가고 있음을 확인하였다.

```
...
printf(1, "[Test 2] scheduler Lock/Unlock Test with interrupt\n");
pid = fork();
if (pid == 0) {
    __asm__("int $129");
}
for(int i = 0; i < 10; i++) {
    if(pid == 0) {
        printf(1, "[child] %d\n", i);
    }
    else{
        printf(1, "[parent] %d\n", i);
    }
}
if(pid == 0){
    __asm__("int $130");
    exit();
}
else{
    while(wait() != -1);
}
printf(1, "[Test 2] finished\n");
...
```

```
$ my_locktest 2
[Test 2] scheduler Lock/Unlock Test with interrupt
[child] 0
[child] 1
[child] 2
[child] 3
[child] 4
[child] 5
[child] 6
[child] 7
[child] 8
[child] 9
[parent] 0
[parent] 1
[parent] 2
[parent] 3
[parent] 4
[parent] 5
[parent] 6
[parent] 7
[parent] 8
[parent] 9
[Test 2] finished
done
```

⇒ Password Error

- 다음은 Password를 틀렸다는 가정하에 했던 테스트이다.

```
printf(1, "[Test 9] schedulerLock / Unlock Wrong Case1 : PASSWORD ERROR\n");
child = fork();
if (child == 0) { //child
  printf(1, "Process %d schedluer Lock\n", getpid());
  printf(1, "This procedure intends to scheduler Lock Error for PASSWORD\n");

  schedulerLock(PASSWORD + 1);
  // 아래부분 실행 안되어야함
  printf(1, "Process %d in Q level %d\n", getpid(), getLevel());
  schedulerUnlock(PASSWORD + 1);
  printf(1, "Process %d schedluer Unlock\n", getpid());
}
else { // parent
  int child2 = fork();
  if (child2 == 0) {
    printf(1, "Process %d schedluer Lock\n", getpid());
    schedulerLock(PASSWORD);
    printf(1, "Process %d in Q level %d\n", getpid(), getLevel());
    printf(1, "This procedure intends to scheduler Unlock Error for PASSWORD\n");
    schedulerUnlock(PASSWORD + 1);
    //아래 부분 실행 안되어야함.
    printf(1, "Process %d schedluer Unlock\n", getpid());
  }
}
exit_children();
printf(1, "[Test 9] finished\n");
```

```
$ my_userapp 9
MLFQ test start
[Test 9] schedulerLock / Unlock Wrong Case1 : PASSWORD ERROR
Process 38 schedluer Lock
This procedure intends to scheduler Lock Error for PASSWORD
38 0 1
Process 39 schedluer Lock
Process 39 in Q level 0
This procedure intends to scheduler Unlock Error for PASSWORD
39 7 0
[Test 9] finished
done
```

- 예상 결과와 같이 password를 틀릴경우 process가 종료되어 해당 process의 남은 코드가 실행이 되지 않을 것을 확인할 수 있었다.

⇒ duplication Lock / Unlock

- 나의 Design에서는 Lock을 여러번 잡는 행위를 할 경우 해당 process를 종료하도록 하고 있지만 / Unlock을 여러번 하는 행위는 막지는 않는다. 이런 경우를 가정하여 Test code를 돌려보았는데 모두 예상에 맞게 나왔다.

```
printf(1, "[Test 10] schedulerLock / Unlock Wrong Case2 : duplication ERROR\n");
child = fork();
if (child == 0) { //child
  printf(1, "Process %d schedluer Lock\n", getpid());

  schedulerLock(PASSWORD);
  printf(1, "Process %d in Q level %d\n", getpid(), getLevel());
  printf(1, "This procedure intends to scheduler Lock Error for duplication\n");
  schedulerLock(PASSWORD);
  printf(1, "Process %d schedluer Unlock\n", getpid());
}
else { // parent
  int child2 = fork();
  if (child2 == 0) {
    printf(1, "Process %d schedluer Lock\n", getpid());
    schedulerLock(PASSWORD);
    printf(1, "Process %d in Q level %d\n", getpid(), getLevel());
    printf(1, "This procedure intends to scheduler Unlock Error for duplication\n");
    schedulerUnlock(PASSWORD);
    printf(1, "Process %d schedluer Unlock\n", getpid());
    schedulerUnlock(PASSWORD);
    printf(1, "Process %d schedluer Unlock Again\n", getpid());

  }
}
exit_children();
printf(1, "[Test 10] finished\n");
```

```
$ my_userapp A
MLFQ test start
[Test 10] schedulerLock / Unlock Wrong Case2 : duplication ERROR
Process 8 schedluer Lock
Process 8 in Q level 0
This procedure intends to scheduler Lock Error for duplication
Duplicate Lock
Process 9 schedluer Lock
Process 9 in Q level 0
This procedure intends to scheduler Unlock Error for duplication
Process 9 schedluer Unlock
Duplicate Unlock
Process 9 schedluer Unlock Again
[Test 10] finished
done
```

⇒ setPriority

- setPriority Test이다. 0 ~ 3이외의 값이 들어갈경우 예외처리를 하였는데 예상 결과대로 나온 것을 확인할 수 있었다.

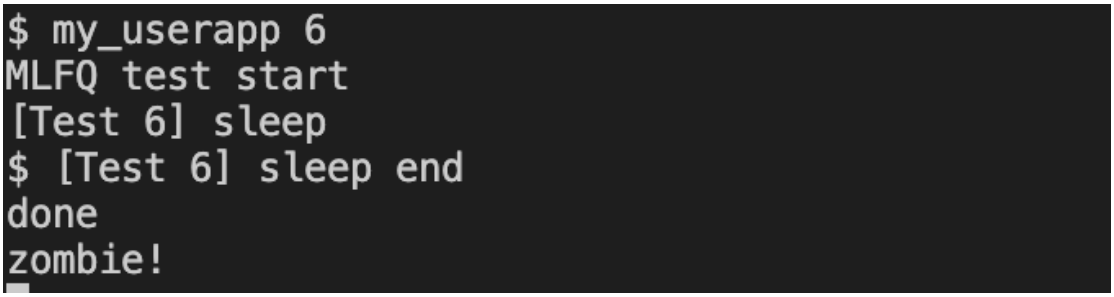
```
printf(1, "[Test 5] setPriority\n");
if (child == 0)
{
  ...
  sleep(20);
  wait();
}
else
{
  int child2 = fork();
  sleep(20);
  if (child2 == 0)
    sleep(10);
  else
  {
    setPriority(child, -1);
    setPriority(child, 11);
    setPriority(child, 10);
    setPriority(child + 1, 10);
    setPriority(child + 2, 10);
    setPriority(parent, 5);
  }
}

exit_children();
printf(1, "done\n");
printf(1, "[Test 5] finished\n");
```

```
$ my_userapp 5
MLFQ test start
[Test 5] setPriority
This priority -1 is not allowed in MLFQ
This priority 11 is not allowed in MLFQ
This priority 10 is not allowed in MLFQ
This priority 10 is not allowed in MLFQ
This priority 10 is not allowed in MLFQ
This priority 5 is not allowed in MLFQ
done
[Test 5] finished
done
```

⇒ no wait exit

- 이번에는 자식을 호출 한 뒤에 부모가 자식을 처리하는 것을 기다리지 않고 exit되었을 경우에 해본 테스트이다.
- 이는 버려진 자식을 xv6가 처리하는 지 알 수 있는 테스트 코드이다.

```
printf(1, "[Test 6] sleep\n");
schedulerLock(PASSWORD);
int pid = fork();
if (pid != 0) {
  exit();
} else {
  sleep(100);
}
printf(1, "[Test 6] sleep end\n");
break;
```



- 아래에는 실제로 xv6에서 버려진 자식 process를 어떻게 처리를 하는 지 알 수 있는데 버려진 자식의 부모를 initproc으로 바꾸고 이를 init.c에서 initproc에서 처리하는 것을 확인 할 수 있었다.

```
// exit in proc.c
// Pass abandoned children to init.
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->parent == curproc){
    p->parent = initproc;
    if(p->state == ZOMBIE)
      wakeup1(initproc);
  }
}

// main in init.c
while((wpid=wait()) >= 0 && wpid != pid)
    printf(1, "zombie!\n");
}
```

4. Trouble Shooting

우선 경험해봤던 오류는 다음과 같다.

- panic : acquire
- panic : release
- panic : zombie
- trap : 13

- trap : 14
- and so on…

1. 우선 locked process가 lock을 풀지 않고 exit하였을 경우에 scheduler에서 처리하도록 하였다. scheduler에서 locked process를 찾고 해당 process가 이미 종료된 process인 경우에는 lock을 풀고 다시 scheduler를 작동하도록 하였다.

```
// in proc.c
// scheduler

if(MLFQ.is_locked == 1){
  ...
  if (p->state == ZOMBIE){
    mlfq_unlocking(p);
    release(&ptable.lock);
    continue;
  }
}
```

2. MLFQ에서 Unused process 처리 방안

   실제 ptable에서는 wait시 zombie가 된 자식 process의 kstack의 할당을 해제하고 그 자리를 다른 process가 쓸 수 있도록 하기 위해 unused로 만든 부분이 있었다. 따라서 MLFQ에 ptable의 process의 pointer를 넣는 나의 디자인에서는 이러한 부분이 문제가 될 수 있다고 판단하였다. (unused process가 중간에 남을경우, 다른 process가 들어올경우 등..)

   이러한 부분을 해결하기 위해 `mlfq_kill` 함수를 디자인하게 되었고 이를 `wait`에서 UNUSED로 처리하기 전에 이 함수를 호출하여 mlfq 내부에서 UNUSED가 될 process를 제거하는 방향으로 디자인 하였고 공간 확보에도 신경을 쓰게 되었다.

   - wait 함수에서 mlfq_kill을 호출하는 코드이다.

```
// proc.c
// wait
if(p->state == ZOMBIE){
  // ZOMBIE가 발견되면 MLFQ내에서 지워버린다.
  mlfq_kill(p);
  // 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;
  release(&ptable.lock);
  return pid;
}
```

3. panic : acquire, release 관련

   - 흔하게 볼 수 있는 오류이며 함수 및 코드를 작성할 때 **acquire/release가** 일어났는데 한 번 더 잡는 경우는 없는지 수정할 때 **acquire/release**를 올바르게 잡거나 해제하는지 확인하면서 코드를 작성하였다. ptable은 공유자원이므로 수정하는 부분에 있어서 acquire 및 release를 하도록 하였다.

- 나의 MLFQ 디자인에서도 결국 ptable을 수정하기 때문에 schdulerLock, schedulerUnlock, setPriority 등의 함수에 `acquire(&ptable.lock);` `release(&ptable.lock);` 을 넣어주어 공유자원을 안전하게 하도록 하였다.

4. runnable process 탐색
   - 처음 runnable process를 탐색할 때 하나의 queue만 탐색하였는데 queue안에 sleep 프로세스가 있을 수도 있다는 생각을 하지 못했다. 따라서 L0부터 L2까지 runnable한 process를 순차적으로 찾도록 하였다.
5. trap 13 / trap 14
   - 결국 잘못된 page에 접근할 때 발생하는 trap이다. 13은 보호중인 page에 접근할 때 14는 없는 page에 접근할 때 발생하는 오류이며 scheduler에서 runnable한 process인지 한 번 더 판단하는 코드를 추가하였고 L2 queue에서 runnable한 process를 탐색할 때 runnable한지 우선 체크하고 priority를 4로 잡고 runnable, 이고 priority 변수보다 작은 priority이면 pid 및 priority업데이트를 통해 올바른 runnable process를 탐색하도록 하였다.

```
struct proc*
find_runnable_proc(void)
{
  ...
  // circular queue이므로 L0, L1인 경우 한바퀴를 돌면서 runnable한 process를 탐색한다.

  // level 2 queue인경우
  // priority를 max_priority + 1로 잡고 순회하면서
  // FCFS, priority 기준으로 적절한 process를 탐색한다.
  else{
    struct proc * temp;
    pid = -1, priority = 4;
    for(int i = 0; i < traversal->size; ++i){
      int idx = (traversal->front+i)%NPROC;
      temp = traversal->data[idx];
      if(temp->state != RUNNABLE) continue;
      if(priority > temp->priority){
        priority = temp->priority;
        pid = temp->pid;
      }
    }

    // 발견하지 못했을 경우
    if (pid == -1)
      break;

    // 발견한 process가 맨 앞에 오도록 circular queue를 회전시킨다.
    r_proc = traversal->data[traversal->front];
    while(pid != r_proc->pid){
      mlfq_dequeue(lev);
      mlfq_enqueue(r_proc, lev);
      r_proc = traversal->data[traversal->front];
    }
  }

  // runnable한 process를 발견했으면 return 한다.
  if (r_proc->state == RUNNABLE) {
    return r_proc;
  }
  ...

}
```

- 또한 scheduler에 의해서 선택된 process가 0인지 아닌지 체크하여 0일경우 실행하지 않도록 하였다.

```
// scheduler
if (p != 0) {
  ...
}
```

6. setPriority에서의 예외처리
    - setPriority는 0~3까지의 priority를 가질 수 있는데 이에 해당하지 않은 priority가 들어오면 예외처리를 하도록 하였다.

```
// setPriority in proc.c
// 지정할 수 없는 priority면 setPriority를 끝낸다.
if (priority < 0 || priority > 3) {
  cprintf("This priority %d is not allowed in MLFQ\n", priority);
  return;
}
```

5. 개선 사항 / 고민 사항

1. L2 queue

- L2 queue를 heap으로 구현하거나 정렬하였으면 priority boosting시에 우선순위가 높은 순서대로 L0 queue에 들어가므로 MLFQ가 좀 더 올바른 방향이 되었을 것 같다.

2. EXIT

- exit된 locked process를 scheduler에서 처리하였는데 exit함수에서 처리하는 방향이 더 나았던 것 같다. exit에서 처리하는게 좀 더 올바른 방향인지는 잘 모르겠지만 scheduler에서 처리하면 lock이 안풀린 zombie process를 탐색하기 위해 시간을 사용하므로 좋은 방향은 아니라는 생각을 하게되었다.

3. Global tick

- global tick은 process가 올바르게 작동하였을 때만 올렸지만 locked 되어있는 process가 존재할 경우에는 예외로 global tick을 계속 올리게 되었는데 sleep, wait등의 함수를 호출하여 다른 processes가 cpu를 점유하지 못하는 상황이 생길 수도 있기 때문에 이러한 Design을 하게 되었는데 이러한 부분이 올바른 Design인지 좀 더 고민하고 있다.

4. SchdulerLock / SchedulerUnlock

- lock의 경우 lock을 여러번 잡는 경우는 혼자만 cpu를 사용하려는 목적으로 판단하고 악의적이라고 판단하였고 kill을 호출하여 종료했다.
- unlock의 경우에는 unlock을 여러번한다고 해서 다른 process의 작동을 방해하는 행위는 아니므로 kill을 호출하지는 않았으며 계속 작동하도록 하였다.