OS_project02_13065_주원웅_2019060546

Design

1. Mlfq - L0 L1 L2 (Round Robin) L3 (priority) 구현

Mlfq를 구현하기 위해서는 우선 4단계의 queue가 필요하다고 생각했습니다. queue를 직접 구현하진 않지만 queue를 나타낼 수 있는 변수 qlevel을 통해서 해당 프로세스가 속한 queue를 표현할 것입니다. 또한 해당 queue로 들어온 순서를 표현하는 order, L3 queue 에서 프로세스처리 우선순위를 담는 priority, 각 queue에서 소모한 시간을 담는 time_q 변수를 proc 구조체에 추가할 것 입니다.

2. 스케줄링 순서 구현

Mlfq 속에서 가장 우선순위가 높은 프로세스는 큐 레벨이 가장 낮은 프로세스 입니다. 그렇기에 현재 RUNNABLE한 모든 프로세스들의 **Qlevel**을 비교할 것입니다. **Qlevel**이 같은 경우는 큐에 들어온 순서를 비교하여 먼저 들어온 순서로 처리할 것이며 **L3** 큐에 속한 프로세스의 경우는 큰 **priority** 값을 갖는 프로세스를 우선적으로 처리할 것 입니다. 각 프로세스는 **round robin** 방식으로 **time interrupt**가 발생하여 **time quantum**을 1 tick 소모시마다 time_q 변수 값을 하나씩 올리며 해당 queue에서 경과한 시간을 기록할 것입니다. 동시에 round robin 이기에 순서를 해당 큐의 맨 뒤로 이동시켜 구현할 것 입니다. 또한 해당 큐에서 할당된 시간만큼의 time_q값을 가질 경우(time quantum 모두 소모) time_q를 0으로 초기화 하고 qlevel 값을 변경하여 프로세스가 속한 큐가 변경되었음을 알릴 것 입니다.

3. priorityBoosting

프로세스가 스케줄링 되지 못하는 것을 막기 위해**(Starvation) priorityBoosting**을 구현합니다. **priorityBoosting**은 tick이 100번이 발생할 때마다 발생하도록 해야 합니다. 그러기 위해서 tick 값 대신 글로벌 변수인 **g_ticks**를 선언하여 사용할 것 입니다. 또한 priorityBoosting이란 모든 프로세스들(moq 제외)의 위치를 L0 큐로 재조정 하는 것 이기 때문에 boosting 되는 순서도 중요합니다. 가장 높은 우선순위를 가지는 프로세스부터 boosting 되도록 할 것이며 이는 위의 스케줄링 방식과 비슷하게 구현할 것 입니다. 추가적으로 이 과정에서 프로세스의 order를 부여하는 q_order의 값 역시 모두 0으로 초기화 하여 프로세스에 부여할 것이기 때문에 q_order의 값이 오버플로우가 발생하는 일이 없어집니다. Flag 값을 통해서 모든 프로세스를 Boosting하는 과정을 진행 할 것 입니다. 또한 Proc 구조체 속 변수 boosting을 이용하여 해당 프로세스가 부스팅 되었음을 표시할 것입니다. 모든 프로세스를 비교하며 RUNNABLE하면서 부스팅 되지 않은 프로세스 중 우선순위가 높은 것 부터 boosting 값을 1로 만들 것입니다. 이 과정을 계속 반복하면 결

국 모든 프로세스가 부스팅 될 것 이며 더 이상 부스팅이 필요한 프로세스가 없게 될 것 입니다. 이 경우 플래그 값을 변경하여 더 이상 부스팅이 필요하지 않음을 나타내고 루프를 빠져 나올 것 입니다. 이후 부스팅 된 프로세스의 부스팅 값을 0으로 만들어 부스팅 과정을 마무리할 것 입니다.

4. Moq(FCFS) 구현

Moq를 구현하기 위해 우선 moq가 활성 상태임을 나타내는 전역변수 moqact, moq속 RUNNABLE한 프로세스의 수를 나타내는 전역변수 moqsize를 사용할 것 입니다. 프로세스가 moq로 이동하는 경우는 setmonopoly함수 호출을 통해 proc구조체 변수 moq를 1로 세팅할 것입니다. 만약 moq 활성이 필요한 경우 moqact 값을 monopolize함수 호출을 통해 1로 세팅할 것 입니다. Moq가 활성 상태인 경우 스케줄러 함수에서 moq 값이 1인 프로세스를 우선으로 스케줄링 할 것 입니다. 또한 moq 속 우선순위는 q_order 배열의 마지막 배열요소를 통해서 나타낼 것 입니다. moq속 모든 프로세스가 종료 될 시 moq활성 상태를 끄기 위해서 moqact 값을 0으로 세팅하여 mlfq를 처리하도록 것 입니다. 이 과정은 moqsize 값이 0인 경우에 unmonopolize 함수를 호출하여 실행되도록 할 것 입니다. 또한 막아두었던 priority boosting을 활성하기 위해 g_ticks 값을 0으로 초기화 할 것입니다. Moq는 FCFS 스케줄링방식 이기에 ticks가 발생하더라도 order를 맨 뒤로 보내지 않을 것 입니다. Moqact 값이 1인 경우 g_ticks 값이 100이되더라도 priority boosting이 일어나지 않도록 할 것입니다.

Implement

```
struct proc {
  uint sz;                     // Size of p
  pde_t* pgdir;                // Page tabl
  char *kstack;                // Bottom of
  enum procstate state;        // Process s
  int pid;                     // Process I
  struct proc *parent;         // Parent pr
  struct trapframe *tf;        // Trap fram
  struct context *context;     // swtch() h
  void *chan;                  // If non-ze
  int killed;                  // If non-ze
  struct file *ofile[NOFILE];  // Open file
  struct inode *cwd;           // Current d
  char name[16];               // Process n
  int qlevel;                   // queue le
  int time_q;                   // time qua
  int priority;                 // process
  uint order;                   // process
  int moq;                      // true fal
  int boosting;                 // flag Boo
};

uint ticks;
uint g_ticks = 0;

int q_order[5] = {0,0,0,0,0};    // variable allocate process queue order
int moqact = 0;                  // variable that moq active
int moqsize = 0;                 // moqsize
```

위에서 설명한 것과 같이 필요한 변수들을 Proc.h의 구조체에 추가하였고, proc.c와 trap.c 에 필요한 전역변수를 만들었습니다.

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->qlevel = 0;
  p->time_q = 0;
  p->priority = 0;
  p->order = q_order[0]++;
  p->moq = 0;
  p->boosting = 0;
```

위는 프로세스가 할당되는 alloproc() 함수중 프로세스의 값들을 초기화하는 부분입니다. 가장 처음엔 L0큐에 속하기에 qlevel을 0으로 설정하고 L3에서만 사용하는 priority 를 0으로, 시간경과를 나타내는 변수인 time_q 역시 0, moq에 속하지 않으며 boosting을 해야할 일 도 없으니 모두 0으로 초기화 합니다. 하지만 order는 이미 L0 큐에 속한 프로세스들이 먼저 자리하고 있을 수 있으니 순서를 지켜야 하기에 L0 큐의 가장 뒤로 넣어주기 위해서 현재 L0큐의 순서 중 마지막을 표현하는 q_order[0]의 값을 할당합니다.

```
// function that compare process execution priority with runnable process
int
comPriority(struct proc* p1, struct proc* p2)
{
    if (p1->qlevel < p2->qlevel)
        return 1;
    else if (p1->qlevel == p2->qlevel) {
        if (p1->qlevel == 3) {
            if (p1->priority > p2->priority)
                return 1;
            if (p1->priority < p2->priority)
                return 0;
        }
        if (p1->order <= p2->order)
            return 1;
    } return 0;
}
```

위는 프로세스사이의 우선순위를 비교할 함수입니다. 매개변수 중 앞의 매개변수인 p1의 우선순위가 더 높을 경우 1을 반환하도록 구현하였습니다.

이 과정은 다음과 같습니다.

1. Qlevel 비교
    A. Qlevel이 더 낮은 경우 우선순위가 더 높다.
2. Qlevel이 동일하면서 3인 경우
    A. Priority값이 더 큰 경우 우선순위가 더 높다.
    B. Priority값이 동일한 경우 order를 통해 우선순위를 정한다.
3. Qlevel이 동일하면서 3이 아닌 경우 or Qlevel이 3이면서 priority가 동일한 경우
    A. Order 값이 더 작은 경우 우선순위가 더 높다.

이 과정을 다 거쳤는데 반환값을 반환하지 않았다면 q1의 우선순위는 q2 보다 낮은 것이므로 0을 반환하게 구현했습니다.

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

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

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    struct proc* np = 0;
    if (moqact == 1 && moqsize == 0)
        unmonopolize();
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE) continue;
      if(moqact == 1 && p->moq == 0) continue;      // first moq execution
      if(!np || comPriority(p, np)) np = p;
    }
    // Switch to chosen process.  It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    if (np)
    {
      c->proc = np;
      switchuvm(np);
      np->state = RUNNING;

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

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

    release(&ptable.lock);
```

다음은 전체 스케줄링 함수 부분을 보겠습니다. 가장 먼저 moqact 활성상태를 체크 후 활성 되어 있지 않은 경우 모든 프로세스의 우선순위를 comPriority() 함수를 통해 확인하여 가장 높은 우선순위를 가진 프로세스를 np에 할당하여 사용합니다.

만약 moqact 가 활성 상태인 경우 모든 프로세스의 moq 값을 확인하여 해당 프로세스가 moq에 해당하는 지 확인후 해당하지 않은 프로세스는 스케줄링에서 제외합니다. Moq에 속한 프로세스는 위와 동일하게 우선순위를 비교하여 moq 속 프로세스 중 가장 우선순위가 높은 프로세스를 선택합니다.

Moqact가 활성중인데 모든 moq속 프로세스가 종료된 경우(moqsize == 0)는 moqact를 0으로 만드는 unmonopolize() 함수를 호출하여 moqact활성을 종료하도록 합니다.

스케줄링 할 프로세스를 선택 한 후 np가 존재하는 경우(스케줄링 해야하는 프로세스가 존재)는 스케줄링을 실시하여 해당 프로세스가 실행되도록 합니다.

```
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  g_ticks++;
  myproc()->time_q++;
  if (myproc()->time_q == ((myproc()->qlevel) * 2) + 2)
  {
    if (myproc()->qlevel == 0)
    {
      if (myproc()->pid % 2 == 1)
        myproc()->qlevel = 1;
      else
        myproc()->qlevel = 2;
    } else if (myproc()->qlevel == 3)
    {
      if (myproc()->priority)
        setpriority(myproc()->pid, myproc()->priority - 1);
    }
    else
      myproc()->qlevel = 3;
    myproc()->time_q = 0;
  }
  if (g_ticks == 100 && moqact != 1)
  {
    priorityBoosting();
    g_ticks = 0;
    if (myproc()->moq != 1){
        myproc()->qlevel = 0;
        myproc()->order = q_order[0]++;
        myproc()->priority = 0;
        myproc()->time_q = 0;
        myproc()->boosting = 0;
    }
  }
  if (myproc()->moq != 1)
      myproc()->order = q_order[myproc()->qlevel]++;
  myproc()->state = RUNNABLE;
  sched();
```

위는 tick이 발생하여 스케줄링 할 프로세스를 교체하기는 yield 함수 내부입니다.

우선 yield 함수는 tick 발생 시 호출되므로 tick값 대신 사용하는 g_ticks값을 하나 늘려줍니다. 이후 현재 실행중인 프로세스에 할당된 time quantum((프로세스가 속한 큐 레벨 * 2) + 2) 역시 하나 소모한 게 되므로 time_q의 값을 하나 늘려줍니다.

다음으로 time_q을 모두 소모하였는지 확인하고 만약 모두 소모했다면 다음 queue로 이동시키는 작업을 수행합니다.
L0 큐에 속한 프로세스라면 Pid 값을 확인한 뒤 홀수라면 L1 큐로, 짝수라면 L2 큐로 이동시킵니다.
L1, L2 큐에 속한 프로세스라면 L3 큐로 이동시킵니다.
L3 큐에 속한 프로세스라면 priority 값을 1감소하도록 setPriority() 함수를 호출합니다. 이 경우 프로세스의 큐 이동은 존재하지 않습니다.

프로세스가 이동과정을 끝냈거나 priority 재조정 이후에는 time_q를 초기화합니다.

프로세스들은 mlfq에 속하고 mlfq는 Round Robin 스케줄링 방식을 선택하므로 order역시 재조정이 필요합니다. 현재 실행중인 프로세스는 맨 마지막 순서로 이동시키기 위해서 현재 속한 큐의 q_order 값을 할당합니다.
단, 프로세스가 moq에 속한 경우 FCFS 스케줄링 방식(하나의 프로세스를 종료할 때까지 실행)을 선택하기에 order 재조정이 필요하지 않습니다.

Tick이 발생할 때마다 yield가 호출되며 g_ticks값이 하나 증가합니다. g_ticks 값이 증가하여 100이 될 경우 starvation을 막기 위해서 priorityboosting을 실행하여야 합니다. 하지만 moqact 활성 상태의 경우는 priorityboosting이 필요 없기에 moqact 값이 1인 경우는 if문이 실행하지 않도록 합니다.
다음은 priorityBoosting 함수입니다.

```
void
priorityBoosting(void)
{
    struct proc* p;
    struct proc* temp;

    for (int i = 0; i < 4; i++)
    {
        q_order[i] = 0;
    }
    int flag = 1;
    while (flag)
    {
        temp = 0;
        flag = 0;
        for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        {
            if (p->state != RUNNABLE || p->boosting || p->moq) continue;
            flag = 1;
            if (!temp || comPriority(p, temp)) temp = p;
        }
        if (!flag) break;
        temp->qlevel = 0;
        temp->time_q = 0;
        temp->priority = 0;
        temp->order = q_order[0]++;
        temp->boosting = 1;
    }

    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if (p->boosting == 1)
        {
            p->boosting = 0;
        }
    }
}
```

Boosting의 경우 모든 프로세스(MOQ 제외)가 L0 큐로 재조정되는 과정이므로 order를 표현하는 q_order 배열의 값을 모두 초기화합니다. 추가로 우선 플래그 값을 0으로 만듭니다.

이후 flag 값을 통해 재조정을 진행합니다. RUNNABLE한 모든 프로세스를 우선순위 그대로 L0 큐에 이동시키기 위해서 우선순위를 모두 비교한 후 가장 우선순위가 높은 프로세스부터 L0큐에 이동합니다. 만약 이동했다면 재조정을 수행한 것이고 프로세스의 boosting변수를 1로 설정하여 L0로 이동하였음을 나타냅니다. 이를 통해 다시 이동하는 일이 없도록 if문에 boosting값을 사용합니다. 추가로 재조정을 하였다면 flag값을 다시 1로 설정하여 반복문을 통해 다른 프로세스들이 재조정 될 수 있도록 합니다.

재조정 할 경우 L0큐로 이동하며 qlevel, time_q, priority, order값 역시 모두 새로 초기화 해 줍니다.

moq에 존재하는 프로세스는 재조정이 필요하지 않기에 이 역시 if문에 추가합니다.

만약 모든 프로세스가 재조정되었고 더이상 재조정이 필요하지 않은 경우 flag값이 1로 설정되지 않을 것이며 반복문을 탈출합니다. 이후 재조정 된 프로세스들의 boosting값을 모두 0으로 만들어 다음 priorityBoosting 과정을 가능하도록 만들어줍니다.

```
if (g_ticks == 100 && moqact != 1)
{
  priorityBoosting();
  g_ticks = 0;
  if (myproc()->moq != 1){
      myproc()->qlevel = 0;
      myproc()->order = q_order[0]++;
      myproc()->priority = 0;
      myproc()->time_q = 0;
      myproc()->boosting = 0;
  }
}
```

다시 yield 함수로 돌아와 현재 실행중인 process의 상태는 RUNNABLE 상태가 아니었기에 재조정 되지 않았으므로 실행중인 process의 재조정 과정을 수행합니다. 또한 g_ticks 값을 초기화하여 다음 Boosting이 가능하도록 합니다.

다음은 SYSCALL 함수들의 구현입니다.

```
int
getlev(void)
{
    if (myproc()->moq)
        return 99;
    return myproc()->qlevel;
}

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

Getlev의 경우 moq프로세스는 99, 그 외의 프로세스는 본인의 큐레벨을 리턴하도록 했습니다.

```
void
unmonopolize(void)
{
    moqact = 0;
    q_order[4] = 0;
    g_ticks = 0;
}
```

```
void
monopolize(void)
{
    moqact = 1;
}

int
```

위는 moqact를 활성하는 monopolize 함수, moqact를 비활성하는unmonopolize함수 입니다.
Monopolize의 경우 간단히 moqact값을 1로 세팅하였습니다.
Unmonopolize의 경우는 moqact를 0으로 하여 비활성화 한 후g_ticks값을 초기화 하여 다음 Boosting에 대비하였고, moq에서 사용하는 order 값을 초기화하여 오버플로우에 대비하였습니다.

```c
int
setmonopoly(int pid, int password)
{
    struct proc* p;
    if (password != 2019060546)
        return -2;

    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if (p->pid == pid)
        {
            p->moq = 1;
            p->order = q_order[4]++;
            moqsize++;
            return moqsize;
        }
    }
    return -1;
}
```

```c
int
sys_setmonopoly(void)
{
    int pid;
    int password;
    if (argint(0, &pid) < 0)
        return -1;
    if (argint(1, &password) < 0)
        return -1;
    return setmonopoly(pid, password);
}
```

위는 경우 해당 프로세스를 moq에 속하도록 하는 setmonopoly 함수입니다. 우선 password를 통해 해당 함수가 실행가능한 지 여부를 판단하고 입력받은 password가 맞을 경우 입력받은 pid에 해당하는 프로세스를 moq에 이동시키는 함수입니다. 모든 프로세스를 확인하여 동일한 pid가 존재 할 시 moq에 속하도록 값을 부여하고, order를 부여합니다. 그리고 전역변수 moqsize값을 증가시켜 moq에 하나 더 추가 되었음을 표시합니다.

Sys_함수의 경우 매개변수를 argint로 pid와 password를 입력받아옵니다. Argint 함수 속 첫 매개변수가 0이면 userapp에서 첫 매개변수를, 1이면 두 번째 매개변수를 받아온다는 의미로 알고 사용했습니다.

```c
int
setpriority(int pid, int priority)
{
    if (priority < 0 || priority > 10)
        return -2;
    struct proc* p;

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

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

위는 priority를 재조정하는 함수입니다. 입력받은 pid와 동일한 프로세스가 있으면 priority 재조정을 진행합니다. 입력받은 priority 값이 음수거나 10을 초과하는 이상한 값이거나 pid가 존재하지 않으면 진행하지 않습니다.

Sys_함수 역시 위의 함수와 동일하게 매개변수를 받아오는 역할을 수행합니다.

```c
int
sys_monopolize(void)
{
    monopolize();
    return 0;
}
```

```c
int
sys_unmonopolize(void)
{
    unmonopolize();
    return 0;
}
```

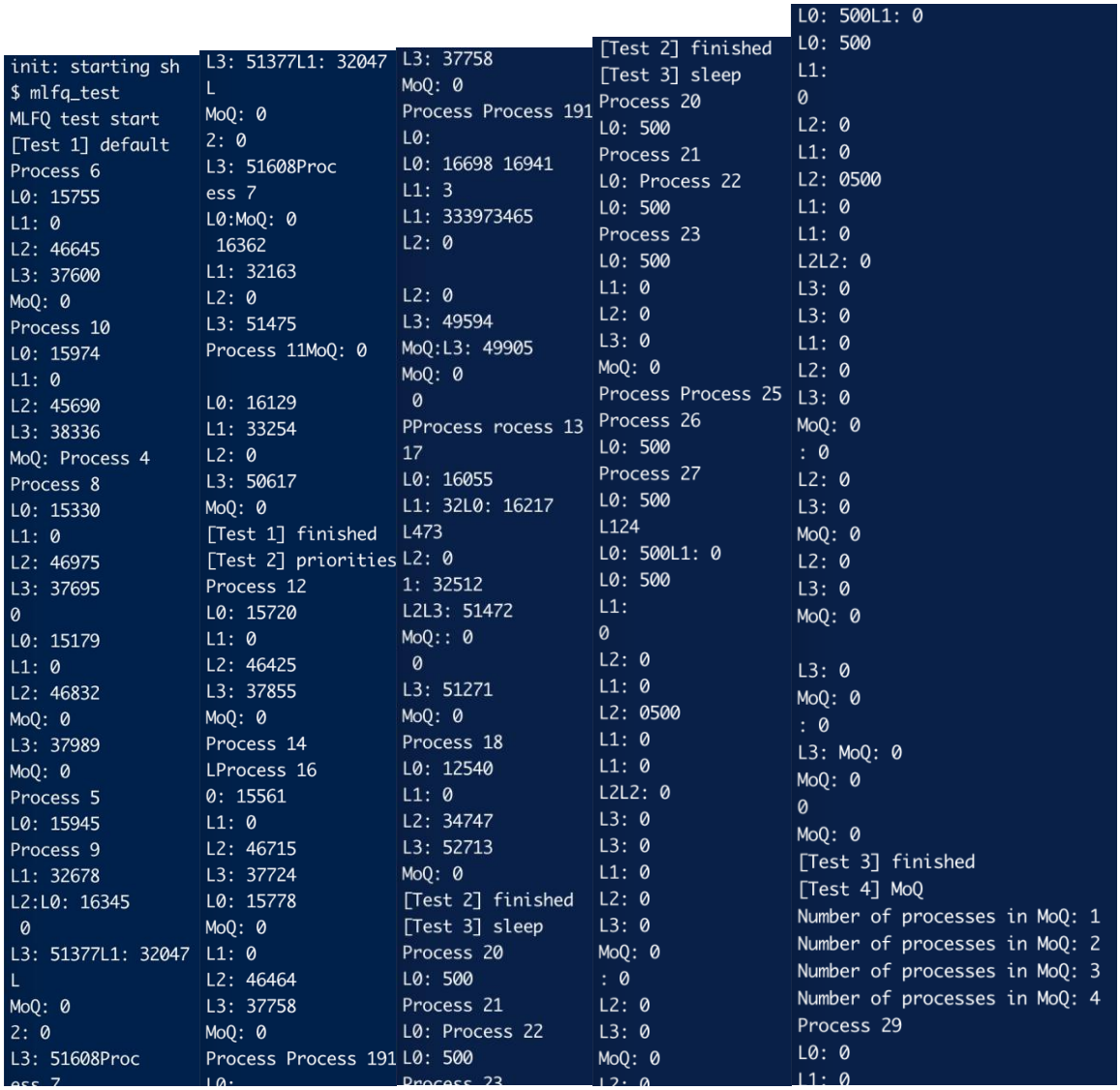
위는 간단한 wrapper 함수입니다.

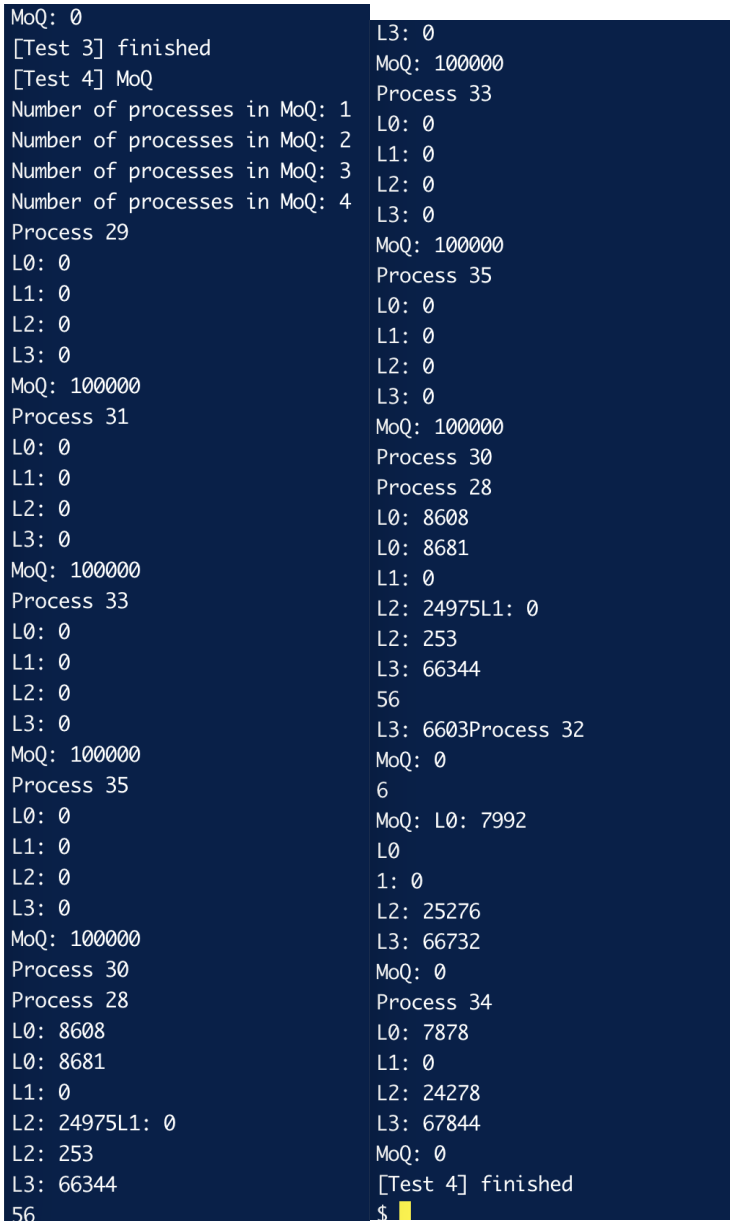
```
  }

  // Jump into the scheduler, never to
  curproc->state = ZOMBIE;
  if (curproc->moq == 1)
      moqsize--;
  sched();
  panic("zombie exit");
}
```

Exit 함수 중 마지막 부분입니다. Moq 에 속한 프로세스의 경우 moqsize를 감소함으로 scheduler 함수의 if문 속 moqsize 판별 가능하게 됩니다. 또한 종료된 프로세스의 size를 줄여 setmonopoly 의 return값이 올바르게 세팅되도록 합니다.

Trouble Shooting 세션 5번에 추가적인 설명과 실행 결과가 있습니다.

Result


init: starting sh
$ mlfq_test
MLFQ test start
[Test 1] default
Process 6
L0: 15755
L1: 0
L2: 46645
L3: 37600
MoQ: 0
Process 10
L0: 15974
L1: 0
L2: 45690
L3: 38336
MoQ: Process 4
Process 8
L0: 15330
L1: 0
L2: 46975
L3: 37695
0
L0: 15179
L1: 0
L2: 46832
MoQ: 0
L3: 37989
MoQ: 0
Process 5
L0: 15945
Process 9
L1: 32678
L2:L0: 16345
0
L3: 51377L1: 32047
L
MoQ: 0
2: 0
L3: 51608Proc
ess 7
L3: 51377L1: 32047
L
MoQ: 0
2: 0
L3: 51608Proc
ess 7
L0:MoQ: 0
16362
L1: 32163
L2: 0
L3: 51475
Process 11MoQ: 0
L0: 16129
L1: 33254
L2: 0
L3: 50617
MoQ: 0
[Test 1] finished
[Test 2] priorities
Process 12
L0: 15720
L1: 0
L2: 46425
L3: 37855
MoQ: 0
Process 14
LProcess 16
0: 15561
L1: 0
L2: 46715
L3: 37724
L0: 15778
MoQ: 0
L1: 0
L2: 46464
L3: 37758
MoQ: 0
Process Process 191
L0:
L3: 37758
MoQ: 0
Process Process 191
L0:
L0: 16698 16941
L1: 3
L1: 333973465
L2: 0
L2: 0
L3: 49594
MoQ:L3: 49905
MoQ: 0
0
PProcess rocess 13
17
L0: 16055
L1: 32L0: 16217
L473
L2: 0
1: 32512
L2L3: 51472
MoQ:: 0
0
L3: 51271
MoQ: 0
Process 18
L0: 12540
L1: 0
L2: 34747
L3: 52713
MoQ: 0
[Test 2] finished
[Test 3] sleep
Process 20
L0: 500
Process 21
L0: Process 22
L0: 500
[Test 2] finished
[Test 3] sleep
Process 20
L0: 500
Process 21
L0: Process 22
L0: 500
Process 23
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process Process 25
Process 26
L0: 500
Process 27
L0: 500
L124
L0: 500L1: 0
L0: 500
L1:
0
L2: 0
L1: 0
L2: 0500
L1: 0
L1: 0
L2L2: 0
L3: 0
L3: 0
L1: 0
L2: 0
L3: 0
MoQ: 0
: 0
L2: 0
L3: 0
MoQ: 0
L0: 500L1: 0
L0: 500
L1:
0
L2: 0
L1: 0
L2: 0500
L1: 0
L1: 0
L2L2: 0
L3: 0
L3: 0
L1: 0
L2: 0
L3: 0
MoQ: 0
: 0
L2: 0
L3: 0
MoQ: 0
L3: 0
MoQ: 0
: 0
L3: MoQ: 0
MoQ: 0
0
MoQ: 0
[Test 3] finished
[Test 4] MoQ
Number of processes in MoQ: 1
Number of processes in MoQ: 2
Number of processes in MoQ: 3
Number of processes in MoQ: 4
Process 29
L0: 0



실행 결과는 위와 같았습니다. Test 1의 경우 제공해주신 test.pdf에서는 홀수 프로세스가 먼저 끝났지만, 본인의 스케줄링방식의 경우는 항상 짝수 프로세스가 먼저 끝났습니다. 이유는 찾지 못했습니다.

Test 2의 경우 pid가 높은 프로세스의 priority를 높게 주어 대부분의 경우 pid가 높은 것이 먼저 종료되었습니다. 이번 결과의 경우 그렇지 않았지만 pid가 높은 프로세스가 먼저 끝나는 경우가 훨씬 많았습니다.

Test 3 는 거의 항상 프로세스가 모두 L0에서 끝나는 결과가 나왔습니다.

Test 4 는 MOQ가 FCFS로 구현되기에 pid가 작은 순서로 결과가 나와야 합니다. 본인의 스케줄링은 잘 구현된 것 같습니다.

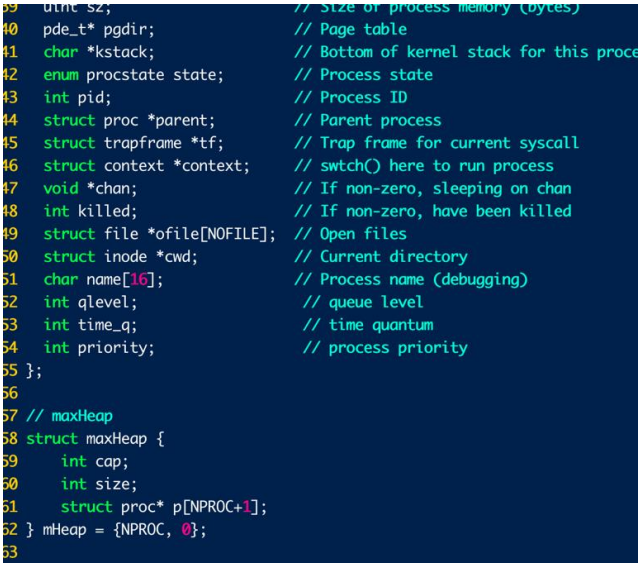
Trouble shooting

1. 인자 값 전달

   Syscall 구현시 argint를 사용하여 인자값을 전달받는데, password의 경우 argint(1, &password) 방식으로 인자를 전달 받아야 합니다. 하지만 이를 인지하지 못하였고 argint(0, &password) 를 사용하여 계속 잘못된 password를 받아와서 프로세스가 진행되지 못하는상황이 발생했습니다.

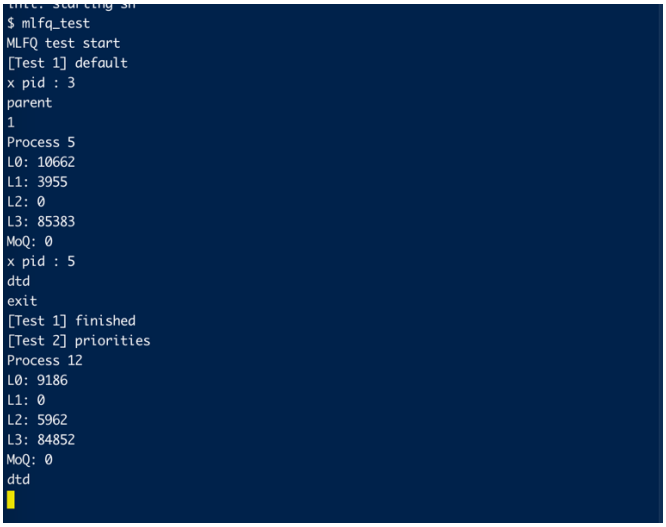
   이를 알맞게 수정하여 문제를 해결했습니다.

2. 스케줄링 방식 잘못 된 접근



```
39   uint sz;                     // Size of process memory (bytes)
40   pde_t* pgdir;                // Page table
41   char *kstack;                // Bottom of kernel stack for this proce
42   enum procstate state;        // Process state
43   int pid;                     // Process ID
44   struct proc *parent;         // Parent process
45   struct trapframe *tf;        // Trap frame for current syscall
46   struct context *context;     // swtch() here to run process
47   void *chan;                  // If non-zero, sleeping on chan
48   int killed;                  // If non-zero, have been killed
49   struct file *ofile[NOFILE];  // Open files
50   struct inode *cwd;           // Current directory
51   char name[16];               // Process name (debugging)
52   int qlevel;                   // queue level
53   int time_q;                   // time quantum
54   int priority;                 // process priority
55 };
56
57 // maxHeap
58 struct maxHeap {
59     int cap;
60     int size;
61     struct proc* p[NPROC+1];
62 } mHeap = {NPROC, 0};
63
```

처음 스케줄링방식은 직접 queue 구조체를 구현하여 적용하려고 했습니다. 하지만 이렇게 구현할 경우 어떤 경우에 process가 생성되고 종료되는지 정확한 부분을 다 세세하게 알 수 없었습니다. 그렇기에 insert와 delete모두 정확히 이루어지지 않았고, 스케줄링 도중 프로세스가 존재하지 않는다 거나 좀비 프로세스를 스케줄링 하는 일이 많았습니다. 이는 본인 스스로 해결할 수 있는 부분이 아님을 깨닫고 다른 방식으로 스케줄링 하는 것이 맞다고 판단하여 따로 구조체를 사용하지 않는 방식으로 다시 작성하여 지금의 결과를 도출하게 되었습니다.



```
$ mlfq_test
MLFQ test start
[Test 1] default
x pid : 3
parent
1
Process 5
L0: 10662
L1: 3955
L2: 0
L3: 85383
MoQ: 0
x pid : 5
dtd
exit
[Test 1] finished
[Test 2] priorities
Process 12
L0: 9186
L1: 0
L2: 5962
L3: 84852
MoQ: 0
dtd
```

Heap구조체를 통한 스케줄링은 위와 같이 정상적이지 않은 결과였습니다.

3. Ticks , g_ticks

   실제 xv6에서 사용되는 값인 ticks를 0으로 초기화 하면 문제가 발생하였기에 새롭게 g_ticks 라는 변수를 사용하여 0으로 초기화할 수 있도록 하여 Boosting에 사용할 수 있었습니다.

4. priorityBoosting 함수 작동하지 않는 경우

   처음 priortyBoosting을 구현했을 때는 이중 for문을 사용하여 구현하였는데, 두 for문 모두 모든 프로세스를 탐색하는 방식으로 구현했습니다. 하지만 계속해서 한 프로세스만 Boosting되는 결과가 나왔습니다. 이를 해결하고자 이중 for문이 아니라 flag를 사용하는 방식을 통해서 priorityBoosting 함수를 구현하였는데 정상적으로 작동함을 확인 하였고 문제를 해결할 수 있었습니다

5. Test 1 짝수 프로세스 먼저 끝나는 문제 해결

위의 실행결과에서 Test 1의 경우 항상 짝수 프로세스가 먼저 끝납니다. 이는 L3의 스케줄링 방식에서 L0, L1, L2에서 사용한 Round Robin 방식을 그대로 사용하여 Order를 재조정 하였기 때문에 일어나는 것을 알았습니다. L3의 프로세스들이 Priority가 같은 경우 order를 재 조정하는 방식을 사용했기에 항상 짝수 프로세스가 먼저 끝나게 된 것입니다. 이를 수정하여 L3의 스케줄링 방식을 FCFS 방식으로 L3속 Order를 재 조정 하지 않도록 수정하였습니다. 그 결과 홀수 프로세스가 먼저 끝나게 되었습니다. 아래의 스크린샷은 L3의 FCFS 구현과 바뀐 실행 결과 입니다.

```
463   }
464   if (myproc()->moq != 1 && myproc()->qlevel != 3)
465       myproc()->order = q_order[myproc()->qlevel]++;
466   myproc()->state = RUNNABLE;
467   sched();
468   release(&ptable.lock);
```

```
$ mlfq_test
MLFQ test start
[Test 1] default
Process 7
L0: 11634
L1: 23680
L2: 0
L3: 64686
MoQ: 0
Process 9
L0: 1Process 5
L0: 11819
L1: 23552
L2: 0
1976
L1: 23640
L2: L3: 64629
MoQ0
L3: 64384: 0
Process 11
L0: 12195
L1: 24680
L2: 0
L3: 63125
MoQ: 0

MoQ: 0
Process 8
L0Process 10
L0: 16291
: 16415
L1Process 6
L1: 0
L2: 48253
L3: : 0
L2: 48153
L0: 16370
35456
Process 4
L0: 16325
L1: 0
```