사용자:KimMinKwan/연습장

목차

Project 02

Design

1. MultiLevel Queue

- 스케줄러는 2개의 큐가 존재하며, 하나는 RR(Round Robin) 스케줄링, 다른 하나는 FCFS(First Come First Served) 스케줄링을 진행합니다.
- RR 큐는 FCFS 큐보다 우선순위이며, RR 큐는 pid 가 짝수인 프로세스들을 스케줄링하며, RR 큐에 있는 프로세스 중 RUNNABLE 상태인 프로세스가 없다면, FCFS 큐로 넘어갑니다.
- FCFS 큐는 pid가 홀수 인 프로세스들을 스케줄링하며, 그 중에서도 pid가 더 작은 것부터 진행합니다.

2. MLFQ

- 스케줄러는 make시 결정되는 k(2 ~ 5)에 따라 L0 부터 Lk-1 개의 큐가 존재합니다.
- 큐의 번호가 작을 수록 우선순위가 높은 것이며, i 번째 큐는 2i+4 ticks의 time quantum을 가집니다.
- 처음 프로세스가 실행되면 가장 높은 레벨인 L0 큐로 들어가며, 같은 큐내에서도 프로세스들의 priority에 따라 높은 priority(0 ~ 10) 일 수록 먼저 스케줄링 합니다.
- priority 값은 setpriority 시스템 콜을 호출함으로서, 결정할 수 있습니다. setpriority 시스템 콜은 자신으로부터 직접 생성된 자식 프로세스에게만 사용 가능 합니다.
- 타이머 인터럽트나 yield , sleep 등을 통해 스케줄러에게 실행 흐름이 넘어올 때마다, 스케줄러는 RUNNABLER한 프로세스가 존재하는 큐 중 가장 레벨이 높은 큐를 선택해야 합니다.
- Li 큐에서 실행된 프로세스의 time quantum을 모두 사용하면 Li+1 큐로 넘어가고 실행 시간을 초기화합니다. 그 후 마지막 큐에서 실행된 프로세스가 time quantum을 모두 다 사용했다면, priority boosting 이 일어나기 전까지 스케줄링을 하지 않습니다.
- priority boosting 이란, 모든 프로세스들을 L0로 올리며 프로세스들의 우선순위는 바뀌지 않습니다. 그리고 이 함수는 100ticks 가 지날때마다 실행됩니다.

Implement

1. MultiLevel Queue

MultiLevel Queue 를 구현하기 위해서는 우선 프로세스의 pid 가 짝수인지 홀수인지를 우선적으로 확인해야합니다.

그래서 nCheckEvenPid 라는 함수를 구현하여, pid가 RUNNABLE한 상태이면서 짝수라면 1을, 그렇지 않다면 0을 return 합니다.

```
// table에 짝수의 pid 가 있을 경우 return 1
// 아니면 return 0
int
nCheckEvenPid(void)
{
  struct proc *p;
  int n_evenCount = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    // pid=2인 프로세스는 쉘이 돌아가기 때문에 제외해준다
    if((p->pid) % 2 == 0 && p->state == RUNNABLE && (p->pid) != 2) n_evenCount++;
  }

  if(n_evenCount == 0) return 0;
  else return 1;
}
```

스케줄링 과정에서 nCheckEvenPid 함수를 사용하여 return 값이 0 이라면, pid 중에서 가장 작은 pid 를 반환하는 FCFS_Least_Odd 구조체 함수를 구현했습니다. 이 구조체 함수는 홀수인 pid 중에서 가장 작은 pid의 프로세스를 반환합니다.

```
struct proc*
FCFS_Least_Odd(void)
{
  struct proc *p;
  struct proc *FCFS_least_proc = ptable.proc;
  int n_min = NPROC;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if((p->pid > 2) && ((p->pid)%2 == 1) && (p->state == RUNNABLE))
    {
      if(p->pid < n_min)
      {
        n_min = p->pid;
        FCFS_least_proc = p;
      }
    }
  }
  return FCFS_least_proc;
}
```

따라서 scheduler 함수에서 nCheckEvenPid을 통해, return 값이 1이라면 RR 스케줄링을, 0이라면 FCFS_Least_Odd 를 실행하여 FCFS 방식으로 스케줄링을 진행합니다.

```
#ifdef MULTILEVEL_SCHED
    struct proc *p;
    struct cpu *c = mycpu();
    struct proc *multilevel_proc = ptable.proc;
    c->proc = 0;
    int n_checkRR = 0;
    for(;;)
    {
      // Enable interrupts on this processor.
      sti();

      // Loop over process table looking for process to run.
      acquire(&ptable.lock);
      n_checkRR = nCheckRR();
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
      {
        multilevel_proc = p;
        // 프로세스의 상태가 Runnable인지 확인
        if(p-> state != RUNNABLE) continue;

        // 처음 보모와 자식 pid
        if(p->pid == 1 || p->pid ==2) multilevel_proc = p;

        //ptable 에 짝수 pid가 존재할경우 RR(Round Robin)
        else if(n_checkRR != 0)
        {
          // 홀수일 경우 넘어가기
          if((p->pid) %2 != 0)  continue;
          multilevel_proc = p;
        }
        // ptable에 홀수 pid가 있을 경우 작은 pid 를 선택 FCFS
        else multilevel_proc = FCFS_Least_Odd();

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

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

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

2. MLFQ

proc.h 와 proc.c 에 추가한 내용

- MLFQ 스케줄링을 하기위해 proc 구조체 안에 priority, ticks, queue_level, isLast_queue, ppid 변수를 추가합니다.

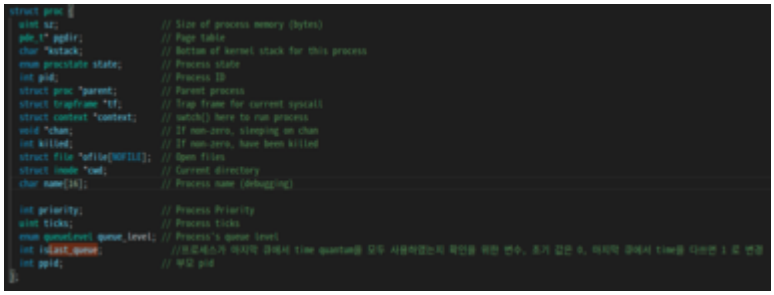
int priority : 프로세스의 priority를 담는 변수

uint ticks : 프로세스의 진행 시간

queue_level : 프로세스가 있는 큐의 level (enum 으로 선언 되어있음: enum queueLevel { L0, L1, L2, L3, L4})

int isLast_queue : 프로세스가 마지막 큐인지 아닌지를 확인해주는 변수

int ppid : 부모 프로세스의 pid



- allocproc 함수에서 처음 프로세스가 시작할 때 설정 값을 추가해줍니다.

프로세스의 queue_level = 0, priority = 0, isLast_queue = 0, ppid = 1, ticks 에는 커널에서 돌고있는 ticks 를 넣어줍니다.

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

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

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  // for first start
  p->queue_level = 0;
  p->priority = 0;
  p->isLast_queue = 0;
  p->ppid = 1;
  acquire(&tickslock);
  p->ticks = ticks;
  release(&tickslock);

  release(&ptable.lock);
```

- priority를 설정해줄 시스템콜 함수인 setpriority 함수를 만들어 줍니다.

setpriority 함수는 pid 와 priority를 인자로 받아 priority 가 0보다 작거나 10보다 크면 -2를 반환, pid 가 존재하지 않거나 자식 프로세스가 없다면 -1을 둘다 아니라면, priority를 설정한 후 0을 반환 합니다.

```
// get pid and priority and if child set prirority 0
// else not exist no child return -1
// else pri < 0 pri >10 return -2
int
setpriority(int pid, int priority)
{
  struct proc *p;
  struct proc *current_p = myproc();
  int n_returnSetValue = -1;

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

- 프로세스의 level 을 가져올 수 있는 getlev 시스템 콜함수를 만들어줍니다.

getlev 함수는 프로세스가 있는 큐레벨을 반환합니다.

```
//get level
int
getlev(void)
{
  struct proc *p = myproc();
  acquire(&ptable.lock);
  int queue_lev = p->queue_level;
  release(&ptable.lock);
  return queue_lev;
}
```

스케줄링이 시작될 때, 현재 있는 queue 중에서 level 이 가장 높은 (L0 ~ Lk-1) 인 큐를 확인을 먼저 합니다.

- 그를 위해서 nReturnLevel 함수를 구현합니다. 이 함수는 level이 존재하는지 확인하면서 만약 존재한다면 그중 가장 높은 레벨을 반환하고 그렇지 않다면 -1을 반환합니다.

```
int
nReturnHighLevel()
{
  struct proc *p;
  int n_check = 0;
  int n_highLevel = 8;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->pid != 1 && p->pid != 2)
    {
      // 숫자가 낮을 수록 높은 레벨
      if(p->queue_level < n_highLevel && p->state == RUNNABLE && p->isLast_queue== 0)
      {
        n_highLevel = p->queue_level;
        n_check++;
      }
    }
    else continue;
  }
  if(n_check == 0) return -1;
  else return n_highLevel;
}
```

- 만약 nReturnLevel 의 반환 값이 -1 이라면 priority boosting 을 진행하는 PriorityBoosting 함수를 사용하여 초기화합니다.

PriorityBoosting 함수는 모든 프로세스의 레벨을 L0로, ticks 는 0으로, 상태는 RUNNABLE 한 상태로 초기화해줍니다.

```
void
PriorityBoosting(void)
{
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->queue_level != L0 && p->pid > 0)
    {
      if(p->state == RUNNING || p->state == RUNNABLE)
      {
        p->state = RUNNABLE;
        p->queue_level = L0;
        p->ticks = 0;
        p->isLast_queue = 0;
      }
    }
  }
}
```

- -1이 아니라 level 이 반환 됬다면, 해당 레벨의 큐중에서 가장 높은 priority의 프로세스를 ReturnHighPriorityP 함수를 사용하여 프로세스를 반환합니다.

ReturnHighPriority는 인자로 받은 큐레벨의 프로세스중 priority(0 ~ 10) 중에서 가장 높은 priority를 가진 프로세스를 반환합니다.

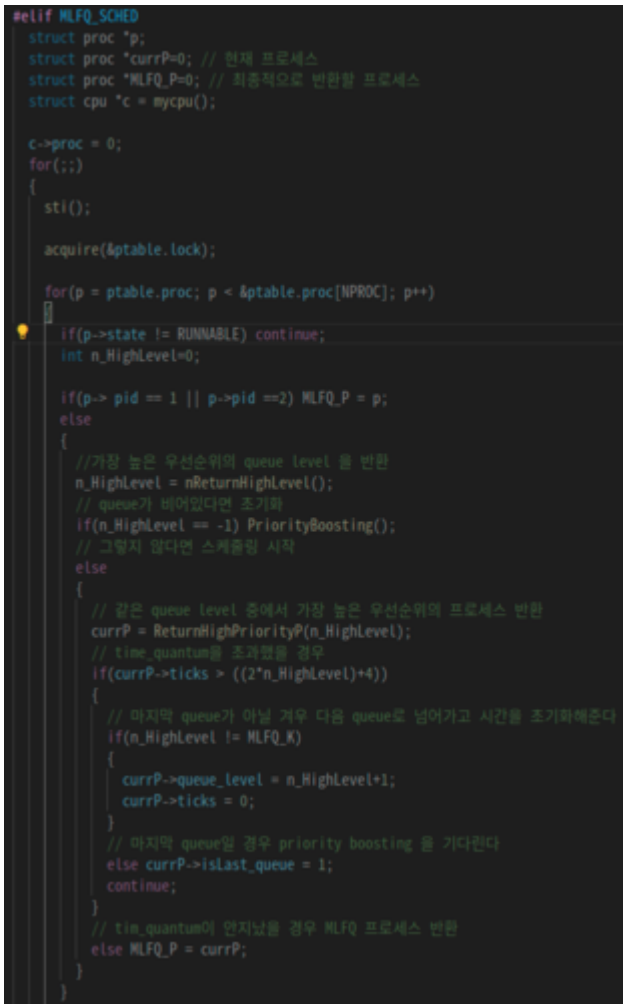
```
struct proc*
ReturnHighPriorityP(int n_queueLev)
{
  struct proc *p;
  struct proc *highPriorityP=0;
  int n_maxPriority = 0;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->pid != 1 && p->pid !=2)
    {
      if((p->state) == RUNNABLE && (p->queue_level) == n_queueLev && p->islast_queue != 1)
      {
        if(p->priority >= n_maxPriority)
        {
          n_maxPriority = p->priority;
          highPriorityP = p;
        }
      }
    }
    else continue;
  }
  return highPriorityP;
}
```

- 프로세스를 반환 받은 뒤, 그 프로세스의 ticks 가 time quantum((2*해당 level)+4) 을 초과했는지 안했는지를 판단 합니다.

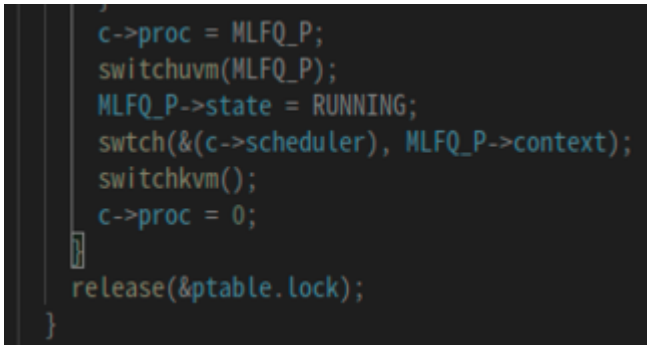
만약 time quantum을 초과 했을 경우, 마지막 큐가 아니라면, queue 의 레벨에 1을 더해 레벨을 올려주고, ticks 는 초기화해 줍니다.

마지막 큐라면, proc 구조체에 마지막 큐라는 last_queue를 1 로 바꾸어줘 priority boosting 을 기다립니다.

time quantum을 초과하지 않았다면, 그 프로세스를 스케줄링합니다.



```
#elif MLFQ_SCHED
  struct proc *p;
  struct proc *currP=0; // 현재 프로세스
  struct proc *MLFQ_P=0; // 최종적으로 반환할 프로세스
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;)
  {
    sti();

    acquire(&ptable.lock);

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if(p->state != RUNNABLE) continue;
      int n_HighLevel=0;

      if(p-> pid == 1 || p->pid ==2) MLFQ_P = p;
      else
      {
        //가장 높은 우선순위의 queue level 을 반환
        n_HighLevel = nReturnHighLevel();
        // queue가 비어있다면 초기화
        if(n_HighLevel == -1) PriorityBoosting();
        // 그렇지 않다면 스케줄링 시작
        else
        {
          // 같은 queue level 중에서 가장 높은 우선순위의 프로세스 반환
          currP = ReturnHighPriorityP(n_HighLevel);
          // time_quantum을 초과했을 경우
          if(currP->ticks > ((2*n_HighLevel)+4))
          {
            // 마지막 queue가 아닐 겨우 다음 queue로 넘어가고 시간을 초기화해준다
            if(n_HighLevel != MLFQ_K)
            {
              currP->queue_level = n_HighLevel+1;
              currP->ticks = 0;
            }
            // 마지막 queue일 경우 priority boosting 을 기다린다
            else currP->islast_queue = 1;
            continue;
          }
          // tim_quantum이 안지났을 경우 MLFQ 프로세스 반환
          else MLFQ_P = currP;
        }
      }
```



```
      c->proc = MLFQ_P;
      switchuvm(MLFQ_P);
      MLFQ_P->state = RUNNING;
      swtch(&(c->scheduler), MLFQ_P->context);
      switchkvm();
      c->proc = 0;
    }
    release(&ptable.lock);
  }
```

trap.c 에 추가한 내용

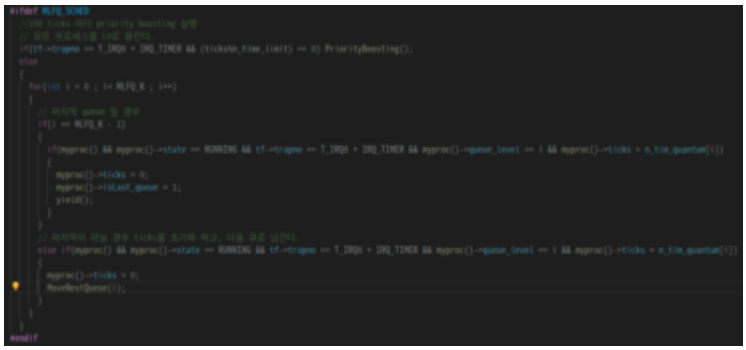
trap 에서는 time interrupt가 발생했을 때와 현재의 프로세스의 ticks를 하나씩 증가시켜주는 부분을 구현해주었습니다.

- void trap 함수에서 case T_IRQ0 + IRQ_TIMER 의 경우에 프로세스의 ticks 를 하나씩 증가 시켜줍니다.

```
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
  // 현재의 프로세스의 ticks를 하나씩 증가시켜준다.
  if(myproc()) myproc()->ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }
  lapiceoi();
  break;
```

- time interrupt가 발생하고, 100ticks 마다 모든 프로세스를 L0로 초기화해주는 priority boosting을 실행하고 그 외의 경우,

마지막 큐에서 진행됐을 경우, ticks 를 초기화해주며 위에서 선언한 구조체 변수중 isLast_queue 를 1로 바꿔줍니다.



마지막 큐가 아닐경우, ticks를 초기화해주고 다음 큐로 이동하는 MoveNextQueue 함수를 실행해줍니다.

MoveNextQueue 함수는 큐의 레벨을 1더해 줍니다.

```
void
MoveNextQueue(int n_queuelevel)
{
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  myproc()->queue_level = n_queuelevel+1;
  sched();
  release(&ptable.lock);
}
```

- MLFQ 스케줄링을 진행할 때는, time interrupt 가 발생할때, yield 하는 부분을 삭제해 줍니다.

```
#ifdef MLFQ_SCHED
  if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
      exit();
```

syscall.c 함수에 추가한 내용

yield(24) 나 sleep(13) 의 시스템콜 함수가 호출되었을 때, 큐 level 은 L0로 ticks는 0 으로 초기화해주는 부분을 추가해줍니다.

```
#ifdef MLFQ_SCHED
  //yeild 나 sleep 이 일어난 경우, L0 로 다시 초기화
  if(num == 13 || num ==24)
  {
    myproc()->queue_level = L0;
    myproc()->ticks = 0;
    myproc()->isLast_queue = 0;
  }
#endif
```

3. Makefile 수정 사항

make 시, 어느 스케줄로 스케줄링을 할지, 그리고 MLFQ 스케줄링을 적용한다면, 큐의 개수와 CPU 수를 인자로 받아야합니다.

따라서 Makefile 에 아래와 같이 내용을 추가해줍니다.

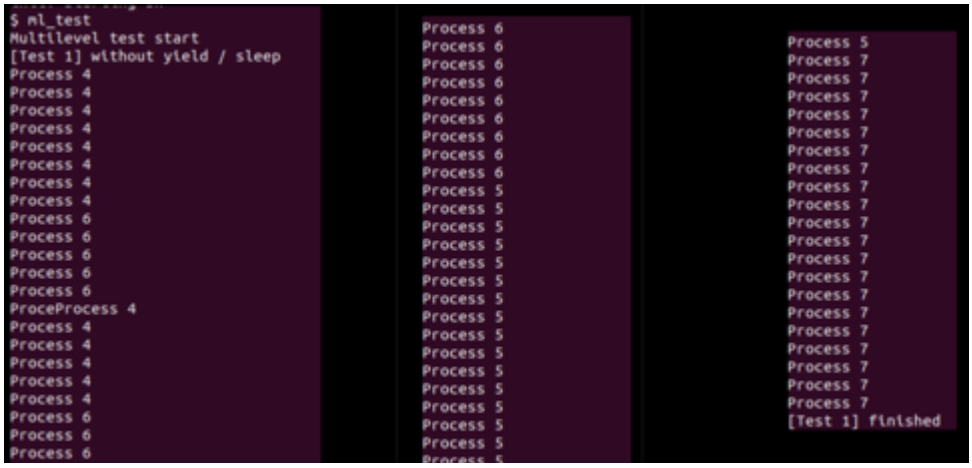
```
# Mecro for scheduling
SCHED_POLICY = ORIGIN
MLFQ_K = 0

# For Scheduling
CFLAGS += -g -Wall -D $(SCHED_POLICY) -D MLFQ_K=$(MLFQ_K)
```

Result

1. MultiLevel Queue

Multilevel 의 결과 값으로 주어진 test user code인 ml_test 를 실행할 경우,

- [test1] 은 짝수 pid 인 프로세스가 우선적으로 실행된 뒤, 홀수의 pid 중 작은 것부터 우선하여 실행됩니다



- [test2] 는 짝수 pid가 작은 순으로 먼저 종료된 뒤, 홀수 pid 인 프로세스 또한 작은 순으로 우선하여 종료 됩니다.

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

- [test3] 는 sleeing 상태인 짝수 pid 가 스케줄링된 후, 홀수 pid 로 이동하고, 홀수 pid 또한 sleeping 상태이므로 다시 짝수 pid가 출력됩니다.

(sleeping 상태는 스케줄링 될 수 없음)

```
[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 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
```

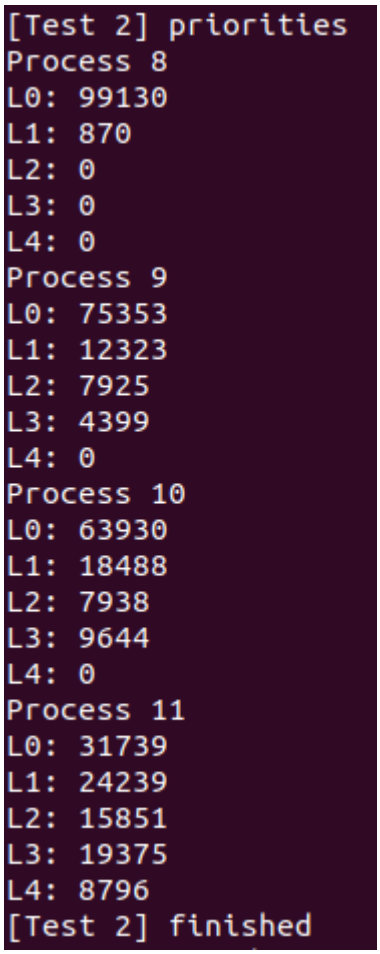
2. MLFQ

make 진행 시, k =5, CPU = 1 로 설정하여 test 진행

- [test1] k =5 이므로 출력 순서에 상관없이 L0. L1. L2. L3. L4 모든 큐에 값이 들어가서 출력 됩니다.

```
$ mlfq_test
MLFQ test start
[Test 1] default
Process 7
L0: 96518
L1: 3482
L2: 0
L3: 0
L4: 0
Process 4
L0: 79021
L1: 6108
L2: 7871
L3: 7000
L4: 0
Process 5
L0: 74703
L1: 7830
L2: 7846
L3: 9621
L4: 0
Process 6
L0: 55899
L1: 18064
L2: 15557
L3: 10480
L4: 0
[Test 1] finished
```

- [test2] time을 사용량은 비슷하므로 끝나는 시간이 비슷합니다.



- [test3] yield가 호출될 때마다, level 을 L0 로 초기화하기 때문에 L0만 값이 출력되고, pid가 클수록 priority 가 크기때문에 큰 순선대로 출력됩니다.

```
[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
```

- [test4] [test3] 과 동일하게 sleep 을하면 L0 로 초기화되기 때문에 pid가 큰 순서대로 출력됩니다.

```
[Test 4] sleep
Process 19
L0: 500
L1: 0
L2: 0
L3: 0
L4: 0
Process 18
L0: 500
L1: 0
L2: 0
L3: 0
L4: 0
Process 17
L0: 500
L1: 0
L2: 0
L3: 0
L4: 0
Process 16
L0: 500
L1: 0
L2: 0
L3: 0
```

```
L4: 0
[Test 4] finished
```

- [test5] pid 가 클수록 낮은 레벨에 스케줄링이 진행되는 경향으로 출력됩니다.

```
[Test 5] max level
Process 20
L0: 99978
L1: 22
L2: 0
L3: 0
L4: 0
Process 21
L0: 44622
L1: 55371
L2: 7
L3: 0
L4: 0
Process 22
L0: 30772
L1: 37555
L2: 31671
L3: 2
L4: 0
Process 23
L0: 26664
L1: 36900
L2: 27509
L3: 8926
L4: 1
[Test 5] finished
```

- [test6] setpriority 가 올바르게 호출되면 0을 반환하기 때문에 wrong error 메세지가 아닌 done 으로 출력됩니다.

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

Trouble Shooting

- 스케줄링을 진행할 때, 처음 생성되는 pid 1과 2는 쉘이 구동하기위한 프로세스므로 스케줄링을 진행할 때는 1, 2 pid 프로세스를 포함시킬 경우, 에러가 발생합니다. 따라서 스케줄링을 할때는 위의 두개를 항상 예외처리를 해주어야합니다.
- 스케줄링 부분인 proc.c 를 수정하는 부분외에도 처음 프로세스가 만들어질 때, 구조체 안에 정의한 변수를 초기화 해주는 부분인 allocproc 부분, 프로세스의 ticks를 계속해서 증가시켜주는 부분은 trap 부분, 그리고 100ticks 마다 초기화 해줘야하는 부분에서 문제가 많았습니다. 처음에는 어느 함수가 어떤 기능을 하는지, 그리고 어디를 수정해야하는지 몰라 문제가 많았지만 하나씩 print 해보고 debugging 을 하면서 수정해 나갔습니다.
- yield 나 sleep 이 발생했을 때, 큐 level과 ticks를 초기화해주는 부분을 처음에는 yield 함수와 sleep 함수 안에서 구현했지만, 이를 더 간단하게 syscall 함수에서 24(yield), 13(sleep) 일 경우에는 예외를 처리하는 방법을 확인했습니다.

- make에 인자를 추가하여 작업하는 부분에서 처음에는 SCHED_POLICY 만 추가하면 되는 줄알았지만, 계속해서 원하는 방향으로 스케줄링이 되지 않아 찾아본 결과 CFLAGS += -g -Wall -D $(SCHED_POLICY) -D MLFQ_K=$(MLFQ_K) 명령어를 추가하여 make 할 시 인자를 추가하여 적용할 수 있었습니다.
