Operating System

Programming Assignment #4: Simplified Linux 2.4 scheduler

2013-11826 임주경

- System call : nice()

Sysproc.c 내부에서 sys_nice는 첫번째 매개변수를 pid로 받으며, pid가 음수이면 -1을 return한다. 두번째 매개변수는 int형 변수 inc에 저장하고, nice(pid, inc)함수를 호출한다.

Proc.c 내부에서 nice(int pid, int inc)함수를 정의하였다. 프로세스 구조체 변수 p와 pid에 해당하는 프로세스가 존재하는지 여부의 int형 변수 found를 선언한다. Pid가 0일 경우, 실행중인 프로세스를 p에 초기화하며, found를 1로 저장한다. Pid가 0이 아닌 경우, for문을 통해 pid에 해당하는 프로세스를 찾으며, 발견 시 found를 1로 저장한다. 이때, 변수 p를 이용한 loop에서 p는 해당 프로세스를 저장하게 된다. Found가 0이면 -1을 return하고, 그렇지 않으면, 새롭게 저장될 nice값의 범위가 -20~19에 유효한지 확인한다. 유효하지 않으면 -1을 return한다. 유효 범위의 nice값일 경우, 해당 프로세스의 nice값을 변경하고 0을 return한다.

nice값의 몇 가지 제약 조건으로 proc.c 코드 일부를 수정하였다. Userinit()에서 init process의 nice를 0으로 초기화한다. 또한, fork()에서 parent process의 nice값과 child process의 nice값을 과제 spec의 공식에 맞게 저장하도록 코드 추가하였다.

- Scheduling algorithm

우선, process의 priority를 결정할 goodness값을 계산하기 위한 함수가 필요하다. Defs.h에 해당 함수를 선언해주었다. goodness(struct proc *p) 함수에서 goodness 공식은 과제 spec과 동일하며, input process p에 대해서, p의 counter값이 0이면, 0을 return. 그렇지 않으면, 공식에 해당하는 goodness값을 return한다.

다음, 기존의 RR방식으로 구현된 scheduler() 함수를 수정하였다. 새롭게 추가된 변수는 struct proc *maxP, int형 변수 new1, new2, found, maxG가 있다. maxP 구조체는 goodness 함수 값을 이용해서 최우선 process를 저장하기 위해 사용된다. new1과 new2는 scheduler가 busy-waiting을 할지, new epoch을 수행할지 결정하기 위한 checking 변수이다. New1은 for문으로 process를 돌며, RUNNABLE한 state인데 counter가 0이 아닌 것이 남아있는 경우를 검사한다. New2는 마찬가지로 for문으로 process를 돌며, 모든 process의 state가 sleep이 아니라는 것을 보장하기 위해 사용된다. 최우선 process를 선별하기 위해 for문으로 구현하였으며, goodness 최대값을 maxG에 저장하고, 해당하는 process를 maxP에 저장한다. 실행할 수 있는 process가 존재한다면, found값은 1로 저장된다. 프로세스 선별이 완료된 후, found값이 1이면, 해당 process state를 RUNNING으로 변경

한다. Cpu의 process를 해당 process로 변경하고, context switch를 수행한다. 만약, 실행할 process가 없을 경우(found값이 0) new1, new2값을 통해 new epoch을 수행할지 busy-waiting을 수행할지 분기점에 놓이게 된다.

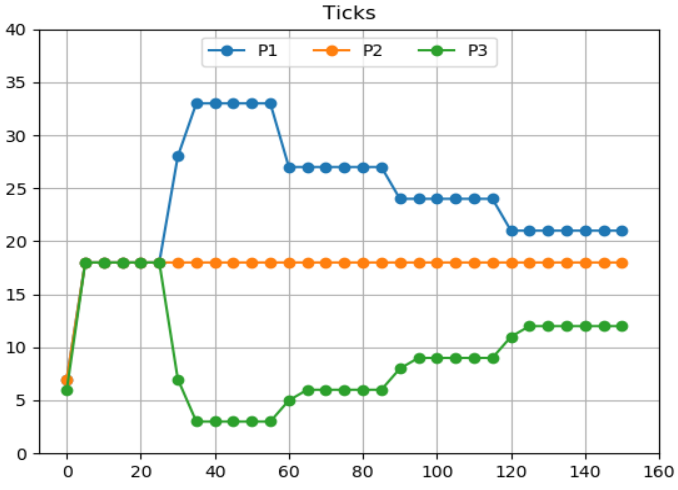
다음, 선별된 process가 실행되는 동안, timer interrupt에 대한 적절한 구현을 위해 trap.c의 코드를 수정하였다. Usertrap()에서 which_dev값이 2인 경우, timer interrupt임을 확인한다. 이때, 실행중인 프로세스의 counter를 하나 감소시킨다. Counter가 0이하라면(fork()로 counter가 0이 된 후, timer interrupt가 발생하면 음수도 가능) yield()호출로 프로세스 실행을 마친다. 이때, yield()내부에서 process의 ticks이 1 증가하므로, 별도의 ticks 증가를 구현하지 않는다. 만약, counter가 아직 남아 있다면, 해당 프로세스의 ticks을 1 증가시키고, yield()하지 않음으로써, process가 다른 process로 넘어가지 않게 유지한다. Kerneltrap()의 timer interrupt에서도 같은 방법으로 구현해주었다.

- System call : getticks()

기존에 구현된 getticks()는 RR scheduler에서 모두 동일한 프로세스의 ticks값을 return하기 때문에, 별도의 acquire()호출이 필요하지 않았다. 기존의 구현은 유지한 채, pid에 해당하는 process를 찾는 for문에서 acquire(), release() 호출을 추가해 lock holding을 control 하도록 수정하였다. 각 process는 별도의 ticks값을 갖고, 위의 scheduling을 통해 ticks값이 개별적으로 변화한다. 따라서, getticks() 값이 각 process에 해당하는 ticks값을 올바르게 출력하도록 구현하였다.

- Schedtest2 Result & Analysis

Schedtest2 실행 결과 xv6.log와 graph.png는 아래에 첨부한 것과 같다.



```
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
```

위 결과는 과제 spec의 sample result와 같게 나왔으며, 올바르게 scheduling이 수행되었음을 확인할 수 있다. 초기에 p1, p2, p3가 실행되고, 각각 ticks가 7, 7, 6에 해당하게 된다. Phase1에서 각 nice는 0/0/0으로 counter는 6/6/6이다. 이때, p1, p2는 초기 counter가 1에서 시작되어, ticks이 p3보다 1 크다. 30에서 nice는 -20/0/19 즉, counter는 11/6/1이다. Phase2에서 ticks는 33/18/3으로 3-cycle씩 돌았음을 확인할 수 있다. 다음 phase3의 nice는 -15/0/15, counter는 9/6/2이므로, 3-cycle씩 수행할 경우 ticks는 27/18/6이 타당하다. Phase4의 nice는 -10/0/10, counter는 8/6/3이므로, ticks는 24/18/9가 맞다. Phase5의 nice는 -5/0/5, counter는 7/6/4이므로, 마찬가지 올바른 ticks 값이 추출되었다. 이때, 30초마다 ticks의 패턴이 불규칙한 양상을 보인다. 이는, nice값이 변화하는 순간, 전 phase의 counter와 다음 phase의 counter가 섞이면서 일어나는 결과로 보인다.

따라서, 올바른 scheduling 결과를 확인할 수 있다.