Operating Systems
- Project02 Wiki–

2017029916
양동해

Ⅰ. FCFS scheduler

1. 디자인

스케줄링은 proc.c 파일 내의 scheduler() 함수를 통해 진행된다. 여기서 기존 스케줄링을 보면, for문을 통해 구조체 ptable 내의 proc 배열을 모두 탐색하여 RUNNABLE인 프로세서를 찾는다. 그 뒤에 선택된 프로세스에게 CPU를 할당하기 위한 코드가 진행되어 있으므로, FCFS 스케줄링을 하기 위해선 이 부분을 새롭게 구현해야 한다.

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

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

  // cprintf("ticks = %d, pid = %d, name = %s\n", ticks, p->pid, p->name);

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

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

[proc.c 파일 안에, scheduler() 함수의 일부]

FCFS의 요구조건을 보면, 먼저 생성된 프로세스가 먼저 스케줄링이 되어야 하고, 그 프로세스가 종료되기 전 까진 switch 되면 안된다. 때문에 먼저 생성된 프로세스를 확인할 수 있는 변수가 필요한데, 여기서 pid의 값이 작을 수록 먼저 생성된 프로세스라는 것을 알 수 있기 때문에, 스케줄링 과정에서 pid를 확인하여 프로세스를 선택해주면 된다.

그리고 두번째 조건은 스케줄링 된 이후 200ticks가 지나면 프로세스를 강제 종료해야하는데, 이는 trap.c 파일을 수정해주어야 한다. 기존의 trap() 함수 중 타이머 interrupt가 동작하는 부분은 다음과 같다.

```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

[trap.c 파일 안에, trap() 함수의 일부]

이 부분에서 현재 프로세스가 200ticks 이상 수행 됐다는 것을 확인하기 위해선 스케줄링 된 시점의 ticks도 proc 구조체에서 갖고 있어야 한다.

마지막으로 실행 중인 프로세스가 SLEEPING 상태가 되면 그 다음으로 생성된 프로세스를 선택해주고, 지금 실행중인 프로세스보다 먼저 생성된 프로세스가 일어나게 되면 먼저 생성된 프로세스를 선택해주어야 한다. 이는 scheduler() 함수를 구현하는 과정에서 포함되는 부분이다.

2. 구현

우선 proc.h 파일내의 proc 구조체에 sTime이라는 변수를 추가해 줬는데, 이것은 프로세스가 스케줄링된 시점을 갖고 있는 변수이다.

```
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 sTime;                  // 프로세스가 스케줄링 된 시점
};
```

[proc.h 파일 안에, proc 구조체]

그 다음에 proc.c 파일내의 scheduler() 함수에 FCFS 스케줄링을 구현했는데, 우선 완성된 모습은 다음과 같다

```
#ifdef FCFS_SCHED
    struct proc *oldP = 0;

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

      // 먼저 만들어진 p를 찾기 위한 코드
      if(oldP == 0) oldP = p; // oldP 비어있다면, oldP에 처음 p 세팅
      else if(p->pid < oldP->pid) oldP = p; // 현재 p가 더 먼저 만들어진 p라면, oldP에 현재 p 세팅
    }

    if(oldP != 0) {
      p = oldP;
      p->sTime = ticks; // 스케줄링 됐을 때 시점 세팅

      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

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

      c->proc = 0;
    }
```

[proc.c 파일 안에, scheduler() 함수 중 FCFS 부분]

일단은 기존의 스케줄링과 구분 짓기 위해, FCFS_SCHED 플래그로 나누어서 구현했다. 여기서 oldP라는 변수를 하나 추가했는데, 이 값은 가장 먼저에 만들어진 프로세스를 담고 있는 포인터 변수이다. 위의 for문에서 먼저 만들어진 프로세스가 oldP에 들어가므로, 그 값이 비어있지 않다면 기존의 스케줄링과 마찬가지로 프로세스를 선택하는 과정을 거친다. 그리고 switch를 하기 전에 proc 구조체에 추가해줬던 sTime의 값에 스케줄링이 된 시점의 ticks 값을 넣어줌으로 써, 후에 trap.c 파일에서 비교하는 값으로 사용될 수 있다.

다음으로 trap.c 파일에서 자원을 양보하는 부분을 수정했는데, 수정된 모습은 다음과 같다.

```
#ifdef FCFS_SCHED
  if(myproc() && myproc()->state == RUNNING && (ticks - myproc()->sTime) >= 200) {
    myproc()->killed = 1;
    cprintf("killed the process %d because it has been over 200 ticks\n", myproc()->pid);
  }
```

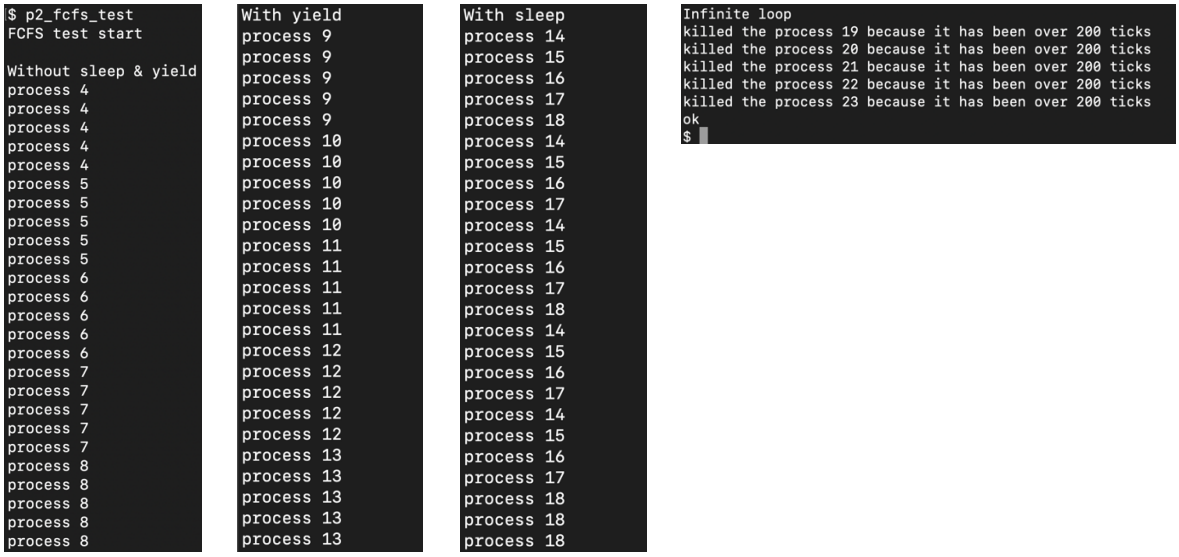
[trap.c 파일 안에, trap() 함수의 일부]

여기도 마찬가지로 FCFS_SCHED 플래그로 구분했으며, 현재의 ticks와 지금 돌고 있는 프로세스가 스케줄링된 시간인 sTime의 차가 200 이상이면, 그 프로세스를 강제 종료한다. 여기서 강제 종료는 proc 구조체에 존재하는 killed라는 변수에 1값을 줌으로써 종료시킬 수 있고, 종료와 관련한 메시지를 출력해 준다.

마지막으로 플래그를 선택하여 make 할 수 있게 관련된 부분을 Makefile안에 추가해 주었고, 스케줄러를 테스트하기 위해 p2_fcfs_test.c 파일도 유저 프로그램으로 추가해 주었다. 자세한 부분은 해당 파일을 통해 확인할 수 있다.

3. 실행 결과

이를 실행하기 위해, make clean, make SCHED_POLICY=FCFS_SCHED, make fs.img를 차례로 수행하고 xv6를 실행시키면 다음과 같은 결과가 나타난다. (권한 설정만 해준다면, xv6의 실행은 실습 시간에 만든 bootxv6.sh의 사용으로도 가능하며, CPU는 1개인 조건에서 실행했다.)



[xv6에서 p2_fcfs_test 실행 결과(왼쪽부터 차례대로)]

첫 번째로 sleep과 yield가 없는 상황에선, 가장 먼저 만들어진 프로세스부터 차례대로 스케줄링 되는 모습을 볼 수 있다.

두 번째로 yield만 있는 상황에선, 자원을 양보한다고 하더라도 스케줄링은 먼저 생성된 프로세스부터 진행되기 때문에 위와 마찬가지 결과를 갖는다.

세 번째로 sleep만 있는 상황에선, 돌아가면서 sleep을 하므로 차례대로 스케줄링이 되는 것처럼 보이지만 중간에 17 다음에 18로 가지 않고 다시 14로 가는 모습을 보면, 14가 18보다 먼저 깨어나 먼저 스케줄링이 되었던 걸 알 수 있다.

마지막으로 infinite loop가 있는 상황에선, 모든 프로세스들이 정상적으로 강제종료 된 모습을 볼 수 있다.

4. 트러블슈팅

우선 이전 project01때와 마찬가지로 존재했던 이슈이지만, 처음에 어떻게 접근해야 할지 막막해서 proc.h 파일 내의 구조체와 proc.c 파일을 차근차근 이해해 보았다. 그리고 scheduler() 함수가 어떤 식으로 동작하는지 확인하고, trap이 발생하는 시점 및 어디에서 이를 처리해야하는지 등을 공부해 보았다.

개발하는 과정에서는 큰 오류는 없었으나, Makefile내에 flag를 어디에 어떻게 추가해야 하는지 조금 헤맸다. 특히 $(SCHED_POLICY) 앞에 -D를 넣지 않아서 make 과정 중 오류가 났었다.

```
CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer $(SCHED_POLICY)
```

[처음에 수정한 Makefile 파일 중 CFLAGS 부분]

Ⅱ. Multilevel Queue

1. 디자인

이전 FCFS과 마찬가지로 proc.c 파일과 trap.c 파일에다가, Multilevel Queue 기능을 구현하면 된다.

Multilevel Queue는 2개의 큐로 이루어져 있는데, pid가 짝수인 프로세스는 RR 스케줄링을 따르며 pid가 홀수인 프로세스는 FCFS를 따른다. 그리고 RR 큐가 FCFS 큐보다 우선순위가 높기 때문에, pid가 짝수인 프로세스가 RUNNABLE 상태라면 우선적으로 실행해주어야 한다.

2. 구현

proc.c 파일 내의 scheduler() 함수를 수정하여 구현했는데, "FCFS scheduler"와 마찬가지로 MULTILEVEL_SCHED 라는 플래그로 구분하여 구현했다.

```
#elif MULTILEVEL_SCHED
    int rr = 0;
    // pid가 짝수인 프로세스가 존재하는지 찾고, 존재한다면 RR 스케줄링
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      if(p->pid % 2 == 0) {
        rr = 1;
        break;
      }
    }
```

[proc.c 파일 안에, scheduler() 함수 중 MULTILEVEL 부분 ①]

위 사진에서 rr이라는 변수는 RR과 FCFS 중 어떤 스케줄링을 선택할지 정해주는 변수이다. rr의 값이 1이면 RR 스케줄링을 하고, RR이 0이면 FCFS 스케줄링을 한다. 때문에 for문에서 pid가 짝수인 RUNNABLE 상태의 프로세스가 존재한다면 rr을 1로 세팅해준다.

```
    if(rr == 1) { // RR 스케줄링
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
          continue;
        if(p->pid % 2 == 1) // pid가 짝수인 프로세스가 있을때, 홀수인 프로세스는 스케줄링 되면 안됨
          continue;

        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;

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

        c->proc = 0;
      }
```

[proc.c 파일 안에, scheduler() 함수 중 MULTILEVEL 부분 ②]

rr이 1이면 RR 스케줄링을 한다. 아래 for문의 내용은 기존 xv6의 스케줄러 내용과 동일하다. 왜냐하면 기존 xv6의 스케줄링은 RR 정책을 따르기 때문이다. 그렇지만 여기서 pid가 홀수인 프로세스들은 우선순위가 없기 때문에 짝수인 것들만 실행 해주기 위해 약간의 조건을 추가했다.

나머지 else 부분은 rr이 0이기 때문에 FCFS 스케줄링을 진행하면 되고, 이 부분은 앞서 구현했던 "FCFS Scheduler"와 완전히 동일하다. 자세한 부분은 해당 파일을 통해 확인할 수 있다.

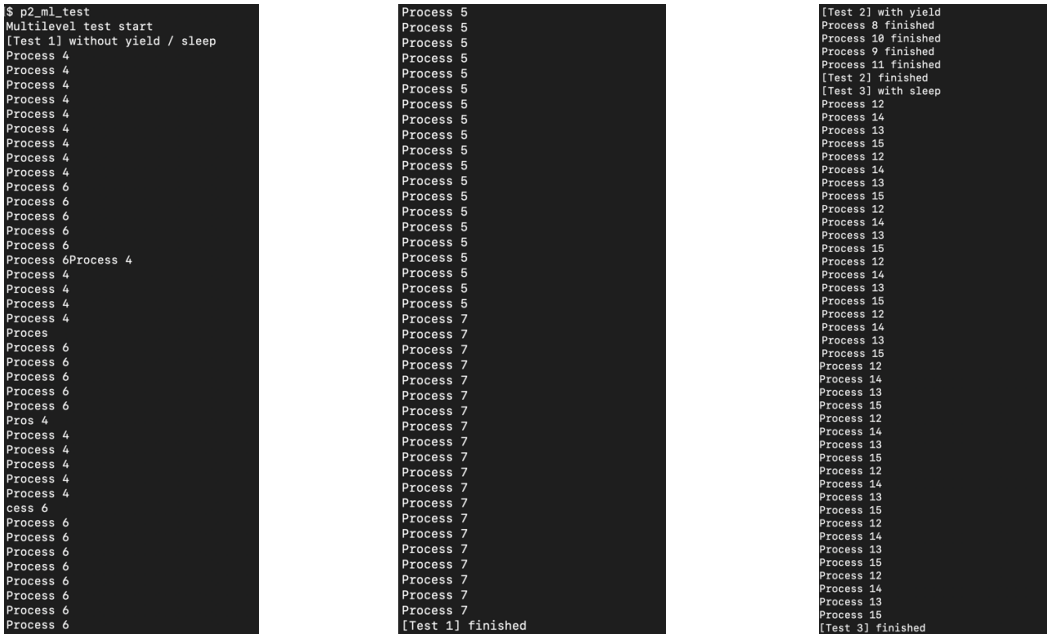
```
#elif MULTILEVEL_SCHED
  if(myproc() && myproc()->state == RUNNING && (myproc()->pid % 2) == 0) { // pid 짝수면 RR
    if(tf->trapno == T_IRQ0+IRQ_TIMER)
      yield();
  }
  // pid 홀수면 FCFS
```

[trap.c 파일 안에, trap() 함수의 일부]

trap.c 파일 내부는 다음과 같이 pid가 짝수인 부분과 홀수인 부분을 나누어 보았다. 짝수인 부분은 기존 xv6와 똑같은 기능을 수행하며, 홀수인 부분은 FCFS이기 때문에 기존에 돌던 프로세스가 계속 처리되어야 하므로 따로 작성하지 않았다.

3. 실행 결과

이를 실행하기 위해, make clean, make SCHED_POLICY=MULTILEVEL_SCHED, make fs.img를 차례로 수행하고 xv6를 실행시키면 다음과 같은 결과가 나타난다. (권한 설정만 해준다면, xv6의 실행은 실습시간에 만든 bootxv6.sh의 사용으로도 가능하며, CPU는 1개인 조건에서 실행했다.)



[xv6에서 p2_ml_test 실행 결과(왼쪽부터 차례대로)]

첫 번째로 yield와 sleep이 없는 상황에선, 짝수 pid를 갖고 있는 4와 6이 번갈아 가며 RR 스케줄링이 되는 것을 볼 수 있고, 홀수 pid를 갖는 5와 7은 짝수 pid가 모두 끝난 뒤 FCFS로 스케줄링 되는 것을 볼 수 있다.

두 번째로 yield가 있는 상황에선, 계속적인 자원 양보가 존재하더라도 짝수 pid가 존재하면 홀수 pid보다 우선 실행되기 때문에 8과 10이 먼저 끝나고 9와 11이 그 뒤에 끝나는 것을 볼 수 있다.

마지막으로 sleep이 있는 상황에선, 돌아가면서 sleep을 하는데 짝수 pid가 모두 SLEEPING 상태에 빠지므로 그 다음 프로세스인 13이 스케줄링 되어 실행되는 모습을 볼 수 있다. 하지만 홀수 pid를 갖는 프로세스들도 차례대로 SLEEPING 상태가 되므로, 이후에 깨어난 12, 14 프로세스들이 스케줄링 되는 것을 볼 수 있다.

4. 트러블슈팅

구현에 있어서는 큰 오류 없이 문제를 해결할 수 있었지만, 이 부분 또한 "FCFS Scheduler"와 마찬가지로 스케줄러에 대한 이해와 공부에 많은 시간을 투자했다.

처음에는 짝수와 홀수 pid를 갖는 프로세스들을 구조체 내에서 구분하거나 proc.c에서 새로운 구조체를 만들어볼까 했지만, 너무 복잡한 것 같아 생각을 다시 해보았고 고민 끝에 위의 방식처럼 방향을 전환하여 문제를 해결했다.

Ⅲ. MLFQ

1. 디자인

문제에서 요구하는 것을 정리해보면 다음과 같다.

1. L0와 L1 두개의 큐로 이루어져 있고, L0의 우선순위가 L1보다 더 높다. (L0에 있는 프로세스가 모두 끝난 뒤에, L1에 있는 프로세스가 스케줄링 된다.)
2. L0는 RR 정책을 따르며 L1은 priority 스케줄링을 따른다. Priority는 0~10의 값을 가지며 숫자가 더 클수록 더 높은 우선순위를 나타낸다. 같은 우선순위 내에선 FCFS을 따른다.
3. L0는 4 ticks의 time quantum을 가지며, L1은 8 ticks의 time quantum을 가진다.
4. 매 200 ticks가 지나면 모든 프로세스들의 큐 level과 priority를 0으로 초기화 해주는 priority boosting이 존재한다.
5. monopolize를 호출한 프로세스는 password가 맞을 시 프로세서를 독점할 수 있다.

전체적인 흐름은 이렇고 이 외에도 자잘하게 신경써야 할 부분들이 존재한다. 이 부분들에 대해서는 구현 부분에서 자세히 설명할 예정이다.

2. 구현

우선 큐의 레벨과 우선순위 값을 가질 변수를 proc.h 파일내의 proc 구조체에 선언해 준다. 또 이 프로세스가 자원을 독점한 상태인지 확인하기 위한 변수로 monopoly도 추가해 주었으며, time quantum을 확인하기 위해 프로세스가 running 한 시간을 담을 변수도 추가해줬다. 완성된 구조체의 모습은 아래 사진과 같다.

```
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 sTime;                  // 프로세스가 스케줄링 된 시점 (FCFS)
  uint rTime;                  // 프로세스가 running 한 시간
  int qLevel;                  // MLFQ에서 큐의 레벨 (0, 1)
  int priority;                // MLFQ에서 우선순위 (0~10)
  int monopoly;                // MLFQ에서 monopolize 확인 (0: monopolize X / 1: monopolize O)
};
```

[proc.h 파일 안에, proc 구조체]

다음으로 proc 구조체에 선언해 주었던 값들을 초기화 해주기 위해, proc.c 파일 내의 allocproc() 함수 안에 초기화를 진행해주었다. 맨 처음 프로세스가 생성되면 L0 큐에 할당되어야 하므로 qLevel을 0으로 초기화 해주었고, 문제 조건에 의해서 처음 생성된 priority값은 0으로 초기화 해주었다.

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  p->sTime = 0;
  p->rTime = 0;
  p->qLevel = 0;    // 맨 처음 프로세스가 생성되면 L0 큐에 할당
  p->priority = 0;  // 맨 처음 프로세스가 생성되면 priority는 0값 가짐
  p->monopoly = 0;
```

[proc.c 파일 안에, allocproc() 함수의 일부]

본격적으로 scheduler() 함수를 수정하여 MLFQ를 구현했는데, "FCFS scheduler"와 "Multilevel Queue"와 마찬가지로 MLFQ_SCHED 라는 플래그로 구분하여 구현했다.

```
#elif MLFQ_SCHED
    int monopoly = 0;
    int L0 = 0;
    // 독점 중인 프로세스가 있는지, L0 큐에 존재하는 프로세스가 있는지 확인
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      if(p->monopoly == 1) monopoly = 1;
      if(p->qLevel == 0) L0 = 1;
    }
```

[proc.c 파일 안에, scheduler() 함수 중 MLFQ 부분 ①]

함수 내의 monopoly 변수는 독점 중인 프로세스가 있는지 확인하는 변수이고, L0는 L0 큐에 프로세스가 존재하는지 확인하는 변수이다.

```
  if(monopoly == 1) { // 독점 중인 프로세스 존재
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      if(p->monopoly == 0) // monopoly 0인 프로세스는 스케줄링 되면 안됨
        continue;

      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

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

      c->proc = 0;
      break;
    }
```

[proc.c 파일 안에, scheduler() 함수 중 MLFQ 부분 ②]

monopoly의 값이 1이라는 것은 현재 독점 중인 프로세스가 존재한다는 것이고, 그 프로세스를 ptable에서 찾아 실행하는 부분이다.

```
  } else if(L0 == 1) { // L0 큐에 프로세스 존재
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      if(p->qLevel == 1) // L1 큐에 있는 프로세스는 스케줄링 되면 안됨
        continue;

      p->rTime = 0; // running time 0으로 셋팅 (4 ticks 확인하기 위해)

      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

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

      c->proc = 0;
    }
```

[proc.c 파일 안에, scheduler() 함수 중 MLFQ 부분 ③]

이 부분은 monopoly가 0인 부분이므로 현재 독점 중인 프로세스가 없기 때문에, 원래 루틴대로 L0 큐의 프로세스들을 RR 스케줄링에 맞춰 실행하는 부분이다.

```
  } else { // L1 큐에 있는 프로세스 처리
    struct proc *highP = 0;

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

      // 더 높은 priority 갖는 p를 찾기 위한 코드
      if(highP == 0) highP = p; // highP 비어있다면, highP에 처음 p 세팅
      else if(p->priority > highP->priority) highP = p; // 현재 p가 더 높은 priority를 가진 p라면, highP에 현재 p 세팅
      else if(p->priority == highP->priority) { // 현재 p와 highP가 같은 priority를 갖는다면
        if(p->pid < highP->pid) highP = p; // 현재 p가 더 먼저 만들어진 p라면
      }
    }

    if(highP != 0) {
      p = highP;
      p->rTime = 0; // running time 0으로 셋팅 (8 ticks 확인하기 위해)

      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

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

      c->proc = 0;
    }
  }
```

[proc.c 파일 안에, scheduler() 함수 중 MLFQ 부분 ④]

여기는 독점 중인 프로세스도, L0에 존재하는 프로세스도 없는 부분으로써, L1 큐에 존재하는 프로세스들이 스케줄링 되는 부분이다. 같은 우선순위끼리의 프로세스들은 FCFS로 스케줄링 되기 때문에, 위에서 구현했던 "FCFS Scheduler"와 "Multilevel Queue"의 FCFS 스케줄링 부분과 거의 유사한 구조를 가지고 있다. 대신 여기선 더 높은 priority를 갖는 프로세스를 찾는 부분이 존재한다.

이 다음으로 구현한 부분은 trap.c 파일이다. 앞선 사진에서 봤듯이 L0와 L1 큐의 스케줄링을 하는 부분에선 rTime을 0으로 초기화는 코드가 있는데, 이는 이 프로세스가 돌아간 시간을 확인하기 위해 매 스케줄링 때마다 0으로 초기화 해준 것이다. 그리고 trap.c 파일에서 매 TIMER 마다 ticks가 증가할 때, rTime 값 또한 같이 증가시켜 주어야 돌아간 시간을 확인할 수 있다. 이와 관련된 코드는 아래 사진에서 볼 수 있다.

```
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    if(myproc() && myproc()->state == RUNNING) myproc()->rTime++;
    wakeup(&ticks);
    release(&tickslock);
  }
  lapiceoi();
  break;
```

[trap.c 파일 안에, trap() 함수의 일부]

그리고 자원을 양보하는 부분은 다음과 같이 수정했다.

```
#elif MLFQ_SCHED
  if(myproc() && myproc()->state == RUNNING && myproc()->monopoly != 1) { // monopoly가 1인 것은 무시하고 진행
    if(myproc()->qLevel == 0 && myproc()->rTime >= 4) { // L0 큐에 있던 프로세스 4 ticks 넘었을 때
      myproc()->qLevel = 1;
      yield();
    } else if(myproc()->qLevel == 1 && myproc()->rTime >= 8) { // L1 큐에 있던 프로세스 8 ticks 넘었을 때
      if(myproc()->priority > 0) myproc()->priority--;
      yield();
    }
  }

  if(ticks%200 == 0) priorityBoosting(); // 처음 ticks은 0부터 시작하므로 200의 나머지로 계산하면, 매 200 ticks 마다 boost 가능
```

[trap.c 파일 안에, trap() 함수의 일부]

독점 중인 프로세스가 존재한다면 이를 무시하고 넘어가며, 독점 중인 프로세스가 없을 때는 그 프로세스가 현재 L0 큐에 있는지 L1 큐에 있는지 확인하여 각각의 루틴을 진행한다. 현재 프로세스가 L0 큐에 있을 경우, 그 running time이 4 ticks가 넘어가면 L1 큐로 내려주고 자원을 양보한다. 현재 프로세스가 L1 큐에 있을 경우, priority를 1 낮추고 자원을 양보한다. 맨 마지막으로 매번 200 ticks가 지나갈 때마다 priority boosting을 해주어야 하므로 이와 관련된 함수를 호출한다.

다음으로 trap.c 파일에서 호출한 priorityBoosting() 함수이다. 이 함수는 proc.c 파일내의 scheduler() 함수 아래 부분에 구현했다. 이 함수는 모든 프로세스들의 큐 레벨과 priority를 초기화 해주는 작업을 진행하며, 자세한 코드는 아래 사진으로 첨부했다.

```
// MLFQ에서 200 ticks 마다 boost 해주는 함수 (ptable 변수를 사용해야하기 때문에 proc.c에 구현)
void priorityBoosting(void) {
  struct proc *p;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    p->qLevel = 0;
    p->priority = 0;
  }

  release(&ptable.lock);
}
```

[proc.c 파일 안에, priorityBoosting() 함수]

마지막으로 유저 프로그램에서 사용되는 시스템 콜 함수 getlev()와 setpriority(), monopolize() 함수들을 구현했다. 처음에는 파일을 따로 만들어서 구현했으나 여러가지 불편함이 존재하여 proc.c 파일 안에 구현했다.

```c
// MLFQ에서 사용되는 시스템 콜 함수 (ptable 변수를 사용해야하기 때문에 proc.c에 구현)
int getlev(void) {
  return myproc()->qLevel;
}
int setpriority(int pid, int priority) {
  if(priority<0 && priority>10) return -2;

  struct proc *p;
  struct proc *thisP = 0;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid) {
      thisP = p;
      break;
    }
  }

  int returnValue = -1;
  if(thisP != 0) { // 존재하는 프로세스인지 체크
    if(thisP->pid >= myproc()->pid) { // 자신 & 자식 프로세스인지 체크
      thisP->priority = priority;
      returnValue = 0;
    }
  }

  release(&ptable.lock);

  return returnValue;
}
```

[proc.c 파일 안에, getlev() 함수와 setpriority() 함수]

getlev() 함수는 단순하게 qLevel을 반환하는 함수이다. setpriority() 함수는 priority가 0에서 10 사이의 값이 아니라면 -2를 리턴하고, 맞다면 인자로 받은 pid에 해당하는 프로세스를 찾는다. 찾은 프로세스가 존재하지 않는다면 -1을 리턴하며, 찾았다면 priority를 변경하고 0을 리턴한다.

```c
void monopolize(int password) {
  acquire(&ptable.lock);

  if(password == 2017029916) { // 비밀번호가 맞았을 때
    if(myproc()->monopoly == 0) myproc()->monopoly = 1;
    else {
      myproc()->qLevel = 0;
      myproc()->priority = 0;
      myproc()->monopoly = 0;
    }
  } else { // 비밀번호가 틀렸을 때
    myproc()->killed = 1;
    cprintf("killed the process %d because the password is incorrect\n", myproc()->pid);
  }

  release(&ptable.lock);
}
```

[proc.c 파일 안에, monopolize() 함수]

monopolize() 함수는 패스워드 값이 틀리면 프로세스를 강제종료하고, 맞다면 현재 독점 중인지를 확인하여 그에 맞게 알맞는 루틴을 처리해준다.

마지막으로, 위에서 구현했던 시스템 콜 함수들(getlev(), setpriority(), monopolize())과 관련된 여러가지 처리들을 진행해 주었고, 테스트 유저 프로그램(p2_mlfq_test)에 관련된 코드도 마저 작성해 주었다. 자세한 코드는 defs.h, sysproc.c, syscall.h, syscall.c, user.h, usys.S, Makefile 파일들을 통해 확인할 수 있다.

3. 실행 결과

이를 실행하기 위해, make clean, make SCHED_POLICY=MLFQ_SCHED, make fs.img를 차례로 수행하고 xv6를 실행시키면 다음과 같은 결과가 나타난다. (권한 설정만 해준다면, xv6의 실행은 실습시간에 만든 bootxv6.sh의 사용으로도 가능하며, CPU는 1개인 조건에서 실행했다.)

```
$ p2_mlfq_test
MLFQ test start

Focused priority
process 8: L0=1448, L1=18552
process 7: L0=2868, L1=17132
process 6: L0=2897, L1=17103
process 5: L0=3616, L1=16384
process 4: L0=4333, L1=15667
```

```
Without priority manipulation
process 9: L0=4559, L1=45441
process 10: L0=4682, L1=45318
process 11: L0=7019, L1=42981
process 12: L0=8357, L1=41643
process 13: L0=8479, L1=41521
```

```
With yield
process 14: L0=10000, L1=0
process 15: L0=10000, L1=0
process 16: L0=10000, L1=0
process 17: L0=10000, L1=0
process 18: L0=10000, L1=0
```

```
Monopolize
process 23: L0=25000, L1=0
process 19: L0=4047, L1=20953
process 20: L0=4265, L1=20735
process 21: L0=3565, L1=21435
process 22: L0=4125, L1=20875
```

[xv6에서 p2_mlfq_test 실행 결과(왼쪽부터 차례대로)]

첫 번째로 priority가 있는 상황에선, 더 나중에 만들어진 프로세스가 pid가 크기 때문에 더 높은 우선순위를 갖는다. 때문에 가장 나중에 만들어진 프로세스부터 차례대로 실행이 끝나는 모습을 볼 수 있다. 뒤에 있는 프로세스들이 상대적으로 L0 큐에 있던 시간이 더 큰 이유는 중간에 200 ticks가 지나 priority boost가 일어난 것을 알 수 있다.

두 번째로 priority가 없는 상황에선, 차례대로 실행이 끝난 것을 알 수 있다. Priority가 없는 상황에선 모두들 같은 우선순위를 가지므로 L1에서도 FCFS 스케줄링이 되어 차례대로 진행된다. 마찬가지로 뒤에 있는 프로세스들이 상대적으로 L0 큐에 있던 시간이 더 큰 이유는 중간에 200 ticks가 지나 priority boost가 일어난 것을 알 수 있다.

세 번째로 yield가 있는 상황에선, 4 ticks가 넘기 전에 yield를 하므로 계속 L0 큐에서만 스케줄링 되는 것을 알 수 있다.

마지막으로 독점 프로세스가 있는 상황에선, 맨 마지막 프로세스가 monopolize를 실행했기 때문에 23 프로세스가 L0에서 먼저 실행이 끝나고 나머지 프로세스들이 차례대로 진행된 것을 알 수 있다.

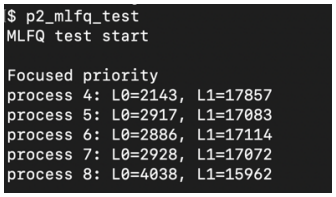
4. 트러블슈팅

우선 매번 존재하는 이슈인 프로그램 이해에 많은 시간을 투자했다. 특히 문제에서 요구하는 조건이 너무 많아 어느 것을 우선적으로 구현해야 할지 많은 시행착오가 있었다. 처음에는 "Multilevel Queue"에서 구현했던 코드의 틀을 가지고 구현했으나 monopolize를 생각하지 못해 중간에 코드를 한번 갈아 엎었다.

그리고 trap.c 파일 내의 priorityBoosting()을 호출하는 부분에서 매번 200 ticks가 지나간 것을 어떻게 해결할지 고민했었다. 맨 처음에는 trap.c 파일 안에 전역변수로 이 시간 값을 가질 변수를 추가해주는 것으로 구현했는데, 중간에 나머지 연산자를 이용하면 쉽게 알아낼 수 있다는 것을 깨닫고 현재 코드와 같이 수정했다.

또한 자잘한 코드 오타가 존재했다. 시스템 콜 함수에서 인자를 처리하는 부분과 trap.c 파일 내의 작은 실수가 존재했는데, 이 부분은 매우 사소한 실수라 따로 에러 사진을 캡쳐 하진 않았다.

프로그램을 처음 완성하고 테스트파일을 실행했을 때, 다음과 같은 결과가 나왔다.



[처음에 작성한 코드로 xv6에서 p2_fcfs_test를 실행했을 때 결과]

우선순위가 적용되었다면 8, 7, 6, 5, 4 순으로 프로세스가 끝나야 하지만 그렇지 않았기 때문에 이와 관련된 함수들을 다시 검토해 보았다. 그리고 proc.c 파일 내의 setpriority() 함수를 확인하여, 자신의 프로세스인 경우도 priority 값을 반영할 수 있게 코드를 수정해서 위의 "3. 실행결과"와 같은 결과를 만들게 되었다.