Fairness 에 중점을 둔 프로세스 스케줄링 기법[1]

강성용*, 장학범**, 최형기**
*성균관대학교 휴대폰학과
**성균관대학교 정보통신공학부
e-mail : { sykang, hbjang, hkchoi}@ece.skku.ac.kr

Process scheduling policy based on fairness

Seong-Yong Kang*, Hak Beom Jang**, Hyoung-Kee Choi**
*Dept. of Mobile Systems Engineering, Sungkyunkwan University
**School of Information and Communication Engineering, Sungkyunkwan University

요 약

운영체제는 여러 가지 프로세스 스케줄링 기법을 지원한다. 본 논문에서는 오픈 소스 운영체제인 Linux 에서 fairness 에 중점을 둔 스케줄링 기법을 설계 및 구현하여 프로세스들간의 priority inversion 과 starvation 을 해결하는 방법을 제안한다.

1. 서론

최근 급격히 발전한 기술의 발전으로 컴퓨터를 사용한 작업이 증가하였다. 컴퓨터를 이용한 업무가 늘어남에 따라 많은 작업이 동시에 하나의 컴퓨터에서 실행된다. 많은 작업을 동시에 처리하기 위해 OS (Operating System)는 대기중인 여러 작업들 중 우선순위에 따라 작업을 처리하게 된다. 하지만 이러한 우선순위에 기반한 스케줄링 기법을 사용함에 따라 특정 프로세스가 오랜 시간 동안 수행되지 못하고 대기상태에 머무르는 기아 (starvation) 현상이 나타날 수 있고, 비선점형 (non-preemptive) 스케줄링 기법을 사용할 경우 우선순위 반전 (priority inversion)이 일어날 수 있다. 따라서 특정 프로세스가 예상된 작업 완료시간보다 늦게 종료되는 현상이 발생할 수 있다. 이러한 현상을 해결하기 위해 본 논문에서는 Fairness 에 중점을 둔 스케줄링 기법을 제안한다.

2. 스케줄링 방법

2.1 FCFS (First Come First Service)

FCFS 는 비선점형 스케줄링 기법으로 다양한 스케줄링 기법 중 가장 간단한 방법이다. 비선점형 스케줄링 기법이기 때문에 타임퀀텀 (time quantum)은 존재하지 않는다. FCFS 의 스케줄링 기준은 프로세스의 레디 큐 (ready queue)에 대한 도착시간이다. FIFO 와 같은 개념으로 레디 큐에 도착한 순서대로 프로세스가 CPU 를 사용할 수 있다. 이러한 스케줄링 기법은 batch system 에 적합하며 리소스 이용률이 높지만 콘보이 이펙트 (convoy effect) 등의 이유로 인하여 응답시간 (response time)이 긴 단점이 있다.

2.2 RR (Round Robin)

RR 은 선점형 스케줄링 기법으로 스케줄링의 기준은 FCFS 와 같지만 타임퀀텀이 존재하여 자신의 타임퀀텀을 모두 사용한 프로세스는 CPU 를 놓고 대기상태로 들어간다. 타임퀀텀의 존재로 인해 특정 프로세스의 CPU 독점을 방지할 수 있지만 문맥교환 오버헤드가 발생한다.

2.3 SPN (Shortest Process Next)

SPN 은 대기중인 프로세스 중 수행시간이 가장 짧은 프로세스를 먼저 수행시키는 비선점형 스케줄링 기법이다. 빨리 종료될 수 있는 프로세스를 먼저 수행하기 때문에 대기중인 프로세스의 수를 최소화 할 수 있고 프로세스의 집합에서 평균 대기시간을 줄일 수 있지만 기아현상이 발생하는 단점이 있다.

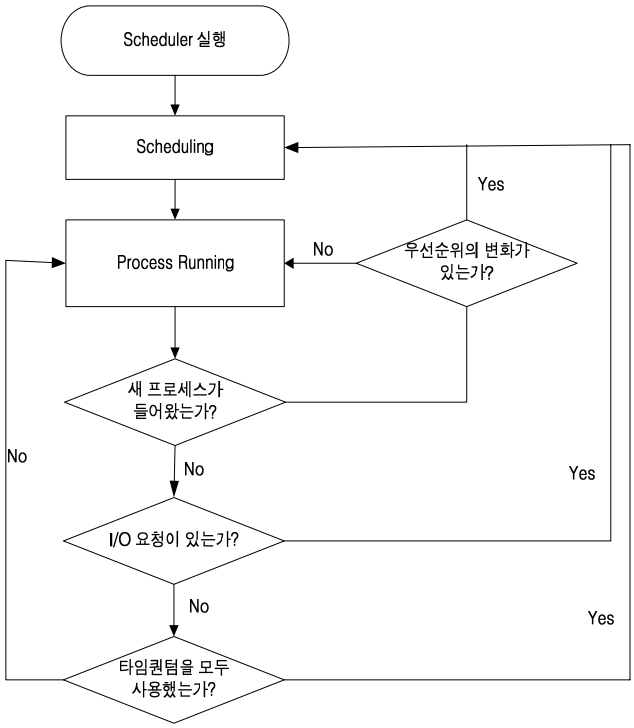
2.4 SRTN (Shortest Remaining Time Next)

SRTN 은 SPN 을 선점형 스케줄링 기법으로 변형시킨 스케줄링 기법이다. 프로세스를 수행 중에 더 적은 수행시간을 갖는 프로세스가 레디 큐에 도착하면 해당 프로세스에게 선점 당하게 된다. 이러한 방법은 SPN 에서와 마찬가지로 수행시간을 예측하기 어렵고 문맥교환 오버헤드가 크다는 단점이 있다.

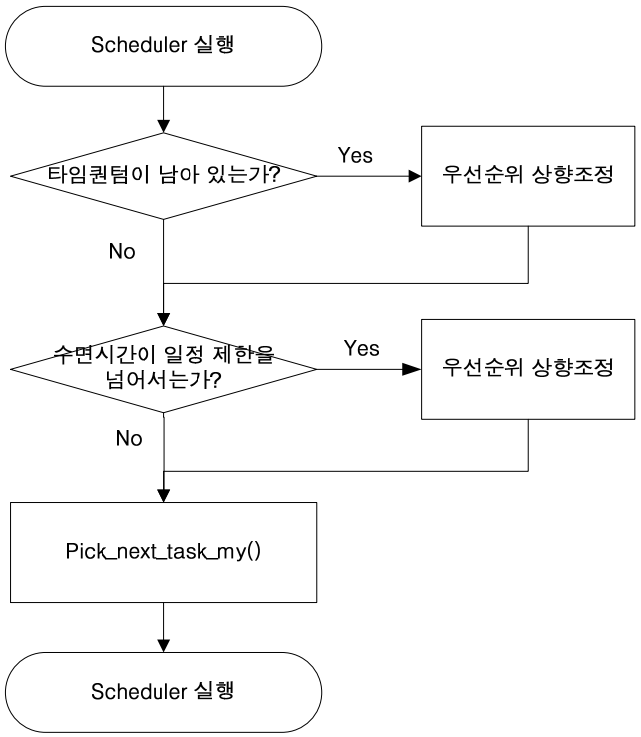
2.5 MFQ (Multi level Feedback Queue)

MFQ 는 현재 대부분의 OS 에서 사용하는 스케줄링 기법으로 선점형이며 우선순위가 동적으로 변한다. 우선순위에 따라 분류되고 각각 다른 스케줄링

[1] 이 논문은 2011 년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임(No.2011-0005037)



(그림 1) 일반적인 스케줄링 Flow



(그림 2) 우선순위 조정 Flow

기법을 가진 다중 레디 큐를 가진다. 또한 피드백(feedback)을 통한 동적 우선순위 기반의 정책을 통해 스케줄링의 대상이 되는 프로세스들에 대한 정보가 전혀 없는 경우에도 효과적인 스케줄링이 가능하다.

3. 제안하는 스케줄링 기법

본 논문에서는 프로세스들 간에 fairness 에 중점을 둔 스케줄링 기법을 제안한다. 우선순위를 고려한 선점형 스케줄링 기법을 사용하여 우선순위 반전을 방지해서 효율적인 스케줄링을 지원하도록 한다. 프로세스가 다른 프로세스에 의해 선점을 당하거나 I/O 작업을 요청할 경우 할당 받은 타임퀀텀을 다 사용하지 못하고 수면 (sleep) 상태로 들어가게 되므로 이러한 경우 우선순위를 상향조정 해서 다음 스케줄링 시점에 CPU 를 우선 사용할 확률을 높이는 방법을 사용하도록 한다. 또한 수면시간이 일정 기준을 넘어설 경우에도 역시 우선순위를 상향 조정하여 기아현상을 방지하도록 한다.

4. 구현

제안한 기법을 실제로 구현하기 위해 Linux kernel 2.6.30 버전을 사용하였다.

4.1 task_tick_my()

매 tick 마다 호출되어 현재 실행중인 프로세스의 time_slice 를 1 만큼 감소시킨 후에 time_slice 가 남아있으면 함수를 빠져나가고, 모두 다 사용했으면 time_slice 를 재할당 해주고 우선순위를 5 만큼 증가시킨다. Time_slice 를 모두 사용한 경우에는 prio_update() 함수를 통해 우선순위를 업데이트 해준 후 resched_task()를 호출하여 스케줄링이 필요함을 알린다.

4.2 prio_update()

본 논문에서 제안하는 바를 수행하는 함수이다. 대기중인 프로세스들의 대기시간에 실행 중이던 프로세스의 실행시간을 더해 대기시간을 갱신해준 후 MAX_SLEEP_TIME 보다 오래 대기했다면 우선순위를 1 만큼 상향 조정한다. 선점을 당해서 time_slice 가 남아있는 경우와 I/O 작업을 위해 수면 상태로 들어간 프로세스의 경우에도 우선순위를 1 만큼 상향 조정한다.

4.3 Test 프로그램

구현한 스케줄링 기법을 테스트하기 위해 간단한 테스트 프로그램을 작성하여 수행하여 보았다. 테스트 프로그램은 두 가지의 기능을 수행하는 프로세스들로 구성되어 있다. 1 번 프로세스는 I/O bound 프로세스로써 파일로부터 데이터를 읽어오는 작업과 화면에 출력하는 작업 위주로 수행하며, 빈번한 I/O 작업을 위해 자주 sleep 된다. 2 번 프로세스는 CPU bound 프로세스로써 CPU 작업을 많이 하도록 하기 위해 백트래킹 기법을 사용한 N-Queens 함수를 수행하도록 하였다.

5. 결과

제안한 스케줄링 기법을 확인하기 위해 테스트 프로그램을 수행한 결과는 (그림 4)과 같다. I/O 작업을 계속 수행하는 1 번 프로세스는 우선순위가 점점 상향 조정되어 우선순위가 100 에

```
unsigned int run_time = TIME_SLICE - rq->curr->time_slice;

/* Update Priority */
list_for_each(pos, &my_tasks) {
    mt = list_entry(pos, struct my_task, my_taskl);

    if (rq->curr != mt->task) {
          /* Update sleep time */
          mt->task->sleep_time += run_time;

          if (mt->task->sleep_time >= MAX_SLEEP_TIME) {

                    mt->task->prio--;
                    mt->task->sleep_time = 0;

                    if (mt->task->prio < 100)
                              mt->task->prio = 100;
          }
    }
}

if (rq->curr->time_slice)
          rq->curr->prio--;

if (rq->curr->state == TASK_INTERRUPTIBLE || rq->curr->state
== TASK_UNINTERRUPTIBLE) {
          rq->curr->prio--;
          printk("   I/O Mode!! PID : %d₩n", rq->curr->pid);
}

if (rq->curr->prio < 100)
          rq->curr->prio = 100;
```

(그림 3) prio_update() 함수의 구현 코드

```
MINE: (enqueue_task_my:114) pid = 5349
MINE: (pick_next_task_my:84) PID: (5349) PRIO: (100)
sleep 1 5349
^II/O Mode!! PID : 5349
MINE: (dequeue_task_my:141) pid = 5349
MINE: (put_prev_task_my:146) PID: (5349) PRIO: (100)
MINE: (pick_next_task_my:84) PID: (5350) PRIO: (117)
MINE: (check_preempt_curr_my:55) CUR_PID: 5350 NEW_PID: 4636 PRIO:
MINE: (check_preempt_curr_my:55) CUR_PID: 5350 NEW_PID: 5330 PRIO:
MINE: (check_preempt_curr_my:55) CUR_PID: 5350 NEW_PID: 4320 PRIO:
TASK TICK MINE: (task_tick_my:174) PID: 5350 Time_Slice : 6 PRIO:
MINE: (check_preempt_curr_my:55) CUR_PID: 5350 NEW_PID: 5296 PRIO:
TASK TICK MINE: (task_tick_my:174) PID: 5350 Time_Slice : 5 PRIO:
TASK TICK MINE: (task_tick_my:174) PID: 5350 Time_Slice : 4 PRIO:
TASK TICK MINE: (task_tick_my:174) PID: 5350 Time_Slice : 3 PRIO:
MINE: (check_preempt_curr_my:55) CUR_PID: 5350 NEW_PID: 5274 PRIO:
TASK TICK MINE: (task_tick_my:174) PID: 5350 Time_Slice : 2 PRIO:
TASK TICK MINE: (task_tick_my:174) PID: 5350 Time_Slice : 1 PRIO:
^ITime slice expired!! PID : 5350
 MINE: (put_prev_task_my:146) PID: (5350) PRIO: (121)
wakeup 5349
MINE: (enqueue_task_my:114) pid = 5349
MINE: (pick_next_task_my:84) PID: (5349) PRIO: (100)
MINE: (check_preempt_curr_my:55) CUR_PID: 5349 NEW_PID: 5296 PRIO:
sleep 1 5349
^II/O Mode!! PID : 5349
MINE: (dequeue_task_my:141) pid = 5349
```

(그림 4) 테스트 프로그램의 커널 로그

도달하고 CPU 작업을 수행하는 2 번 프로세스는 상향과 하향을 반복하여 일정 수준의 우선순위를 유지하는 것을 확인 할 수 있다.

6. 결론

기존의 스케줄링 기법은 정해진 우선순위에 따른 스케줄링을 통해 서로 다른 작업을 수행하는 프로세스간에 unfair 한 수행 결과를 보였다. 본 논문은 상대적으로 대기상태에 오래 머무르거나 빈번한 I/O 작업으로 인해 CPU bound 프로세스에 비해 늦은 수행결과를 보이는 프로세스의 우선순위를 상향 조정하는 기법을 사용하여 fairness 에 중점을 둔 스케줄링 기법을 구현하였다.

참고문헌

[1] Baruah, S.K. “Strong P-fairness: a scheduling strategy for real-time applications”, Real-Time Applications, 1994., Proceedings of the IEEE Workshop on

[2] Baruah, S.K. “Fairness in periodic real-time scheduling”, Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE

[3] M. Ajtai, J. Aspnes, M. Naor, Y. R.abani, L. Schulman, and O. Waarts. “Fairness in scheduling”. In Proceedings of the Sixth Annual ACMSIAM Symposium on Discrete Algorithms, .January 1995.

[4] D.P. Bovet and M. Cesati, “Understanding the Linux Kernel”, O’REILLY, 2007

[5] Abraham Silberschatz and Peter B. Galvin and Greg Gagne, “Operating System Concepts – 8/E”, WILEY, 2009

[6] http://www.kernel.org