멀티코어 시스템에서 공정성 향상을 위한 Virtual Runtime 기반 로드 밸런싱 메커니즘

허승주 O1), 유종훈 2), 홍성수 1), 2)

1) 서울대학교 융합과학기술대학원 지능형 융합시스템 학과, 2) 전기컴퓨터공학부

{sjhuhO, jhyoo, sshong}@redwood.snu.ac.kr

Virtual Runtime based Load Balancing for Improving Fairness on Multicore Systems

Sungju Huh, Jonghun Yoo, Seongsoo Hong

1) Department of Intelligent Convergence Systems, Graduate School of Convergence Science and Technology

2) School of Electrical Engineering and Computer Science, Seoul National University

요 약

리눅스의 기본 스케줄러인 CFS는 리눅스의 보급과 더불어 기업용 서버와 클라우드 서버에서 널리 사용되고 있다. 그런데 시스템의 규모가 극도로 커짐에 따라 CFS에 의해 스케줄링된 태스크들은 종종 심각한 기아 현상을 겪는다고 알려져 있다. 이 논문은 CFS에 대한 심도 깊은 분석을 통해 이러한 문제의 원인이 멀티코어에서 공정성 보장의 실패에 있음을 규명한다. 그리고 이를 해결하기 위한 virtual runtime 기반의 로드 밸런싱 메커니즘을 제안한다. 제안된 메커니즘은 태스크들의 진행 속도를 균등하게 만들기 위해 주기적으로 태스크 이주를 수행한다. 우리는 제안된 메커니즘을 리눅스 커널 상에 구현하고 일련의 실험을 수행하였다. 그 결과 기존의 CFS로 스케줄링된 태스크들의 virtual runtime 차이는 선형적으로 증가하는데 반해 제안된 기법은 virtual runtime의 차이를 상수로 바운드할 수 있음을 보였다.

1. 서 론

CFS(complete fair scheduler)[1]는 2.6.23 커널 이래로 리눅스의 주된 태스크 스케줄러로서 사용되어 왔다. CFS의 목적은 태스크들의 가중치(weight)에 비례하여 CPU 시간을 할당해 주는 것이다. 일반적인 작업 환경에서 이는 기존 스케줄러에 비해 높은 반응성과 공정성을 보이기 때문에 CFS는 모바일 장치에서부터 대규모의 클라우드 서버까지 다양한 분야의 컴퓨팅 시스템에서 사용되고 있다.

불행히도 CFS는 시스템의 규모가 극도로 커짐에 따른 scalability 지원이 미흡한 것으로 알려져 있다. 특히 수천에서 수만 개의 태스크가 공존하는 시스템에서 CFS에 의해 스케줄링되는 태스크들은 종종 심각한 기아현상을 겪는다고 보고된 바 있다 [2]. 그럼에도 불구하고 CFS에 대한 분석적이고 비평적인 평가는 그리 많지 않으며 대규모 시스템에서 scalability 지원 미흡의 원인을 명확하게 설명하지 못한다 [3][4].

본 논문에서 우리는 CFS를 명확하게 분석하여 멀티코어에서 다양한 가중치를 갖는 태스크들을 처리할 때 CFS가 공정성을 보장하지 못함을 보인다. 우리의 분석 결과는 CFS로 스케줄링된 태스크들이 멀티코어 환경에서 예측할 수 없는 CPU 시간 분포를 갖는 이유를 설명한다. 분석에 기반하여 우리는 virtual runtime에 기반한 새로운 로드 밸런싱 메커니즘을 제안한다. 이 메커니즘은 주기적으로 태스크들을 이주하여 임의의 두 태스크의 virtual runtime 차이를 상수로 바운드한다.

이 논문의 나머지 부분은 다음과 같이 구성된다. 2장에서는 CFS의 동작원리를 명확히 분석한다. 이어서 3장은 멀티코어에서 CFS가 공정성을 성공적으로 달성하지 못하는 문제점을 보인다. 4장에서는 이러한 문제를 해결하기 위한 virtual runtime 기반 로드 밸런싱 알고리즘을 설명한다. 5장은 제안된 메커니즘의 구현과 실험 결과를 보인다. 마지막으로 6장에서는 논문의 결론을 내린다.

2. CFS의 분석

CFS는 태스크의 가중치를 이용하여 공정성을 달성하는 스케줄링 알고리즘이다. 태스크의 가중치는 시스템 관리자에 의해 각 태스크에 부여되는 nice 값에

의해 정해진다. Nice 값은 -20과 19 사이의 정수로서 작은 nice 값은 큰 가중치에 상응한다.

CFS는 SMP(symmetric multi-processor)를 지원하기 위해 각 CPU마다 태스크의 실행큐를 유지한다. 실행큐 내의 태스크들은 각 태스크에 부여된 virtual runtime이 증가하는 순서로 정렬된다. 이때 virtual runtime은 태스크의 가중치에 의해 역으로 스케일링된 누적 실행 시간이다. 구체적으로, CFS에서 시간 t에 태스크 τ_i의 virtual runtime $VR(\tau_i, t)$은 아래와 같이 정의된다.

$$VR(\tau_i,t) = \frac{\omega_0}{\omega_i} \times PR(\tau_i,t)$$

여기서 ω_0은 nice 값 0의 가중치를, ω_i는 태스크 τ_i의 가중치를 의미한다. 또한 $PR(\tau_i,t)$는 시간 t일 때 태스크 τ_i의 실제 누적 실행시간을 의미한다.

CFS는 실행큐 내의 모든 태스크들에게 가중치에 비례하는 time slice를 할당하여 태스크가 이 기간 동안 선점 당하지 않고 실행될 수 있게끔 한다. CFS에서 태스크 τ_i의 time slice TS_i는 아래와 같이 계산된다.

$$TS_i = \frac{\omega_i}{\sum_{j\in\varphi}\omega_j} \times P$$

여기서 φ는 실행큐 내에 있는 실행 가능한 태스크들의 집합을 의미하고, P는 주어진 작업량에 대해서 상수이다. P는 다음과 같이 정의된다.

$$P = \begin{cases} sysctl\_sched\_latency & \text{if } n < nr\_latency \\ min\_granularity \times n & otherwise \end{cases}$$

여기서 n은 실행큐 내의 실행 가능한 태스크의 개수를 의미한다. $sysctl\_sched\_latency$, $nr\_latency$, $min_granularity$는 시스템 전체에 대한 상수이며 현재 리눅스 상에는 6, 8, 0.75로 정의되어 있다.

태스크가 자신에게 할당된 time slice를 소진하면, NEED_RESCHED flag가 설정된다. 스케줄링 주기마다 CFS는 현재 수행 중인 태스크의 virtual runtime을 갱신하고 NEED_RESCHED flag를 확인한다. 만약 이 flag가 설정되어 있으면 CFS는 현재 실행 중인 태스크를 선점하고 실행큐 내의 가장 작은 virtual runtime을 갖는 태스크를 스케줄링 한다.

CFS는 멀티코어 환경에서 보다 효율적으로 CPU 자원을 활용하기 위해 로드 밸런싱을 수행한다. CFS는 실행큐 마다 load 값을 유지하여 시스템 내의 실행큐에 대해 균형을 맞춘다. 실행큐 Q_i의 load는 아래와 같이 정의된다.

$$load_{Q_i} = \sum_{j\in\varphi_i}\omega_j$$

여기서 φ_i는 실행큐 Q_i 내의 실행 가능한 태스크들의 집합을 의미한다.

CFS가 로드 밸런싱을 수행하는 시점은 다음과 같다: (1) 실행큐 내의 태스크들이 모두 수행을 종료하여 실행큐가 유휴 상태가 되었을 때, (2) 태스크가 새롭게 생성되거나(fork, exec) 수면 혹은 대기 상태에서 깨어날 때(wakeup), (3) 서로 다른 실행큐 사이에 load의 불균형이 발생했을 때. 첫 번째 경우, 실행큐 Q_i가 유휴 상태가 되었을 때 CFS는 load가 가장 큰 실행큐 $Q_{busiest}$를 찾아 태스크들을 Q_i로 이주시킨다. 이 때 CFS는 이주시킨 결과가 더 큰 불균형을 야기하지 않을 만큼의 태스크들을 한번에 이주시킨다. 두 번째 경우, CFS는 새로 생성되거나 깨어난 태스크 τ_i를 load가 가장 작은 실행큐 Q_{idlest}에 이주시킨다. 이때 태스크 τ_i의 virtual runtime은 이주된 Q_{idlest}에서 가장 작은 virtual runtime보다 조금 작은 값으로 설정되어 τ_i가 빠른 응답시간을 가질 수 있도록 보장한다. 세 번째 경우, CFS는 실행큐 Q_i에 대해서 특정 주기마다 load가 가장 큰 실행큐 $Q_{busiest}$로부터 태스크들을 가져올 수 있게 한다. 첫 번째 경우와 마찬가지로 이주시킨 결과가 더 큰 불균형을 야기하지 않을 만큼의 태스크를 한번에 이주시키며 로드 밸런싱을 수행하는 주기는 Q_i와 $Q_{busiest}$의 캐시 locality에 따라 다르다. 현재 리눅스 상에는 최소 1분에 한번씩은 주기적인 로드 밸런싱을 수행하도록 정의되어 있다.

3. 멀티코어 환경에서 CFS의 문제점

본 장에서는 CFS가 멀티코어에서 공정성을 보장하지 못함을 명확히 보인다.

임의의 시간 구간 $[t_1, t_2]$에서 태스크 τ_i의 이상적인 실행시간은 다음과 같다.

$$IR_{\tau_i}(t_1,t_2) = \frac{\omega_i}{\sum_{j\in\Phi}\omega_j} \times (t_2 - t_1) \times M$$

여기서 Φ는 시스템 전체의 실행 가능한 태스크들의 집합을 의미하고 M은 CPU의 개수를 나타낸다.

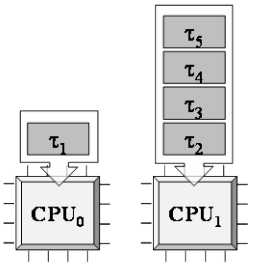
반면 CFS는 임의의 구간 $[t_1, t_2]$에서 태스크 τ_i에게 다음의 실행시간을 할당한다.

$$R_{\tau_i}(t_1,t_2) = \frac{\omega_i}{\sum_{j\in\varphi_i}\omega_j} \times (t_2 - t_1)$$

여기서 φ_i는 τ_i가 속한 실행큐 내의 태스크들의 집합을 의미한다. 계산의 단순화를 위해 (t_2-t_1)은 P의 배수라고 가정한다.

CFS가 멀티코어에서 공정한 스케줄링을 하기 위해서 모든 실행큐의 load가 모두 동일해야 한다. 하지만 현실적으로 태스크의 개수와 가중치는 다양하기 때문에 모든 실행큐의 load를 동일하게 만드는 것은 어렵다. 그림 1은 멀티코어에서 CFS가 공정성을 보장하지 못하는 예를 보인다. 이 예에서 CPU의 개수는 두 개이고 태스크들은 CFS의 로드 밸런싱 정책에 따라 최대한 균등하게 분포되어 있다. 태스크들은 무한 루프를 수행하는 CPU 바운드 태스크들이다. 이 경우

Task	τ_1	τ_2	τ_3	τ_4	τ_5
Nice	0	5	5	5	5
Weight	1024	335	335	335	335



(a) 태스크 집합의 명세와 초기 분포 상태

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2920 sjhuh     20   0  2752 1316  948 R  0.7  0.3   0:07.32 top
 2914 sjhuh     25   5 42912  276  216 R 23.8  0.1   8:31.07 test
 2913 sjhuh     25   5 42912  276  216 R 24.1  0.1   8:29.78 test
 2912 sjhuh     25   5 42912  276  216 R 24.1  0.1   8:25.32 test
 2911 sjhuh     25   5 42912  276  216 R 24.1  0.1   8:23.76 test
 2910 sjhuh     20   0 42912  276  216 R 88.6  0.1  32:21.08 test
```

(b) Linux **top** 명령어를 통한 CPU 시간 확인

그림 1. 멀티코어에서 CFS의 공정성 보장 실패의 예

어떤 태스크를 어떤 CPU로 이주시켜도 더 큰 불균형을 야기하기 때문에 CFS는 더 이상 로드 밸런싱을 수행하지 않는다. 결과적으로 태스크 τ_1는 나머지 태스크들보다 3배 더 높은 가중치를 가졌음에도 불구하고 실제로는 4배 더 많은 CPU 시간을 할당 받게 된다. 다시 말해서, τ_1를 제외한 나머지 태스크들은 자신에게 필요한 CPU 시간을 할당 받지 못하여 반응이나 처리가 늦어지는 결과를 초래한다.

4. Virtual Runtime 기반 로드 밸런싱

이 장에서 우리는 CFS가 멀티코어에서도 공정성을 보장할 수 있도록 만들기 위한 virtual runtime 기반 로드 밸런싱을 제안한다.

제안된 로드 밸런싱 알고리즘의 개괄은 그림 2와 같다. 이 알고리즘은 매 λ마다 주기적으로 수행된다. 알고리즘의 목적은 임의의 두 태스크 (τ_i, τ_j)에 대해서 virtual runtime의 차이가 상수로 바운드될 수 있도록 태스크를 CPU에 할당하는 것이다. Virtual runtime의 정의에 의해서 태스크들이 자신의 가중치에 비례한 만큼 CPU 시간을 할당 받았다면 모든 태스크들의 virtual runtime은 동일하다. 따라서 virtual runtime의 차이를 상수로 바운드 시키는 것은 공정한 스케줄링의 달성을 의미한다. 이를 위해 제안된 알고리즘은 두 CPU의 실행큐 내의 태스크 집합을 취합하여 virtual runtime이 큰 순서대로 정렬한다. 정렬된 태스크 집합 Φ'는 PARTITION 알고리즘을 통해 진행이 빠른 태스크 집합 φ_{fast}와 진행이 느린 태스크 집합 φ_{slow}로 분할된다. 이 때 φ_{fast}의 가중치 합은 시스템 전체의 평균 가중치 합보다 크도록, φ_{slow}의 가중치 합은 시스템 전체의 평균 가중치 합보다 작도록 분할한다. 이러한 분할은 로드 밸런싱을 수행한 후에 φ_{fast}에 속한 태스크들의 진행을 느리게 하고 φ_{slow}에 속한 태스크들의 진행을 빠르게

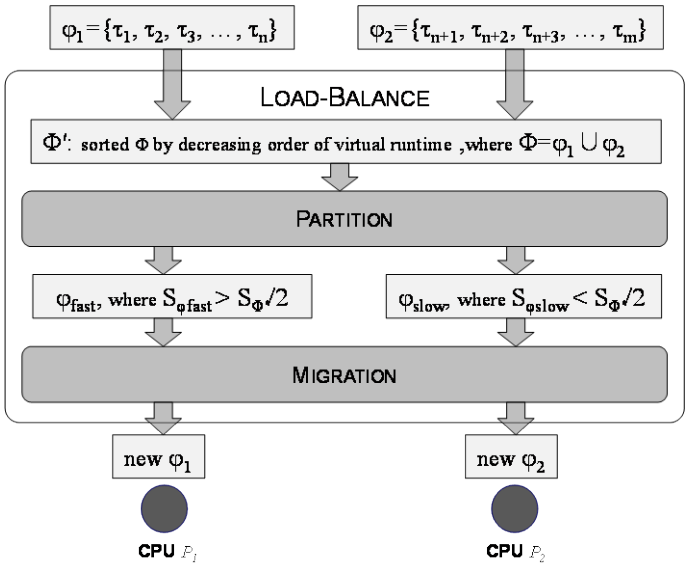


그림 2. Virtual Runtime 기반 로드 밸런싱의 개괄

한다. 이는 어떠한 임의의 두 태스크에 대해서 특정 상수 값 이상 virtual runtime의 차이가 벌어지지 않도록 보장한다. PARTITION 알고리즘을 통해 얻은 φ_{fast}와 φ_{slow}는 MIGRATION 알고리즘을 통해 각 태스크 집합이 어떤 CPU에서 수행될지 결정된다. MIGRATION 알고리즘은 태스크 이주 횟수를 최소화할 수 있도록 태스크 이주를 수행시킨다. 그림 3은 제안된 알고리즘을 상세히 설명한다.

Input:
Φ: Set of tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$,
W: $\Phi \rightarrow \mathbf{R}$ (weights of tasks)
φ_1, φ_2: Set of tasks running in CPU 1 and 2 ($\varphi_1 \cap \varphi_2 = \phi$, $\varphi_1 \cup \varphi_2 = \Phi$)

LOAD-BALANCE(Φ,W,φ_1,φ_2)
1 $\Phi' \leftarrow$ sort Φ by decreasing order of virtual runtime
2 (φ_{slow}, φ_{fast}) $\leftarrow$ **PARTITION** (Φ',W)
3 **MIGRATION** (φ_1,φ_2,φ_{slow},φ_{fast})

PARTITION(Φ',W)
1 $\varphi_{slow} \leftarrow \phi$, $\varphi_{fast} \leftarrow \phi$,
2 List L $\leftarrow \Phi'$
3 Integer S = 0
4 **WHILE** S < $(1/2)\Sigma_{k\in\Phi'}$W(k)
5 t $\leftarrow$ remove head of L
6 S = S + W(t)
7 $\varphi_{slow} \leftarrow \varphi_{slow} \cup \{t\}$
8 $\varphi_{fast} \leftarrow$ tasks in L

MIGRATION(φ_1,φ_2,φ_{slow},φ_{fast})
1 $\delta_{f-1} \leftarrow \varphi_{fast} - \varphi_1$, $\delta_{f-2} \leftarrow \varphi_{fast} - \varphi_2$, $\delta_{s-1} \leftarrow \varphi_{slow} - \varphi_1$, $\delta_{s-2} \leftarrow \varphi_{slow} - \varphi_2$
2 **if** $|\delta_{f-1}+\delta_{s-2}| > |\delta_{f-2}+\delta_{s-1}|$ **then**
3 migrate δ_{f-2} from CPU 1 to CPU 2
4 migrate δ_{s-1} from CPU 2 to CPU 1
5 **else**
6 migrate δ_{s-2} from CPU 1 to CPU 2
7 migrate δ_{f-1} from CPU 2 to CPU 1
8 **endif**

그림 3. Virtual Runtime 기반 로드 밸런싱 알고리즘

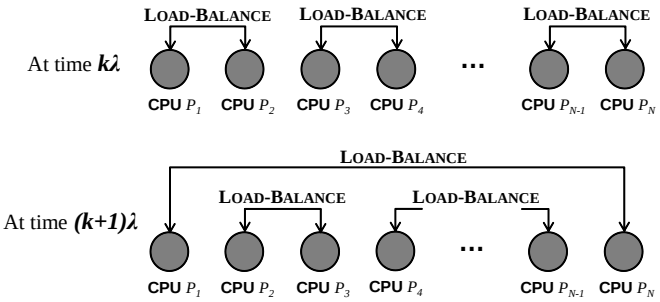


그림 4. 제안된 알고리즘의 N개 CPU로의 확장

이 알고리즘은 2개의 CPU들를 위한 로드 밸런싱을 지원한다. 우리는 제안된 알고리즘이 N개의 CPU들을 지원할 수 있게 하기 위해서 CPU들을 로드 밸런싱 시점마다 교차적으로 짝을 짓는다. 그림 3에서 보듯, $k\lambda$일 때와 $(k+1)\lambda$일 때 서로 다르게 짝을 지어 N개의 CPU에서도 어떠한 두 태스크 쌍의 virtual runtime의 차이가 상수로 바운드 될 수 있게 한다.

5. 실험 및 검증 결과

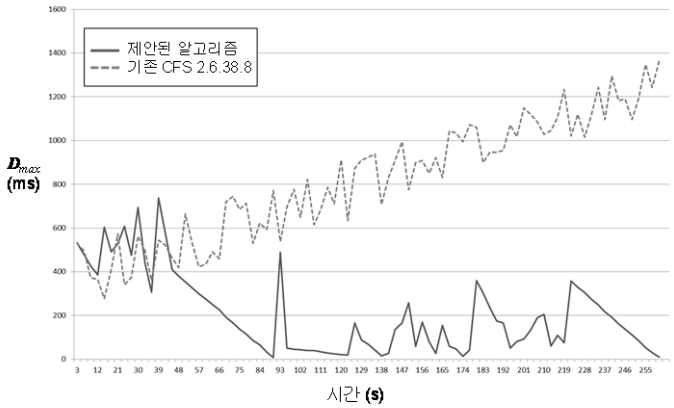
제안된 로드 밸런싱 알고리즘의 성능을 평가하기 위해서 실험을 수행하였다. 우리는 2-CPU로 구성된 데스크톱과 8-CPU로 구성된 서버에서 리눅스 2.6.38.8 커널에서 실험하였다. 제안된 알고리즘의 공정성을 평가하기 위해 우리는 임의의 두 태스크의 최대 virtual runtime 차이 $D_{max}(t)$을 측정하였다. $D_{max}(t)$는 아래와 같이 정의된다.

$$D_{\max}(t) = \max_{\tau_i,\tau_j \in \Phi}(|VR(\tau_i,t) - VR(\tau_j,t)|)$$

그림 4는 실험에 사용한 태스크 집합의 명세와 시간의 흐름에 따른 $D_{max}(t)$의 값을 확인한 결과이다.

Task	τ_0	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9
Nice	0	0	6	1	3	3	5	2	1	0
Weight	1024	1024	272	820	526	526	335	655	820	1024

(a) 태스크 집합의 명세



(b) 제안된 알고리즘을 통한 virtual runtime의 차이 바운드 달성 결과

그림 6. 멀티코어에서 제안된 알고리즘의 공정성 달성

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2478 sjhuh     25   5 42912  276  216 R 23.1  0.0   3:05.99 test
 2477 sjhuh     25   5 42912  276  216 R 23.1  0.0   3:01.40 test
 2476 sjhuh     25   5 42912  276  216 R 23.1  0.0   3:04.45 test
 2475 sjhuh     25   5 42912  276  216 R 23.1  0.0   3:03.56 test
 2474 sjhuh     20   0 42912  276  216 R 98.7  0.0   9:24.77 test
 2473 sjhuh     20   0 42912  276  216 S  0.0  0.0   0:00.00 test
```

그림 5. 리눅스 top 명령어를 통한 CPU 시간 확인

로드 밸런싱 주기 λ는 3초로 설정하였다. 기존 리눅스 2.6.38.8의 CFS는 이 태스크 집합에 대해서 $D_{max}(t)$가 선형적으로 증가하는 반면 제안된 알고리즘을 적용한 CFS는 $D_{max}(t)$을 일정 수준 이하로 유지함을 알 수 있다.

그림 5는 제안된 알고리즘이 CFS가 공정성 보장을 실패했던 그림 1의 경우에도 성공적으로 공정성을 보장함을 보인다. 이 실험을 통해 다른 태스크들보다 가중치가 3배 높은 태스크 τ_1는 실제로 3배 더 많은 CPU 시간을 할당 받았음을 알 수 있다.

6. 결론

우리는 리눅스 CFS의 동작원리를 분석하여 CFS가 멀티코어에서 공정성의 측면에서 문제점이 있다는 것을 밝혔다. 분석에 기반하여 본 논문은 멀티코어 리눅스에서 보다 향상된 공정성을 위한 로드 밸런싱 알고리즘을 제안하였다. 제안된 알고리즘은 어떤 CPU에 있는 어떠한 태스크에 대해서 virtual runtime의 차이를 상수로 바운드한다. 우리는 실험 및 검증을 통해 제안한 알고리즘이 멀티코어에서 공정성을 향상시킴을 보였다. 향후 연구로서, 우리는 제안된 알고리즘을 확장하여 캐시 affinity를 고려한 태스크 이주를 지원 할 예정이다.

Acknowledgement

본 연구는 삼성전자 DMC 연구소의 지원을 받아 수행하였음. (No. 0418-20110011, HW SW 융복합 원천기술 개발)

참고 문헌

[1] I. Molnar. The Completely Fair Scheduler, http://people.redhat.com/mingo/cfs-scheduler/.

[2] http://www.mattheaton.com/?p=222.

[3] L. A. Torrey, J. Coleman, and B. Miller, “A comparison of interactivity in the Linux 2.6 scheduler and an MLFQ scheduler” Software: Practice and Experience, vol. 37(4), pp. 347~364, 2007.

[4] S. Wang, Y. Chen, W. Jiang, P. Li, T. Dai, and Y. Cui, “Fairness and Interactivity of Three CPU Schedulers in Linux” Proceeding of RTCSA, pp. 172~177, August, 2009.