OS_project02_2018008313

Project02

1. Xv6 scheduler 구현해야함. 구현 해야할 scheduler는 총 2가지.
   - Multilevel queue scheduler
     - Round-Robin
     - FCFS
   - MLFQ scheduler
     - Priority scheduler

Multilevel queue scheduler

1. Design
   A. 조건
      i. 스케줄러는 2개의 큐로 이루어져 있어야 한다. Pid의 홀짝을 기준으로 두가지 스케줄러가 작동하도록 하여야한다.
      ii. 짝수인 pid의 프로세스는 Round-Robin 형식으로 스케줄링 되어야 한다.
      iii. 홀수인 pid의 프로세스는 FCFS 형식으로 스케줄링 되어야 한다.
      iv. 우선순위는 RR이 항상 FCFS보다 높다.
      v. FCFS는 먼저 생성된 프로세스(pid값이 낮은)를 먼저 처리한다.
      vi. SLEEPING 상태의 프로세스는 무시한다.
   B. 구현 방법
      i. 일단 xv6는 기본적으로 RR 정책을 사용하고있다. 이는 인터럽트가 발생하면 실행중인 프로세스를 RUNNABLE 상태로 바꾸고 프로세스의 배열에서 다음 인덱스의 프로세스를 실행시킨다. 마지막까지 실행되면 처음으로 다시 돌아온다.
      ii. 따라서 Round-Robin을 따로 구현해줄 필요는 없고 짝수번째 pid의 프로세스가 xv6의 기본 스케줄러에서 동작할 수 있도록 해주면 된다.
      iii. FCFS는 먼저 오는 프로세스를 먼저 처리하는 방식이기 때문에 pid값을 비교해서 작은 프로세스를 먼저 완료시켜주기만 하면 된다.

2. Implement
   A. 스케줄러 구현 부분
      i. 내가 짠 코드로는 2가지 함수를 구현해주어야 했다. 짝수 pid가 있는지 확인하는 함수와, 가장 pid값이 작은 프로세스를 찾아주는 함수였다.
      ii. 기본으로 proc.c에 있는 scheduler는 단순히 RUNNABLE이 아닌 프로세스는 지나치고 RUNNABLE인 프로세스를 switchuvm에 넣어주고 RUNNING 상태로 바꿔준다. 그리고 해당 프로세스의 context를 scheduler context로 전환해준다. 그리

고 이후에 다시 restore할 수 있도록 해준다.

iii. 전환하는 내용은 구지 바꿀 필요가 없다고 판단했다. 그리고 위에서 pid를 확인하고 어떤 스케줄러로 들어갈지 정해줄 필요가 있다고 생각했다.

iv. 일단 RUNNABLE이 아닌 부분은 그대로 지나가고 RUNNABLE인 상황에 대해서 이야기를 해주어야 했다.

v. 어떤 프로세스를 실행시킬지 선택해야했기 때문에 프로세스 테이블에 있는 프로세스라는 의미의 proc_to_run을 선언해주었다.

vi. 먼저 프로세스 1번과 2번은 처음 실행할때 필요했기 때문에 그대로 둔다.

vii. 프로세스 테이블에 pid가 짝수인 프로세스가 하나라도 있으면 우선적으로 실행해야 한다. 그래서 pid가 짝수인 프로세스가 1개 이상 있으면 1을 return하는 check_even이라는 함수를 만들어줬다. 모든 프로세스 테이블에 대하여 피드값이 짝수이고, 2가 아니면서 RUNNABLE 상태인 프로세스가 있으면 count값을 늘려주어 판별한다.

viii. 그래서 만약 짝수가 있다고 판별을 하면 그 안에서 또 홀수인 pid가 있는지 확인해주었다. 홀수일 경우에는 continue로 넘어가고 짝수먼저 실행할 수 있도록 해주었다.

ix. 이 모든 경우를 다 제외하고 남아있는게 홀수인 pid만 있는 경우이기 때문에 이때는 먼저 온 프로세스 순서대로 처리를 해주면 될 것이다. 먼저온 프로세스는 pid값을 비교하면 되기 때문에 최솟값을 구하는 함수와 비슷하게 짜서 최솟값인 pid를 가지고 있는 프로세스를 리턴하는 형식으로 find_min_proc함수를 만들어줬다. 최대로 처리할 수 있는 프로세스 갯수가 몇개인지 잘 몰라서 그냥 int의 최대범위를 넘어가지 않는다는 가정으로 pid의 min 값을 찾아주었다. 그리고 물론 이때도 1이나 2가 포함되지 않게 해 주었다.

x. find_min_proc에서 리턴한 프로세스를 proc_to_run에 넣어 실행할 수 있도록 했다.

B. Make 부분

i. Make를 할때 Multilevel Queue와 MLFQ중에서 선택을 할 수 있도록 하려면 코드에 분기할 수 있는 부분을 만들어줘야한다. 이건 #ifdef를 사용하여 스케줄러 코드 안에서 3부분으로 쪼개서 만들어줬다. (Multilevel Queue, MLFQ, default)

ii. Ｍｕｌｔｉｌｅｖｅｌ을 선택했을땐 다른 조건이 필요 없지만 MLFQ를 선택할땐 MLFQ_K 즉 큐의 갯수에 대한 정보가 있어야하기 때문에 그 부분을 만들어줘야한다.

C. 코드 내용

D. 실행 결과

i. Makefile 부분

ii. proc.c 에서 pid값 홀수일때 가장 작은 proc를 return하는 함수. 이때도 1,2번 프로세스는 shell을 실행하는데 사용되기때문에 제외한 경우에서 최소를 찾도록 했다.

iii. proc.c에서 pid값이 짝수인 프로세스가 있는지 판별하는 함수. 이때도 위와 같은 이유로 2번 proc는 제외하고 생각해주었다.

```
95 SCHED_POLICY = DEFAULT
96 MLFQ_K = 0
97 CFLAGS += -D $(SCHED_POLICY) -D MLFQ_K=$(MLFQ_K)
```

iv. proc.c의 스케줄러 부분. #ifdef로 분기점을 구분해주었다. 실행될 프로세스 테이

```
//function to findout the min pid proc of odd pid proc
struct proc* find_min_proc(void){
    struct proc* min_proc = ptable.proc;
    struct proc* p;
    int min =2147483647;
    for(p = ptable.proc; p<&ptable.proc[NPROC];p++){
        if((p->pid) % 2 != 0 && (p->pid) != 1 &&
                (p->state) == RUNNABLE && (p->pid)<min){
            min = p->pid;
            min_proc = p;
        }
    }
    return min_proc;
```

```
#ifdef MULTILEVEL_SCHED
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  struct proc *proc_to_run = ptable.proc;
  c->proc = 0;

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

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        proc_to_run = p;
      if(p->state != RUNNABLE)
        continue;
```

블에 담길 프로세스의 이름을 proc_to_run이라고 해 주었다. 기본적으로 proc_to_run에 담기는 프로세스들은 xv6의 RR정책을 따라갈 것이니 짝수인 경우는 순서에 맞게 실행할 수 있도록 넣어주기만 하면 된다. RUNNABLE이 아닌 경우는 고려할 필요가 없으므로 continue로 지나가고, 1번 2번 프로세스는 shell 실행을 위해 그대로 두었다. 그 밖의 경우중에서 홀수로 판별된 프로세스가 있는 경우에는 그 중에서 홀수인 프로세스를 제외하고 우선적으로 테이블에 넣어서 실행시켜준다. 그리고 마지막에 홀수인 프로세스 중에서 가장 pid값이 작은 프로세스부터 테이블에 넣어서 실행할 준비를 해준다. 테이블에 들어있는 프로세스들은 기본적으로 XV6에 있는 스케줄러의 코드를 그대로 활용하여 context switcing을 시켜준다.

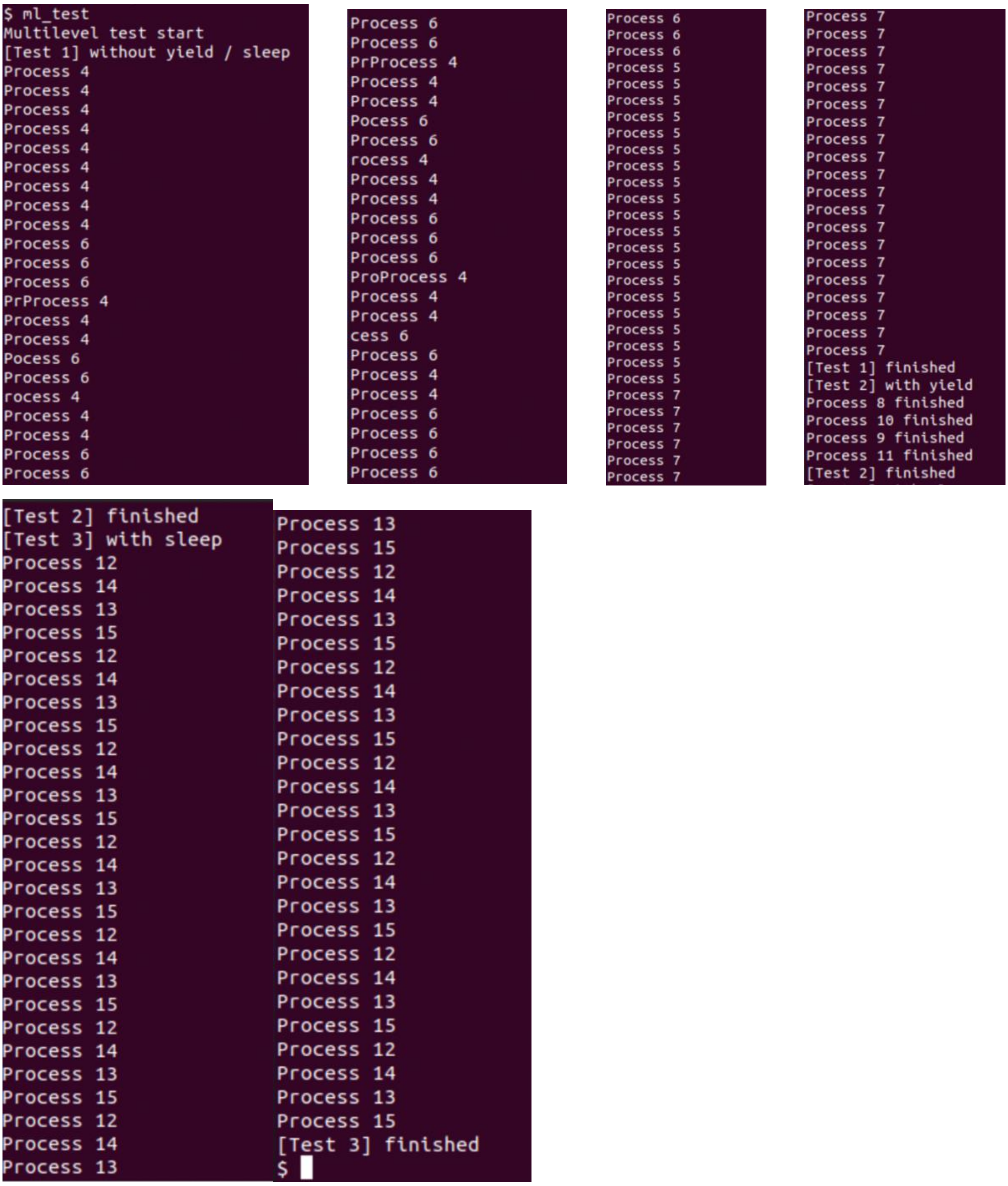
```
      else{
        if(p->pid == 1 || p->pid == 2){
            proc_to_run = p;
        }
        else if(check_even()==1){
            if((p->pid)%2!=0){
                continue;
            }
            proc_to_run = p;
        }
        else{
            proc_to_run = find_min_proc();
        }
      }

      // 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 = proc_to_run;
      switchuvm(proc_to_run);
      proc_to_run->state = RUNNING;


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

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

3. Result



```
$ ml_test
Multilevel test start
[Test 1] without yield / sleep
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 6
Process 6
Process 6
PrProcess 4
Process 4
Process 4
Pocess 6
Process 6
rocess 4
Process 4
Process 4
Process 6
Process 6
```

```
Process 6
Process 6
PrProcess 4
Process 4
Process 4
Pocess 6
Process 6
rocess 4
Process 4
Process 4
Process 6
Process 6
Process 6
ProProcess 4
Process 4
Process 4
cess 6
Process 6
Process 4
Process 4
Process 6
Process 6
Process 6
Process 6
```

```
Process 6
Process 6
Process 6
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
```

```
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
[Test 1] finished
[Test 2] with yield
Process 8 finished
Process 10 finished
Process 9 finished
Process 11 finished
[Test 2] finished
```

```
[Test 2] finished
[Test 3] with sleep
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
```

```
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
Process 12
Process 14
Process 13
Process 15
[Test 3] finished
$
```

4. Trouble shooting

   i. check_even 함수가 짝수인 pidrk 있다고 판별한 상태에서 그대로 RR로 돌리기 위해서 proc_to_run을 실행할 수 있도록 지정을 해 주었는데, 이대로 실행하면 어떤 시점에서는 홀수 pid가 짝수 pid보다 먼저 실행되는것을 확인할 수 있었다.

   ii. 이때 사실 짝수 pid가 있다는게 판별이 되더라도 그게 그 pid보다 더 먼저온 홀수 pid가 없다는 말은 아니기 때문에 짝수 pid가 하나라도 있다고 판별이 되면

그 테이블 안에 다른 홀수 pid가 있을때 그대로 흘려주는 구간이 필요하다는 사실을 깨달았다. 그래서 다시 그 pid가 홀수인지 확인을 하고, 홀수이면 continue로 보내고 그 밖의 경우에서 즉 먼저온 홀수 pid가 없는 경우에는 당연히 proc_to_run으로 넣어 먼저 실행될 수 있도록 해 주었다.

iii. Makefile의 너무 앞부분에서 make 시에 sched_policy를 입력 받도록 해주어서 다른 파일들이 컴파일 되기전에 선택되어 실행이 되지 않았던 것 같다. 그래서 분명 Multilevel Queue만 짜고 분기점이 없이 그대로 실행되게 했을때는 잘 실행되었지만, 분기점을 만들고 Make를 하면 제대로 실행되지 않았다.

iv. 분기점 설정에도 문제가 있었다. Make시에 sched_policy를 설정해준다면 그에 맞게 스케줄러를 선택해서 실행할 수 있었지만, 아무것도 넣지 않았을때에는 선택할 수 있는 스케줄러가 없었다. 그래서 아무런 조건을 선택하지 않고 make를 했을때도 실행될 수 있도록 분기점을 #else 의 경우까지 만들어줘야 했다.

MLFQ scheduler

1. Design

   A. 조건

      i. 최소 2개에서 최대 5개의 큐를 가지는 MLFQ를 설계하는것이 최종적인 목표이다.

ii. 여기서 큐의 갯수는 make 시에 입력을 받는다.

iii. 각 큐는 고유의 time quantum을 가지고, i번 큐라 할때 time quantum은 4i+2이라 한다.

iv. 처음 실행되는 프로세스는 가장 높은 0번 큐로 들어간다.

v. 각 큐는 priority가 높은 프로세스를 먼저 스케줄링한다.

vi. Priority는 0이상 10이하이고 높을수록 우선순위가 높다. 처음생긴 프로세스의 Priority는0이다.

vii. Priority가 같은 것 끼리는 어떤것을 먼저해도 괜찮다.

viii. Priority를 변경하는 setpriority시스템 콜을 만들어줘야하고, 이 시스템콜은 자신으로부터 직접 fork된 자식만 사용할 수 있다.

ix. 타이머 인터럽트, yield, sleep시에 RUNNABLE 프로세스가 존재하는 큐중 가장 높은 레벨을 선택.

x. 큐마다 이전실행한 프로세스의 time quantum이 남았다면 계속실행, 넘겼다면 priority가 가장 높은 프로세스를 선택하여 실행한다.

xi. 큐에서 프로세스 time quantum을 모두 사용했다면 다음큐로 넘어가고 실행시간은 초기화한다.

xii. 보유 큐중 가장 마지막 큐에서도 모든 time quantum을 사용했다면 이후 priority boosting 전까지는 스케줄링하지 않는다.

B. 구현 방법

i. 일단 여기에서 우리가 프로세스로 하여금 필요로하는 정보가 무엇인지를 확인하여 추가해준다.

ii. 필요한 시스템콜을 작성한다.

iii. 스케줄러에서 명세에 알맞게 구성한다.

iv. Sleep이나 yield에서 필요한 동작을 해준다.

v. Makefile부분을 작성하고 스케줄러에 분기점을 설정한다.

2. Implement

A. Proc 구조체 구현

i. 프로세스가 포함된 큐에 대한 정보, 프로세스의 priority에 대한 정보, 해당 프로세스의 tick이 얼마만큼 되었는지 확인할 수 있는 정보가 필요하다. 이건 기본적으로 구현 되어있지 않기 때문에 프로세스 구조체에 따로 구현한다.

B. 시스템콜 구현

i. 현재 프로세스의 레벨 정보를 불러오는 getlev 시스템콜과 priority를 설정해주는 setpriority 시스템콜을 만들어줘야한다.

```
// 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)
  uint ticks;
  int priority;
  int level;
};
```

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

int sys_setpriority(void){
    int pid,priority;
    if(argint(0,&pid)<0){
        return -1;
    }
    if(argint(1,&priority)<0){
        return -1;
    }
    return setpriority(pid,priority);
}
```

```
5 int getlev(void){
6       struct proc* p = myproc();
7       return p->level;
8 }
9
 //pid should be child process of current process
 int setpriority(int pid, int priority){
     struct proc* p;
     //process to change priority(child)
     struct proc* current_process=myproc();
     //process that call this function(parent)

     if(priority<0 || priority>10){
         return -2;
     }
     else{
         for(p=ptable.proc;p<&ptable.proc[NPROC];p++){
             if(p->parent->pid == current_process->pid){
                 acquire(&ptable.lock);
                 p->priority = priority;
                 release(&ptable.lock);
                 return 0;
             }
             else{
                 continue;
             }
         }
     }
     return -1;
 }
```

ii. getlev 시스템콜은 단순히 현재 프로세스의 레벨을 리턴해준다.

iii. setpriority 시스템콜은 과제 명시에서 말한 조건들을 기반으로 만들어준다. 이 시스템콜을 호출하는건 부모이고 이 시스템콜에서 인자로 받는 프로세스의 pid는 해당 프로세스의 자식에 해당하는 프로세스의 pid이다. 일단 priority가 0과10 사이에 없다면 -2를 return 해주고 그 밖의 경우에 대해 고민해야 한다.

iv. 명세에서 말한 것처럼 만약 입력받은 pid의 프로세스가 자식 프로세스가 아니면 설정하지 못하고 -1을 return 하도록 해준다. 이때 부모의 pid와 자식의 parent의 pid를 비교하여 부모자식 관계를 확인한다. 일치한다면 priority를 변경할 수 있게끔 한다. 그리고 변경에 성공한 경우에는 0을 return한다.

C. 필요한 함수 구현

i. 스케줄러를 구성하기 전에 priority_boosting함수를 만들어주었다. 이는 해당 프로

```
int check_queue_level(void){
    struct proc* p;
    int max_level = 4;
    int count=0;
    for(p=ptable.proc; p<&ptable.proc[NPROC];p++){
        if((p->state) == RUNNABLE && (p->level) < max_level
                && (p->pid) > 2){
            max_level = p->level;
            count++;
        }
    }
    if(count==0){
        return -1;
    }
    return max_level;
}
void priority_boosting(void){
    struct proc* p;

    for(p=ptable.proc; p<&ptable.proc[NPROC];p++){
        if(p->state == RUNNABLE){
            p->level = 0;
            p->ticks = 0;
        }
    }
}

struct proc* high_priority(int q_level){
    struct proc* p;
    struct proc* highest=0;
    int max=0;

    for(p=ptable.proc;p<&ptable.proc[NPROC];p++){
        if(p->priority > max && p->state == RUNNABLE
                && p->level == q_level && p->pid>2){
            max = p->priority;
            highest = p;
        }
    }
    return highest;
}
```

세스의 큐 level과 ticks를 초기화 해주는 기능을 한다. 테이블에 있는 모든 프로세스로 하여금 초기화를 해주어야 하기 때문에 for문으로 RUNNABLE한 모든 프로세스에 동일한 동작을 실행해준다.

ii. 같은 큐에서 동작할때에는 가장 priority가 높은 프로세스를 실행해주어야 하기 때문에 가장 priority가 높은 프로세스를 return하는 high_priority라는 함수를 하나 만들어줬다.

iii. 동작은 Multilevel Queue에서 만든 find_min_proc과 비슷하게 만들었다. 단 여기서는 priority값을 비교하여 가장 높은 priority를 가진 프로세스를 return 한다.

iv. 큐의 level을 확인해주는 함수도 하나 만들어주었다. 만약 처음 생성하였다면 -1을 return하여 없다는 것을 확인하여주고, 다른 경우에는 가장 높은 큐의 level을 return할 수 있도록 해주었다.

D. 스케줄러 구현

i. 스케줄러를 구성할땐 실행할 프로세스가 들어갈 프로세스 테이블을 만들어준다.

ii. 총 ticks를 다 세어주는 int변수를 0으로 초기화 해서 시간마다 하나씩 늘려준다.

```
#elif MLFQ_SCHED
void scheduler(void){
    struct proc *p;
    struct cpu *c = mycpu();
    struct proc *proc_to_run = ptable.proc;
    c->proc = 0;
    int ticks_count = 0;
    struct proc *change=0;
    for(;;){
        // Enable interrupts on this processor.
        sti();

        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            int queue_level=0;
            if(p->state != RUNNABLE){
                continue;
            }
```

iii. 이 경우에도 RUNNABLE이 아닌 프로세스는 continue로 넘기고 생각한다.

iv. 만약 pid가 1이나 2이면 shell을 만들면서 생성되는 프로세스이기 때문에 그대로 넘어가준다.

v. 그리고 그밖의 경우에 큐의 level을 가지고 명세에 나오는 내용을 하나씩 실행한

```
            else{
                if(p->pid == 1 || p->pid ==2){
                    proc_to_run = p;
                    //first started proc
                }
                else{
                    queue_level = check_queue_level();
                    //queue
                    if(queue_level==-1 || ticks_count>100){
                        //no queue available
                        //or ticks are over 100
                        priority_boosting();
                    }
                    else{
                        change=high_priority(queue_level);
                        if(change->ticks > ((4*queue_level)+2)){
                            if(queue_level!=MLFQ_K-1){
                                change->level = queue_level+1;
                                change->ticks = 0;
                            }
                            else{
                                continue;
                            }
                        }
                        else{
                            proc_to_run = change;
                        }
                    }
```

```
                }
            }

            //cprintf("pid : %d, priority : %d, level : %d\n", proc_to_


            // 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 = proc_to_run;
            switchuvm(proc_to_run);
            proc_to_run->state = RUNNING;
            proc_to_run->ticks++;
            ticks_count++;
            swtch(&(c->scheduler), proc_to_run->context);
            switchkvm();

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

다.

vi. Tick이 100이 넘거나 큐가 존재하지 않으면 priority_boosting을 호출한다. 그리고 그밖에는 해당 레벨의 가장 높은 우선순위에 있는 프로세스의 tick이 해당 큐의 tick제한을 넘어간 상황이면 큐의 레벨을 확인해 마지막 큐일때는 priority_boosting이 실행되기전에는 더이상 진행되면 안되므로 그대로 넘기고, 마지막 큐가 아니면 해당 큐의 다음큐로 레벨을 이동시키고 ticks를 초기화한다.

vii. 이 모든 경우를 지나서는 지금 가장 높은 priority를 가진 프로세스를 테이블에 올린다.

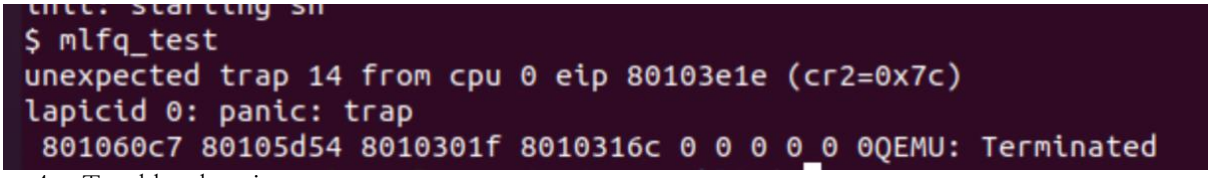
viii. 프로세스 테이블에 있는 프로세스를 switch해주면서는 ticks을 1 증가시켜준다. 그리고 총 ticks인 ticks_count도 마찬가지로 늘려준다.

E. sleep, yield 에서의 초기화 구현

i. 직접 sleep과 yield에서 level과 ticks를 구현하면 커널에 바로 접근하게되기 때문에 시스템콜 수준에서 초기화를 시켜주어야 한다.

ii. sys_yield와 sys_sleep에 해당 프로세스의 level과 ticks를 초기화하는 코드를 넣어준다.

3. Result



4. Trouble shooting

i. 현재 코드를 실행했을때 지속적으로 trap14가 뜨는 상황이다. 스케줄러의 조건문 부분부분마다 프린트를 해서 어느부분에서 실행이 끊기는지 확인 해보았으나 정확한 해결책을 찾지 못했다. 일단 shell 창이 실행되는걸 보면 1,2번 pid에서는 문제없이 동작하는 것 같으나 어떤 문제가 발생하는지 특정짓지 못했다.

ii. proc 구조체에 추가된 변수들의 allocation 문제인줄 알고 allocproc 함수에서 초기화 시켜 주었으나 변화는 없었다.