Project2

Project2. Scheduler - 2018008877 이상원

Before beginning

시작하기에 앞서 기존 xv6의 scheduler는 Round-Robin 정책을 사용하고 있다. timer interrupt가 발생하면 `yield()` 가 호출되어 현재 실행 중인 process가 RUNNABLE 상태로 전환되고 `sched()` 가 호출되어 scheduler로 switching이 발생하게 된다. 이후 ptable 에서 현재 process의 다음에 위치해 있는 RUNNABLE process를 실행시키게 된다. 이번 project는 multilevel queue scheduler와 multilevel queue feedback scheduler를 구현하는 것이 목표이다.

일단 project를 시작하기 전 system call로 등록 되어있지 않은 `yield()` 를 system call에 등록 해주어야 한다.

```
// sysproc.c
int sys_yield() { return yield(); }
```

그리고 *syscall.h* 에 `#define SYS_yield 24` 를 넣어주었고, *syscall.c*의 syscalls 배열에 `[SYS_yield] sys_yield` 를 추가해 주었다.

Project2-1: Multilevel queue scheduler

실행하는 법 (xv6-public 폴더 내부에서)

1. make clean
2. make SCHED_POLICY=MULTILEVEL_SCHED -j
3. make fs.img SCHED_POLICY=MULTILEVEL_SCHED -j
4. ./bootxv6.sh 혹은 qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512

Design

먼저 scheduler는 Round-Robin과 FCFS 두 가지의 queue로 이루어져 있다. Round-Robin은 tick이 발생할 때마다 다음 순서의 process가 실행되는 기존의 정책을 따르고, FCFS는 아직 먼저 생성된 (즉, process id가 작은) 프로세스의 작업이 모두 끝나야만 다음

process가 실행되는 정책을 따른다. 이 두 queue에 담기는 기준은 process id 이다. pid가 짝수인 process는 Round-Robin, 홀수인 process는 FCFS로 scheduling 된다. 내가 떠올린 design은 다음의 순서로 구성되어 있다.

1. **proc structure에 queue의 종류를 구분하기 위한 멤버 변수 추가**

   levelOfQueue 멤버 변수를 추가할 것이며 level이 낮을 수록 우선 순위가 높은 queue 이다.

2. **process 생성 시, 담겨질 queue 구분**

   처음 `allocproc()` 에서 process가 만들어질 때, pid에 따라 levelOfQueue 멤버 변수를 설정해 줄 것이다.

3. **ptable 순회하며 실행해야 할 process가 담긴 queue 선택**

   `scheduler()` 내부에서 ptable lock을 획득한 이후 ptable을 순회하며 levelOfQueue가 0인 RUNNABLE process가 존재할 경우 기존 scheduler의 Round-Robin 방식을 활용해 scheduling 한다.

4. **Round-Robin queue에서의 scheduling은 기존의 방법 유지**

5. **FCFS queue 선택해야 한다면, pid가 제일 낮은 process 선택**

   FCFS queue에서는 pid가 낮은 process의 모든 작업이 끝나야 다음으로 pid가 작은 process가 실행될 수 있다.

6. **FCFS queue에 있는 process 선택 시, non-preemption**

   FCFS queue에 있는 process가 선택되고 실행되면, timer interrupt가 생겨도 `yield()` 를 호출하지 않고 작업이 끝날 때까지 실행되도록 할 것이다.

Implement

Design 순서를 바탕으로 구현을 완료하였다.

1. **proc structure에 queue의 종류를 구분하기 위한 멤버 변수 추가**

   levelOfQueue 변수는 Round-Robin의 경우 0, FCFS의 경우 1 이 담겨야 한다.

   ```
   // proc.h
   struct proc {
     ...
     int levelOfQueue;
   };
   ```

2. **process 생성 시, 담겨질 queue 구분**

p→pid 가 짝수인 경우는 levelOfQueue가 0이 되고, 홀수인 경우엔 1이 된다.

```
// proc.c
  ...
found:
  p->state = EMBRYO;
  p->pid = nextpid++;

#ifdef MULTILEVEL_SCHED
  p->levelOfQueue = (p->pid % 2);
#endif
  ...
```

3. **ptable 순회하며 실행해야 할 process가 담긴 queue 선택**

   ptable를 순회하며, p→state이 RUNNABLE이고 levelOfQueue 가 0, 즉 Round-Robin queue에 담긴 process가 존재한다면 roundRobin flag를 1로 바꾸어 준 후 순회를 종료한다.

```
// proc.c
...
char roundRobin = 0;

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state == RUNNABLE && p->levelOfQueue == 0) {
    roundRobin = 1;
    break;
  }
}
...
```

4. **Round-Robin queue에서의 scheduling은 기존의 방법 유지**

   roundRobin flag가 1이라면, Round-Robin queue에서 scheduling을 해야 한다. timer interrupt가 발생하면 `yield()` 가 호출되어 현재 실행 중인 process가 RUNNABLE 상태로 전환되고 `sched()` 가 호출되어 scheduling이 실행된다. 이후 ptable 에서 현재 process의 다음에 위치해 있는 RUNNABLE 하고 Round-Robin queue에 담겨있는 process를 실행시키게 된다.

```
// proc.c
...
if(roundRobin){
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE || p->levelOfQueue != 0)
          continue;
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;
```

```
        swtch(&(c->scheduler), p->context);
        switchkvm();
        c->proc = 0;
      }
    }
...
```

5. **FCFS queue 선택해야 한다면, pid가 제일 낮은 process 선택**

   roundRobin flag가 1이 아니라면, FCFS queue에서 scheduling을 해야 한다. 이때 FCFS queue에 담겨있는 process 중 pid가 가장 작은, 즉 가장 먼저 생성된 process가 선택되어야 한다. 적절한 process가 선택되면 해당 process가 실행된다.

```
// proc.c
...
else{
      struct proc *point = 0;
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE || p->levelOfQueue != 1)
          continue;
        if(!point || (point->pid > p->pid))
          point = p;
      }
      if(point){
        c->proc = point;
        switchuvm(point);
        point->state = RUNNING;
        swtch(&(c->scheduler), point->context);
        switchkvm();
        c->proc = 0;
      }
    }
...
```

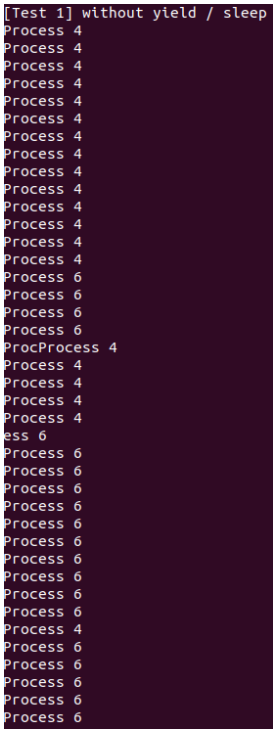
6. **FCFS queue에 있는 process 선택 시, non-preemption**

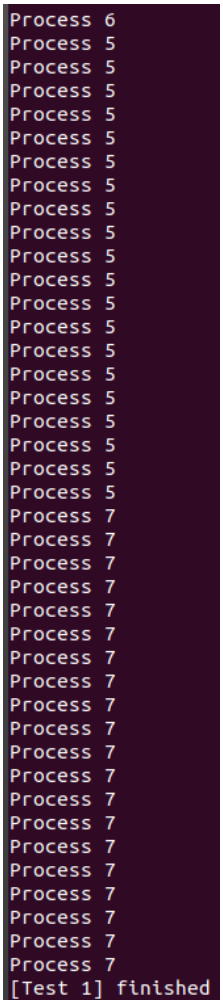
   timer interrupt가 발생하였을 때, 현재 실행 중인 process가 Round-Robin queue에 담긴 process 일 때에만 `yield()` 를 호출하도록 하였다. 이렇게 하면 FCFS queue에 있는 process는 실행 중에 cpu 선점을 빼앗기지 않게 된다.

```
// trap.c
...
#ifdef MULTILEVEL_SCHED
  if(myproc() && myproc()->levelOfQueue == 0 && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
#endif
...
```

Result

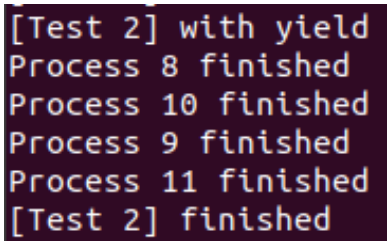
- **Test1**





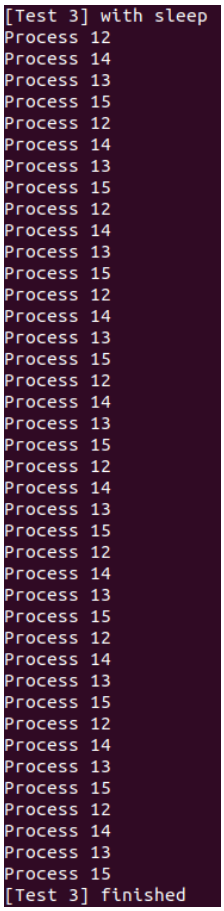
왼쪽 → 오른쪽 순으로 실행 되었다. process 4와 6은 Round-Robin queue에 담겨있기 때문에, FCFS queue에 담겨있는 process 5와 7보다 먼저 실행된다. ptable에서의 index가 빠른 process 4가 먼저 실행이 되고 tick이 증가할 때마다 6과 context switch가 발생한다. process 4와 6의 작업이 모두 끝나면 Round-Robin queue가 비게 되고 이제 FCFS queue에 있는 process가 실행될 수 있게 된다. pid가 적은 process 5부터 실행되고 모든 작업이 끝날 때까지 이어서 실행이 되는 모습을 볼 수 있다.

- **Test2**



Round-Robin queue에 담겨있어서 yield가 발생할 때마다 번갈아 실행되는 process 8과 10은 동시에 작업이 끝나는 것을 확인할 수 있었고, FCFS queue에 담겨있는 process 9와 11은 9가 끝난 뒤 시간 간격을 두고 11이 끝나는 것을 확인할 수 있었다.

- **Test3**



SLEEPING 상태인 process들은 우선순위가 높은 queue에 들어있더라도 scheduling에 영향을 주지 않는다. 따라서 작업이 끝나지 않은 process 12와 14가 존재하더라도 SLEEPING 상태라면 FCFS queue에 있는 process 13과 15가 실행되는 것을 확인할 수 있다.

Trouble shooting

FCFS scheduling의 non-preemptive한 성질을 잊어버렸다. 따라서 FCFS의 경우에도 tick이 증가할 때 yield가 발생하도록 하였고 context switching의 overhead가 줄어든다는 FCFS의 장점을 활용하지 못했다. 다시 이론 수업 내용을 복습하면서 이 부분을 바꾸었다.

Project2-2: Multilevel feedback queue scheduler

실행하는 법 (xv6-public 폴더 내부에서)

1. make clean
2. make SCHED_POLICY=MLFQ_SCHED MLFQ_K=숫자 CPUS=1 -j
3. make fs.img SCHED_POLICY=MULTILEVEL_SCHED MLFQ_K=숫자 CPUS=1 -j
4. ./bootxv6.sh 혹은 qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512

Design

먼저 process가 담기는 queue는 2개에서 5개로 유동적이다. queue의 level이 낮을 수록 우선순위가 높은 queue이기 때문에 RUNNABLE한 process가 담긴 queue들 중 가장 낮은 level의 queue에서 process가 선택되어야 한다. queue 내부에서도 우선순위에 따라서 선정되는 process가 달라진다. 각 queue 마다 time quantum이 정해져 있는데, 이 time quantum을 모두 사용하게 되면 한 단계 높은 level의 queue로 이동하게 된다. 내가 떠올린 design은 다음의 순서로 구성되어 있다.

1. **global ticks 처리**
2. **proc structure에 priority와 tick 변수 추가**

   해당하는 process가 어떤 queue에 있는지는 Multilevel scheduler에서 구현한 levelOfQueue를 사용하면 된다. 추가로 queue 내부에서 우선순위를 판단하기 위한 priority 변수와 얼마의 tick 만큼 실행되었는지 판단하기 위한 ticks 변수가 추가되어야 한다.

3. `allocproc(void)` **시 ticks와 priority, levelOfQueue 변수 초기화**
4. `setpriority(int, int)`, `getlev(void)` **system call 구현**

   과제 명세에서 요구하는 system call을 구현해야 한다. `setpriority()`는 해당하는 id를 가진 process의 priority를 인자로 들어온 priority로 바꾸어 주는 것이다. 이때 오로지 자식 process의 priority만 변경할 수 있다. `getlev()` 는 현재 process가 어떤 level의 queue에 있는지 반환하는 system call이다.

5. `priority_boosting(void)` **구현**

timer interrupt가 발생하면 global ticks가 1씩 증가하게 되는데, 이때 100 tick마다 모든 process의 levelOfQueue와 ticks 변수를 0으로 초기화 해 주어야 한다. 또한 실행할 수 있는 process가 존재하지 않는다면 priority boost를 시행할 것이다.

6. `sys_sleep(void)`, `sys_yield(void)` **변경**

   `sys_sleep()` 내부에서 return 하기 전과 `sys_yield()`에서 `yield()`를 호출하기 전에 현재 process의 levelOfQueue와 ticks를 0으로 초기화 해야 한다. process의 작업이 끝났다고 판단하기 때문이다.

7. **process→ticks 처리**

   timer interrupt가 발생하면 global ticks가 1씩 증가하게 되는데, 이때 process→ticks도 1씩 증가해야 할 것이다. 또한 process의 ticks가 time quantum을 초과하였을 때 levelOfQueue 를 1 증가 시키고, priority는 유지한 채 ticks를 0으로 초기화 해야 한다.

8. `scheduler(void)` **변경**

   ptable을 순회하며 RUNNABLE하고 levelOfQueue가 MLFQ_K 보다 작은 process들 중에 우선순위가 가장 높은 process를 선택해야 한다. 우선순위는 다음의 순서와 같다.

   a. levelOfQueue가 작을수록 우선순위 높음.

   b. p→ticks가 0보다 크면 (실행된 적이 있지만 time quantum이 아직 남아있으면) 우선순위가 높음.

   c. p→priority가 클수록 우선순위 높음.

   d. p→pid 가 작을수록 우선순위 높음.

   즉, a→b→c→d 순서로 우선순위가 높다. 즉, a가 같으면 b로 판단하고, b가 같으면 c로 판단한다. a, b, c가 모두 같으면 d로 판단하는데, 이는 먼저 만들어진 process를 선택하는 것이다. (과제 명세에서는 a, b, c에서 우선순위를 판별할 수 없을 때 어떤 process를 선택해도 상관없다고 하였다.)

Implement

Design 순서를 바탕으로 구현을 완료하였다.

1. **global ticks 처리**

   timer interrupt를 받아 ticks 가 증가할 때 100의 나머지가 0인 경우 priorityBoostingFlag를 1로 만들어 준다. 이때 uint의 최댓값이 4,294,967,295 이기 때문에 4,294,967,200 까지만 올려주도록 한다.

```
// trap.c 의 trap(struct trapframe* tf)
...
switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
#ifdef MLFQ_SCHED
      if(ticks % 100 == 0) {
        priorityBoostingFlag = 1;
        if(ticks == (uint)4294967200)
          ticks = 0;
      }
#endif
    ...
```

2. **proc structure에 priority와 tick 변수 추가**

```
// proc.h
struct proc {
  ...
  int levelOfQueue;
  uint ticks;
  int priority;
};
```

3. `allocproc(void)` **시 ticks와 priority, levelOfQueue 변수 초기화**

```
// proc.c
static struct proc* allocproc(void)
{
  ...
  #elif MLFQ_SCHED
    p->levelOfQueue = 0;
    p->priority = 0;
    p->ticks = 0;
  #endif
  ...
}
```

4. `setpriority(int, int)`, `getlev(void)` **system call 구현**

   먼저 `getlev(void)` 는 간단하게 myproc()→levelOfQueue를 return 하면 된다. `setpriority(int, int)`는 인자로 받은 priority가 0부터 10까지의 수인지 확인을 하고 ptable의 lock을 획득한 뒤에 인자로 받은 pid에 해당하는 process를 찾고, 해당하는 process의 부모 프로세스가 이 system call을 호출한 process인지 확인한다. 적절한 process를 찾았다면 해당 process의 priority를 인자로 받은 priority로 설정한 후 return 해준다.

`sys_setpriority(void)` 의 경우 `argint()` 를 통해 인자가 정상적으로 들어왔는지 확인해주어야 한다.

```
// proc.c
int getlev(void)
{
  if(myproc()) return myproc()->levelOfQueue;
  return -1;
}

int setpriority(int pid, int priority)
{
  struct proc *parent, *p;

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

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

```
// sysproc.c
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. `priority_boosting(void)` **구현**

   이 함수는 queue들 중 어디에도 실행 가능한 process가 존재하지 않을 때, 혹은 tick이 100만큼 늘어났을 때 호출되는 함수이다. 내부에서 ptable을 참조하기 때문에 호출하기

전에 ptable의 lock을 획득해야 한다.

```
// proc.c
void priority_boosting(void)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid > 0){
      p->levelOfQueue = 0;
      p->ticks = 0;
    }
  }
}
```

6. `sys_sleep(void)`, `sys_yield(void)` **변경**

   system call `sleep()` 과 `yield()`가 호출되었을 경우 작업이 끝났다고 판단하여 priority를 제외한 ticks와 levelOfQueue를 초기화 해 주어야 한다.

```
int sys_sleep(void)
{
  ...
  while(ticks - ticks0 < n){
    ...
    sleep(&ticks, &tickslock);
  }
  ...

  #ifdef MLFQ_SCHED
  acquire_ptable_lock();
  myproc()->levelOfQueue = 0;
  myproc()->ticks = 0;
  release_ptable_lock();
  #endif

  return 0;
}

void sys_yield(void)
{
#ifdef MLFQ_SCHED
  acquire_ptable_lock();
  myproc()->levelOfQueue = 0;
  myproc()->ticks = 0;
  release_ptable_lock();
#endif
  return yield();
}
```

7. **process→ticks 처리**

timer interrupt가 발생하였을 때, 현재 실행 중인 process의 ticks를 1 증가 시킨 후, 만약 해당하는 time quantum을 넘으면 levelOfQueue를 한 단계 올려주어야 한다. 이때 만약 위에서 priorityFlag가 set 되었다면 `priority_boosting()` 을 호출하기만 한다.

```
// trap.c 의 trap(struct trapframe* tf)
...
switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      ...
    }
#ifdef MLFQ_SCHED
    acquire_ptable_lock();
    if(priorityBoostingFlag){
      priority_boosting();
    }
    else{
      struct proc *p = myproc();
      if(p && (p->state == RUNNING) && (p->levelOfQueue < MLFQ_K)){
        p->ticks++;
        if((p->ticks) >= (4 * p->levelOfQueue + 2)){
          p->levelOfQueue++;
          p->ticks = 0;
        }
      }
    }
    release_ptable_lock();
#endif
...
```

8. `scheduler(void)` **변경**

우선 순위를 판단하며 `struct proc *point` 에 실행해야 할 process의 주소를 담아준다. 이때 만약 실행할 수 있는 process가 존재하지 않다면 `priority_boosting()` 을 호출한다.

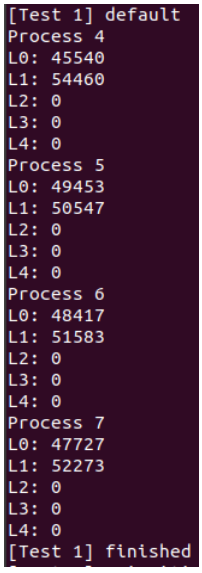
```
// proc.c 의 scheduler(void)
...
#elif MLFQ_ SCHED
    int level;
    int K = MLFQ_K;
    struct proc *point = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if((p->state != RUNNABLE) || ((level = p->levelOfQueue) >= K))
        continue;
      if(!point)
        point = p;
      else if(point->levelOfQueue != level){
        if(point->levelOfQueue > level)
          point = p;
      }
```

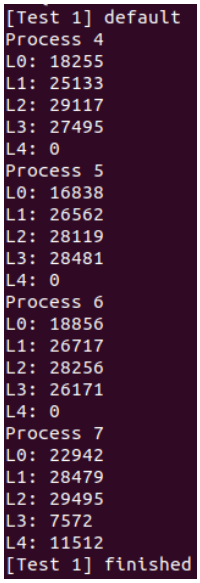
```
      else if((point->ticks > 0) != (p->ticks > 0)){
        if(p->ticks > 0)
          point = p;
      }
      else if(point->priority != p->priority){
        if(point->priority < p->priority)
          point = p;
      }
      else if(point->pid > p->pid){
        point = p;
      }
    }

    if(!point)
      priority_boosting();
    else{
      c->proc = point;
      switchuvm(point);
      point->state = RUNNING;
      swtch(&(c->scheduler), point->context);
      switchkvm();
      c->proc = 0;
    }
#else
...
```

Result

- Test1





MLFQ_K 가 2(좌측)와 5(우측)일 때 모두 잘 동작한다.

- Test2

```
[Test 2] priorities
Process 11
L0: 46808
L1: 53192
L2: 0
L3: 0
L4: 0
Process 10
L0: 48824
L1: 51176
L2: 0
L3: 0
L4: 0
Process 8
L0: 50538
L1: 49462
L2: 0
L3: 0
L4: 0
Process 9
L0: 48720
L1: 51280
L2: 0
L3: 0
L4: 0
[Test 2] finished
```

```
[Test 2] priorities
Process 11
L0: 14391
L1: 22896
L2: 28208
L3: 34505
L4: 0
Process 10
L0: 19375
L1: 21770
L2: 27179
L3: 31676
L4: 0
Process 9
L0: 17795
L1: 25386
L2: 35460
L3: 21359
L4: 0
Process 8
L0: 18587
L1: 26335
L2: 37944
L3: 7477
L4: 9657
[Test 2] finished
```

pid가 높은 process에게 높은 priority를 부여하기 때문에 큰 process가 더 빨리 끝나는 경향을 보인다.

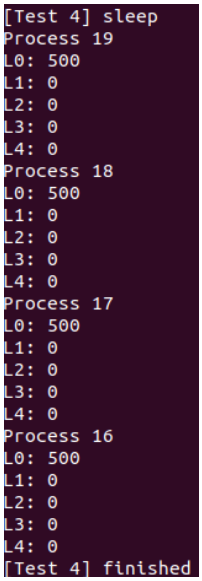
- Test3

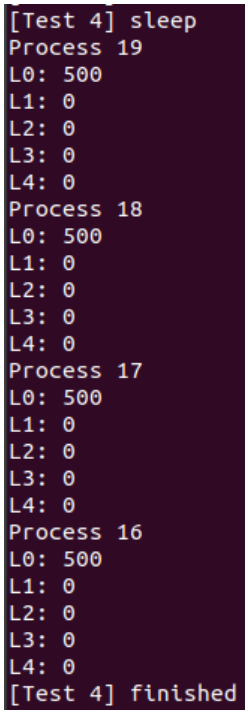
```
[Test 3] yield
Process 15
L0: 20000
L1: 0
L2: 0
L3: 0
L4: 0
Process 14
L0: 20000
L1: 0
L2: 0
L3: 0
L4: 0
Process 13
L0: 20000
L1: 0
L2: 0
L3: 0
L4: 0
Process 12
L0: 20000
L1: 0
L2: 0
L3: 0
L4: 0
[Test 3] finished
```

```
[Test 3] yield
Process 15
L0: 20000
L1: 0
L2: 0
L3: 0
L4: 0
Process 14
L0: 20000
L1: 0
L2: 0
L3: 0
L4: 0
Process 13
L0: 20000
L1: 0
L2: 0
L3: 0
L4: 0
Process 12
L0: 20000
L1: 0
L2: 0
L3: 0
L4: 0
[Test 3] finished
```

MLFQ_K가 2, 5일 때 모두 결과가 잘 나온다. `yield()` system call을 계속해서 호출하기 때문에 모든 process들이 L0에 머물러 있고, 높은 pid를 가진 process가 우선순위가 더 높으므로 pid가 클수록 작업이 빨리 끝난다.

- Test4





Test3과 비슷하게 이번엔 `sleep()` system call을 계속해서 호출한다. sleep 시스템 콜을 호출하면 해당 process가 자는 동안 다른 process도 일을 할 수 있기 때문에 거의 동시에 끝나는 것을 확인할 수 있었다.

- Test5

```
[Test 5] max level
Process 20
L0: 99975
L1: 25
L2: 0
L3: 0
L4: 0
Process 21
L0: 50403
L1: 49597
L2: 0
L3: 0
L4: 0
Process 22
L0: 50042
L1: 49958
L2: 0
L3: 0
L4: 0
Process 23
L0: 44930
L1: 55070
L2: 0
L3: 0
L4: 0
[Test 5] finished
```

```
[Test 5] max level
Process 20
L0: 99974
L1: 26
L2: 0
L3: 0
L4: 0
Process 21
L0: 46006
L1: 53984
L2: 10
L3: 0
L4: 0
Process 22
L0: 30798
L1: 35342
L2: 33856
L3: 4
L4: 0
Process 23
L0: 24553
L1: 28357
L2: 27556
L3: 19532
L4: 2
[Test 5] finished
```

pid가 작은 process부터 최대 queue level 제한을 두어 해당하는 level에 도달하면 `yield()`를 호출하게 된다. MLFQ_K가 5인 오른쪽의 경우 process 20은 L1, process 21은 L2, process 22는 L3, process 23은 L4에 도달할 때 `yield()` system call을 호출하는 것을 알 수 있다. MLFQ_K가 2인 왼쪽의 경우 process 20만 최대 제한 level에 도달할 수 있게 된다.

- Test6

```
[Test 6] setpriority return value
done
[Test 6] finished
```

`setpriority()` system call을 정상적으로 구현하였다.

Trouble shooting

ptable에 접근할 때 ptable lock을 획득하고 접근하려다 두 번 이상 lock을 잡으려고 시도하여 error를 경험했다.