Operating Systems, Project 4

2013-11392 김지현

개요

이번 프로젝트의 목표는 간단한 버전의 리눅스 2.4 스케줄러를 구현하는것이다. 주어진 xv6 뼈대코드에 기본으로 탑재되어있는 스케줄러는 우선순위를 신경쓰지 않는 Round-robin 알고리즘을 사용하고있다. 이번 과제는 크게 nice syscall 구현하기, 스케줄러 구현하기 두 단계로 이뤄져있다. 수업시간에 배운 지식들을 활용해 어렵지 않게 과제를 마칠 수 있었다.

구현 설명

nice() 구현하기

```c
int nice(int pid, int inc) {
  if (pid < 0) { return -1; }
  struct proc *p = find_by_pid_then_lock(pid);
  if (p == 0) { return -1; }
  int next_nice = p->nice + inc;
  if (next_nice < -20 || 19 < next_nice) {
    release(&p->lock);
    return -1;
  }
  p->nice = next_nice;
  release(&p->lock);
  return 0;
}
```

Figure 1. Simplified code of nice() implementation

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int
main(int argc, char **argv)
{
  if(argc < 3){
    fprintf(2, "usage: nice <pid> <inc>\n");
    exit(1);
  }
  int ret = nice(atoi(argv[1]), atoi(argv[2]));
  printf("%d\n", ret);
  exit(0);
}
```

Figure 2. 테스트용 user/nice.c

nice 시스템콜을 구현하는 일 자체는 setpgid, getpgid와 비슷하게 proc 구조체에 새 필드 하나를 추가하는 일에 불과하기 때문에, Figure 1과 같이 이전 과제에서 했던것과 유사한 방식으로 구현하였다. 특별히 신경써줘야 하는 점들은 아래와 같았다.

- Nice를 inc 값으로 변경하는 것이 아니라, 현재 있는 nice 값에 inc 값을 더하는 것임
- inc의 범위가 아니라 nice+inc 의 범위가 [-20, 19] 안에 있는지 확인해야함
- freeproc() 에서 nice를 0으로 초기화시켜야 새 프로세스의 nice 값이 항상 0이 됨
- fork()에서 자식 프로세스가 부모 프로세스의 nice를 상속받도록 해야함

특기할 점으로는 이번 과제부터는 친절하게 시스템콜을 테스트하는 용도의 유저 유틸리티가 별도로 제공되지 않기 때문에, Figure 2와 같이 직접 유틸리티를 만들어 정합성을 검사하였다.

스케줄러 구현: Clock tick 마다 counter와 tick 업데이트하기

xv6의 기본 스케줄러는 timer interrupt를 만나면 무조건 yield하도록 구현되어있다. 먼저 이를 timer slice를 모두 소진했을때만 yield 하도록 고칠 필요가 있었다. 그래서 Figure 3과 같이 decrement_counter_or_yield() 라는 이름의 새로운 함수를 만들어, Figure 4와 같이 기존의 yield 호출을 decrement_counter_or_yield() 호출로 대체하였다.

```
static void decrement_counter_or_yield(void) {
  struct proc *p = myproc();
  acquire(&p->lock);
  p->counter -= 1;
  p->ticks += 1;
  const int no_more_timeslice = p->counter <= 0;
  release(&p->lock);
  if (no_more_timeslice) yield();
}
```

Figure 3. Simplified version of decrement_counter_or_yield()

```
@@ -78,7 +97,7 @@ usertrap(void)

   // give up the CPU if this is a time
   if(which_dev == 2)
-    yield();
+    decrement_counter_or_yield();

   usertrapret();
 }
@@ -151,9 +170,9 @@ kerneltrap()

   // give up the CPU if this is a time
   if(which_dev == 2 && myproc() != 0 &
-    yield();
+    decrement_counter_or_yield();
```

Figure 4. Use of decrement_counter_or_yield()

decrement_counter_or_yield() 함수는 매 타이머 틱마다 호출되어야 하는 함수이고, 호출될경우 단순무식하게 counter를 1 감소시키고 ticks을 1 증가시킨 뒤, counter가 0이 되어 더 이상의 타임슬라이스가 남아있지 않는 것이 확인될 경우 마침내 yield하는 함수이다.

여기까지만 구현하면 Time slice가 소진될 때까지 프로그램이 yield하지 않고 연속으로 실행되는 부분은 만족하지만, 스케줄러를 고치지 않았기때문에 여전히 우선순위를 무시하고 Round-robin으로 스케줄링된다. 이를 해결하기위해 scheduler() 함수의 수정이 필요하다.

스케줄러 구현: Goodness 기반 스케줄링

먼저 Runnable한 프로세스중 가장 goodness가 좋은 프로세스를 선정하는 코드가 필요하다. 이 때, Runnable한 프로세스가 단 한 개도 존재하지 않는 상황을 가정하고 작성해야한다. Figure 5와 같이 O(n)으로 모든 프로세스를 순회하는 방식으로 구현하였다.

```
int max_goodness = 0x80000000; // INT_MIN
struct proc *next_p = 0;
for(struct proc *p = proc; p < &proc[NPROC]; p++) {
  acquire(&p->lock);
  if(p->state == RUNNABLE) {
    const int goodness = p->counter == 0 ? 0 : p->counter + (20 - p->nice);
    if (max_goodness < goodness) {
      max_goodness = goodness;
      next_p = p;
    }
  }
  release(&p->lock);
}
```

Figure 5. 가장 Goodness가 높은 프로세스 선정하기

Proc 배열을 goodness를 기준으로 정렬하거나, goodness를 키로 갖는 Max heap을 만들거나, goodness로 정렬된 자가균형 탐색트리로 바꿀 경우 시간복잡도가 훨씬 좋아지지만, 이 과제의 범위를 넘어가는 것 같아 구현하지 않았다.

Figure 5의 프로세스 O(n) 탐색을 거치면 아래의 세 결과중 하나가 나오게된다.

A. Runnable한 프로세스가 없음 (next_p = 0)
B. Runnable한 프로세스가 있고, counter가 양수인 프로세스가 존재함
   (next_p != 0, max_goodness > 0)
C. Runnable한 프로세스가 있으나, 모든 프로세스의 counter가 0임
   (next_p != 0, max_goodness = 0)

```
if (next_p == 0) {
  // No runnable process, busy wait
} else if (max_goodness > 0) {
  // Runnable process exists, schedule
  acquire(&next_p->lock);
  if(next_p->state == RUNNABLE) {
    next_p->state = RUNNING;
    c->proc = next_p;
    swtch(&c->scheduler, &next_p->context);
    c->proc = 0;
  }
  release(&next_p->lock);
} else {
  // All runnable processes' counter is 0, NEW EPOCH
  for (struct proc *p = proc; p < &proc[NPROC]; ++p) {
    acquire(&p->lock);
    if (p->state == RUNNABLE)
      p->counter = ((20-(p->nice)) >> 2) + 1;
    else if (p->counter == SLEEPING)
      p->counter = (p->counter >> 1) + ((20-(p->nice)) >> 2) + 1;
    release(&p->lock);
  }
}
```

Figure 6. A, B, C 각 경우에서 어떻게 동작하는지 코드로 간단하게 표현한것

A 경우에선 과제가 요구한대로 busy_wait 해야한다. 별도의 동작을 수행하지 않고, scheduler() 함수의 무한루프를 계속 실행하는 방식으로 구현하였다. 이렇게 구현할 경우 Runnable한 프로세스가 하나라도 생기는 즉시, 해당 프로세스가 실행된다.

B 경우는 정상적으로 스케줄할 프로세스가 선정된 상황으로, 선정된 프로세스로 컨텍스트 스위치 하면 된다. 뼈대코드에 있는 이미 있었던 구현을 그대로 사용하였다.

C 경우에선 스케줄할 프로세스가 없으나, Runnable한 프로세스가 존재하는 상태이기때문에 busy wait 하며 시간을 버려서도 안된다. 이 경우엔 모든 프로세스를 순회하며 counter 값을 초기화시키는 동작을 수행해야한다. Figure 6과 같이 과제에서 요구한것과 동일하게 구현하였다.

여기까지 구현하면 스케줄러의 큰 부분은 정상적으로 동작하지만, 아래의 두 지점을 간과해선 안된다.

- Nice와 마찬가지로, freeproc()에서 counter 값과 ticks 값을 모두 0으로 초기화 시켜줘야, 다음에 새 프로세스를 생성할 때 counter와 ticks 필드의 초기값이 0이 된다.
- Fork()시, counter의 경우 counter를 ½로 나눠 자식 프로세스와 부모 프로세스가 나눠가져야한다. counter값이 홀수일 경우 자식이나 부모 프로세스중 하나가 1을 더 가져야 하는데, 과제에 있는것과 동일한 결과가 나오게 하려면 자식 프로세스가 1을 갖도록 해야한다.
- Fork()시, 자식 프로세스의 ticks이 0부터 시작하도록 만들어야한다.

여기까지 구현할 경우, 스케줄러와 getticks() 시스템콜 모두 과제에서 요구하는 결과를 정확하게 재현할 수 있게된다. Getticks() 시스템콜의 경우 기존 뼈대코드에 있던 ticks 업데이트 하는 코드를 삭제하고 Figure 3과 같이 수정하면 getticks() 시스템콜을 고치지 않아도 정상적인 결과가 출력된다.

스케줄러 구현의 테스트를 돕기 위해 과제에서 schedtest1, schedtest2 두 유틸리티를 제공했지만, 언제 타이머 틱이 발생하는지 보다 편리하게 파악하기 위해 아래와 같은 간단한 유틸리티를 추가로 구현해 과제를 수행했다.

```
int main(int argc, char **argv)
{ for (;;); }
```

Figure 7. loop_user.c, 유저 모드에서 무한루프를 실행하는 유틸리티

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main(int argc, char **argv) {
  for (;;) write(1, "", 0);
}
```

Figure 8. loop_kernel.c, 유저모드와 커널모드를 오가며 무한루프하는 유틸리티

실험 결과 및 분석

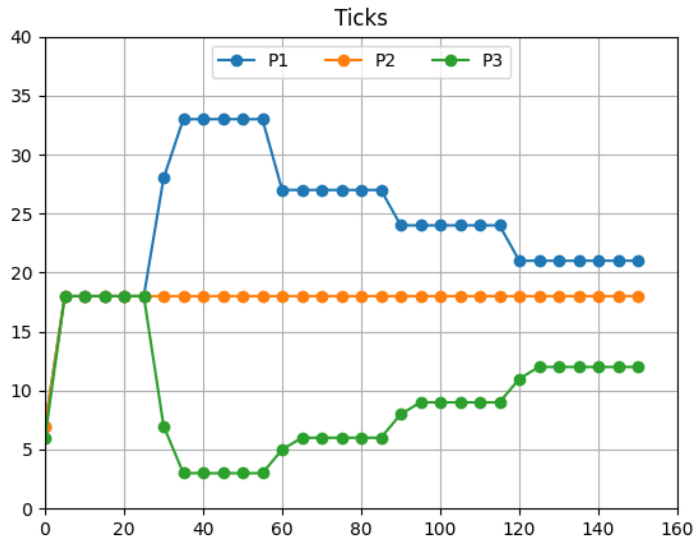
다음 페이지의 Figure 9와 같이, 과제에서 요구한것과 정확하게 동일한 실험 결과가 재현된다.

schedtest2는 유저모드에서 무한루프를 도는 자식프로세스 세개를 생성한 뒤, 세 프로세스의 nice값을 시간에 따라 조정하며 각 프로세스들이 약 5초동안 얼마만큼의 tick을 사용하는지 측정해주는 프로그램이다. Sleep() 시스템콜이 정확히 5초 뒤에 깨어나는 함수가 아니고, 타이머 인터럽트 또한 매번 정확히 같은 타이밍에 발생하지 않기 때문에 실행 결과가 완전히 결정적일수는 없지만, 여러 번 실행해도 거의 항상 같은 결과가 나왔다.

Figure 9의 그래프에서 0초부터 30초까지는 Phase 1로, 세 프로세스의 nice가 모두 동일한 상황인데 이때엔 세 프로세스가 항상 같은 tick을 소진하는것을 볼 수 있다.

30초부터 60초까지는 P1 P2 P3의 nice가 각각 -20, 0, 19인데, 제일 높은 우선순위를 가진 P1이 제일 높은 ticks를 소진하고, 제일 낮은 우선순위를 갖게된 P3는 몹시 적은 tick을 사용하는 것을 볼 수 있다.

60초 이후로는 30초마다 Phase 3, 4, 5가 이어지는데 각 페이즈에선 P1, P2, P3의 우선순위가 각각 -15/0/15, -10/0/10, -5/0/5 로 차이가 줄어든다. 이에 따라 프로세스들이 실제로 소진하는 tick의 차이도 줄어드는 것을 알 수 있다.




```
$ 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
```

Figure 9. graph.png and part of xv6.log

여기서 특기할 점은 프로세스들의 우선순위를 항상 같은 5씩 조정함에도 불구하고 프로세스들이 실제로 사용하는 tick이 차이에 정비례해서 줄어들지는 않는 것을 볼 수 있다. 이 과제에서 구현한 스케줄러는 리눅스의 CFS와는 달리 프로세스끼리 우선순위에 차이가 있더라도 우선순위 차이에 정확하게 비례하는 시간을 배정해주지는 못하고 있다는 것을 Figure 9의 그래프로 확인할 수 있다.

결론

수업시간에 배웠던 Counter 기반의 간단한 스케줄러를 xv6에 직접 구현하여, xv6에 기능도 추가하고 수업에서 배웠던 개념도 몸으로 익힐 수 있었던 좋은 실습이었다. 동시에 counter 기반의 스케줄러가 시간 분배 측면에서 어떤 한계점을 갖는지 또한 확인할 수 있었다. 거의 기능이 없었던 xv6에 나만의 기능을 점점 추가해나갈 수 있어서 재미있었고, 앞으로의 과제들도 몹시 기대된다.