운영체제 Project 2 Wiki

정상윤
2016025032

1. FCFS Scheduler 구현

• 디자인

FCFS 같은 경우 xv6의 기본 스케줄러인 Round Robin과 매우 비슷하기 때문에 구현하는데 큰 어려움을 겪지는 않았습니다. 실제로 스케줄러 코드도 매우 유사합니다. 다만 주의했어야 하는 부분은 프로세스가 SLEEPING 상태가 되지 않는 이상 종료되기 전까지는 switch-out 되지 않도록 break문을 넣어줌으로써 프로세스가 RUNNING 상태에서 빠져나오면 다시 ptable을 조회함으로써 다시 가장 작은 pid를 가진 프로세스를 스케줄러 대상으로 선택하도록 했습니다.
또한, 프로세스가 스케줄링 된 이후 200 ticks가 지나고도 종료되거나 SLEEPING 상태가 되지 않으면 종료되는 것을 구현하기 위해서 프로세스 구조체 (Proc)에 스케줄링 된 시간을 저장하는 Int 변수를 추가해주었습니다. (srtime) 그래서 trap.c에서 인터럽트 이벤트 처리 코드를 넣어서 현재 ticks에서 srtime을 뺐을 때 200이 넘으면 kill()함수를 호출하도록 추가해줬습니다.

• 구현

```
37 // Per-process state
38 struct proc {
39   uint sz;                     // Size of process memory (bytes)
40   pde_t* pgdir;                // Page table
41   char *kstack;                // Bottom of kernel stack for this process
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
53   uint srtime;           // tick when process started running by scheduler
54
55   // for MLFQ
56   // priority: process priority, higher -> bigger
57   // level: queue level (for MLQ, too)
58   // ismono: is this process monopolizing? 0:no 1:yes
59   // qtime: time quantum of this process
60   int priority;
61   int level;
62   int ismono;
63   int timeq;
64 };
```

스케줄러에게 선택된 시간을 저장하기 위한 srtime

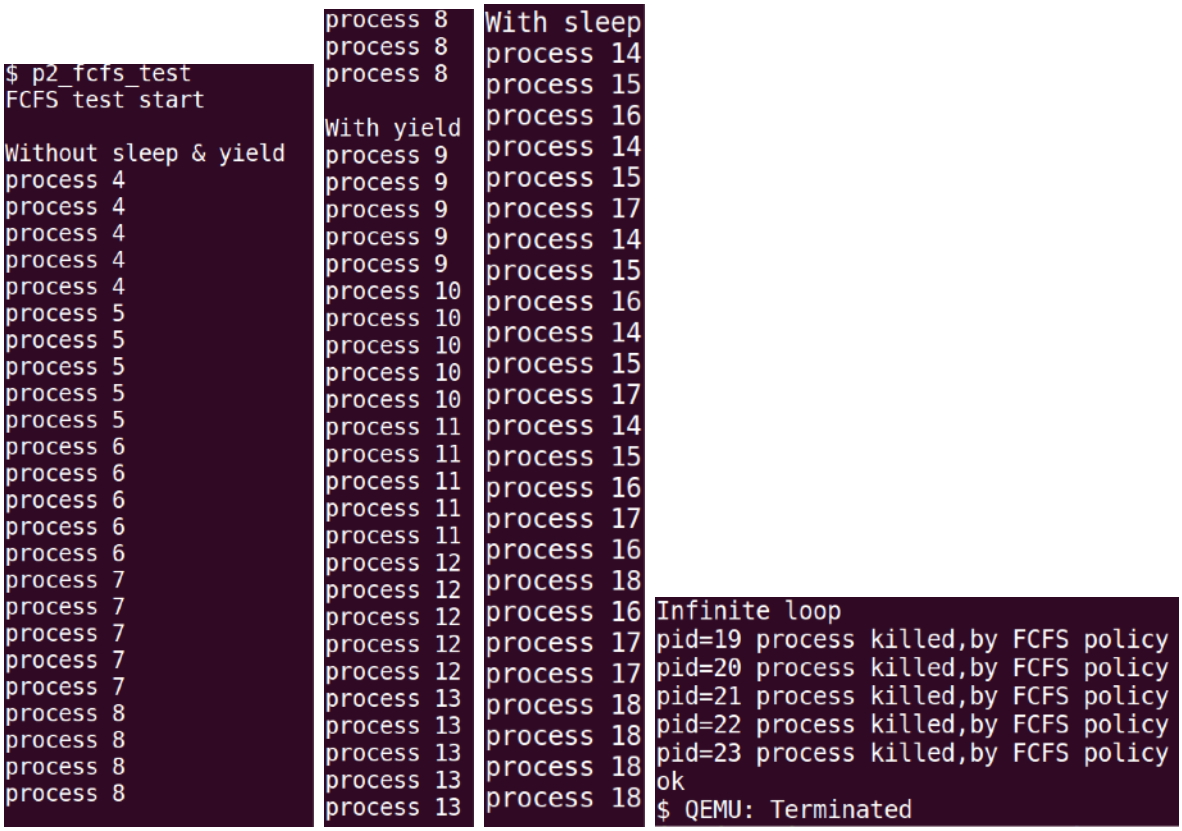
```
378 #ifdef FCFS_SCHED
379 void
380 scheduler(void)
381 {
382     struct cpu *c = mycpu();
383     struct proc *p;
384     c->proc = 0;
385
386     for(;;){
387         sti();
388         //loop throguth ptable
389         acquire(&ptable.lock);
390         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
391             if(p->state != RUNNABLE)
392                 continue;
393
394             p->srtime = ticks; // started running time
395             c->proc = p;
396             switchuvm(p);
397             p->state = RUNNING;
398
399             swtch(&(c->scheduler), p->context);
400             switchkvm();
401
402             c->proc = 0;
403             break; //to prevent switch-out unless it sleeps
404         }
405         release(&ptable.lock);
406     }
407 }
```

switch-out 되지 않도록 추가된 break문

```
124 #elif FCFS_SCHED
125   //200 ticks and still running,then kill
126   if(myproc() && myproc()->state == RUNNING &&
127      (ticks - myproc()->srtime) >= 200){
128     cprintf("pid=%d process killed,by FCFS policy\n", myproc()->pid);
129     kill(myproc()->pid);
130   }
```

trap.c 모습. kill되면 메시지를 출력해준다.

• 실행결과



pid가 작은 순서대로 잘 출력되는 것을 확인할 수 있습니다. 단, sleep의 경우 순서가 섞이긴 하지만 결국 뒤로 갈수록 pid가 큰 프로세스들이 남는 것을 확인할 수 있습니다. 또한, 무한루프가 발생했을 시 kill되는 것도 확인할 수 있습니다.

• 트러블슈팅

큰 어려움은 없었으나, 초반에 프로세스가 스케줄링되는 흐름을 이해해나가는데 시간이 많이 걸렸습니다. 또한 trap.c 파일에서 이벤트 처리 코드 또한 이해하는데 시간을 썼습니다. 그리고 break문을 추가해줘야 한다는 것을 아는데 조금 시간이 걸렸습니다.

2. 2. Multilevel Queue Scheduler 구현

• 디자인

Multilevel 같은 경우, 프로세스 구조체에 level 변수를 추가하여 0값이면 첫 번째 큐, 1이면 두 번째 큐인 것으로 생각해서 구현했습니다. (MLFQ 또한 그렇죠) Pid가 짝수인 프로세스들은 우선순위가 높기 때문에 ptable을 조회하면서 짝수 프로세스들은 큰 조건 확인 없이 바로 스케줄링하도록 코드를 짰습니다. 기본 RR코드를 그대로 넣어줬습니다. 타임 인터럽트 코드 또한 마찬가지입니다.
홀수 프로세스인 경우, ptable에 짝수 프로세스가 없는지 확인해야하기 때문에 ptable을 조회하는 코드를 다시 넣어줬습니다. 그를 위해 임시 프로세스 변수 t 를 넣었습니다. 두번째 큐는 FCFS이기 때문에 가장 작은 pid를 먼저 스케줄링해야합니다.. 그래서 iseven, minpid 변수를 추가해줬습니다. 조회를 마치면 내가 해당 프로세스가 가장 작은 pid고 짝수 프로세스가 없다는 것이 확인되면 FCFS 스케줄링에 들어갑니다. trap.c 에서 인터럽트 처리 코드도 FCFS의 것을 그대로 가져와 추가해줬습니다.

• 구현

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
        continue;
    //even pid, RR, level = 0, just run it
    if((p->pid) % 2 == 0){
        p->level = 0;
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;
        swtch(&(c->scheduler), p->context);
        switchkvm();
        c->proc = 0;
    }
```

짝수 프로세스들은 조회되는 즉시 스케줄링된다

```
//odd pid, FCFS, level = 1, should check
//is there no even pid?
//am lowest pid?
else if((p->pid) % 2 == 1){
    struct proc *t; //temp proc for searching
    //assume this process pid is min
    int minpid = p->pid;
    int iseven = 0; //is there even pid?
    for(t = ptable.proc; t < &ptable.proc[NPROC]; t++){
        if(t->state != RUNNABLE)
            continue;
        //there is smaller pid
        if(t->pid < minpid)
            minpid = t->pid;
        //there is even pid
        if(t->pid % 2 == 0)
            iseven = 1;
    }
    //if this p's pid is lowest and no even pid
    //start fcfs scheduling
    if(p->pid == minpid && iseven == 0){
        p->srtime = ticks;
        p->level = 1;
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;
        swtch(&(c->scheduler), p->context);
        switchkvm();
        c->proc = 0;
        break;
```

홀수 프로세스 코드. 조건을 확인해야할 게 많다.

```
#elif MULTILEVEL_SCHED
  //if myproc()'s level = 0 : RR, default time interrupt
  if(myproc() && myproc()->state == RUNNING &&
     myproc()->level==0  && tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();

  //if myproc()'s level = 1 : FCFS, 200ticks kill
  if(myproc() && myproc()->state == RUNNING &&
     myproc()->level==1  &&
     (ticks - myproc()->srtime) >= 200){
    //cprintf("pid=%d killed due to FCFS policy\n",myproc()->pid);
    kill(myproc()->pid);
  }
```

MULTILEVEL 스케줄링 시 trap.c 코드

• 실행결과

```
$ p2 ml test
Multilevel test start
[Test 1] without yield / sleep
Process 4
Process 4
Process 4
Process 4
Process 4
Process 6
Process 6
Process 6
Process 6
Process 4
Process 4
Process 4
Process 4
Process 4
Process 4
Process 6
Process 6
PrProcess 4
Process 4
Process 4ocess 6
Process 6
Process 6
Process 6
```

```
PrProcess 4
Process 4
Process 4ocess 6
Process 6
Process 6
Process 6

Process 4
Process 4
Process 4
Process 4
Process 4
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
```

```
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
```

```
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
[Test 1] finished
[Test 2] with yield
Process 8 finished
Process 10 finished
Process 9 finished
Process 11 finished
[Test 2] finished
[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
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
$ QEMU: Terminated
```

짝수가 우선적으로 스케줄링되며, RR이기 때문에 뒤죽박죽이다. 홀수들은 FCFS로 잘 스케줄링된다. yield에서도 짝수가 남아있기 때문에 홀수가 스케줄링되지 않는 모습도 확인 가능하다. SLEEPING 상태가 되면 무시하는 것도 확인 가능합니다.

• 트러블슈팅

RR과 FCFS는 구현이 돼있는 상태였기 때문에 조건문을 신경을 많이 쓰는 것이 가장 주된 일이였던 것 같습니다. 짝수 프로세스가 yield 하더라도 홀수 프로세스가 스케줄링 되지 않도록 하는 것이 가장 중요했던 것 같습니다.

3. MLFQ Scheduler 구현

• 디자인

MLFQ 같은 경우, 프로세스 구조체에 총 4개의 변수가 추가됩니다. 우선순위 priority, 큐 레벨 level, 이 프로세스가 CPU를 독점하고 있는지 저장하는 ismono, 그리고 Time quantum timeq를 각각 추가해줬습니다.
MLFQ 같은 경우 trap.c 코딩이 가장 중요했던 것 같습니다. 일단 현재 프로세스가 RUNNING 이면 timeq를 1씩 감소시켜줬으며, L0에서 timeq가 0 이하가 될 시 모든 시간을 쓴 것이기 때문에 L0에서 L1으로 넘어가도록, 즉 level을 1로 바꿔주고 timeq 또한 8로 바꿔줍니다. L1에서 timeq가 0 이하가 되면 priority가 0보다 큰지 먼저 검사 후 그러면 1씩 감소시켜줍니다.
모든 프로세스가 생성될 때 L0 큐로 들어오기 때문에 allocproc() 함수에서 초기값들을 설정해줄 때 level을 0으로 해줬고, timeq 를 4로 설정해줬습니다. 스케줄러 함수 내부에는 일단 islevzero라는 변수가 추가되며, 이 변수는 ptable에서 Runnable한 프로세스 중 L0에 들어있는 프로세스가 있는지 확인합니다. 발견되면 break를 걸고 바로 스케줄링에 들어갑니다. (RR)
만약 L0에 더 이상 프로세스가 없다면, 가장 큰 priority값을 가진 프로세스를 찾아야 하며, 우선순위 값이 같으면 pid가 더 작은 프로세스를 선택해야 하기 때문에 maxpriority, minpid 변수를 추가해줬습니다. 그래서 ptable을 돌며 조건문들을 지나 가장 큰 우선순위와 pid 값이 찾아지면 마지막으로 유효한 pid인지 (L1에도 프로세스가 없을 수 있는 경우) 확인 후 스케줄링해줍니다.

• 구현

```
37 // Per-process state
38 struct proc {
39   uint sz;                     // Size of process memory (bytes)
40   pde_t* pgdir;                // Page table
41   char *kstack;                // Bottom of kernel stack for this process
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
53   uint srtime;           // tick when process started running by scheduler
54
55   // for MLFQ
56   // priority: process priority, higher -> bigger
57   // level: queue level (for MLQ, too)
58   // ismono: is this process monopolizing? 0:no 1:yes
59   // qtime: time quantum of this process
60   int priority;
61   int level;
62   int ismono;
63   int timeq;
64 };
```

```
int islevzero = 0; //is there proc in L0?

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
        continue;
    if(p->level == 0){
        islevzero = 1;
        break;
    }
}
//if there is L0 processes
//RR with time quantum 4
if(islevzero == 1){
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
            continue;
        if(p->level == 0){
```

```
//if there is L0 processes
//RR with time quantum 4
if(islevzero == 1){
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
            continue;
        if(p->level == 0){
            c->proc = p;
            switchuvm(p);
            p->state = RUNNING;
            p->timeq = 4;

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

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

```
//there's no L0 process
//find highest priority, if same, FCFS (lower pid first)
else if(islevzero == 0){
    //search for highest priority and smallest pid
    int minpid = 0;
    int maxpriority = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
            continue;
        if(p->level == 1){
            if(minpid == 0){
                minpid = p->pid;
                maxpriority = p->priority;
            }
            else{
                if(p->priority > maxpriority){
                    minpid = p->pid;
                }
                else if(p->priority == maxpriority){
                    if(p->pid < minpid){
                        minpid = p->pid;
                    }
                }
            }
        }
    }
```

```
//it exists
if(minpid > 0){
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == minpid){
            c->proc = p;
            switchuvm(p);
            p->state = RUNNING;
            //p->timeq = 8;

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

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

trap.c에서 L1으로 넘어오면서 timeq를 8로 주기 때문에 주석처리

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

  p->level = 0; // in MLFQ, all new process go to L0
  p->timeq = 4; // L0 -> time quantum is 4
  p->priority = 0;
  p->ismono = 0;
```

allocproc() 함수 변경점

```
#elif MLFQ_SCHED
  //every 200 ticks perform priority boosting
  if(ticks % 200 == 0)
      priority_boosting();

  if(myproc() && myproc()->state == RUNNING)
      myproc()->timeq--;

  //if process running in L0 (level == 0)
  //if it is monopolizing (ismono==1), then it should not yield()
  //and it's time quantum all consumed (timeq)
  //then go down to L1
  if(myproc() && myproc()->state == RUNNING &&
     myproc()->level==0 &&
     myproc()->ismono==0 &&
     myproc()->timeq <= 0){
    myproc()->level = 1;
    myproc()->timeq = 8;
    yield();
  }
```

trap.c 모습. 200초마다 priority boosting을 해준다.

```
  //if process running in L1 (level == 1)
  //and not monopolizing (ismono == 0), same as L0
  //and running time is over time quantum
  //then priority - 1, over 0
  if(myproc() && myproc()->state == RUNNING &&
     myproc()->level==1 &&
     myproc()->ismono==0 &&
     myproc()->timeq <= 0){
      if(myproc()->priority > 0)
          myproc()->priority--;
      yield();
  }
#endif
```

• 실행결과

```
MLFQ test start

Focused priority
process 4: L0=659, L1=19341
process 5: L0=1355, L1=18645
process 6: L0=1384, L1=18616
process 7: L0=2073, L1=17927
process 8: L0=2731, L1=17269

Without priority manipulation
process 9: L0=2146, L1=47854
process 10: L0=4359, L1=45641
process 11: L0=4459, L1=45541
process 12: L0=6627, L1=43373
process 13: L0=6519, L1=43481

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=2248, L1=22752
process 20: L0=2143, L1=22857
process 21: L0=2197, L1=22803
process 22: L0=4257, L1=20743
```

• 트러블슈팅

Monopolizing을 어떻게 구현해야하나 많은 고민을 했고, 결국 CPU를 독점하고 있다는 것은 yield()를 하지 않는 것이라고 생각했기 때문에 trap.c에서 ismono == 0 (독점하고 있지 않으면)을 추가로 조건문에 넣어줬습니다. 즉, 만약 ismono == 1 이면 독점을 하고 있다는 소리기 때문에 yield()를 호출하지 않습니다.
또한 L1 큐에 대한 조건문이 굉장히 많기 때문에 작은 실수에도 결과가 잘못 나와 큰 신경을 써야했습니다.

시스템 콜 구현 코드들

```
int
getlev(void)
{
    return myproc()->level;
}

int
setpriority(int pid, int priority)
{
    //if not 0<=priority<=10
    if(priority < 0 || priority > 10){
        return -2;
    }

    acquire(&ptable.lock);
    struct proc *p;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        //if process with pid exist and it is child of myproc()
        if(p->pid == pid && p->parent->pid == myproc()->pid){
            p->priority = priority;
            release(&ptable.lock);
            return 0;
        }
    }
    release(&ptable.lock);
    return -1;
}
```

```
void
monopolize(int password)
{
    //if password is correct
    if(password==2016025032){
        //if it was already monopolize
        if(myproc()->ismono == 1){
            myproc()->level = 0;
            myproc()->priority = 0;
            myproc()->ismono = 0;
        }
        //if not
        else if(myproc()->ismono == 0){
            myproc()->ismono = 1;
        }
    }
    //wrong password
    else if(password != 2016025032){
        kill(myproc()->pid);
    }
}
```

```
void
priority_boosting(void)
{
    struct proc *p;
    acquire(&ptable.lock);
    for(p=ptable.proc; p < &ptable.proc[NPROC]; p++){
        p->level = 0;
        p->priority = 0;
    }
    release(&ptable.lock);
}
```