환경에서의 리눅스 스케줄러 성능분석

*김도한, 신동군
성균관대학교 정보통신대학
e-mail : amos_doan@nate.com, dongkun@skku.edu

A Study of Linux Schedulers in Multi-core Environment

*Do-Han Kim, Dongkun Shin
College of Information and Communication Engineering
Sungkyunkwan University

Abstract

These days, Multi-core processor is generalized in the desktop market. Therefore, in terms of operating system, task scheduler needs new strategies to improve response time and turnaround time in the multi-core environment. In this paper, main algorithms of two popular Linux schedulers, CFS(Completely Fair Scheduler) and BFS(Brain Fuck Scheduler) are examined, and, finally, the performance of the schedulers are analysed.

I. 서론

멀티코어 프로세스가 데스크탑 시장에 보편화되며 단순히 증가된 코어 수만큼의 성능 향상뿐만 아니라 여러 개의 코어수를 좀 더 효율적으로 활용하여 성능을 극대화 할 수 있는 전략이 필요하게 되었다.

이에 따라 운영체제 계층에서는 단순하게 증가된 코어에 프로세스를 스케줄링 하기보다, 여러 개의 코어로 인한 프로세스 공유 문제, 코어 간 로드의 불균형, 캐쉬 메모리의 지역성의 원칙(locality) 등을 고려하여 스케줄링을 좀 더 효율적으로 할 수 있어야 한다. 스케줄링 알고리즘에 따라 멀티코어 환경에서 프로세스가 준비 상태로부터 첫 실행을 시작하는 반응시간과 실행을 끝마치기까지의 반환시간이 많은 차이를 보이며, 따라서 사용자는 컴퓨터의 사용용도에 따라 멀티코어 환경에서 적절한 스케줄러를 선택하여야 한다.

만약 사용자가 주로 사용하는 프로세스가 CPU에게 서비스 받는 시간이 짧고, I/O를 기다리는 시간이 빈번한 입출력 중심작업이라면, 반응시간이 작을수록 유리할 것이지만, 반대로 CPU 중심 작업이라면 반응성이 유리한 스케줄러보다는 전체 작업이 완료되는 반환시간이 중요한 요인이 될 것이다.

본 논문에서는 리눅스 커널 버전 2.6.23 이후로 현재까지 메인스트림 리눅스 커널에서 사용되고 있는 CFS(Completely Fair Scheduler)와 일반 사용자를 위하여 반응성을 향상한 BFS(Brain Fuck Scheduler)의 작동 알고리즘을 살펴보고, 멀티코어 환경에서 두 스케줄러의 성능을 분석하여 결과에 대한 그 원인에 대해서 기술한다.

II. 본론

2.1 리눅스 스케줄러 관련 연구

리눅스 커널 2.6.23 버전부터 등장한 CFS(Completely Fair Scheduler) 스케줄러는 성능을 저해하지 않으며, 기존 O(1) 스케줄러에서 프로세스의 우선순위에 절대적인 CPU 서비스 시간 할당으로 인한

프로세스간 불균형을 해소할 목적으로 설계되었다. 이에 따라 CFS와 O(1) 스케줄러를 비교하는 여러 논문[1]들과 벤치마킹 프로그램들이 소개되어[2], CFS가 설계 목적을 충족시켜주고 있음을 보이게 되었다.

그 이후 Con.Kolivas는 알고리즘을 단순화하여 반응성에 초점[3]을 둔 BFS(Brain Fuck Scheduler)스케줄러를 발표하였고, 논문[4]의 연구결과, 반응시간 측면에서 CFS보다 큰 향상이 있음을 알 수 있었다.

이외에도 스케줄링 알고리즘의 공정성 달성하기 위한 여러 연구들이 있어왔으며[5], 다양한 측면에서 스케줄러에 대한 연구가 진행되고 있다.

본 논문에서는 멀티 코어의 환경에서 CFS와 BFS를 비교하여본다.

2.2 CFS(Completely Fair Scheduler)

CFS는 프로세스의 우선순위에 따라 절대적인 수행시간을 할당하는 것이 아니라 상대적인 가중치를 이용하여 수행시간을 할당한다. 프로세스들의 가중치는 nice값에 기반을 둔 값으로, 다음과 같은 식을 이용되어 할당되는 수행시간이 정해진다.

$$ime\,slice = \frac{task.load}{cfs.rq.load} \times period$$

여기서 time slice는 프로세스에 할당 될 수행시간이고, period는 현재 CFS 실행 큐의 모든 프로세스들을 서비스 할 CPU의 단위시간이다. 즉 CPU의 단위시간 동안 CFS 실행 큐에 있는 모든 프로세스들의 가중치 합(cfs.rq.load)과 해당 프로세스의 가중치(task.load)의 상대적인 비에 따라 타임 슬라이스가 동적으로 할당된다. 이와 동시에 CPU의 단위 시간도 프로세스의 개수에 따라 동적으로 변화한다.[2] 이러한 동적 시간 할당을 통하여 CFS는 단위시간 내에 모든 프로세스들이 한 번씩 실행됨을 보장함으로써 공평성을 충족시킨다.

이러한 동적 시간 할당뿐만 아니라 CFS는 다음에 실행 될 프로세스를 virtual runtime라는 값을 인덱스로 하여 레드 블랙 트리(red-black tree)로 실행 큐를 구성한다는 특징을 가지고 있다. 스케줄링 시점이 되면, 레드 블랙 트리 실행 큐에서 virtual runtime값이 가장 작은 프로세스를 선택하여 실행한다.

큐의 인덱스로 사용되는 virtual runtime은 프로세스의 현재까지 누적시간을 가중치로 역 스케일링한 값으로 다음과 같이 정의된다.

$$T = \frac{task_0.load}{task.load} \times task.exec$$

여기서 task.exec는 프로세스의 현재까지 누적 실행시간이며, 이를 nice값이 0인 프로세스의 가중치(task_0.load)와 현재 프로세스의 가중치(task.load)의 비를 역 스케일링하여 virtual runtime을 구한다. 식에서 알 수 있듯이 가중치가 큰 프로세스일수록 vitrual runtime이 작다는 것을 알 수 있다. 그러므로 우선순위가 높은 프로세스는 낮은 프로세스보다 virtual runtime의 증가폭이 작으며, 이로써 우선순위가 큰 프로세스일수록 좀 더 오래 CPU를 점유할 수 있는 기회가 보장된다.

마지막으로, 멀티코어 환경에서의 CFS는 코어별로 실행 큐를 가지고 있다. 프로세스가 생성되면 코어의 상황과 캐쉬의 지역성에 따라 적절한 코어로 배치되며, 한 코어에서 실행 후 코어 별 로드의 양을 비교하여 다른 코어에 스케줄링 되기도 한다. 또한 주기적으로 코어별 로드의 불균형을 체크하여 여러 개의 프로세스를 한꺼번에 이동시키는 로드 밸런싱이 이루어지기도 한다.[5]

위와 같은 메커니즘들을 통하여 CFS는 각 프로세스가 우선순위에 근거하여 일정시간 내에 서비스 받을 수 있게 보장하며, 코어 간 로드의 불균형을 조절하여 공평한 스케줄링을 구현하였다.

2.3 BFS(Brain Fuck Scheduler)

BFS[3]는 하나의 시스템 상에 오직 하나의 실행 큐만을 가지고 있으며 매우 간단한 알고리즘으로 구성되어있다.

BFS는 단순하게 이중 연결 리스트로 큐를 구성하며, virtual deadline first 정책으로 다음에 실행 할 프로세스를 선택한다. virtual deadline이란 같은 nice값을 가진 프로세스 2개가 있을 때 하나의 프로세스가 실행되기 전까지 기다려야 하는 최대 시간 값으로 다음과 같이 정의 된다.

$$VD = jiffies + (user_priority \times rr_interval)$$

여기서 jiffies는 프로세스가 큐에 들어가는 시점의 시간이고, user_priority는 nice값에 기반을 둔 값이며 여기에 BFS가 프로세스에 할당하는 고정시간 rr_interval을 곱하여 구한다. Deadline에 virtual이 붙은 이유는 스케줄러가 virtual deadline의 값에 의거하여 다음에 스케줄 될 프로세스를 선택하지만, 그 deadline내에 반드시 스케줄 될 것을 보장하지 못하기 때문이다. 어쨌든 BFS는 작은 virtual deadline을 갖는 프로세스를 높은 것 보다 먼저 선택하고, 선택된 프로세스는 rr_interval만큼 할당되어 실행하게 되어, 큐에

일찍 들어오고, 우선순위가 높은 프로세스일수록 먼저 실행되게 된다.

멀티코어 환경에서도 BFS는 하나의 큐만을 유지하며, 스케줄 시점에서 가장 작은 virtual deadline을 갖는 프로세스는 현재 서비스를 받을 수 있는 코어가 있는지, 현재 실행중인 다른 프로세스를 선점할 수 있는지를 체크하고 전역 락(global lock)을 걸어 해당 코어에 의해 서비스 된다.

멀티코어 환경에서 코어를 선택하는 데에 있어 BFS는 캐쉬의 지역성보다는 유휴(idle)한 코어를 우선시한다.[6] 만약 이전에 서비스 받은 코어가 유휴상태가 아니라면 다른 유휴상태 코어에 우선적으로 서비스를 받고, 이전에 서비스 받은 코어와 다른 코어가 같이 유휴상태라면 이 시점에서는 캐쉬의 지역성을 고려하여 기존의 코어에서 서비스를 받게 된다.

위와 같이 시스템에 하나만 존재하는 전역 큐 구조로 인하여 lock contention문제가 발생하기도 하지만 Kolivas는 적은 코어수를 갖는 일반적인 데스크탑 환경에서는 그러한 페널티보다 간단한 프로세스 할당으로 인한 성능 향상이 더 크게 나타난다고 주장한다.

정리하면 BFS는 CFS에서의 동적 시간 할당, 레드블랙 트리의 균형 조절, 그리고 코어별 실행 큐 구조로 인한 복잡한 로드 밸런싱 등의 알고리즘을 탈피하여 간단한 큐 구조와 알고리즘을 사용함으로써 반응시간 측면에서 이점을 가졌다고 할 수 있다.

Ⅲ. 실험

실험은 CFS와 BFS의 반응시간(Response Time)과 반환시간(Turnaround Time)을 측정하며, 그 정의는 다음 그림과 같다. 반응시간은 프로세스가 실행 큐에 들어간 시점으로부터 디스패치되어 CPU에게 서비스 받는 것을 시작하기까지의 시간이며, 반환시간은 프로세스의 작업이 모두 끝날 때까지의 시간이다.

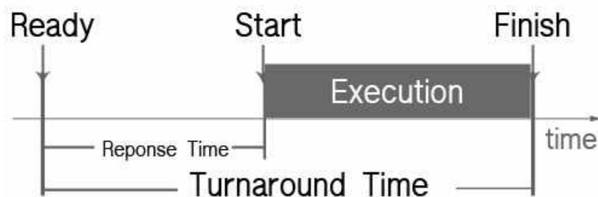

그림 1. 반응시간과 반환시간

두 스케줄러를 비교하기 위한 사양은 다음과 같다 : Intel Core 2duo Q9550, 4GB RAM, Ubuntu Linux 11.10(Kernel 3.2) with CFS, Ubuntu Linux 11.10 with BFS(3.1-sched-bfs-414.patch)

실험은 latt.c 벤치마킹 툴[5]을 이용하였다. latt.c는 자식프로세스(Clients)를 지정한 수만큼 생성하고, 사용자가 지정한 임의의 부하(Work load) 하에 자식프로세스들이 무작위로 데이터를 압축을 여러 번 시행하여, 자식 프로세스의 반응시간과 반환시간의 평균값을 출력한다.

본 실험은 임의의 부하로 3초간 sleep을 로드로 주었다.

다음 그래프는 CFS와 BFS의 반응시간 비로, CFS의 값에서 BFS의 값을 나눈 결과이다.

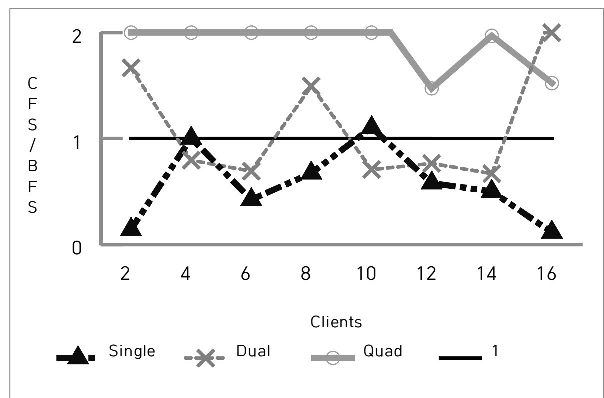

그림 2. 코어 수에 따른 CFS와 BFS의 반응시간 비

실험결과, 반응시간의 경우 싱글 코어에서는 CFS가 BFS보다 대체적으로 유리한 결과를 보였다. 듀얼 코어에서는 반응시간의 격차가 싱글 코어보다 줄어들고 쿼드 코어에서는 BFS가 CFS를 역전하여 확연하게 유리하다는 것을 알 수 있다.

다음 그래프는 CFS와 BFS의 반환시간의 비로, 역시 CFS의 값에서 BFS의 값을 나눈 결과이다.

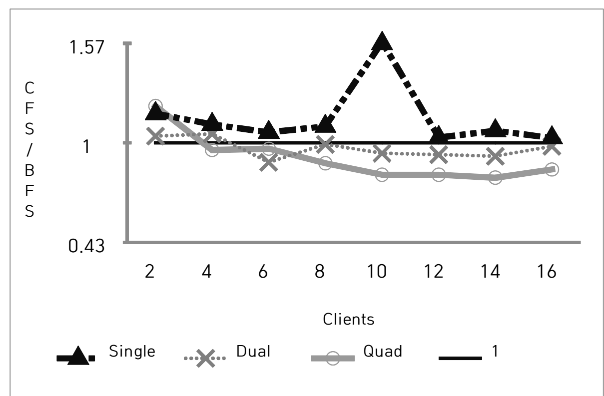

그림 3. 코어 수에 따른 CFS와 BFS의 반환시간 비

반환시간의 경우 단일 코어 환경에서는 큰 차이가 없었지만 BFS가 근소하게 유리하였고 듀얼 코어나 쿼드 코어 환경에서는 CFS가 근소하게 유리하게 측정되었다.

이에 커널 트레이싱 툴 trace-cmd를 이용하여 latt.c가 실행되는 동안 커널의 스케줄링 이벤트를 트레이싱 해보았다.

스케줄링 이벤트 발생 빈도를 확인한 결과, 단일 코어의 경우 큰 차이가 없었지만, 듀얼 코어와 쿼드 코어환경에서 각 프로세스의 코어 이동횟수에서 큰 차이를 보였다. 결과 그래프는 다음과 같으며, BFS에서의 프로세스들의 평균 코어 이동 횟수를 CFS의 값으로 나눈 값이다.

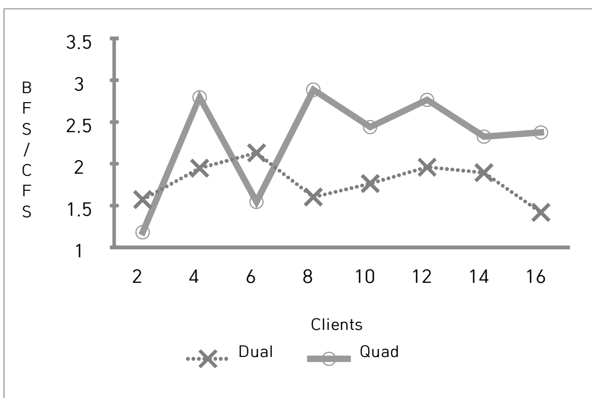

그림 4. 코어 수에 따른 프로세스의 코어 이동 횟수 비

듀얼 코어의 경우 최대 2배까지 BFS가 코어 이동이 잦았으며, 쿼드 코어의 경우에는 최대 3배에 가깝게 나타났다.

그 이외의 스케줄링 이벤트 수는 크게 차이나지 않았다.

Ⅳ. 결론 및 향후 연구 방향

두 스케줄러 CFS와 BFS를 비교한 결과, 반응시간은 싱글 코어 환경에서 CFS가 BFS보다 근소하게 유리하였지만 코어 수가 늘어감에 따라 BFS가 점차 CFS를 따라잡는 양상을 보였다. 이와 함께 반환시간의 경우 싱글 코어에서는 BFS가, 멀티 코어 환경에서는 CFS가 유리한 경향을 보였다.

이 결과는 코어 수가 늘어감에 따라 프로세스가 시작하는 시간은 BFS가 더 빠르지만, 실질적인 실행시간은 더 길어졌다는 것을 의미한다.

커널 트레이싱 결과, 멀티 코어 환경에서 나타난 코어 이동횟수 차이가 반응시간의 역전현상을 보이는 요인으로 작용함을 확인하였고, 이로 미루어보았을 때, BFS는 유휴상태 코어를 바로 사용하여 스케줄을 하고, CFS는 한 코어에 좀 더 오랫동안 머물러, 다른 코어를 바로 이용하지 않는 다는 점에서 반응시간이 손해라는 것을 알 수 있다.

이는 BFS를 설계원리 중 유휴상태인 코어를 이용하는 것이 캐쉬의 지역성을 이용하는 것보다 반응시간의 향상에 큰 영향을 미친다는 것의 결과[6]이다.

그러나 BFS는 빈번하게 새로운 코어에 스케줄 됨에 따라 캐쉬 miss로 인한 miss penalty로 CFS보다 실제 실행시간이 길어져, 반환시간에서는 손해를 본다는 것을 알 수 있다.

결론적으로 멀티코어 환경에서, 반응성이 중요시 되는 I/O중심의 프로세스는 반응시간이 작은 BFS를 사용하는 것이 유리하다 할 수 있으며, 반면에 반응성보다 CPU에 할당 될 시간이 많아야 하는 계산 중심의 프로세스는 반환시간이 작은 CFS가 유리하다고 할 수 있겠다. 이로써 사용자는 주로 사용할 프로세스의 성향에 따라 두 스케줄러 중 적절한 것을 선택함으로써 효율을 높일 수 있을 것이다.

참고문헌

[1] C.S. Wong, I.K.T. Tan, R. D. Kumari, J. W. Lam, W. Fun, Fairness and Interactive Performance of O(1) and CFS Linux Kernel Schedulers, Information Technology ITSim, 2008

[2] 김용수, 리눅스 스케줄링 알고리즘 CFS를 위한 시뮬레이션 모델, 한국정보기술학회논문집 제 9권 제 7호, 2011.

[3] http://ck.kolivas.org/patches/bfs/bfs-faq.txt

[4] Tyalor Groves, Jeff knockel, Eric Schutlte, BFS vs CFS - Scheduler Comparison, 11, December , 2009.(http://www.cs.unm.edu/~eschulte/data/bfs-v-cfs_groves-knockel-schulte.pdf)

[5] 허승주, 유종훈, 홍성수, 멀티코어 시스템에서 공정성 보장을 위한 가상런타임 기반 로드 밸런싱 알고리즘, 정보과학회논문지 : 컴퓨팅의 실제 및 레터 제 18 권 제 4 호, 2012

[6] http://ck.kolivas.org/patches/3.0/3.1/3.1-ck2/patches/3.1-sched-bfs-414.patch