Operating System

Project 4

2014-15703 송화평

1. 프로젝트 개요

본 프로젝트의 목적은 Linux 2.4에서 쓰이던 O(n) 스케쥴러를 xv6 위에 구현하는 것이다. xv6의 기본 스케쥴러는 Round-Robin 방식으로 구현되어 있어, timer interrupt가 발생할 때마다 스케쥴러가 호출되어 원형 큐 위의 다음 번 runnable 프로세스에게 CPU를 넘겨줌으로써 스케쥴링을 수행한다. 새로 구현한 O(n) 스케쥴러는 각 epoch마다 각 프로세스에게 nice value와 기타 정보에 기반하여 time slice를 분배해주고 각 프로세스의 goodness를 계산한 뒤 goodness가 가장 높은 프로세스 순으로 CPU를 할당한다. 각 프로세스는 pre-empt되지 않으며 주어진 time slice동안 CPU를 사용한다. 주어진 time slice를 다 사용하거나 프로세스 상태가 blocked로 변경되면 스케쥴러가 호출되고, 다시 goodness를 기준으로 CPU를 할당할 프로세스를 찾는다. 만약 모든 프로세스가 주어진 time slice를 다 사용하면, 이는 한 epoch가 끝난 것이고 새 epoch를 시작하여 처음부터 다시 시작한다. 이를 위하여 다음 세 가지를 수행하였다.

> 1. nice() system call 구현
> 2. O(n) 스케쥴러 구현
> 3. getticks() system call 수정

2. 구현 내용

2.1. 기존 코드 수정

nice(), getticks(), 스케쥴러를 구현하기에 앞서서, 이를 위하여 기존의 xv6 코드의 여러 부분을 수정하였다. 먼저, kernel/proc.c에 새로이 findproc() 함수를 구현하였다. findproc()은 입력으로 pid를 받아, pid에 해당하는 proc 구조체의 포인터를 반환하며, 그러한 proc 구조체가 발견되지 않으면 0을 반환한다.

```
struct proc*
findproc(int pid)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->pid == pid){
      release(&p->lock);
      return p;
    }
    else {
      release(&p->lock);
    }
  }
```

```
  return 0;
}
```

또한 프로세스를 fork할 때 부모 프로세스의 남은 counter를 반으로 나눠 자식 프로세스와 나누어 가지게 하기 위해 동일 파일 내의 fork()함수에 다음과 같은 내용을 추가하였다.

```
np->counter = (p->counter + 1) >> 1;  // PA4
p->counter = (p->counter) >> 1;
```

동일 파일의 userinit() 함수에도 init 프로세스에 counter를 할당하는 내용을 추가하였다.

```
p->counter = ((20 - (p->nice)) >> 2) + 1; // PA4
```

2.2. nice() system call 구현

nice system call은 kernel/sysproc.c에 sys_nice() 함수에 구현하였다. sys_nice() 함수는 system call 22번에 등록되어 있고, a0 레지스터에 첫 번째 패러미터로서 pid를, a1 레지스터에 두 번째 패러미터로서 inc를 저장한 상태로 실행된다. sys_nice()는 패러미터로 들어오는 pid가 0이면 현재 프로세스의 pid를 사용한다. 이 pid를 findproc() 함수에 넣어 pid에 해당하는 프로세스 구조체를 얻는다. 만약 pid < 0이거나 findproc() 함수를 통해 프로세스 구조체를 얻지 못하면 비정상 동작으로 간주하여 -1을 반환하고 종료한다.

이렇게 얻은 프로세스 구조체의 nice를 실제로 inc만큼 증가시키기에 앞서, 먼저 그 더한 값이 nice value의 range, 즉 [-20, 19] 범위 안에 들어가는지를 확인한다. 만약 그렇지 않으면 역시 비정상 동작이므로 -1을 반환하고 종료하며, 그렇지 않고 범위 안에 들어간다면 프로세스 구조체의 nice를 inc만큼 증가시킨다. 여기서 주의할 점은 자기 자신의 nice 값을 수정하는 것이 아니라면, 반드시 해당 프로세스에 acquire()을 통해 lock을 걸고 진행해야 하며, 모든 과정이 끝났거나 오류가 생겨 종료해야 하는 경우에는 release()를 통해 lock을 다시 풀어주어야 한다는 것이다. 이 과정에 해당하는 코드는 아래와 같다.

```
if(pid != curr_p->pid)
  acquire(&p->lock);

// Check the nice value range.
if(p->nice + inc > 19 || p->nice + inc < -20){
  if(pid != curr_p->pid)
    release(&p->lock);
  return -1;
}

p->nice += inc;

if(pid != curr_p->pid)
  release(&p->lock);
```

2.3. O(n) 스케쥴러 구현

xv6의 스케쥴러는 kernel/proc.c에 구현되어 있다. 어떤 프로세스를 스케쥴링 하기 위해서, 스케쥴을 시행할 해당 프로세스 구조체의 state를 RUNNING으로 변경하고, cpu 구조체의 proc에 해당 프로세스 구조체를 넣

어준 후 swtch() 함수를 통해 context를 바꿔준다.

2.3.1. Round-Robin 스케쥴러

이 때 어떤 프로세스를 스케쥴링 할 것인지 정하기 위해 xv6에서는 Round-Robin 스케쥴러를 사용하였다. 스케쥴링 이벤트가 발생하면 swtch() 함수에서 컨트롤이 반환되고, cpu 구조체를 비운 후 다음 RUNNABLE 상태의 프로세스를 실행한다. 만약 프로세스가 등록되어있는 proc 배열의 끝에 도달했다고 하더라도 무한루프 안이기 때문에 맨 처음으로 돌아가 RUNNABLE 상태의 프로세스를 찾게 된다.

2.3.2. O(n) 스케쥴러

새로 구현한 O(n) 스케쥴러에 사용하기 위해, 다음과 같은 변수들을 추가적으로 사용하였다.

```
struct proc *best_proc;
int max_goodness;
int goodness;
int counter_exhausted;
int no_runnable_proc;
```

O(n) 스케쥴러는 프로세스마다 goodness를 계산하여 goodness가 가장 큰 프로세스를 스케쥴링한다. 이를 위해 스케쥴러는 다음과 같이 프로세스 리스트를 순회하며 가장 높은 goodness 값과 이를 갖는 프로세스를 기록한다.

```
// Choose the best runnable process.
for(p = proc; p < &proc[NPROC]; p++) {
  acquire(&p->lock);

  if(p->state == RUNNABLE){
    goodness = p->counter == 0 ? 0 : p->counter + (20 - p->nice);
    if(goodness > max_goodness){
      max_goodness = goodness;
      best_proc = p;
    }
  }

  release(&p->lock);
}
```

이 작업이 끝나고 RUNNABLE 프로세스가 적어도 하나는 있다면, best_proc은 다음에 스케쥴링할 프로세스의 포인터를 갖게 된다. 이 경우 스케쥴러는 정상적으로 스케쥴링할 프로세스의 state와 cpu 구조체를 설정하고 swtch()를 통해 해당 프로세스에게 CPU 점유를 넘겨주게 된다.

```
// The best runnable process found: there is at least one runnable pr
// Give CPU to the best runnable process.
if(best_proc != 0 && best_proc->state == RUNNABLE){
  acquire(&best_proc->lock);

  best_proc->state = RUNNING;
  c->proc = best_proc;
  swtch(&c->scheduler, &best_proc->context);
```

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

    release(&best_proc->lock);
  }
```

이렇게 스케쥴링된 프로세스가 반환되면 걸어두었던 lock을 해제하고, 다음 스케쥴링할 프로세스를 정하기 전에 RUNNABLE 프로세스가 있는지, 만약 있다면 그 프로세스들의 counter가 고갈되지 않았는지를 다음과 같이 체크한다.

```
  // Check if there is no runnable process,
  // and if there is one, check if counters are exhausted.
  no_runnable_proc = 1;
  counter_exhausted = 1;
  for(p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);

    if(p->state == RUNNABLE){
      no_runnable_proc = 0;
      if(p->counter != 0)
        counter_exhausted = 0;
    }

    release(&p->lock);
  }
```

만약 RUNNABLE 프로세스가 없다면 no_runnable_proc은 1을 가지고 있을 것이고, 모든 RUNNABLE 프로세스의 counter가 고갈되었다면 counter_exhausted는 1을 가지고 있을 것이다. 이를 통해 아래와 같이 RUNNABLE 프로세스가 있는지 검사하여, 만약 그러하다면 다시 모든 RUNNABLE 프로세스의 counter가 고갈되었는지도 체크한다. 이 경우도 참이라면 모든 프로세스에게 새로이 counter를 분배함으로써 새 epoch를 시작한다. 이 때 SLEEPING 상태인 프로세스는 counter를 모두 쓰지 않았음을 고려하여 이전 epoch에서 남은 counter의 절반만큼을 분배 결과에 더하여 보정해준다.

RUNNABLE 프로세스가 없어 상술한 분기점이 실행되지 않았다면 스케쥴러는 for(;;)문을 계속해서 돌며 다른 변화가 일어나 탈출조건이 성립되기 전까지 busy-wait을 하게 된다.

2.4. getticks() system call 수정

getticks system call은 kernel/sysproc.c에서 sys_getticks() 함수를 통해 23번 system call로서 호출된다. sys_getticks() 함수는 패러미터로 전달된 pid만 추출하여 이를 kernel/proc.c에 정의된 getticks() 함수에 전달한다. 현재 코드에서는 CPU를 1개만 사용하고 스케쥴러도 O(n) 스케쥴러 하나만 사용하기 때문에 각 프로세스 구조체에 접근할 때 lock을 걸지 않은 기존의 getticks() 함수를 그대로 사용하였다.

다만 달라진 것은 기존의 Round-Robin 스케쥴러에서는 RUNNING 프로세스가 매 tick마다 yield()를 호출하고 yield() 함수 안에 ticks를 증가시키는 코드가 있었지만, O(n) 스케쥴러에서는 yield()가 매 tick마다 호출되는 것이 아니기 때문에 문제가 발생한다는 것이다. 이 문제는 kernel/trap.c의 usertrap() 또는 kerneltrap() 둘 중 하나(대부분의 경우 usertrap)가 매 tick마다 불린다는 점을 이용하여 해결할 수 있었다. timer interrupt가 매 tick마다 발생하면 각 프로세스의 counter와 ticks는 주기적으로 발생하는 timer interrupt마다 1씩 감소/증가해야 한다. 만약 counter가 0에 도달하면 프로세스는 yield()를 통해 CPU의 점유를 중단한다. 이 timer interrupt는 kernel/trap.c에 정의된 r_scause()와 devintr()를 통해 usertrap()

이나 kerneltrap()에서 처리한다. timer interrupt가 발생하면 dev_intr() 함수가 2를 반환하게 되고, 이 경우 usertrap()에서는 다음과 같이 처리하여 counter를 감소시키고 counter가 고갈되었는지 확인하여 고갈되었다면 yield()를, 그렇지 않다면 ticks를 1 늘리도록 구현하였다. 이 때 yield ()함수 안에 자체적으로 ticks를 1 늘리는 코드가 포함되어 있으므로 counter가 고갈되었을 때는 따로 ticks를 늘리지 않아도 되었다.

```
// PA4
// give up the CPU if this is a timer interrupt.
// correction: just decrement counter. yield if counter == 0.
if(which_dev == 2){
  p->counter--;

  if(p->counter == 0){
    yield();
  }

  // yield() also increments ticks; so incrementing ticks should be d
  else{
    p->ticks++;
  }
}
```

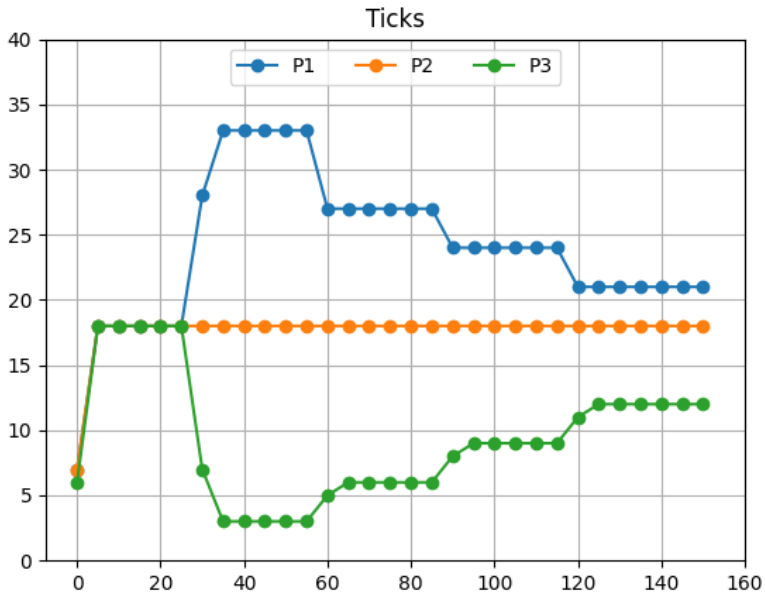
kerneltrap()의 경우에도 비슷하게 구현하였다.

```
// PA4
// give up the CPU if this is a timer interrupt.
// correction: just decrement counter. yield if counter == 0.
if(which_dev == 2 && myproc() != 0 && myproc()->state == RUNNING){
  p = myproc();
  p->counter--;

  if(p->counter == 0){
    yield();
  }

  // yield() also increments ticks; so incrementing ticks should be d
  else{
    p->ticks++;
  }
}
```

따라서 Round-Robin이 아닌 O(n) 스케쥴러의 경우에도 각 프로세스의 ticks를 증가시킬 수 있고, 이를 getticks() system call을 통해 제대로 가져올 수 있었다.

3. 결과

```
xv6 kernel is booting

init: starting sh
$ schedtest2
0, 7, 7, 6
5, 18, 18, 18
10, 18, 18, 18
15, 18, 18, 18
20, 18, 18, 18
25, 18, 18, 18
30, 28, 18, 7
35, 33, 18, 3
40, 33, 18, 3
45, 33, 18, 3
50, 33, 18, 3
55, 33, 18, 3
60, 27, 18, 5
65, 27, 18, 6
70, 27, 18, 6
75, 27, 18, 6
80, 27, 18, 6
85, 27, 18, 6
90, 24, 18, 8
95, 24, 18, 9
100, 24, 18, 9
105, 24, 18, 9
110, 24, 18, 9
115, 24, 18, 9
120, 21, 18, 11
125, 21, 18, 12
130, 21, 18, 12
135, 21, 18, 12
140, 21, 18, 12
145, 21, 18, 12
150, 21, 18, 12
$ QEMU: Terminated
```



schedtest2 실행 시 위와 같이 과제에서 제시한 예시와 동일한 결과를 얻을 수 있었다. 또한 schedtest1와 usertests의 경우에도 문제없이 통과할 수 있었다. 비록 기능이 제한되어 있는 운영체제이기 때문에 스케쥴러에게 더 많은 상황을 부여해볼 수는 없었지만, 테스트에서 요구하는 정도의 작업은 수행할 수 있는 것으로 보인다.