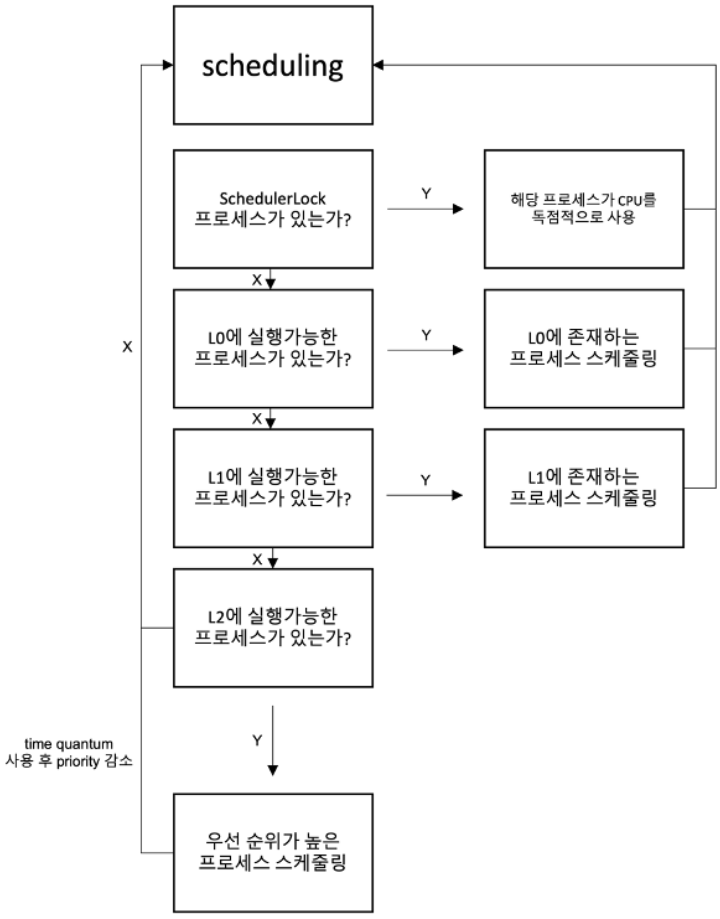
운영체제 Project 01 Wiki

해당 문서는 운영체제 Project 01(MLFQ 구현)에 대한 wiki문서이다.

한양대학교 컴퓨터소프트웨어학부 2019067429 한승우

1. Design



구현 할 스케줄러의 도식화

1) scheduler

- 우선 scheduler 함수가 실행이 되게 된다면, ptable을 한번 순회하여 L0, L1, L2 큐에 존재하는 프로세스의 개수를 각각 세어줄 것이다.

- **만약, 순회도중 schedulerLock이 된 프로세스가 발견 될 경우 무한루프를 통해 독점적으로 CPU를 사용하게 할 것이다.**

a. L0

- 프로세스가 L0큐에 존재한다면 ptable을 순회하면서 실행가능 하면서 L0큐에 존재하고 RUNNABLE상태인 프로세스에게 CPU를 할당해 줄 것이다.
- **해당 큐는 Round-Robin방식으로 동작하게 할 것이다.**

b. L1

- L0큐에 프로세스가 없고, L1큐에 프로세스가 존재한다면 L1큐의 스케줄링을 진행해준다.
- **L1큐의 스케줄링 방식도 L0과 큰 차이가 없기 때문에 Round-Robin방식으로** 같은 맥락으로 동작하게 할 것이다.

A(L0)	B(L0)	A(L0)	B(L0)	A(L0)	B(L0)	A(L0->L1)	B(L0->L1)	A(L1)	B(L1)

A,B 두개의 프로세스가 존재 할 때, Round-Robin방식으로 1tick씩 사용되는 순서를 도식화 하였다. (L0, L1 큐 동일)

c. L2

- L0큐, L1큐 모두 프로세스가 없고, L2큐에 프로세스가 존재한다면 L2큐의 스케줄링을 진행해준다.
- L2큐의 스케줄링 방식은 앞선 L0, L1의 방식과는 조금 다른데, **priority 스케줄링과 FCFS 스케줄링 방식을 모두 도입하여야 한다.**
- 이를 위해 ptable을 한번 순회하면서, **우선순위가 제일 높은 프로세스 그리고 우선순위가 같다면 가장 먼저 들어온 프로세스를 선택해서 CPU를 할당해줄 것이다.**

Priority 스케줄링										FCFS 스케줄링									
A(3)	B(3)	A(3->2)	B(3->2)	A(2)	B(2)	A(2->1)	B(2->1)	A(1)	B(1)	A(1->0)	B(1->0)	A(0)	A(0)	A(0)	A(0) 종료	B(0)	B(0)	B(0)	B(0) 종료

A,B 두개의 프로세스가 존재 할 때, L2큐에서 스케줄링이 진행되는 방식을 도식화 하였다.
(이때, L2의 time quantum은 2, A가 B보다 먼저 스케줄러에 들어왔다고 가정.)

2) schedulerLock & schedulerUnlock

a. schedulerLock

- schedulerLock이 실행된다면, **해당 프로세스의 queue Level을 -1로 체크해주어, CPU를 독점할 수 있도록 할 것이다.**

- 이때, 인자로 들어온 password가 학번과 일치하지 않다면, exit()를 사용해서 해당 프로세스를 바로 종료 할 수 있도록 할 것이다.
- 또한, 129번 인터럽트 호출 시에도 해당 함수가 실행되게 해야하는데, 이는 trap.c에서 **129번 인터럽트 호출 시 무조건 schedulerLock 함수에 학번과 일치하는 인자를 넘겨주어 무조건 schedulerLock이 동작하게 구현 할 것이다.**

b. schedulerUnlock

- schedulerUnlock이 실행된다면, 인자로 들어온 password가 학번과 일치할 때, **프로세스를 L0에 다시 넣어주고 다시 정상적으로 MLFQ가 실행 될 수 있도록 할 것이다.**
- 학번과 일치하지 않다면 exit()를 사용해서 해당 프로세스를 바로 종료하게 할 것이다.
- 또한, 130번 인터럽트 호출 시에도 해당 함수가 실행되게 해야하므로, schedulerLock과 **마찬가지로 130번 인터럽트 호출 시 scheduelrUnlcok 함수에 학번과 일치하는 인자를 넘겨주어 무조건 schedulerUnlock이 동작하게 구현할 것이다.**

3) priority_boosting

- 타이머 인터럽트 발생 시, **global tick이 100인지 확인해준 후 100ticks**가 되었다면 ptable을 모두 순회하면서 process들을 L0큐, priority는 3, time quantum을 0으로 모두 초기화 해줄 것이다.

4) Yield

- Yield는 xv6에서 기존 존재하는 함수를 그대로 사용할 것이며, **시스템 콜에 등록만 해서 유저 모드에서도 사용 할 수 있게 할 것이다.**

5) getLevel

- getLevel 시스템 콜은 간단히 현재 실행중인 프로세스가 속한 **queueLevel을 return**하게 해줄 것이다.

6) setPriority

- setPriority 시스템 콜은 pid와 priority 두 가지를 인자로 받아온다.
- ptable을 순회하면서 인자로 들어온 pid와 일치하는 프로세스가 있는지 확인 후, **존재한다면 해당 프로세스의 priority를 인자로 받아온 priority로 바꾸어 줄 것이다.**

---

2. Implement

1) proc.h

- proc.h에는 process마다 pid, queue, priority, time quantum 등이 정의 되어 있다. 각각의 기능은 다음과 같다.
  - pid → 프로세스의 큐 레벨
  - priority → 프로세스의 priority
  - ticks → 프로세스의 time quantum 소진 여부를 파악
  - ctime → FCFS를 위해 process가 각 큐에 들어온 시간을 저장
  - isLock → 만약 해당 프로세스가 Lock 후 스케줄러로 복귀 했을 때, 체크해주기 위한 변수

```
// 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 queue;                   // 프로세스의 queue level
  int priority;                // level 2에서 priority 스케줄링을 위한 변수
  int ticks;                   // 각 프로세스의 사용한 time quantum
  int ctime;                   // FCFS를 위해 process가 큐에 할당된 시간을 저장
  int isLock;                  // 만약 해당 프로세스가 lock 후 스케줄러로 복귀 했을 때, 체크해주기 위한 변수
};
```

2) proc.c

a. 변수 선언

- proc.c에서 새로 선언한 전역 변수는 queueLevel, ticks, ctime , L0, L1, L2, Lock, isLock이 있다. 각각의 기능은 다음과 같다.
  - queueLevel → 현재 스케줄링이 진행되고 있는 큐의 레벨을 나타낸다. trap.c에서도 접근할 수 있게 하기 위해 extern 변수로 선언하였다.

- ticks → priority_boosting을 위한 global ticks이다.
- ctime → 프로세스가 각 큐에 할당되는 시간을 저장하기 위한 ticks이다.
- L0, L1, L2 → 각 레벨 큐에 존재하는 프로세스의 개수를 세어주기 위한 변수이다.
- Lock → schedulerLock 후 다시 MLFQ로 돌아온 프로세스를 체크해주기 위한 변수이다.
- isLock → schedulerLock이 발생했는지 확인하는 변수

```
extern int queueLevel; //현재 스케줄링 중인 큐의 레벨을 나타냄
uint ticks; //priority_boosting을 위한 global ticks
uint ctime; //프로세스가 각 큐에 들어가는 시간을 저장하기 위한 ticks
int L0 = 0; //L0큐에 존재하는 프로세스의 개수
int L1 = 0; //L1큐에 존재하는 프로세스의 개수
int L2 = 0; //L2큐에 존재하는 프로세스의 개수
int Lock = 0; //schedulerLock 후 다시 MLFQ로 돌아온 프로세스를 체크
int isLock; //scheduelrLock이 발생했는지 확인하는 변수
```

b. allocproc

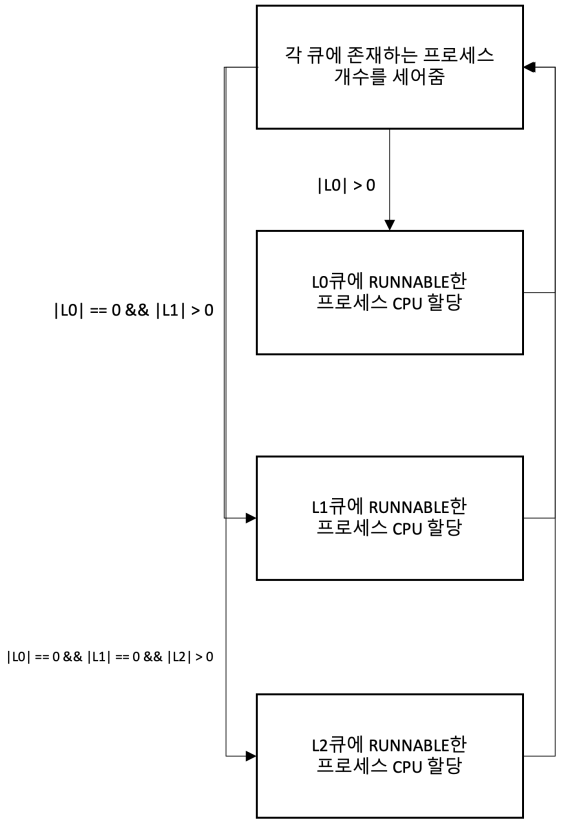
- allocproc은 프로세스가 fork, init등으로 새로 만들어질 때, 새로운 프로세스를 할당하고 초기화 하는 역할을 한다.
- **이때, ptable에 빈 곳을 찾으면 프로세스를 할당해 주는데, 이때 프로세스들의 변수를 초기화 해주었다.**

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

```
  //변수 초기화
  p->queue = 0;
  p->priority = 3;
  p->ticks = 0;
  p->ctime = ctime;
  release(&ptable.lock);
  //새로 프로세스가 할당 받을 때, L0큐의 프로세스 개수는 증가
  L0++;
```

c. scheduler

- scheduler는 기존 xv6의 round-robin 방식을 L0, L1 스케줄링에서 조금 차용을 했지만, 전반적으로 모든 부분을 바꾸어 주었다.



scheduler함수의 전반적인 진행 순서

1. **ptable을 한번 순회하면서 각 큐 레벨에 프로세스가 몇개 존재하는지 세어준다.**

```
//레벨별로 프로세스가 몇 개 존재하는지 저장하기 위한 변수
    int Lock_p = 0;
    int L_p0 = 0;
    int L_p1 = 0;
    int L_p2 = 0;

    //각 레벨별로 프로세스가 몇 개 존재하는지 세어줌
    for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
      if((p1->state == RUNNABLE || p1->state == RUNNING) && p1->pid > 0){
          //schedulerLock 상태인 프로세스가 있을 경우
          if(p1->queue == -1){
            //해당 프로세스가 종료 혹은 priority_boosting이 될 때 까지 실행
              for(;;){
                //프로세스가 종료된 경우 priority_boosting 종료
                if(p1->pid < 1 || p1->queue != -1 || p1->state != RUNNABLE){
                  isLock = 0;
                  goto end;
                }
                c->proc = p1;
                switchuvm(p1);
                p1->state = RUNNING;

                swtch(&(c->scheduler), p1->context);
                switchkvm();
```

```
                c->proc = 0;
              }
              goto end;
            }
          //schedulerLock 상태에서 MLFQ로 돌아온 프로세스를 체크해줌
          if(p1->isLock == 1){
            Lock_p++;
          }
          //L0큐에 존재하는 프로세스의 개수를 세어줌
          if(p1->queue == 0){
            L_p0++;
          }
          //L1큐에 존재하는 프로세스의 개수를 세어줌
          else if(p1->queue == 1){
            L_p1++;
          }
          //L2큐에 존재하는 프로세스의 개수를 세어줌
          else if(p1->queue == 2){
             L_p2++;
          }
        }
      }

      //전역 변수 Lock, L0, L1, L2에 큐 마다의 프로세스 개수를 저장해줌
      Lock = Lock_p;
      L0 = L_p0;
      L1 = L_p1;
      L2 = L_p2;
```

- ptable을 순회하면서 각 큐에 존재하는 프로세스의 개수를 세어 주었다.
- 이때, schedulerLock이 된 프로세스 즉, **프로세스의 큐 레벨이 -1인 프로세스가 찾아지면 무한루프를 사용하여 해당 프로세스가 CPU를 독점할 수 있도록 해준다.**
- 또한, 프로세스의 큐 레벨이 0이고, schedulerLock이 되었다가 다시 MLFQ로 복귀한 프로세스가 있을 경우 **Lock변수를 사용하여 확인 해주었다.**

2. L0큐에 프로세스가 있는 경우

```
//L0큐에 프로세스가 존재하는 경우, Round Robin 방식
    if(L0 > 0){
      //schedulerLock 상태에서 다시 MLFQ로 돌아온 프로세스가 있는 경우
      if(Lock > 0){
        for(p = ptable.proc; p< &ptable.proc[NPROC]; p++){
          //프로세스가 RUNNABLE 상태가 아니면 continue
          if(p->state != RUNNABLE)
            continue;
          //프로세스의 isLock 변수가 1이 아니라면
          if(p->isLock != 1){
            continue;
          }
          //스케줄링 도중 schedulerLock이 발생하면 해당 큐의 스케줄링 종료
          if(isLock == 1){
            goto end;
          }
          //context switching
```

```
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;

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

        c->proc = 0;
      }
    }
    //스케줄링 중인 큐 레벨을 0으로 설정
    queueLevel = 0;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      //프로세스가 RUNNABLE 상태가 아니면 continue
      if(p->state != RUNNABLE){
        continue;
      }
      //프로세스가 L0큐에 속하지 않으면 continue
      if(p->queue != 0){
        continue;
      }
      //스케줄링 도중 schedulerLock이 발생하면 해당 큐의 스케줄링 종료
      if(isLock == 1){
        goto end;
      }

      //다시 스케줄러로 돌아온 프로세스는 앞서서 CPU를 사용했기에 continue;
      if(Lock > 0 && p->isLock == 1){
        continue;
      }
      //context switching
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

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

      c->proc = 0;

    }
  }
```

- L0큐에 프로세스가 존재하는 경우 round-robin방식으로 스케줄링이 진행되게 해주었다.
- Round-Robin방식은 ptable을 순회하면서 ptable에 저장된 프로세스 순서대로 1ticks씩 할당해주는 방식으로 구현하였다.
- trap.c에서 현재 스케줄링 중인 큐의 레벨을 알 수 있도록 하기 위해서 queueLevel 변수를 0으로 설정해 주었다.
- **이때, L0큐의 스케줄링 도중 schedulerLock이 발생하면 isLock 변수를 통해 확인해주고, 발생한 경우 바로 L0 스케줄링을 종료하게 해주었다.**
- **또한, schedulerLock 상태였다가 다시 MLFQ로 복귀한 프로세스가 있는지 Lock 변수와 프로세스 구조체에 선언한 isLock 변수를 통해 확인해주고, 있는 경우 L0 큐에서 가장 우선적으로 스케줄링 되게 해주었다.**

3. L0큐에는 프로세스가 존재하지 않고, L1큐에 프로세스가 존재하는 경우

```
//L0큐에 프로세스가 존재하지 않고 L1큐에 프로세스가 존재하는 경우, Round Robin 방식
    else if(L1 > 0){
      //schedulerLock 상태에서 다시 MLFQ로 돌아온 프로세스가 있는 경우
      if(Lock > 0){
        for(p = ptable.proc; p< &ptable.proc[NPROC]; p++){
          //프로세스가 RUNNABLE 상태가 아니면 continue
          if(p->state != RUNNABLE)
            continue;
          //프로세스의 isLock 변수가 1이 아니라면
          if(p->isLock != 1){
            continue;
          }
          //스케줄링 도중 schedulerLock이 발생하면 해당 큐의 스케줄링 종료
          if(isLock == 1){
            goto end;
          }
          //context switching
          c->proc = p;
          switchuvm(p);
          p->state = RUNNING;

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

          c->proc = 0;
        }
      }
      //스케줄링 중인 큐 레벨을 1로 설정
      queueLevel = 1;
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        //프로세스가 RUNNABLE 상태가 아니라면 continue
        if(p->state != RUNNABLE){
          continue;
        }
        //만약 순회 중 Level이 0인 프로세스가 있다면, L1큐의 스케줄링을 종료
        if(L0 > 0)
          goto end;
        //프로세스의 큐 레벨이 1이 아니라면 continue
        if(p->queue != 1)
          continue;
        //스케줄링 도중 schedulerLock이 발생하면 해당 큐의 스케줄링 종료
        if(isLock == 1){
          goto end;
        }
        //다시 스케줄러로 돌아온 프로세스는 앞서서 CPU를 사용했기에 continue;
        if(Lock > 0 && p->isLock == 1){
          continue;
        }

        //context switching
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;

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

```
        switchkvm();

        c->proc = 0;

      }
```

- L0큐에는 프로세스가 존재하지 않고, L1큐에 프로세스가 존재하는 경우 L0큐와 마찬가지로 Round-Robin방식으로 동작하게 해주었다.
- Round-Robin방식은 ptable을 순회하면서 ptable에 저장된 프로세스 순서대로 1ticks씩 할당해주는 방식으로 구현하였다.
- ptable을 순회하면서 process가 RUNNABLE하고 큐 레벨이 1이라면 cpu를 할당받도록 해주었다.
- trap.c에서 현재 스케줄링 중인 큐의 레벨을 알 수 있도록 하기 위해서 queueLevel 변수를 1로 설정해 주었다.
- **이때, L1큐의 스케줄링 도중 schedulerLock이 발생하면 isLock 변수를 통해 확인해주고, 발생한 경우 바로 L1 스케줄링을 종료하게 해주었다.**
- **만약, L1큐를 스케줄링 하는 도중 init혹은 fork를 통해 새로운 프로세스가 들어오게 된다면 무조건 L1큐의 우선순위 보다 높은 L0큐에 할당이 되기에 예외처리를 위해 L0 변수를 사용하여 확인해 주었다.**

4. **L0, L1큐에는 프로세스가 존재하지 않고, L2큐에 프로세스가 존재하는 경우**

```
//L0,L1큐에 프로세스가 없고, L2에 프로세스가 있다면 priority를 기반으로 한 스케줄링 실행
else if(L2 > 0){
  //비교를 위한 변수 설정
  int priority = 100;
  uint p_time = 4294967295;
  p = 0;
  //스케줄링 중인 큐 레벨을 2로 설정
  queueLevel = 2;
  struct proc *p1;
  for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++) {
    //만약 순회 중 Level이 0, 1인 프로세스가 있다면, 종료
    if(L1 > 0 || L0 > 0)
      goto end;
    //우선순위가 더 높은 프로세스가 있다면, 비교 후 먼저 실행
    if(p1->state == RUNNABLE && p1->priority < priority && p1->queue == 2) {
      priority = p1->priority;
      p_time = p1->ctime;
      p = p1;
    }
    //우선 순위가 같은 프로세스 끼리는 FCFS 방식이기에 비교 후 먼저 실행
    else if(p1->state == RUNNABLE && p1->priority == priority && p1->queue == 2) {
      if(p1->ctime < p_time) {
        p_time = p1->ctime;
        priority = p1->priority;
        p = p1;
      }
    }
    //스케줄링 도중 schedulerLock이 발생하면 해당 큐의 스케줄링 종료
    if(isLock == 1){
      goto end;
```

```
        }
      }

      if(p != 0) {
        //context switching
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;
        swtch(&(c->scheduler), p->context);
        switchkvm();
        c->proc = 0;
      }
    }
```

- L2는 앞선 L0, L1과 다른 방식으로 priority와 FCFS를 통해서 스케줄링이 진행 된다.
- **cpu를 프로세스에게 할당해주기 전에 ptable을 한번 순회해 주면서, L2큐에 존재하는 프로세스 중에 우선순위가 가장 높고, 가장 먼저 L2큐에 들어온 프로세스에게 cpu를 할당해준다.**
- trap.c에서 현재 스케줄링 중인 큐의 레벨을 알 수 있도록 하기 위해서 queueLevel 변수를 2로 설정해 주었다.
- **이때, L2큐의 스케줄링 도중 schedulerLock이 발생하면 isLock 변수를 통해 확인해주고, 발생한 경우 바로 L2 스케줄링을 종료하게 해주었다.**
- **만약, L2큐를 스케줄링 하는 도중 L0 혹은 L1큐에 프로세스가 할당이 된다면, L0, L1변수를 통해 확인해주고 바로 L2 스케줄링을 종료하게 해주었다.**
- **이때, ctime변수는 프로세스가 해당 큐에 들어온 시간이며, priority가 같은 프로세스가 있을 때, L2큐에 들어온 시점이 더 빠르다면 해당 프로세스에게 CPU를 할당하게 해주었다.**

xv6에서 기본적으로 제공하는 yield에 레벨 별로 yield를 차별을 주기 위해 yield0_1, yield2 시스템 콜을 추가해 주었다.

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

```
//L2큐에 있는 프로세스에서 yield 실행 된 경우 priority를 감소시켜줌
void
yield2(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  if(myproc()->priority > 0){
    myproc()->priority--;
  }
```

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

```
//L0큐 혹은 L1큐에 있는 프로세스에서 yield가 실행 된 경우 큐 레벨을 늘려줌.
void
yield0_1(void)
{
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  //L1, L2큐에 존재하는 프로세스의 개수를 조정해줌
  if(myproc()->queue == 1){
    L1--;
    L2++;
  }
  //L0, L1큐에 존재하는 프로세스의 개수를 조정해줌
  if(myproc()->queue == 0){
    L0--;
    L1++;
  }
  // myproc()->isLock = 0;
  //해당 큐에 할당된 시간을 나타내는 ctime 변수 조정
  myproc()->ctime = ctime;
  //큐 레벨 증가
  myproc()->queue++;
  sched();
  release(&ptable.lock);
}
```

- yield 시스템 콜은 xv6에서 기본적으로 제공하는 시스템 콜과 동일하다.
- yield2 시스템 콜은 L2큐에서 **스케줄링 중 타임 인터럽트가 발생하면 해당 프로세스의 priority를 한단계 낮추어 주고, priority가 0보다 작거나 같을 경우 단지 CPU를 양보하도록 해주었다.**
- yield0_1 시스템 콜은 L0, L1큐에서 **스케줄링 중 타임 인터럽트가 발생하면 해당 프로세스의 queue 레벨을 한단계 높여주고, 각 큐 마다 해당 프로세스가 들어간 시간을 저장해 주었다.**

e. priority_boosting

```
//priority_boosting
void
priority_boosting(void)
{
  struct proc *p;
  acquire(&ptable.lock);
  //ptable을 모두 돌면서 time quantum, priority, Q0으로 초기화 시켜줌
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    //ptable을 모두 돌면서 time quantum, priority, 큐 레벨을 초기화 시켜줌
    if(p->queue == -1){
      p->isLock = 1;
    }
    p->ticks = 0;
    p->priority = 3;
```

```
    p->queue = 0;
    isLock = 0;
    p->ctime = ctime;
  }
  queueLevel = 0;
  release(&ptable.lock);
}
```

- priority_boosting 함수는 ptable을 모두 돌면서 각 프로세스의 time quantum, priority, 큐 레벨 등을 초기화 해준다.
- 이때, schedulerLock이 되었다가 priority_boosting이 발생해 다시 MLFQ로 돌아가는 경우 isLock 변수를 통해서 체크 해준다.
- 또한, **schedulerLock인 프로세스가 존재한다는 상태를 나타내는 isLock 변수를 0으로 초기화 해주었다.**

f. setPriority

```
//setPriority
void
setPriority(int pid, int priority)
{
  struct proc *p;
  int possible = 0;
  int p_possible = 1;
  if(priority < 0 || priority > 3){
    p_possible = 0;
  }
  if(p_possible == 0){
    cprintf("Since the PRIORITY is invalid, the program will continue to execute.\n");
  }
  else{
    acquire(&ptable.lock);
    //ptable을 돌면서 pid가 인자와 같은 프로세스를 찾고 priority를 바꾸어 줌
    for(p = ptable.proc ; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
          p->priority = priority;
          possible = 1;
          break;
        }
   }
   //인자로 들어온 pid와 같은 프로세스가 없으면 오류메시지 출력 후 아무런 변화도 안줌
   if(possible == 0){
     cprintf("Since the PID is invalid, the program will continue to execute.\n");
   }
   release(&ptable.lock);
  }
}
```

- setPriority 시스템 콜은 pid와 priority를 인자로 받아온다. ptable을 순회하면서 인자로 들어온 pid와 일치하는 process가 있을 경우 해당 프로세스의 priority를 변경해준다.

- **이때, priority가 명세의 조건인 0~3 사이가 아닐 경우 예외처리를 통해 오류 메시지를 출력해주고 아무런 행동을 취하지 않는다.**
- **ptable을 순회하면서 만약 인자로 들어온 pid와 일치하는 process가 없을 경우 오류메시지를 출력해주고 아무런 행동을 취하지 않는다.**

g. getLevel

```
//현재 프로세스의 level을 return
int
getLevel(void)
{
  struct proc *p = myproc();
  return p->queue;
}
```

- **해당 시스템콜은 현재 process가 속한 큐 레벨을 리턴해준다.**

h. schedulerLock

```
//schdulerLock
void
schedulerLock(int password)
{

  struct proc *p = myproc();
  //인자로 받은 password와 학번이 같은 경우
  if(password == 2019067429){
    if(isLock != 0){
      cprintf("There is already a schedulerLock process\n");
    }
    else{
      acquire(&tickslock);
      //global ticks를 초기화
      ticks = 0;
      wakeup(&ticks);
      release(&tickslock);
      acquire(&ptable.lock);
      //해당 프로세스의 큐 레벨을 -1로 체크해줌
      p->queue = -1;
      //schedulerLock이 발생 했음을 체크해줌
      isLock = 1;
      release(&ptable.lock);
    }
  }
  //다를 경우 오류 메시지를 출력하고 현재 process의 pid, time_quantum, queue를 출력하고 강제 종료
  else{
    //p가 schedulerLock 상태인 프로세스인 경우 예외 처리를 위해 작성
    if(p->queue == -1){
      p->queue =0;
      isLock = 0;
```

```
    }
    cprintf("Invalid password, exiting the process.\n");
    cprintf("pid \t time_quantum \t level \n");
    cprintf("%d \t %d \t \t %d \n", myproc()->pid, myproc()->ticks, myproc()->queue);
    exit();
  }
}
```

- schedulerLock은 password를 인자로 받아온다.
- **만약 인자로 받은 password와 학번(2019067429)가 일치한다면 해당 프로세스의 큐 레벨을 임의적으로 -1로 변환해주어서 체크 해준다.**
- **또한, isLock 전역 변수를 통해 schedulerLock상태인 프로세스가 존재한다는 것을 알려준다.**
- 인자로 받은 password와 학번(2019067429)가 일치하지 않는다면 오류메시지를 출력하고 현재 프로세스의 pid, time_quantum, queue를 출력하고 **exit()를 통해 종료하게 해준다.**

i. schedulerUnlock

```
//schdulerUnlock
void
schedulerUnlock(int password)
{
  struct proc *p = myproc();
  //인자로 받은 password와 학번이 같은 경우
  if(password == 2019067429){
    //프로세스의 큐 레벨이 -1, 즉 schedulerLock 상태가 아닌 경우 오류 메시지 출력 후 exit 호출
    if(myproc()->queue != -1){
      cprintf("This Process is not Locked, exiting the process.\n");
      cprintf("pid \t time_quantum \t level \n");
      cprintf("%d \t %d \t \t %d \n", myproc()->pid, myproc()->ticks, myproc()->queue);
      exit();
    }
    acquire(&ptable.lock);
    //프로세스의 queue, ticks, priority를 초기화
    p->queue = 0;
    p->ticks = 0;
    p->priority = 3;
    //scheduerLock 상태에서 MLFQ로 돌아왔음을 체크
    p->isLock = 1;
    //schedulerLock 상태가 아님을 체크
    isLock = 0;
    release(&ptable.lock);
  }
  else{
    //p가 schedulerLock 상태인 프로세스인 경우 예외 처리를 위해 작성
    if(p->queue == -1){
      p->queue =0;
      isLock = 0;
    }
    //다를 경우 오류 메시지를 출력하고 현재 process의 pid, time_quantum, queue를 출력하고 강제 종료
    cprintf("Invalid password, exiting the process.\n");
    cprintf("pid \t time_quantum \t level \n");
    cprintf("%d \t %d \t \t %d \n", myproc()->pid, myproc()->ticks, myproc()->queue);
```

```
    exit();
  }
```

- schedulerUnlock은 password를 인자로 받아온다.
- 만약, 인자로 받은 password와 학번(2019067429)가 일치한다면 해당 프로세스의 isLock 변수를 1로 체크해주어서 L0큐에서 제일 먼저 스케줄링의 대상이 되게 한다.
- 학번과 password가 일치하다면 해당 프로세스의 queue, ticks, priority를 초기화 시켜준다. 그리고, **schedulerLock 상태인 프로세스가 없다는 것을 알려주기 위해 isLock 변수를 0으로 초기화 해준다.**
- **또한, unlock을 실행시킨 프로세스가 Lock상태가 아니라면 오류메시지를 출력하고 해당 프로세스를 종료하게 해주었다.**
- 인자로 받은 password와 학번(2019067429)가 일치하지 않는다면 오류 메시지를 출력하고 해당 프로세스의 pid, time quantum, 큐 레벨을 출력하고 프로세스를 **exit()를 통해 종료하게 해주었다.**

3) trap.c

a. 변수 선언

```
extern uint ticks; //priority boosting을 위한 global ticks
extern uint ctime; //계속 증가하고 있는 global ticks
int queueLevel = 0; //현재 진행하고 있는 queueLevel
const int Q0_ticks = 4; //Q0에서의 time quantum
const int Q1_ticks = 6; //Q1에서의 time quantum
const int Q2_ticks = 8; //Q2에서의 time quantum
const int global_Limit = 100; //priority_boosting을 위한 global time quantum
```

- trap.c에서는 extern 변수로 ticks, ctime 변수와 queueLevel, Q0_ticks, Q1_ticks, Q2_ticks, global_Limit 변수를 추가해 주었다.
- ticks는 priority_boosting을 위한 global ticks이다.
- ctime은 xv6의 부팅부터 종료까지 계속해서 증가하고 있는 tick이다.
- Q0, Q1 Q2_ticks는 각 큐 레벨에서의 time quantum을 const변수로 선언해 주었다.
- global_Limit는 priority_boosting을 위해서 100틱을 const로 선언해 주었다.

b. 타임 인터럽트

```
switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      //타임 인터럽트가 발생할 때 마다 글로벌 틱과 프로세스의 틱을 증가시켜줌
```

```
      ticks++;
      ctime++;
      if(myproc()){
        myproc()->ticks++;
      }
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
```

- 타임 인터럽트 발생 시 현재 프로세스의 tick과 global tick, 그리고 ctime을 증가하게 해주었다.

c. 각 큐 레벨에서의 time quantum

```
//타임 인터럽트가 발생한 경우
    if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER){
      //L0큐에 존재하는 프로세스가 time quantum을 모두 사용했을 때
      if(myproc()->queue == 0 && myproc()->ticks > Q0_ticks && queueLevel == 0){
        //프로세스의 time quantum을 초기화 시켜주고 yield0_1 호출
        myproc()->ticks = 0;
        yield0_1();
      }
      //L1큐에 존재하는 프로세스가 time quantum을 모두 사용했을 때
      else if(myproc()->queue == 1 && myproc()->ticks > Q1_ticks && queueLevel == 1){
        //프로세스의 time quantum을 초기화 시켜주고 yield0_1 호출
        myproc()->ticks = 0;
        yield0_1();
      }
      //L2큐에 존재하는 프로세스가 time quantum을 모두 사용했을 때
      else if(myproc()->queue == 2 && myproc()->ticks > Q2_ticks && queueLevel == 2){
        //프로세스의 time quantum을 초기화 시켜주고 yield2 호출
        myproc()->ticks = 0;
        yield2();
      }
      //위의 모든 상황이 아니라면 yield 호출 (1ticks마다 CPU 양도)
      else{
        yield();
      }
     }
```

- 타임 인터럽트 발생 시, 현재 스케줄링 중인 큐의 레벨을 queueLevel을 통해 알아낸 뒤, **만약 현재 프로세스가 속한 큐의 time quantum을 모두 사용 했을 때,** 각각 L0, L1큐는 yield0_1(), L2큐는 yield2()를 호출하게 해주었다.
- **모든 프로세스는 각 큐의 time quantum을 모두 사용하는 것이 아니라 1 ticks마다 CPU를 양보하는 방식으로 진행되기 때문에 앞서 말한 3가지 상황이 아니라면 xv6에 기본적으로 있는 yield 시스템 콜을 호출하게 해주었다.**
-

d. prioirty_boosting

```
//글로벌 틱이 100틱을 넘어 갈 때, priority_boosting을 실행
    if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER && ticks > global_Limit){
      ticks = 0;
      priority_boosting();
     }
```

- **글로벅 틱이 100틱을 넘어간 경우, 글로벌 틱을 0으로 초기화 해주고 Priority_boosting을 호출하게 해주었다.**

e. 인터럽트 추가

```
//129번 인터럽트일 경우 schedulerLock을 무조건 실행
  if(tf->trapno == 129){
    schedulerLock(2019067429);
    return;
  }
  //130번 인터럽트일 경우 schedulerUnlock을 무조건 실행
  if(tf->trapno == 130){
    schedulerUnlock(2019067429);
    return;
  }
```

- trap.c에서 129번, 130번 인터럽트를 추가해주어서 schedulerLock, Unlock을 인터럽트로 실행 가능하게 해주었다.
- 이때, **항상 schedulerLock, Unlock이 실행될 수 있도록 인자로는 정확한 학번을 넘겨주었다.**

---

3. Result

1) 테스트 코드

- 아래 코드는 테스트를 위해 fork를 CHILD의 개수만큼 실행하고, 각 자식 프로세스마다 LOOP번 반복되며 해당 프로세스의 큐 레벨을 저장하여 출력해주는 파일을 작성한 코드이다.

```
//test.c
#include "types.h"
#include "stat.h"
#include "user.h"

#define CHILD 5
#define LOOP 500000

int me;
int create_child(void){
  for(int i =0  ; i<CHILD; i++){
```

```
    int pid = fork();
    if(pid == 0){
      me = i;
      sleep(10);
      return 0;
    }
  }
  return 1;
}

void exit_child(int parent) {
  if (parent)
    while (wait() != -1); // wait for all child processes to finish
  exit();
}

int main()
{
  //just scheduling

  int p;
  //Test 1

  //child 프로세스를 LOOP번 진행
  p = create_child();

  if (!p) {
    int pid = getpid();
    int cnt[3] = {0, };
    for (int i = 0; i < LOOP; i++) {
        cnt[getLevel()]++;
    }
    printf(1, "process %d: L0=%d, L1=%d, L2=%d\n", pid, cnt[0], cnt[1],cnt[2]);
  }
  exit_child(p);


  //Test 2

  //1개의 child 프로세스가 yield() 시스템 콜을 계속 진행
  p = create_child();

  if (!p) {
    int pid = getpid();
    int cnt[3] = {0, };
    for (int i = 0; i < LOOP; i++) {
      if(me == CHILD-1)
        yield();
      cnt[getLevel()]++;

    }
    printf(1, "process %d: L0=%d, L1=%d, L2=%d\n", pid, cnt[0], cnt[1],cnt[2]);
  }

  exit_child(p);

  // Test 3

  //1개의 child 프로세스가 schedulerLock 시스템 콜을 호출
  p = create_child();

  if(!p){
```

```
    int pid = getpid();
    int cnt[3] = {0, };

    for(int i= 0 ; i<LOOP; i++){
      if(me == CHILD - 3 && i == 10000){
        schedulerLock(2019067429);
      }
      cnt[getLevel()]++;
    }
    printf(1, "process %d : L0=%d, L1=%d, L2=%d\n", pid, cnt[0], cnt[1], cnt[2]);

  }
  exit_child(p);

  //Test 4

  //1개의 child 프로세스의 priority를 setPriority를 통해 우선순위가 계속 높여줌
  p = create_child();

  if (!p) {
    int pid = getpid();
    int cnt[3] = {0, };
    for (int i = 0; i < LOOP; i++) {
      cnt[getLevel()]++;
      if(me == CHILD-3){
        setPriority(pid, 0);
      }
    }
    printf(1, "process %d: L0=%d, L1=%d, L2=%d\n", pid, cnt[0], cnt[1], cnt[2]);
  }

  exit_child(p);
}
```

a. Test 1(child 프로세스들이 LOOP번 진행)

1개의 자식 프로세스를 5만번의 루프를 진행한 결과

- 1개의 자식 프로세스를 5만번 실행하였을 때, **L0, L1, L2의 비율이 4:6:56** 정도가 나왔는데, 이론적으로는 1개의 프로세스를 실행 시켰을 때, L0, L1, L2의 비율이 4:6:100정도가 나오는 것이 맞다고 결론을 지었다. **왜냐하면 L2에서는 큐 자체의 time quantum은 8tick 이지만, 한 개의 프로세스 만을 진행한다면 해당 프로세스만 스케줄링 되며 priority_boosting이 발생하기 전까지 L2에서 스케줄링 될 것 이기 때문이다**.
- 이에 대한 이유는 **루프 횟수가 작을 수록 차지하는 오버헤드의 비율이 높아질 것**이라고 예상하였고, **그래서 반복횟수를 한번 늘려보기로 하였다.**

1개의 자식 프로세스를 50만번의 루프를 진행한 결과

```
process 4: L0=50198, L1=69960, L2=879842
```

1개의 자식 프로세스를 100만번의 루프를 진행한 결과

- 1개의 자식 프로세스를 50만번 실행하였을 때, L0, L1, L2의 비율이 4:6:68정도가 나왔으며, 100만번 실행시에는 L0, L1, L2의 비율이 4:6:72정도가 나왔다. **루프의 반복 횟수가 높아질 수록 예상한 대로 4:6:100의 비율로 수렴하는 결과값을 도출 할 수 있었다.**

```
process 4: L0=47305, L1=66015, L2=386680
process 5: L0=77140, L1=107844, L2=315016
process 6: L0=97978, L1=135996, L2=266026
process 7: L0=113698, L1=158801, L2=227501
process 8: L0=125269, L1=175380, L2=199351
```

자식 프로세스가 총 5개이고 50만번의 루프를 진행한 결과

- 후에 실행된 프로세스로 갈 수록 L0의 비율이 더 높고, L2의 비율이 더 낮은 것으로 나오는데, **이는 L2의 스케줄링 방식에 의한 결과값이다.**
- **L2큐의 스케줄링 방식은 priority가 만약 같다면 먼저 L2큐에 들어오게 된 프로세스에게 CPU를 할당하는 방식이다.**
- L2큐에 모든 프로세스가 존재하는 시점을 가정하자. (이때, 프로세스들이 L2큐에 들어가는 순서는 L0, L1큐의 round-robin 방식 때문에 pid의 순서와 동일하다.)

A(3)	A(3)	A(3)	A(3)	A(3)	A(3)	A(3)	A(3->2)	A(2)	A(2)	A(2)	A(2)	A(2)	A(2)	A(2)	A(2->1)	A(1)	A(1)	A(1)	A(1)

A, B, C, D, E 프로세스는 모두 priority가 3인 상태이며, A의 큐에 들어온 시점이 가장 빠르다.

- 위의 그림은 L2큐에서 스케줄링 되고 있는 모습을 도식화 하였으며, **모두 priority가 3인 시점에서는 A가 가장 큐에 빨리 들어왔기 때문에 FCFS로 인해 CPU를 할당받게 된다.**
- 이와 같은 방식으로 진행되게 된다면, A의 priority가 가장 먼저 priority가 감소 할 것이다. **하지만 감소한 시점에서도 priority가 낮을수록 우선순위가 높기에 가장 먼저 CPU를 할당받게 된다.**
- 이러한 진행 순서로 L2에서는 A만 스케줄링 되게 되며, **priority_boosting이 호출될 때까지 나머지 프로세스들은 starvation 현상을 겪게 된다.**
- 요약하자면 **자연스럽게 A 프로세스가 종료되기 전까지는 나머지 프로세스들은 L0, L1에서만 스케줄링 될 수 밖에 없으며, A가 종료된 후 이후에도 B, C, D, E 순서로 L2를 독점하기 때문에 후에 실행된 프로세스일수록 L2의 비율이 낮게 나오는 것이다.**

```
process 4: L0=249204, L1=250796, L2=0
process 5: L0=250271, L1=249729, L2=0
process 7: L0=271387, L1=228613, L2=0
process 6: L0=269909, L1=230091, L2=0
process 9: L0=272437, L1=227563, L2=0
process 8: L0=271698, L1=226932, L2=1370
process 10: L0=2710process 00, L1=224760,11: L0=2719process  L2=4240
18, L1=2280814: L0=272, L2=0
1097, L1=22process 8903, L2=0
13: L0=272379, L1=227621, L2=0
process 12: L0=272872, L1=226635, L2=493
```

11개의 프로세스를 50만번 실행시킨 결과

- **앞서서 실행된 프로세스들은 L2큐에서 스케줄링 되지 않고, 후에 실행된 프로세스들만 L2에 작은 비율로 스케줄링 되는 모습을 볼 수 있다.**
- **이에 대한 이유는 10개 이상의 프로세스가 실행된다면 L2에 스케줄링 되기 전에 priority_boosting이 실행되어서 프로세스 개수가 많을 때에는 L2큐에서 스케줄링 되지 않다가, 앞서서 프로세스들이 종료된 후 후에 실행된 프로세스들은 L2에서 스케줄링 되기 때문이다.**

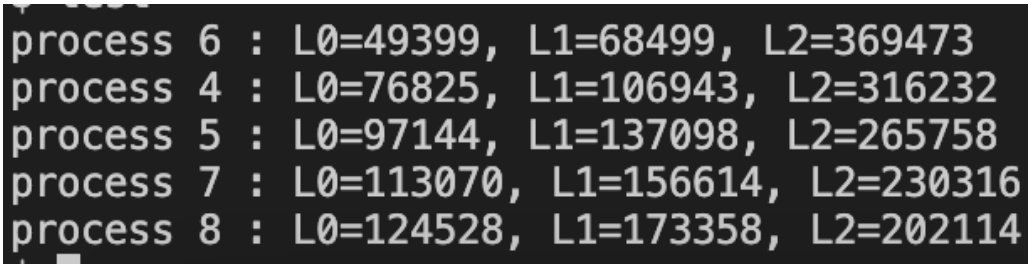
b. Test 2 (yield)

```
process 4: L0=52969, L1=74060, L2=372971
process 5: L0=84425, L1=118088, L2=297487
process 7: L0=103106, L1=143889, L2=253005
process 8: L0=119742, L1=168530, L2=211728
1 sleep  init 80104293 80105319 801064bd 801061f2
2 sleep  sh 80104293 80105319 801064bd 801061f2
3 sleep  test 80104293 80105319 801064bd 801061f2
6 run    test
1 sleep  init 80104293 80105319 801064bd 801061f2
2 sleep  sh 80104293 80105319 801064bd 801061f2
3 sleep  test 80104293 80105319 801064bd 801061f2
6 run    test
process 6: L0=64840, L1=87819, L2=347341
```

5개의 프로세스를 50만번 실행시킨 결과(process 6이 계속해서 yield를 호출)

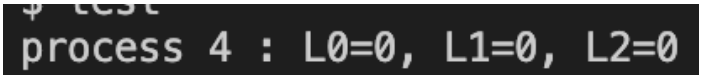
- 예상한 결과 값이 나오게 되었다. **6번 프로세스는 계속 yield함수를 호출하기에 가장 마지막으로 프로세스가 종료 되고, 나머지 4개의 프로세스들은 정상적으로 스케줄링이 되는 모습이다.**
- 하지만 이때, 6번 프로세스는 다른 프로세스보다 훨씬 긴 실행시간이 걸렸는데, 이는 **아마 yield 시스템 콜을 호출하면서 오버헤드 비율이 급격하게 커져 이에 따른 결과라고 추론을 했다.**

c. Test 3 (schedulerLock & schedulerUnlock)



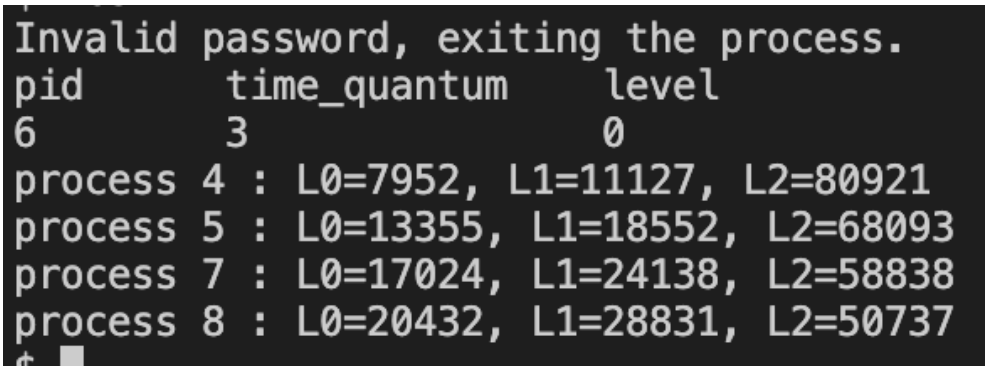
1개의 프로세스가 schedulerLock을 한번 실행

- process 4,5,7,8은 L0, L1, L2의 합이 Loop의 개수인 50만개 이지만, **process 6번은 48만개 정도의 결과 값이 나왔다.** 이러한 이유는 **schedulerLock이 실행 된다면 MLFQ에 속해서 스케줄링이 되는 것이 아닌 독점적으로 CPU를 점유하기 때문이다.**
- 이때 또한, process 6이 가장 먼저 종료가 되었는데, 이에 대한 이유는 **나의 구현 상 코드에는 ptable을 순회하면서 L0, L1을 스케줄링 해주는데, 각 큐를 스케줄링 하기 전에 schedulerLock 후 MLFQ로 돌아온 프로세스가 우선적으로 스케줄링 되기 때문이다.**



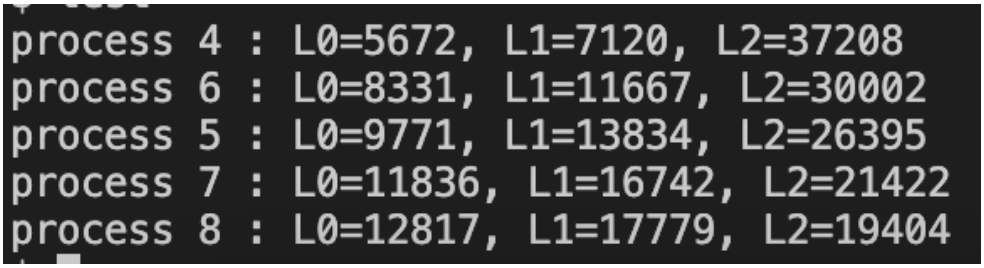
1개의 프로세스가 계속해서 schedulerLock 호출시킨 결과

- 1개의 프로세스가 계속해서 **schedulerLock을 실행시킨다면 MLFQ에서 프로세스가 진행되는 것이 아닌 독점적으로 CPU를 할당받기 때문에**, L0, L1, L2큐에서 스케줄링 된 값이 0이 나온다.



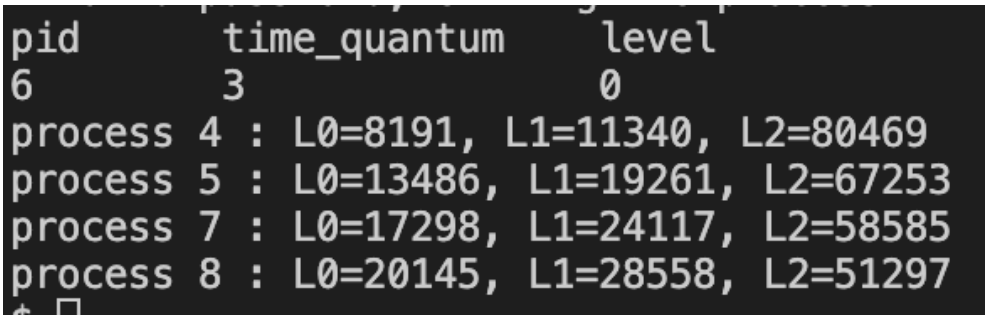
schedulerLock에 잘못된 인자를 주었을 때(process 6)

- schedulerLock에 잘못된 인자를 주었을 때는, 해당 프로세스의 pid, time_quantum, 큐 레벨을 출력하고 **프로세스를 종료하게 해주었다.**
- process 6의 출력 값이 나오지 않는 것으로 보아 정상적으로 수행 되는 것을 알 수 있다.



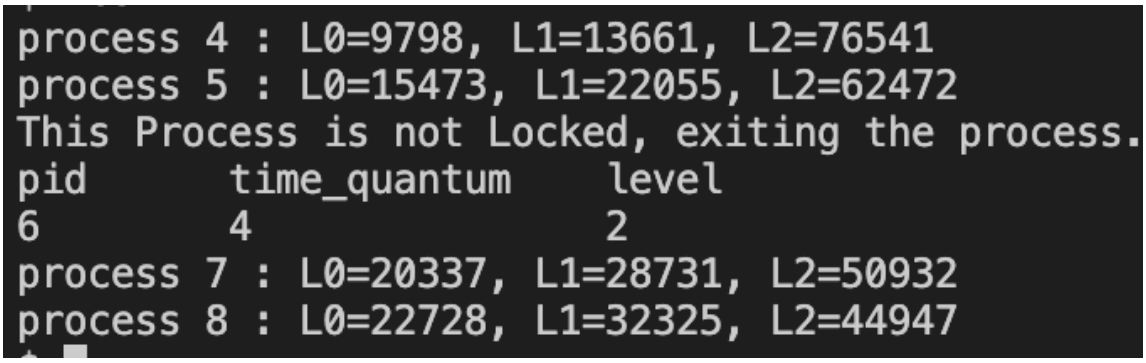
schedulerLock을 호출한 후 바로 schedulerUnlock을 호출 했을 때(process 6)

- schedulerLock을 호출한 후 바로 schedulerUnlock을 호출 했을 때는, **process 6의 L0, L1, L2의 합이 정상적으로 5만이 나오는 것을 알 수 있다.**
- 이때, 6번 프로세스가 5번 프로세스보다 먼저 종료되는 이유는 **앞선 경우와 같이 L0, L1에서 가장 우선권을 가지기 때문이다.**



schedulerUnlock에 잘못된 인자를 주었을 때(process 6)

- schedulerUnlock에 잘못된 인자를 주었을 때는, 해당 프로세스의 pid, time_quantum, 큐 레벨을 출력하고 **프로세스를 종료하게 해주었다.**
- process 6의 출력 값이 나오지 않는 것으로 보아 정상적으로 수행 되는 것을 알 수 있다.



schedulerUnlock을 호출 했지만 Lock 상태인 해당 프로세스가 프로세스가 아닐경우(process 6)

- schedulerUnlock 시스템 콜에 정상적인 인자를 전달 하였지만, 해당 프로세스가 Lock 상태가 아닐 경우, pid, time_quantum, 큐 레벨을 출력하고 **프로세스를 종료하게 해주었다.**

- process 6의 출력 값이 나오지 않는 것으로 보아 정상적으로 수행 되는 것을 알 수 있다.

```
process 4 : L0=10010, L1=13828, L2=76162
process 5 : L0=16191, L1=22299, L2=61510
There is already a schedulerLock process
process 6 : L0=20063, L1=27243, L2=39448
process 7 : L0=22336, L1=31095, L2=46569
process 8 : L0=25287, L1=34768, L2=39945
```

schedulerLock을 호출 했지만 이미 Lock 상태인 프로세스가 있을 경우(process 6→7)

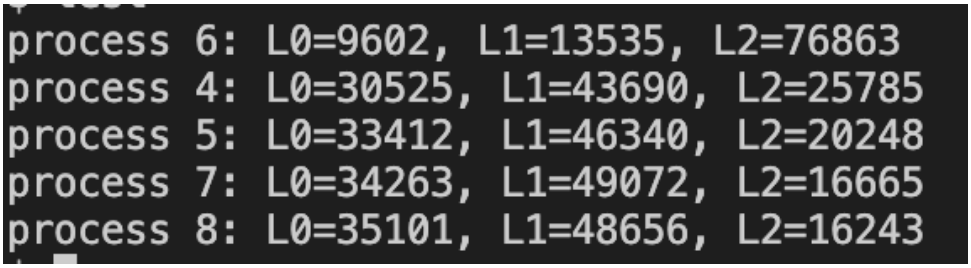
- schedulerLock 시스템 콜을 호출 했지만 이미 Lock 상태인 프로세스가 있는 경우 오류 메시지를 출력하고 아무것도 동작하지 않고 시스템 콜을 종료하도록 하였다.
- 먼저 schedulerLock을 호출한 process 6은 L0, L1, L2의 합이 86000정도로 정상적으로 schedulerLock이 수행된 모습을 알 수 있고, **process 7은 후에 schedulerLock을 수행했지만 schedulerLock 함수에서 아무것도 동작하지 않았기 때문에 합이 10만이 나왔다.**

```
process 4 : L0=8665, L1=13575, L2=77760
process 5 : L0=14925, L1=22326, L2=62749
Invalid password, exiting the process.
pid      time_quantum    level
6        8               -1
process 7 : L0=19939, L1=28548, L2=51513
process 8 : L0=22233, L1=32600, L2=45167
```

Lock상태인 프로세스에게 잘못된 인자로 schedulerLock or schedulerUnlock 호출(process 6)

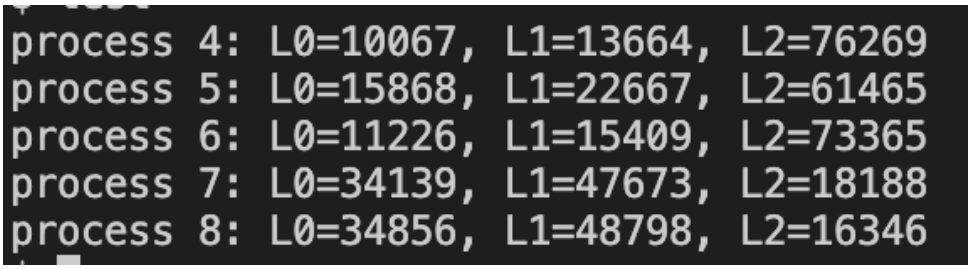
- process 6에게 schedulerLock 시스템 콜을 호출하여 Lock 상태로 만들어 준 후, **잘못된 인자로 schedulerLock이나 schedulerUnlock 시스템 콜을 호출하였을 때의 결과 값이다.**
- 이때, 6번 프로세스의 time_quantum이 8이 나오는 것으로 보아, **정상적으로 CPU를 독점하고 있음을 알 수 있다.**

d. Test 4 (setPriority)



process 6의 priority를 setPriority 시스템 콜을 통해 priority를 계속해서 0으로 조정해준 결과

- **process 6이 계속해서 setPriority 시스템 콜을 호출해서 priority를 0, 즉 가장 높은 우선순위에 배정한 결과 process 6이 가장 끝난 것은 물론 L2에서의 스케줄링 비율이 비정상적으로 높은 것을 알 수 있다.**
- L0, L1에서는 정상적으로 스케줄링이 되지만, priority 기반의 스케줄링을 하는 L2큐에 들어가는 순간 가장 우선순위가 높기 때문에, **L2큐에서는 process 6번이 priority_boosting 전까지 CPU를 독점하기 때문이다.**



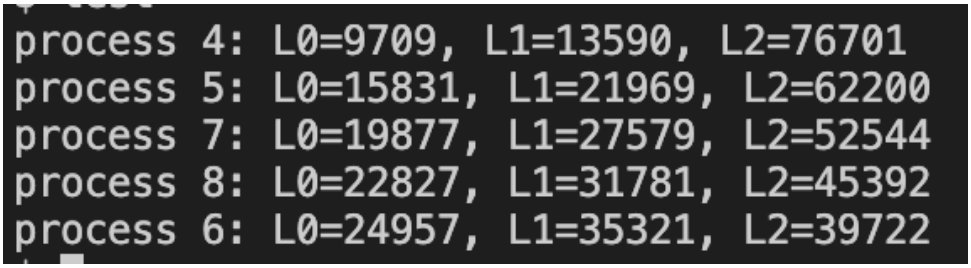
process 6의 priority를 setPriority 시스템 콜을 통해 priority를 계속해서 3으로 조정해준 결과

- process 6이 계속해서 setPriority 시스템 콜을 호출해서 priority를 3, **즉 가장 낮은 우선순위에 배정한 결과 process 6의 L2 스케줄링 비율이 높아지고, 7, 8 번의 L0, L1 스케줄링 비율은 비정상적으로 높아지고 L2 스케줄링 비율은 비정상적으로 낮아졌다.**
- 예상 결과는 process 6이 가장 나중에 종료되는 결과를 예상했지만 예상과는 다른 결과 값이 나왔다.
- 이에 대한 이유를 추론해 보았을 때,

  - process 6이 시스템 콜을 반복해서 호출하면서 오버헤드 비율이 커져 다른 프로세스들 보다 실행 시간이 길다.

  - 스케줄링 방식 자체가 가장 먼저 L2큐에 들어온 프로세스가 끝나야 후에 실행된 프로세스들이 L2에 스케줄링이 되기 때문이다.

  - 즉, process 6의 실행 시간이 길어져서 7, 8 프로세스가 L0, L1에 스케줄링 되는 비율이 높아지고, process 6의 L2 스케줄링 비율이 높아진 것이다.



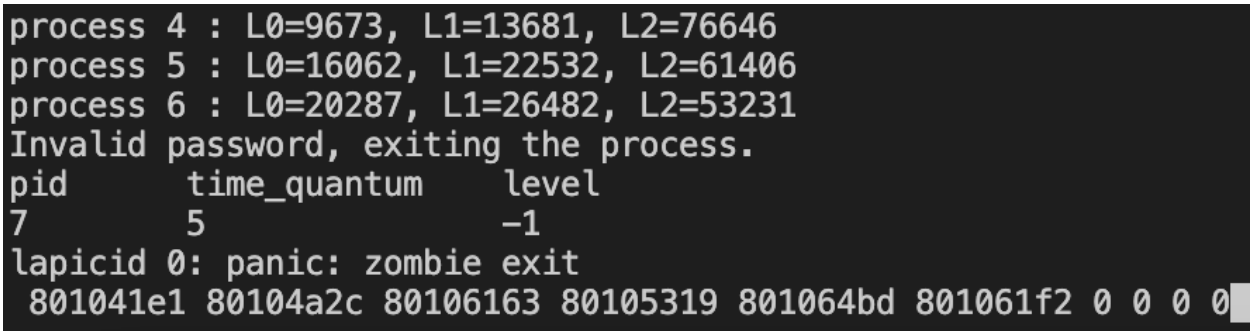
계속해서 process 4,5,7,8의 priority를 2으로 조정, process 6의 priority를 3로 조정

- 앞서 말했던 이유를 증명하기 위해서 각 프로세스들의 오버헤드 비율을 대략적으로 맞춰주어 실행시켜 보았다.
- 모든 프로세스들의 시스템 콜 호출로 인한 오버헤드 비율을 맞춰주기 위해, **계속해서 process 4,5,7,8의 priority를 2로 조정하였고, process 6의 priority를 3으로 조정해주었다.**
- 예상한 결과인 process 6번이 가장 늦게 종료 되는 것을 알 수 있다.

---

4. Trouble shooting

1) schedulerLock

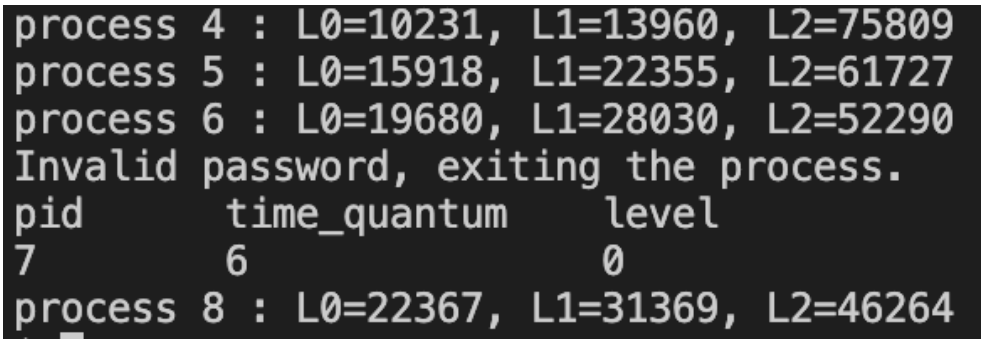


process 7번에 schedulerLock(정상 인자) 호출 후 schedulerLock or Unlock (비정상 인자) 호출한 결과

- schedulerLock 시스템 콜에 정상적인 인자를 전달한 후, 바로 schedulerLock 혹은 schedulerUnlock을 비정상적인 인자로 전달하였을 때, **panic: zombie exit이라는 오류문이 출력되었다.**
- zombie exit의 경우에는 크게 2가지가 있다. **자식프로세스 보다 먼저 부모프로세스가 종료되거나, 자식프로세스가 종료되었지만 부모 프로세스가 wait 시스템 콜을 호출하지 않는 경우이다.**
- 하지만, 해당 에러의 경우는 어느 곳에도 해당되지 않았고, 단계적으로 디버깅을 진행해 보았다.
  - 해당 에러는 **Lock 상태인 프로세스를 exit()**하는 과정에서 발생했다. → schedulerLock 이후에 exit()시스템 콜을 호출했을 때, 같은 에러가 발생하였다.
  - Lock인 상태인 프로세스만 exit()하는 과정에서 에러가 발생한다면, **Lock 상태인 프로세스가 좀비 상태로 들어간 후, 좀비 상태인 프로세스를 스케줄링해서 문제가 생긴 것이다.**

- **코드를 검토해 보았을 때, 실제로 Lock인 프로세스가 있는데 해당 프로세스에 exit() 시스템 콜을 호출한 다면 해당 프로세스의 큐 레벨은 여전히 -1이고, Lock 상태인 프로세스가 있는지 체크하는 isLock변수가 0으로 초기화 되지 않아서 발생하는 문제였다.**

```
if(p->queue == -1){
      p->queue =0;
      isLock = 0;
    }
```



정상적으로 출력되는 모습

2) 타임 인터럽트의 문제

- **처음 구현한 타임 인터럽트는 아래 코드와 같이 타임 퀀텀을 다 사용했을 때, yield함수를 호출하게 해 주었다.**

```
//queue level 0인 프로세스가 time quantum을 다 사용 했을 때, yield0_1을 실행
    if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER && myproc()->queue == 0 && myproc()->ticks > Q0_ticks && queueLevel == 0){
      myproc()->ticks = 0;
      yield0_1();
     }


    //queue level 1인 프로세스가 time quantum을 다 사용 했을 때, yield0_1을 실행
    if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER && myproc()->queue == 1 && myproc()->ticks > Q1_ticks && queueLevel == 1){
      myproc()->ticks = 0;

      yield0_1();
     }

    //qeueue level 2인 프로세스가 time quantum을 다 사용 했을 때, yield2를 실행
    if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER && myproc()->queue == 2 && myproc()->ticks > Q2_ticks && queueLevel == 2){
      myproc()->ticks = 0;
```

```
      yield2();
     }
```

A(0)	A(0)	A(0)	A(0->1) Yield	B(0)	B(0)	B(0)	B(0->1) Yield	A(1)	A(1)

위의 코드를 도식화 한 그림

- 위의 그림과 같이 해당 코드가 진행되는 방식은 프로세스가 해당 큐의 **time quantum을 모두 사용하여야 yield를 호출하여 CPU를 양도하는 방식이었다.**

```
if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER){
      if(myproc()->queue == 0 && myproc()->ticks > Q0_ticks && queueLevel == 0){
        myproc()->ticks = 0;
        yield0_1();
      }
      else if(myproc()->queue == 1 && myproc()->ticks > Q1_ticks && queueLevel == 1){
        myproc()->ticks = 0;
        yield0_1();
      }
      else if(myproc()->queue == 2 && myproc()->ticks > Q2_ticks && queueLevel == 2){
        myproc()->ticks = 0;
        yield2();
      }
      else{
        yield();
      }
     }
```

A(0)	B(0)	A(0)	B(0)	A(0)	B(0)	A(0->1) Yield	B(0->1) Yield	A(1)	B(1)

위의 코드를 도식화 한 그림

- 위의 그림과 같이 수정한 코드의 진행되는 방식은 **타임 인터럽트 발생 시에 무조건 CPU를 양도하지만 각 큐의 time quantum을 모두 사용했을 때에는 각각 큐 레벨에 맞는 yield를 호출하게 해주었다.**

3) 테스트 케이스의 문제

- 어느정도의 구현을 마친 후, **MLFQ가 제대로 동작하는지 확인할 수 있는 테스트 케이스가 없다는 것이 큰 문제였다.**
- 적절한 테스트를 위해서는 **여러개의 프로세스가 동작해야하고, 각 프로세스 마다 어느정도의 실행 시간이 보장 되어야 유의미한 결과**가 나올 것이라는 결론을 내렸다.
- **fork() 시스템 콜을 통해 프로세스를 여러개 만들어 주고, for문을 통해 각각의 프로세스마다 실행 시간을 보장해 주었다.**

- ~~테스트 케이스를 만들어서 주실줄은 몰랐다..~~

4) schedulerLock시 context switching의 문제

- schedulerLock 호출 시에 현재 스케줄링 중이던 상황을 멈추지 않고, 각 큐의 스케줄링이 모두 끝나야지만 schedulerLock을 호출한 프로세스가 CPU를 점유한다는 문제를 발견했다.
- schedulerLock 함수에서 context switching을 동작해주어, **CPU를 점유하는 프로세스를 바꾸어 주려고 시도했으나 오류 메시지가 출력이 되었다.**

```
//스케줄링 도중 schedulerLock이 발생하면 최우선적인 스케줄링을 위해 처음부터 순회
if(isLock == 1){
  goto end;
}
```

- **이를 위해 각 큐의 스케줄링 동작 중에 schedulerLock이 호출 되었다는 것을 알 수 있도록 isLock변수를 추가해 주었고, 만약 isLock변수가 1, 즉 schedulerLock을 호출한 프로세스가 있다면 바로 해당 큐의 스케줄링을 종료하게 해주었다.**

5) L1, L2 스케줄링 중 새로운 프로세스가 들어왔을 때의 문제

- 완성한 코드를 읽어내려가던 중 한 가지 의문이 들었다. **나의 코드에 의하면 L1, L2에서 스케줄링 하고 있는 상황에서 새로운 프로세스가 들어오게 된 경우 해당 프로세스의 큐 레벨은 L0이기 때문에, 최 우선적으로 스케줄링 해주었지만 그러지 않는 것처럼 보였다.**
- 특히 L1에서는 round-robin방식으로 동작하기에, **ptable의 순회가 끝나야지만 L0에 존재하는 프로세스를 스케줄링 해줄 수 있었다.**
- 이를 해결해주기 위해서 **L0, L1, L2의 개수를 저장하는 전역변수를 지정해 주었고, 새로운 process가 할당될 때 마다 L0의 개수를 하나씩 늘려주었다.**

```
//만약 순회 중 Level이 0인 프로세스가 있다면, 종료
if(L0 > 0)
  goto end;
```

- **L0, L1 큐에서 해당 코드를 추가해주어 새로운 프로세스가 할당되게 된다면 바로 해당 큐의 스케줄링을 종료하게 해주었다.**

6) schedulerLock 후 다시 MLFQ로 돌아왔을 때 문제와 구조적인 문제

- Circular Queue로 구조를 가져가지 않은 것에 대해서 후회하게 만든 부분이었다.
- L0큐에 가장 앞으로 이동을 하여야 하는데, **그 뜻은 모든 L0, L1큐의 스케줄링에서 우선 권한을 얻는 말이랑 같은 의미이다.(ptable의 순서대로 순회하기 때문)**
- 처음에는 단순히 프로세스 자체의 **isLock 변수로 L0에서 우선적으로 스케줄링 되게 구현을 하였다.(4 ticks 모두 사용)**
- 하지만, 단순히 해당 프로세스가 해당 큐에서 우선권한을 가지는 것 뿐아니라, **스케줄링 방식은 Round-Robin 방식을 충족해야한다.**
- **즉, 1ticks를 먼저 사용하는 것이지 해당 큐의 time quantum을 모두 사용하는 것이 아니라는 것이다.**

```
//schedulerLock 상태에서 다시 MLFQ로 돌아온 프로세스가 있는 경우
if(Lock > 0){
  for(p = ptable.proc; p< &ptable.proc[NPROC]; p++){
    //프로세스가 RUNNABLE 상태가 아니면 continue
    if(p->state != RUNNABLE)
      continue;
    //프로세스의 isLock 변수가 1이 아니라면
    if(p->isLock != 1){
      continue;
    }
    //스케줄링 도중 schedulerLock이 발생하면 해당 큐의 스케줄링 종료
    if(isLock == 1){
      goto end;
    }
    //context switching
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;

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

    c->proc = 0;
  }
}
```

- 각 큐의 스케줄링을 진행하기 전에, 만약 schedulerLock 상태에서 MLFQ로 돌아온 프로세스가 있는 경우, ptable을 한번 순회해 주면서 해당 프로세스를 찾아주었다.
- **해당 프로세스를 한번 CPU를 할당 해준 후 정상적으로 L0의 스케줄링을 진행해 주었다.**
- 구현 한 후 고민이 추가가 되었는데, **만약 다시 MLFQ로 돌아온 프로세스가 여러개인 상황에서는 큐에 들어온 순서를 보장 할 수 없다는 것이다.**
- ~~이를 해결하고보자 ptable자체를 건드려, 프로세스끼리의 swap으로 해결해 보고자 했으나 많은 오류에 직면했다..~~
- 이 문제를 해결하려면 프로세스를 관리하는 구조 자체를 바꿔야 한다는 결론에 이르렀고, **ptable을 순회하면서 스케줄링 해주는 방식이 아닌, 연결 리스트를 사용해서 queue를 구현한 뒤 직접 프로세스를 관리해주는 방식으로만 해결 가능하다는 생각이 들었다.**

- **구조 자체를 바꾸기에는 과제를 마무리 할 시간이 부족할 것 같아 제출한 코드에서는 schedulerLock에서 MLFQ로 돌아간 프로세스가 항상 1개보다 작거나 같은 경우에만 큐의 순서대로 동작하는 스케줄러가 완성되었다.**
- 하지만, 한 가지 든 생각은 독점적으로 CPU를 사용해서라도 일을 끝내야만하는 프로세스라면, 대부분의 경우에는 **프로세스의 실행 시간이 짧아야 만 한다는 것이다.**
- 100틱 정도를 한 프로세스가 사용한다면, 프로세스의 관점에서는 적은 시간이 아니고 **그 해당 시간동안 할 일을 마치지 못했고 L0, L1큐에서도 우선순위를 가장 높게 가진다면, 이로 인해 다른 프로세스들의 효율이 떨어질 것이라는 결론이다.**
- **즉, 여러 프로세스들이 schedulerLock을 실행했는데, 실행시킨 프로세스 모두가 100틱 안에 끝내지 못했다면 이는 나머지 프로세스들의 관점에서는 비효율적인 방식이라는 것이다.**

7) schedulerLock 상태에서 종료된 프로세스가 있는 경우의 문제

```
if(p1->queue == -1){
 //해당 프로세스가 종료 혹은 priority_boosting이 될 때 까지 실행
   for(;;){
     if(p1->pid < 1 || p1->queue != -1 || p1->state != RUNNABLE){
       goto end;
     }
     c->proc = p1;
     switchuvm(p1);
     p1->state = RUNNING;

     swtch(&(c->scheduler), p1->context);
     switchkvm();
     c->proc = 0;
   }
   goto end;
}
```

- 처음에는 위의 코드와 같이 작성을 하였는데 만약 **schedulerLock 상태인 프로세스가 schedulerLock 상태에서 종료된다면 나머지 프로세스가 제대로 스케줄링이 동작하지 않는 것을 발견했다.**
- 이는, 전역변수로 **schedulerLock 상태를 표시하는 isLock 변수를 조정할 때, schedulerLock 상태에서 종료된 프로세스를 고려하지 않아서 라고 생각이 들었다.**
- 결국, 프로세스가 만약 종료된다면, isLock 변수를 초기화 시켜주어 문제를 해결했다.

```
 if(p1->queue == -1){
 //해당 프로세스가 종료 혹은 priority_boosting이 될 때 까지 실행
   for(;;){
     //프로세스가 종료된 경우 priority_boosting 종료
     if(p1->pid < 1 || p1->queue != -1 || p1->state != RUNNABLE){
       // isLock = 0;
       goto end;
     }
```

```
            c->proc = p1;
            switchuvm(p1);
            p1->state = RUNNING;

            swtch(&(c->scheduler), p1->context);
            switchkvm();
            c->proc = 0;
          }
          goto end;
        }
```

5. 후기

- 구조적인 문제를 먼저 고려하고 구현을 시작해야 된다는 것을 깊게 깨달은 프로젝트였다.
- 사실, 아직 부족한 부분이 많이 보이지만 고질적인 구조적 문제이기 때문에 고치기 쉽지 않았다. (특히, ptable에 들어있는 순서대로 round-robin을 구현했기 때문에 L0, L1 큐에서의 동작 방식이 조금은 미흡하다.)
- 구조적으로 미흡한 부분을 보완하기 위해서 많은 고민을 하였다. 이 덕분에 OS에 대한 이해를 더 잘할 수 있었고, 프로젝트를 처음 시작할 때는 xv6에 있는 코드를 수정하기 무서웠지만 ~~(부팅조차 안되는 경우가 많았다)~~ 현재는 그에 비해 많이 발전 하였다는 것을 깨달았다.