Project02 Wiki

2020081958 송재휘

1. Design

이번 Project02는 크게 RR 방식을 따르는 MLFQ와 FCFS 방식의 MOQ 두 부분으로 나눌 수 있다.

우선 첫번째로 4개의 layer를 가지는 MLFQ를 구현하기 위해 명세에 제공된 관련 시스템 콜 (yield, getlev, setpriority) 3개를 먼저 추가할 것이다. 시스템 콜을 선언한 이후에는 MLFQ가 말 그대로 Multi-Layer Feedback Queue이기 때문에 mlfq를 위한 구조체를 구현할 것이다. 위의 행위들은 proc.h에서 구현할 것이다. 구조체에는 특정 layer 큐의 timequatum을 나타내기 위한 int형 변수와 해당 큐에 속한 프로세스들을 가리키기 위한 proc 구조체 기반의 process 배열을 추가할 것이다. 추가적으로 MLFQ내부에서 큐 level 이동 및 L3에서 이루어질 priority 기반의 처리를 위해 proc 구조체에도 int 형의 priority, qlevel, ticks 변수를 추가할 것이다. qlevel은 해당 프로세스가 속한 qlevel을 나타내는 것으로 0, 1, 2, 3, 99(0~3은 MLFQ, 99는 MOQ에 속한 경우)의 값을 가진다. Priority의 경우 0~10까지의 값을 나타낸다. Ticks는 각 프로세스가 사용한 ticks를 나타낸다. MLFQ의 각 level 큐의 time quantum과의 비교를 위해 사용한다. 이후 본격적인 처리를 위해 proc.h에서 정의한 mlfq 구조체 4개 사이즈의 구조체 배열을 proc.c에서 선언하여 4개의 level을 가진 mlfq를 만들 것이다. 추가적으로 mlfq처리 과정에서 필요한 spinlock과 Global tick 카운트를 위한 변수의 선언도 해줄 것이다. 이후 각 process가 시작 될 때 아까 새로 선언한 process 구조체 내부의 변수 qlevel, ticks, priority를 0으로 초기화해 줄 것이다. 이후 프로세스가 생성될 때마다 가장 상위의 레벨인 L0에 들어가도록 설계할 것이다. 이렇게 되면 생성된 모든 프로세스들은 우선 L0큐에 쌓일 것이고, 스케줄러가 처음 시작될 때 각 mlfq level의 time quantum을 2 * i +2로 설정해줄 것이고, 나중에 구현할 MOQ처리를 나타내는 monopolized 변수가 false이면 프로세스 테이블을 돌면서 가장 높은 level의 큐에 있는 RUNNABLE한 프로세스를 차례로 실행한다. 상위 level의 RUNNABLE process를 우선적으로 스케줄링하며 상위 level에 실행 가능한 프로세스가 없다면, 하위 레벨을 실행하는 구조를 가진다. 이때 각 프로세스의 tick을 확인해 해당 level의 mlfq time quantum을 넘었다면 해당 프로세스를 하위 큐로 이동시킨다. 이때 기준은 L0의 경우 pid가 홀수면 L1으로 pid가 짝수면 L2로 내려가며, time quantum 즉, 프로세스의 ticks를 초기화해 줄 것이다. L1, L2의 경우 L3로 내려가며, L3의 경우 하위의 큐가 없으므로 priority를 1 빼준다(최소값은 0). 이렇게 time quantum test를 통과한 경우, 상위 level의 실행가능한 process를 결정한다. 이렇게 골라

진 process이 L3에 해당하는 경우 priority까지 고려해야 하기 때문에 다시 for문을 돌며 높은 priority의 프로세스를 찾아 해당 프로세스를 스케줄링한다. (즉, process이 L0, L1, L2에 해당하는 경우 그냥 ptable에 들어온 process를 차례로 뒤져 가장 상위 레벨에 해당하는 프로세스를 실행하고, L3에 해당하는 경우에는 priority가 가장 높은 process를 찾아서 실행한다.) 이후 MLFQ 내의 프로세스가 스케줄링 될 때마다 global ticks와 해당 프로세스의 ticks를 1 올려준다. 이후 priority boosting을 위해 global ticks가 100이 넘는지를 체크하고 만약 넘는다면 priority boosting 함수를 실행 할 것이다. 이 과정에서 MLFQ level 간 process의 이동 및 priority boosting 처리를 위한 함수 등은 아래 구현 파트에서 차례로 설명할 것이다.

두번째로는 MOQ를 만들어주어야 하는데 MOQ내의 프로세스 처리를 위해선 monopolize가 되어야하기 때문에 cpu라는 구조체에 monopolized라는 int형 변수를 추가해줘 현재. Monopolized된 상태인지를 체크할 것이다. (MLFQ 처리를 위해서는 monopolized가 0이어야 하기 때문에 MLFQ를 구현하는 과정에서 미리 cpu 구조체에 monopolized를 추가하였다. 하지만 설명의 편의를 위해 MOQ파트에 언급한다.) 이후 명세의 시스템 콜인 setmonopoly, monopolize, unmonopolize를 추가할 것이다. 또한, 앞서 MLFQ 구조체를 만든 것처럼, proc.h에 MOQ 구조체도 구현할 것이다. 해당 구조체에는 나중에 setmonopoly 시스템 콜에서 암호가 일치할 경우 MoQ 내부의 종료되지 않은 프로세스의 개수를 반환하기 위해 len이라는 int형 변수와 MoQ에 속하는 process를 저장할 MLFQ와 동일한 형태의 proc 구조체 배열을 넣어주었다. 똑같이 proc.c에서 moq라는 구조체 변수를 선언해주었고, monopolize 시스템콜을 통해 cpu 구조체의 monopolized가 1로 설정된 경우 스케줄러에서 moq 구조체의 proc 구조체 배열을 차례로 돌며 실행하도록 할 것이다. (MoQ에 먼저 들어온 프로세스가 MoQ의 proc 구조체의 앞쪽부터 차례로 저장될 것이기 때문에 순서대로 돌면서 수행해주었다.(FCFS 방식) 이후, 더 이상 실행할 process가 없다면 unmonopolize 시스템 콜을 호출해 cpu의 Monopolized 값과 global tick를 0으로 바꾸어준다. MLFQ에 있는 process를 MoQ로 이동시키는 함수는 구현 파트에서 설명하겠다.

마지막으로, 유저 프로세스가 잘못된 행동을 하여 강제 종료되는 경우는 이를 필터링하는 코드가 trap.c에 구현되어 있기 때문에 해당 부분에 강제종료를 시켰다는 메시지를 추가해 줄 계획이다.

2. Implementation

Design 파트에서 전반적으로 처리할 flow에 대한 언급을 했기 때문에, 여기서는 코드 위주로 구현을 설명하고자 한다.

우선 MLFQ와 MOQ 중 어떤 큐의 프로세스를 실행할 지 결정하기 위해 proc.h에 선언된 cpu 구조체에 int monopolized;를 추가해주었다. 또 proc 구조체에 process에 int qlevel, int priority, int ticks를 추가해주었다. 또 mlfq와 moq 구조체를 선언해 주었는데 기능은 앞의 design에서 언급했음으로 코드는 다음과 같다.

```
// MLFQ struct
struct mlfq {
  int timequantum;
  struct proc *proc_list[NPROC];
};

// MOQ struct
struct moq {
  int len;  // # of unfinished processes
  struct proc *proc_list[NPROC];
};
```

이번엔 해당 프로젝트를 위해 구현한 system call을 설명할 것이다. proc.c에 시스템 콜의 함수를, sysproc.c에는 해당 함수를 위한 wrapper function을 정의하였다. 각 시스템 콜을 정의한 후 defs.h, syscall.h, syscall.c, user.h. usys.S, 에 차례로 추가해주었다! (시스템 콜 등록에 대한 설명은 project01과 겹침으로 생략했다.)

우선 yield의 경우는 이미 proc.c에 구현이 되어있기 때문에, sysproc.c에서

```
// make function yield to system call
int
sys_yield(void)
{
  yield();
  return 0;
}
```

형태의 wrapper function을 만들어주었다.

두번째로, getlev는 현재 process의 큐 레벨을 리턴하는 함수로 return mypro()->qlevel;을 통해 구현하였다. 이후 sysproc.c에 int sys_getlev(void)라는 wrapper function을 정의하고 return getlev();를 추가했다.

세번째로, setpriority의 경우, pid와 priority를 인자로 받는데, 인자로 들어온 pid의 process에 priority를 설정하는 시스템콜로 입력된 priority 값이 0~10의 범위가 아니면 -2를 반환한다. 그게 아니라면 ptable을 돌며 해당 pid의 process를 찾고, p->priority를 입력 받은 priority 값으로 설정하고, 0을 반환한다. 해당 pid의 프로세스가 없을 경우, -1을 반환한다.

```
// set priority to a process with specific pid.
int
setpriority(int pid, int priority)
{
  if (priority < 0 || priority > 10) {
    return -2;
  }

  struct proc *p;
  acquire(&ptable.lock);

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

sysproc.c에서는 pid와 priority를 argint를 통해 각각 받아와 이를 proc.c의 setpriority 함수에 인자로 넘겨주는 wrapper function을 정의했다.

```
// wrapper function for int setpriority(int pid, int priority) in proc.c
int
sys_setpriority(void)
{
  int pid, priority;
  argint(0, &pid);
  argint(1, &priority);
  return setpriority(pid, priority);
}
```

네번째로, setmonopoly 시스템 콜의 경우, 인자로 입력받은 password가 내 학번인 2020081958과 같은지 판단하고, 아니면 에러 문구와 -2를 반환한다. 암호가 맞다면, ptable을 돌며 인자의 pid를 가진 process를 찾고 해당 process를 push_moq함수를 통해 moq로 이동시킨다. (push_moq함수는 밑에서 설명할 예정이다.) 이후 moq의 길이를 1추가해준다. 해당 pid가 없다면 -1을 반환한다.

```
// move a process with certain pid to MoQ
int
setmonopoly(int pid, int password) {
  if (password != 2020081958) {
    cprintf("Wrong password! Unable to move this process into MoQ.\n");
    return -2;
  }

  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->pid == pid) {
      push_moq(p, p->qlevel);
      release(&ptable.lock);
      return ++moq.len;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

Setpriority의 wrapper function과 마찬가지로 argint를 통해 pid와 password를 입력받고, setmonopoly 함수에 인자로서 이를 전달하며 시행시킨다 그리고, 그 결과 값을 반환한다.

```
// wrapper function for setmonopoly in proc.c
int
sys_setmonopoly(void)
{
  int pid, password;
  argint(0, &pid);
  argint(1, &password);
  return setmonopoly(pid, password);
}
```

다섯번째로, monopolize 시스템콜의 경우는 MoQ가 CPU를 독점해 사용하도록 하는 시스템 콜이기 때문에, 앞서 design에서 명시한 것처럼 이를 판단하는 mycpu()의 monopolized의 값을 1로 바꾸는 과정을 수행한다.

```
// set MoQ process monopolize CPU
void
monopolize(void) {
  pushcli();
  mycpu()->monopolized = 1;
  cprintf("Successfully monopolized code: %d\n", mycpu()->monopolized);
  popcli();
}
```

```
// wrapper function for monopolize
int
sys_monopolize(void)
{
  monopolize();
  return 0;
}
```

마지막으로, unmonopolize 시스템 콜의 경우는 monopolize와 반대로 monopolized 값을 0으로 바꾸고, global_ticks를 초기화한다.

```
// return to MLFQ
void
unmonopolize(void) {
  pushcli();
  mycpu()->monopolized = 0;
  popcli();
  global_ticks = 0;
}
```

```
// wrapper function for unmonopolize
int
sys_unmonopolize(void)
{
  unmonopolize();
  return 0;
}
```

추가적으로, MLFQ에서 프로세스의 큐 간 이동을 위해 proc.c에 해당 함수들을 추가로 정의해주었다.

1) push_proc: MLFQ의 특정 level의 큐로 process를 추가하기 위한 함수이다. 우선 큐 레벨이 0~3 범위인지 확인하고, 아니면 -1을 반환한다. 범위 테스트를 통과하면 mlfq 인자로 들어온 level의 queue의 proc_list를 돌면서 빈 자리를 발견하고, 발견했다면 해당 영역에 process를 추가한다. 이때 ticks를 0으로 초기화하고, qlevel을 해당 queue의 level로 바꿔준다. 이때 flag를 추가한 이유는, 나중에 process 간의 큐 level 이동을 위해 push, pop이 반복해서 일어나는 경우, 기존 큐에서의 pop은 성공했지만 새로운 큐로의 push에 실패한 경우, 기존의 ticks 값을 초기화하지 않고, 이전 상태로 돌아가기 위함이다. (flag가 0이면 기본 push, 1이면 pop 이후 push 실패를 의미)

```
// push process into specific level MLFQ
int
push_proc(struct proc *proc, int level, int flag)
{
  if (level < 0 || level > 3) {
    cprintf("Push error, L%d queue does not exist!\n", level);
    return -1;
  }
  for (int i=0; i < NPROC; i++) {
    if (mlfq[level].proc_list[i] == 0) {
      if (flag == 0)
        proc->ticks = 0;
      proc->qlevel = level;
      mlfq[level].proc_list[i] = proc;
      return 0;
    }
  }
  cprintf("No space in L%d queue!\n", level);
  return -1;
}
```

2) pop_proc: MLFQ의 특정 level의 큐에서 특정 process를 삭제하기 위한 함수이다. Level 범위 체크는 push_proc과 동일하고, 이번엔 인자로 받은 level의 mlfq의 proc_list를 차례로 돌며, 인자로 받은 process를 찾아 해당 위치를 비우고, 0을 반환한다. 해당 큐에 해당 프로세스가 없다면, -1을 반환한다.

```
// pop process from specific level MLFQ
int
pop_proc(struct proc *proc, int level)
{
  if (level < 0 || level > 3) {
    cprintf("Pop error, L%d queue does not exist!\n", level);
    return -1;
  }
  for (int i = 0; i < NPROC; i++) {
    if (mlfq[level].proc_list[i] == proc) {
      mlfq[level].proc_list[i] = 0;
      return 0;
    }
  }
  cprintf("L%d queue does not contain process with pid=%d!\n", level, proc->pid);
  return -1;
}
```

3) change_queue: pop_proc과 push_proc을 차례로 수행하며 process의 queue 간 이동을 지원한다. 둘 중 하나라도 실패하면 -1을 반환한다. 성공 시 0을 반환.

```
// move proc from a queue to another queue.
int
change_queue(struct proc *proc, int level)
{
  if (pop_proc(proc, proc->qlevel) == -1) {
    cprintf("Unable to move proc with pid=%d from L%d queue to L%d queue.\n", proc->pid, proc->qlevel,
level);
    return -1;
  }
  else {
    if (push_proc(proc, level, 0) == -1) {
      push_proc(proc, proc->qlevel, 1);
      return -1;
    }
  }
  return 0;
}
```

4) push_moq: 특정 level의 MLFQ에 있는 프로세스를 moq로 이동시킨다. (setmonopoly함수 호출 시 사용됨) 기존 MLFQ에서 pop을 수행하고, 성공 시 moq의 proc_list에서 빈 곳을 찾아 해당 프로세스의 qlevel을 99로 설정한 뒤 proc_list에 추가한다. MoQ에 공간이 없을 경우 -1을 반환한다.

```
// push process to moq
int
push_moq(struct proc *proc, int level)
{
  if (pop_proc(proc, level) == -1) {
    cprintf("Failed to setmonopoly.\n");
    return -1;
  }
  for(int i = 0; i < NPROC; i++) {
    if (moq.proc_list[i] == 0) {
      proc->qlevel = 99;
      moq.proc_list[i] = proc;
      return 0;
    }
  }
  cprintf("No space in MoQ!\n");
  return -1;
}
```

5) priority_boosting: 스케줄러 코드에서 global_ticks가 100이 넘은 경우 호출 되는 함수로, ptable을 돌면서 qlevel이 0~3인 (MLFQ에 해당하는) 프로세스를 L0 큐로 재조정하고, 각 프로세스의 ticks를 0으로 초기화한다. 이미 L0인 경우에 있는 경우는 재조정없이 process의 ticks만 0으로 초기화한다.

```
// priority boosting
int
priority_boosting(void) {
  struct proc *p;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p == 0) {
      continue;
    }
    if (p->qlevel > 0 && p->qlevel < 4) {
      change_queue(p, 0);
    }
    else {
      p->ticks = 0;
    }
  }
  return 0;
}
```

해당 함수들을 추가로 정의한 뒤 defs.h의 proc.c영역에 해당 함수들을 추가해주었다.

```
void            yield(void);
int             push_proc(struct proc*, int, int);
int             pop_proc(struct proc*, int);
int             change_queue(struct proc*, int);
int             priority_boosting(void);
int             push_moq(struct proc*, int);
int             getlev(void);
int             setpriority(int, int);
int             setmonopoly(int, int);
void            monopolize(void);
void            unmonopolize(void);
```

이제 필요한 기능 및 자료구조에 대한 구현을 마쳤고, 이제는 이를 활용한 스케줄러의 구현에 대해 설명하려고 한다. 스케줄러의 구현은 proc.c에서 이루어진다.

우선 proc.c의 코드 윗부분에 전역변수로 4크기의 mlfq 구조체 배열, global_ticks, 하나의 moq 구조체, mlfq 배열의 값을 변경할 때 사용할 spinlock을 선언하였다.

```
// initialize 4 mlfqs
struct mlfq mlfq[4];
// for counting global ticks, initialized with 0
int global_ticks;
// spinlock for mlfq
struct spinlock mlfq_lock;
// initialize MoQ
struct moq moq;
```

이후 프로세스가 생성될 때마다, 앞서 proc 구조체에 추가한 특성들인 qlevel, ticks, priority를 0으로 초기화 해주고, 해당 process를 L0 큐에 넣어줘야 하는데, 이를 처리하기 위해 **allocproc 함수**에 해당 프로세스의 추가된 특성들을 초기화하는 코드를 넣어주었다.

```
// initialize new process's qlevel, tick, prioirity with 0
p->qlevel = 0;
p->ticks = 0;
p->priority = 0;
```

추가적으로 해당 함수가 끝나기 전에 프로세스를 L0 큐에 넣기 위해, 앞에서 정의한 push_proc(p, 0, 0)이라는 코드를 추가해주었다.

```
// push new proc to level 0 mlfq
push_proc(p, 0, 0);
```

앞서 언급한 것처럼, moq 구조체에는 MoQ에 존재하는 종료되지 않은 프로세스의 개수를 의미하는 len이라는 변수가 있었는데, setmonopoly를 통해 process가 MoQ로 이동할 때 해당 값이 1 늘어났다. 하지만, 프로세스가 종료되는 경우, 이 값을 1 줄여주어야 하는데, 이를 처리하기 위해 proc.c의 exit 함수(프로세스의 종료를 처리하는 함수)의 끝부분에 현재 프로세스의 qlevel이 99라면 moq의 길이를 1 감소시키는 코드를 넣어주었다.

```
if (curproc->qlevel == 99)
  moq.len--;
```

scheduler 함수에서는 스케줄링을 위한 전반적인 처리를 다룬다. 해당 코드의 길이가 길기 때문에 한번에 코드를 첨부하는 대신 부분적으로 설명하고자 한다. 우선 scheduler 함수의 상단에 이후 MLFQ 처리를 위한 proc 구조체 포인터들을 선언했다. *p, *p2, *p3는 각각 ptable을 돌며 process를 가리키기 위해 사용된다. *lowest_queue_proc은 이후 상위 큐의 RUNNABLE한 프로세스의 우선 수행을 보장하기 위한 포인터이며, *highest_priority_proc은 L3에 해당하는 프로세스 중 priority가 가장 높은 프로세스를 가리키기 위한 포인터이다.

```
void
scheduler(void)
{
  struct proc *p, *p2, *p3;
  struct proc *lowest_queue_proc, *highest_priority_proc;
  struct cpu *c = mycpu();
  c->proc = 0;
```

이후엔 MLFQ의 각 level 별로 time quantum(2*i+2라는 규칙으로)을 정의해주었다.

```
// set the time quantum for each MLFQ
acquire(&mlfq_lock);
for (int i = 0; i < 4; i++)
  mlfq[i].timequantum = 2 * i + 2;
release(&mlfq_lock);
```

추가적으로, moq의 길이, global_ticks도 0으로 초기화해주었다.

```
// set moq->len with 0
moq.len = 0;
global_ticks = 0;
```

이젠 무한 루프를 돌며 프로세스를 특정 규칙에 맞게 스케줄링 하는 과정만 남았는데, 앞서 cpu 구조체에 추가한 monopolized라는 변수를 통해 MLFQ와 MoQ 중 어떤 것을 시행할지 결정하도록 만들었다. monopolized가 1이면, MoQ기반의 스케줄링을 의미하고, 그 반대의 경우에는 MLFQ 스케줄링을 진행한다. 이 경우는 MLFQ기반 스케줄링의 코드이다. 첫번째 루프를 통해 ptable을 돌며 프로세스를 읽어오고, 해당 프로세스의 state이 RUNNABLE하지 않거나, 정의되지 않았다면 다음으로 넘어간다.

```
for(;;){
  sti();
  acquire(&ptable.lock);
  // case that cpu does not monopolized
  if (!c->monopolized) {
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if (c->monopolized == 1)
        break;
      if (p->state != RUNNABLE || !p)
        continue;
```

이번엔 상위 level에 존재하는 RUNNABLE 프로세스를 찾기 위해 두번째 루프를 돈다. 코드는 첫번째와 동일하다. 이때 qlevel의 비교를 위해 lowest_queue_proc에 p를 넣어준다. 두번째 루프를 통해 다시한번 ptable을 돌며 p2에 process를 읽어온다. P2의 state 판단은 위와 같다.

```
// find RUNNABLE process in lowest MLFQ
lowest_queue_proc = p;
for (p2 = ptable.proc; p2 < &ptable.proc[NPROC]; p2++) {
  if (c->monopolized == 1)
    break;
  if (p2->state != RUNNABLE || !p2)
    continue;
```

이제는 차례로 읽어온 p2에 대해서 time quantum을 비교한다. P2의 time quantum이 해당 프로세스가 속한 level의 mlfq의 time quantum보다 크다면 하위 큐로 조정한다. 이 때 조정 방식은 앞서 design에서 언급한 것과 같다. (L0일 경우 pid 기준 짝수->L2 홀수-> L1로 조정, L1,L2일 경우 L3로 조정, L3일 경우 별도 조정 없이 priority -1) Time quantum 테스트를 통과하면 qlevel을 비교해 더 작은 프로세스를 lowest_queue_proc에 저장한다.

```
  // check proc ticks surpassed time quantum of MLFQ
  int lev = p2->qlevel;
  if (p2->ticks > mlfq[lev].timequantum) {
    p2->ticks = 0;
    if (lev == 0) {
      int pid = p2->pid;
      // even number case
      if (pid % 2 == 0)
        change_queue(p2, 2);
      else
        change_queue(p2, 1);
    }
    else if (lev == 3) {
      if (p2->priority > 0)
        p2->priority--;
    }
    else if (lev == 1 || lev == 2) {
      change_queue(p2, 3);
    }
    continue;
  }

  if (p2->qlevel < lowest_queue_proc->qlevel)
    lowest_queue_proc = p2;
}
```

만약 qlevel이 L3인 경우에는 priority까지 고려해야하기 때문에, 또 다른 루프를 돌며 L3 큐의 프로세스 중 가장 높은 priority를 가진 프로세스를 찾는다. 이후 조건문을 통해 lowest_queue_proc의 level이 3일 경우 다음 실행할 process를 나타내는 p를 highest_priority_proc으로 설정한다. 그게 아니라면 lowest_queue_proc으로 설정한다.

```
// priority scheduling for L3 MLFQ
highest_priority_proc = p;
for (p3 = ptable.proc; p3 < &ptable.proc[NPROC]; p3++) {
  if (c->monopolized == 1)
    break;
  if (p3->state != RUNNABLE || !p3)
    continue;

  if (highest_priority_proc->priority < p3->priority && p3->qlevel == 3)
    highest_priority_proc = p3;
}

// if L3 MLFQ, do priority scheduling
if (lowest_queue_proc->qlevel == 3 && highest_priority_proc->qlevel == 3)
  p = highest_priority_proc;
// else, sheduling process in lower queue first.
else
  p = lowest_queue_proc;
```

이후엔 기존 스케줄러와 코드와 같이 p->state을 RUNNING으로 바꾸고 프로세스를 실행한다. 해당 코드는 기존 스케줄러와 같음으로 생략한다.

Process가 실행되고 나면 global_ticks 값을 1추가 하고, 해당 프로세스의 ticks도 1 추가한다.

```
// Process is done running for now.
// It should have changed its p->state before coming back.
c->proc = 0;
// Increment global and proc ticks
acquire(&mlfq_lock);
global_ticks++;
release(&mlfq_lock);
p->ticks++;
```

이후에는 priority boosting의 시행 조건을 판단하기 위한 코드를 추가했다. Global_ticks가 100이 되면 priority boosting 함수를 호출하고, global_ticks를 0으로 초기화한다.

```
// priority boosting for preventing starvation.
if (global_ticks == 100) {
  priority_boosting();
  acquire(&mlfq_lock);
  global_ticks = 0;
  release(&mlfq_lock);
}
```

여기까지가 MLFQ의 처리 코드이다. 이후엔 MoQ 부분이 남았는데, 이는 MLFQ보다 간단하다. Monopolized 값이 1이면 이는 MoQ가 CPU를 독점함을 의미하고, MoQ 스케줄링을 실행한다. MoQ는 FCFS 방식을 따르기 때문에 moq의 proc_list를 차례로 돌면서 해당 process가 RUNNABLE한 경우 실행하면 된다(차례로 실행하면 FCFS 충족). 이후 더 이상 실행할 프로세스가 없는 경우에는 자동으로 unmonopolize 시스템콜을 호출해 MLFQ 스케줄링의 상태로 바꿔준다.

```
else if (c→monopolized == 1) {
  struct proc *monop;
  for (int i = 0; i < NPROC; i++) {
    monop = moq.proc_list[i];
    if (monop == 0 || monop→state ≠ RUNNABLE)
      continue;
    if (!c→monopolized)
      break;
    // 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 = monop;
    switchuvm(monop);
    monop→state = RUNNING;

    swtch(&(c→scheduler), monop→context);
    switchkvm();

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

마지막으로 추가한 코드가 있는데, 유저프로세스가 잘못된 행동을 하여 강제 종료를 시키는 경우 trap.c에서 해당 로직을 처리한다. 이때 trap 함수의 default 부분이 이에 해당하는데, 강제로 종료 시켰다는 메시지를 출력하기 위해 해당 부분에 cprintf로 해당 문구를 출력하도록 했다.

```
//PAGEBREAK: 13
default:
  if(myproc() == 0 || (tf->cs&3) == 0){
    // In kernel, it must be our mistake.
    cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
            tf->trapno, cpuid(), tf->eip, rcr2());
    panic("trap");
  }
  // In user space, assume process misbehaved.
  cprintf("User process misbehaved! Start killing this process.\n");
  cprintf("pid %d %s: trap %d err %d on cpu %d "
          "eip 0x%x addr 0x%x--kill proc\n",
          myproc()->pid, myproc()->name, tf->trapno,
          tf->err, cpuid(), tf->eip, rcr2());
  myproc()->killed = 1;
}
```

이것이 추가한 코드이다.

```
// In user space, assume process misbehaved.
cprintf("User process misbehaved! Start killing this process.\n");
```

3. Result

우선 해당 코드를 테스트하기 위해 조교님께서 올려주신 mlfq_test.c를 등록해야 했다. 이를 위해 Makefile의 UPROGS에 _mlfq_test를 추가해주었고, EXTRA에도 mlfq_test.c를 추가해주었다. 이후 컴파일 및 실행을 위해 cpu의 개수를 하나로 가정하고 make CPUS=1, make fs.img, ./bootxv6 명령어를 차례로 실행해 컴파일 및 실행을 했다. 이렇게 테스트한 결과는 다음과 같다

1) Test 1

Test 1에서는 프로세스가 실행되면서 자신에게 주어진 quantum을 전부 쓰게 되면 낮은 레벨의 큐로 재조정되기 때문에, 차례로 하위 큐로의 재조정이 이루어지는 걸 확인할 수 있었다.

```
$ mlfq_test
MLFQ test start
[Test 1] default
Process 4
L0: 21761
L1: 0
L2: 49932
L3: 28307
MoQ: 0
Process 6
L0: 21667
L1: 0
L2: 51067
L3: 27266
MoQ: 0
Process 8
L0: 21719
L1: 0
L2: 56937
L3: 21344
MoQ: 0
Process 10
L0: 20910
L1: 0
L2: 57032
L3: 22058
MoQ: 0
Process 5
L0: 21928
L1: 41615
L2: 0
L3: 36457
MoQ: 0
Process 11
L0: 23015
L1: 40477
L2: 0
L3: 36508
MoQ: 0
Process 9
L0: 25436
L1: 45365
L2: 0
L3: 29199
MoQ: 0
Process 7
L0: 24099
L1: 45534
L2: 0
L3: 30367
MoQ: 0
[Test 1] finished
```

2) Test 2

Test 2에서는 pid가 큰 프로세스에게 더 높은 우선순위를 부여한다. 물론 전체적인 시간 사용량이 결국 비슷해져 아닌 경우도 존재하지만, pid가 높은 프로세스가 조금 더 먼저 끝나는 경향을 확인할 수 있었다.

```
[Test 2] priorities
Process 19
L0: 16771
L1: 30007
L2: 0
Process 18
L0: 16630
L1: 0
L2:L3: 53222
MoQ: 0
 42168
L3: 41202
MoQ: 0
Process 17
L0: 20351
L1: 36087
L2: 0
L3: 43562
MoQ: 0
Process 16
L0: 17691
L1: 0
L2: 44475
L3: 37834
MoQ: 0
Process 14
L0: 25287
L1: 0
L2: 58875
L3: 15838
MoQ: 0
Process 15
L0: 23828
L1: 42067
L2: 0
L3: 34105
MoQ: 0
Process 12
L0: 26569
L1: 0
L2: 58933
L3: 14498
MoQ: 0
Process 13
L0: 27396
L1: 48869
L2: 0
L3: 23735
MoQ: 0
[Test 2] finished
```

3) Test 3

Test 3에서는 각 프로세스 루프를 돌 때마다 sleep 시스템 콜을 호출합니다. 제시된 예시와 숫자는 다르지만, pid가 작은 숫자들이 먼저 끝나는 경향을 확인할 수 있었다.

```
[Test 3] sleep
Process 20
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 21
L0: 120
L1: 200
L2: 0
L3: 180
MoQ: 0
Process 22
L0: 120
L1: 0
L2: 280
L3: 100
MoQ: 0
Process 24
L0: 120
L1: 0
L2: 280
L3: 100
MoQ: 0
Process 26
L0: 120
L1: 0
L2: 280
L3: 100
MoQ: 0
PProcess 23
L0: 120
L1: 200
L2: 0
Process 25
L0: 120
L1: 200
L2: 0
L3: 180
MoQ: 0
rocess L3: 180
MoQ: 0
27
L0: 120
L1: 200
L2: 0
L3: 180
MoQ: 0
[Test 3] finished
```

4) Test 4

Test 4는 monopolize 시스템 콜이 호출되는 예시이다. Monopolize 시스템 콜이 호출되면 cpu를 MoQ가 독점해야 하기 때문에 MoQ 프로세스 처리 후 MLFQ가 실행되어야 한다. 하지만, 위의 코드 처리 과정에서 c->monopolized값을 계속해서 확인해주도록 했음에도 MoQ가 곧바로 시작되지 않아서 아이러니하다. Unmonopolize는 정상적으로 이루어지는 것을 확인할 수 있었다.

```
[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
Successfully monopolized code: 1
Process 28
L0: 8264
L1: 0
L2: 15311
L3: 76425
MoQ: 0
Process 30
L0: 10547
L1: 0
L2: 17767
L3: 71686
MoQ: 0
Process 32
L0: 10778
L1: 0
L2: 25037
L3: 64185
MoQ: 0
Process 34
L0: 14294
L1: 0
L2: 33474
L3: 52232
MoQ: 0
Process 35
L0: 0
L1: 0
L2: 0
L3: 0
MoQ: 100000
Process 31
L0: 0
LProcess 29
L0: 0
L1: 0
L2: 0
L3: 0
MoQ: 11: 0
L2: 0
L3: 0
MoQ: 100000
00000
Process 33
L0: 0
L1: 0
L2: 0
L3: 0
MoQ: 100000
[Test 4] finished
```

4. Trouble Shooting

우선 CPU 스케줄링이 이루어지는 과정을 파악하는데 정말 많은 시간이 걸렸다. 실습 시간에 조교님께서 주신 파일들을 하나씩 정독해가면서 과정을 파악하려고 노력했는데, 과정이 쉽지는 않았던 것 같다. 또한 과제를 하면서 셀 수 없이 많은 error를 만났다. 대표적으로 생각 나는 건 moq를 정의하는 과정에서 struct moq *moq;를 사용해 정의했었는데, moq->len 방식으로 moq에 접근하려고 했더니 계속 trap 14 error가 발생해 이 문제를 파악하는 데에 많은 시간이 걸렸던 것 같다. 이후 포인터 대신 struct moq moq;를 사용하니 해결되었다. 추가적으로, mycpu()에 접근해 monopolized 설정을 해주는 과정에서 pushcli(), popcli()를 사용하지 않고 접근 했을 때 trap error가 발생했다. 코드를 하나씩 프린트해가면서 문제점을 파악하느라 많은 시간을 소비했다. 명세에서 요구하는 조건들에 대해 빠짐 없이 구현했다. 그러나, Cpu 구조체의 monopolized 변수를 통한 MLFQ와 MoQ의 판단을 하려고 했는데, 또 그렇게 구현했다고 생각하는데, 3번의 Test 4 결과가 예상한대로 나오지 않아, 조금 당황스럽다.