공학박사학위논문

서버 시스템을 위한 공정하고 효율적인 스케줄링 알고리즘

A Fair and Efficient Scheduling Algorithm for Server Systems

2014년 2월

서울대학교 대학원
전기 · 컴퓨터 공학부
정 진 만

공학박사학위논문

서버 시스템을 위한 공정하고 효율적인 스케줄링 알고리즘

A Fair and Efficient Scheduling Algorithm for Server Systems

2014년 2월

서울대학교 대학원
전기 · 컴퓨터 공학부
정 진 만

서버 시스템을 위한 공정하고 효율적인 스케줄링 알고리즘

A Fair and Efficient Scheduling Algorithm for Server Systems

지도교수 조유근

이 논문을 공학박사 학위논문으로 제출함
2013년 11월

서울대학교 대학원
전기 · 컴퓨터 공학부
정 진 만

정진만의 공학박사 학위논문을 인준함
2013년 12월

위 원 장	신 현 식	(인)
부위원장	조 유 근	(인)
위 원	민 상 렬	(인)
위 원	박 근 수	(인)
위 원	홍 지 만	(인)

초 록

네트워크와 하드웨어의 발전으로 서버 시스템은 그 활용 범위가 넓어지면서 사용자의 요구가 다양해지며 동시에 높은 수준의 서비스 품질을 요구하고 있다. 이를 위해 서버 시스템은 서로 다른 요구사항을 포함하는 다양한 태스크들에게 한정된 자원을 공정하게 할당하고 대규모 요구에도 안정적으로 지원하도록 높은 효율성을 제공해야 한다. 하지만 기존 공정 스케줄링 알고리즘들은 편향 지분 분포를 가진 태스크 집합에 대해 태스크의 수가 커질수록 공정성이 낮거나 스케줄링 오버헤드가 커져 서버 시스템에 그대로 적용할 수 없다.

본 논문에서는 서버 시스템에서 공정하고 효율적인 스케줄링 알고리즘인 K-LZF를 제안한다. 새로운 태스크 만족 지수를 정의한 후, 그 기반에서 높은 공정성을 제공하며 $O(N)$의 시간 복잡도를 보이는 LZF 알고리즘을 제시하고 분석한다. 분석된 결과를 통해 높은 공정성을 유지하면서 상수 시간의 시간 복잡도를 갖도록 LZF 알고리즘을 근사화한다. 또한 K-LZF 알고리즘에서 K 값의 변화를 통해 서버 시스템의 용도에 따라 공정성과 효율성의 성능을 조절할 수 있다. 다양한 분포를 가진 태스크 집합에 대한 시뮬레이션 결과들은 K-LZF 알고리즘이 대규모 편향 지분 분포에서도 LZF 알고리즘에 근사한 우수한 성능을 제공함을 보인다. 또한, AVOS 운영체제와 LINUX 운영체제에 구현하여 성능 평가한 결과는 K-LZF 알고리즘의 시간 복잡도가 서버 시스템에서 효율적으로 동작하기 적합한 $O(K)$임을 보인다. 이러한 결과들을 통해 다양한 종류의 태스크들이 혼재하거나 태스크 수가 많아지더라도 K-LZF는 확장성을 제공하여 서버 시스템의 스케줄링 알고리즘으로 적합하면서도 우수한 성능을 제공함을 입증한다.

주요어: 서버 시스템, 자원 할당, 태스크 스케줄링, 공정 스케줄링, 공정성, 확장성

학번: 2008-20975

목 차

표 목 차

그 림 목 차

제 1 장 서론

1.1 연구 배경

최신 무선 네트워크 기술의 발전으로 서버 시스템 내에서 그 활용 범위가 넓어지면서 서버 시스템은 더 높은 수준의 공정성과 효율성이 강조되고 있다 [LPB13]. 특히 서버 시스템들은 다양한 유무선 통신 기술을 접목하여 서로 연동 및 공유되면서 더욱 다양화되고 지분, 자원 예약률, 실행 시간 등의 서로 다른 요구사항을 포함하게 되었다 [ASMH00]. 이러한 다양한 태스크들이 서버 시스템 내에 혼재하는 경우 자원 할당 문제는 더욱 어렵게 된다. 한정된 자원을 다양한 태스크들에게 차등화여 할당할 수 있는 동시에 서버 시스템은 대규모 서비스 요구에도 안정적으로 지원하도록 효율성을 제공해야 한다 [AMB11, NAR+11].

이러한 높은 수준의 공정성과 효율성의 요구는 임베디드 서버 시스템부터 아마존 EC2 같은 대규모 서버 시스템까지 모두 중요한 문제이다. 많은 임베디드 서버 시스템은 홈 / 빌딩 시스템, 카 인포테인먼트(car infortainment) 시스템 등에 응용되면서 다양한 태스크들에게 차등화된 서비스를 지원하기 위해 한정된 자원의 효율적 관리가 더욱 중요해지고 있다. 예를 들어, 홈 / 빌딩 서버 시스템에는 멀티미디어 서비스, 자동 제어 서비스, 보안 감시 서비스 등의 다양한 서비스를 통합적으로 제공한다 [AS, RB07]. 멀티미디어 스트리밍을 제공하기 위한 데몬 태스크들 뿐만 아니라 자동 제어, 화재 경보, 침입 방지을 위한 많은 데몬 태스크들도 단일 물리 서버 내에서 혼재하여 동시에 동작하고 있다. 이러한 임베디드 서버 시스템은 태스크의 시간

요구사항에 맞게 적시에 처리될 수 있도록 공정하게 한정된 자원을 할당해야 한다. 또한 아마존 EC2와 같은 클러스터 기반 서버 시스템의 경우, 물리적인 서버 컴퓨팅 자원에 대한 다중 가상 머신 인스턴스를 제공해야 하므로 공정한 자원 분배는 비용과 직결된다. 이러한 종류의 클러스터 기반 서버 시스템에서는 사용자의 과금 정도에 따라 예측 가능하게 컴퓨팅 자원을 할당해야 한다.

최신 Window NT 계열 및 MAC OS X 과 같은 범용 운영체제들은 멀티레벨 피드백 큐(multi-level feedback queue) 스케줄링 기법을 사용한다 [RSI09]. 태스크 관리를 위해 $0 \sim 31$의 우선순위를 정해 놓고 우선순위 기반으로 자원을 할당한다 [NSM10]. 이 우선순위 기반의 기법에서는 처리기 자원에 대한 시간할당량이 우선순위 차이에 정확히 비례하지 않기 때문에 주어진 시간 동안 태스크에게 할당할 컴퓨팅 자원의 양을 정확히 예측할 수 없어 높은 공정성을 기대하기 어렵다 [KLL97, RJS00]. 또한 한정된 우선순위는 제한된 서비스 차별화 단계를 제공하므로 다양한 서비스 품질을 제공하는데 적합하지 않다. 리눅스 운영체제에서는 CFS(complete fair scheduler) [Pab09]가 2.6.23 이후에 적용되었다. CFS에서도 태스크들의 지분은 태스크들에게 부여된 -20에서 19까지 정수 범위의 nice 값에 의해 결정되므로 다양한 서비스 품질을 제공하는데 한계가 있다. 또한 이 알고리즘에서는 태스크들의 누적 실행 시간을 각 태스크의 지분 비율에 의해 정규화한 가상 런타임(virtual runtime) 값을 유지하여 스케줄링이 필요한 시점에 실행큐에서 가장 작은 가상 런타임 값을 갖는 태스크를 선택한다 [NSM10]. 이를 위해 CFS는 레드-블랙 트리(red-black tree)를 사용하는데 스케줄링 시점마다 $O(logN)$의 시간 복잡도를 보인다 [CRC13].

높은 공정성을 보장하기 위하여 많은 공정 스케줄링 알고리즘들이 제안되었다 [Nag85, SV95, NVZ01, DKS89, BZ96, Zha91, WW94, SAWJ+96, GVC96, SV98, RP03, CCN+05]. 공정 스케줄링 알고리즘들은 대부분 태스크들에게 지분(share),

티켓 (ticket), 크레딧(credit)이라는 가중치 속성을 태스크마다 각각 정의하고 이 지분에 비례하여 자원을 할당하는 방법을 사용한다 [CAGS00, CCN+05].

공정 스케줄링 알고리즘에서 가장 중요한 평가 기준 중 하나는 공정성(fairness)으로서 얼마나 정확하게 그 태스크 지분에 비례하여 자원을 서비스 해줄 수 있는가에 있으며, 일반적인 경우 기대 실행 시간과 실제 실행 시간과의 차이인 서비스 시간 오차(service time error)로서 평가된다 [SAWJ+96, CAGS00, NVZ01, SCK06, SWH+13]. 실제 구현에서는 모든 태스크들에게 항상 서비스 시간 오차를 0으로 하는 것은 불가능하다. 근본적인 문제는 컴퓨팅 자원은 퀀텀(quantum)과 같은 이산적인 단위 시간으로 서비스하기 때문에 임의의 시간 구간에서 사용자들이 요구한 자원의 지분만큼 보장하기 어렵다는 점이다 [CCN+05]. 따라서 기존 공정 스케줄링 알고리즘들은 서비스 시간 오차를 최소화하는 것에 목표를 두고 있다.

하지만, 다양한 분포를 가진 태스크 집합으로 수행된 실험에서 기존 공정 스케줄링 알고리즘들은 지분 분포 편향될수록 공정성의 정확도가 낮거나, 태스크의 수가 커질수록 스케줄링 오버헤드(scheduling overhead)가 커져 서버 시스템에 그대로 적용할 수 없음을 확인하였다. 따라서 서버 시스템에서 높은 수준의 공정성과 효율성을 동시에 제공하는 새로운 공정 스케줄링 알고리즘이 필요하다.

1.2 연구 목적 및 범위

공정 스케줄링 알고리즘에서 태스크들의 시간적인 요구 사항은 지분을 통해 부여되므로 서버 시스템 환경에 따라 다양한 지분 분포를 가질 수 있다. 서버 시스템 특성상 상대적으로 높은 지분을 요구하는 연성 실시간성 태스크들이 존재하고, 낮은 지분을 요구하는 다양한 일반 태스크들이 혼재하는 경우 지분 분포는 고르지

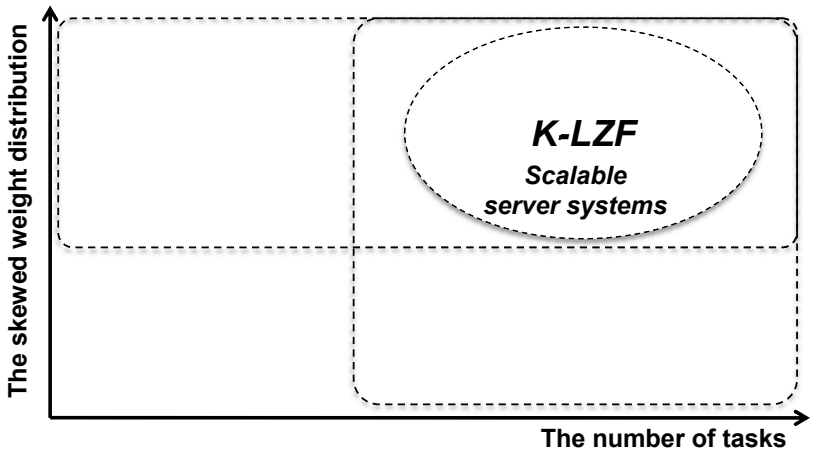


그림 1.1: 확장성 있는 서버 시스템을 위한 공정하고 효율적인 스케줄링 알고리즘

않고 편향적일 수 있다. 확장성 있는 서버 시스템을 위해 지분의 분포가 편향되는 상황에서 태스크의 수가 증가하더라도 높은 공정성을 효율적으로 제공해야 된다.

본 논문에서는 다양한 지분 분포가 기존 공정 스케줄링 알고리즘들의 공정성에 미치는 영향을 분석하였다. 분석한 결과, 태스크가 많고 그 지분 분포가 편향되어 있는 서버 시스템 환경에서 공정성이 악화되는 문제를 가지고 있다. 이러한 문제를 해결하기 위해 태스크들의 자원 할당 요구의 차이가 큰 태스크들을 다루는 서버 시스템에 적합한 스케줄링 연구가 필요하다. 본 논문의 목적은 그림 1.1과 같이 확장성있는 서버 시스템을 위해 태스크 수 및 지분 분포에 상관없이 높은 공정성을 유지하고 상수 시간의 시간 복잡도를 갖도록 스케줄링 알고리즘을 설계하는 것이다. 본 논문에서는 확장성 있는 서버 시스템을 위한 공정하고 효율적인 스케줄링 알고리즘 K-LZF를 제안한다.

따라서 단일 물리 서버 시스템 내에서 다양한 지분 분포를 가진 태스크들을 공정하고 효율적으로 스케줄링하기 위한 문제를 다루는 내용을 담고 있다. 제안된 스케줄링 알고리즘은 단일처리기 기반 서버 시스템과 다중처리기 기반 서버 시스템에

모두 적용 가능하다. 다중처리기 기반 서버 시스템에서 실행큐 모델은 이들의 구조에 따라 크게 전역 실행큐 [NVZ01, CCN+05]과 분산 실행큐 [CNS06, LBH09] 모델로 나뉜다. 전역 실행큐 모델에서 모든 태스크는 단일 실행큐에서 관리되어, 시스템 전체의 태스크들을 고려하여 중앙 스케줄러에 의해 스케줄링되므로 높은 공정성을 얻을 수 있다. 하지만, 이 방식은 처리기가 많은 경우 모든 처리기들이 동시에 이 전역 실행큐에 접근 할 수 있으므로 동기화 문제로 인한 성능 저하 문제가 발생할 수 있다. 반면, 분산 실행큐 모델에서는 각 처리기 별로 실행큐가 관리되므로, 동기화 문제가 덜 발생하며 캐쉬의 효과도 기대할 수 있다. 하지만 처리기간 부하 불균형 문제가 발생할 수 있으므로 부하 상황에 따라 효과적으로 부하 조절(load balancing)을 해야 하는 어려움이 있다. 제안 알고리즘인 K-LZF는 전역 실행큐로 사용되거나 분산 실행큐 모델에서 처리기별 실행큐로서 적용될 수 있다. 본 논문에서는 효과적인 공정성 분석을 위해 가상 시간 개념과 다른 새로운 스케줄링 기준으로 태스크 만족 지수(task satisfaction index, z)를 정의한다. 태스크 만족 지수 기반의 LZF 알고리즘을 제시하고 공정성과 효율성을 분석한다. LZF 알고리즘에 기반을 두어 높은 공정성을 유지하면서 K개의 리스트로 구성된 다중리스트를 사용하여 상수 시간의 시간 복잡도를 갖도록 LZF 알고리즘을 근사화한다. K-LZF 알고리즘에서 K 값의 변화를 통해 서버 시스템의 용도에 따라 공정성과 효율성의 성능을 조절할 수 있도록 한다. 본 논문에서는 다중처리기 기반 시뮬레이션 실험으로 제안 알고리즘 K-LZF의 공정성을 평가한다. 또한 서버 시스템에 활용 가능한 AVOS [KCK12] 운영체제와 LINUX 운영체제에서 구현하여 상수 시간의 시간 복잡도를 가지는 것을 확인한다.

1.3 논문의 구성

본문의 구성은 다음과 같다. 2장에서는 관련 연구로 공정 스케줄링 알고리즘을 동작 방식에 따라 가상 시간(virtual time) 기반, 라운드 로빈(round robin) 기반 및 혼합(hybrid) 기반 스케줄링 알고리즘의 세 가지로 분류하여 설명한다. 분류된 알고리즘들을 비교 분석한 후, 기존 알고리즘들의 문제점을 기술한다. 3장에서는 시스템 모델을 제시하고, 공정 스케줄링 알고리즘을 평가하기 위한 기준에 대해 논한다. 새로운 스케줄링 기준으로 태스크 만족 지수를 정의 한 후, 그 기반에서 높은 공정성을 제공하고 $O(N)$의 시간 복잡도를 보이는 LZF 알고리즘을 제시하고 분석한다. 4장에서는 높은 공정성을 유지하면서 상수 시간의 시간 복잡도를 갖도록 LZF을 근사화하는 K-LZF 알고리즘을 제안한다. 5장에서는 제안 알고리즘과 기존 공정 스케줄링 알고리즘들에 대한 시뮬레이션 및 구현 결과를 보인다. 서비스 시간 오차와 스케줄링 함수의 평균 실행 시간의 비교 분석을 통해 K-LZF 스케줄링 알고리즘의 성능을 확인할 수 있다. 마지막으로 6장에서는 결론을 맺고, K-LZF 스케줄링 알고리즘과 관련한 향후 연구 계획에 대해 설명한다.

제 2 장 관련 연구

본 장에서는 먼저 다양한 공정 스케줄링(fair scheduling)과 관련된 연구를 소개한다. 공정 스케줄링 알고리즘들의 목적은 각 태스크들에 포함된 지분(share), 티켓(ticket), 크레딧(credit)과 같은 미리 정의한 자원 명세를 통해 제공되는 자원의 비율에 비례하여 공정한 서비스를 보장하는 것이다. 공정 스케줄링 알고리즘은 각각의 동작 방식에 따라 크게 가상 시간(virtual time) 기반 알고리즘, 라운드 로빈(round robin) 기반 알고리즘, 그리고 혼합(hybrid) 기반 알고리즘으로 분류할 수 있다. 각 분류된 공정 스케줄링 알고리즘들을 비교 분석한 후, 서버 시스템에 적용하는데 발생가능한 기존 알고리즘들의 문제점들을 기술한다.

2.1 가상 시간 (virtual time) 기반 스케줄링 알고리즘

WFQ 알고리즘 [DKS89] 이후 Virtual-Clock [Zha91], EEVDF [SAWJ+96], SFQ [GVC96], 및 SPFQ [SV98] 등 많은 가상 시간 기반 알고리즘들이 연구되었다. 가상 시간(virtual time)은 태스크들의 지분에 의해 정규화된 누적 실행 시간을 의미한다. 가상 시간 기반 공정 스케줄링 알고리즘들은 가상 시간을 기반으로 과거에 서비스 받은 시간할당량들을 기억하여 상대적으로 가장 덜 서비스 받은 태스크를 다음 수행할 태스크로 선택하는 방법이다 [DKS89, WW94, BZ96]. 가상 시간 기반 알고리즘들의 특징은 최소의 가상 시간 값을 선택하는 문제와 관련이 있으므로 비교적 높은 공정성을 보이나 구현에 따라 $O(logN) \sim O(N)$의 시간 복잡도를 가진다. 따라서

태스크의 수가 작을 때 효과적이며, 태스크 수가 많은 서버 시스템에서는 적합하지 않다.

2.1.1 WFQ(weighted fair queuing) 알고리즘

WFQ [DKS89]에서 가상 종료 시간(virtual finishing time, VFT)을 처음으로 도입하였다. 가상 종료 시간을 설명하기 위해서 가상 시간(virtual time, VT)의 개념을 먼저 설명해야 한다. 가상 시간은 태스크의 지분을 고려하여 태스크가 상대적으로 받은 서비스 시간으로 계산된다. 임의의 t 시간에서 어떤 태스크 T_i의 가상 시간 $VT_i(t)$은 다음과 같이 누적 실행 시간을 자신의 지분의 역수로 정규화된 값이다.

$$VT_i(t) = \frac{w_i(t)}{\phi_i}$$

여기서 ϕ_i는 T_i의 지분이고, $w_i(t)$는 T_i가 t 시간까지 수행한 누적 실행 시간이다. 가상 종료 시간은 단위 시간을 할당 받은 후의 가상 시간으로 정의되며, WFQ 알고리즘은 스케줄링 시점에 최소의 가상 종료 시간을 가진 태스크를 선택한다. 예를 들어 4개의 태스크 A, B, C, D가 각각 4:3:2:1의 지분을 가질 때, WFQ 스케줄링 알고리즘에 의한 실행 순서는 [ABACBAABCD]이다. 최소의 가상 종료 시간을 선택하므로 WFQ 스케줄링 알고리즘은 $O(N) \sim O(logN)$의 시간 복잡도를 가진다. 또한 이 알고리즘은 음의 서비스 시간 오차 중 -1의 하한 값을 보장하지만, 태스크가 편향될수록 양의 서비스 시간 오차가 크게 증가한다. WFQ 알고리즘은 CPU 태스크 스케줄링에서 스트라이드(stride) 알고리즘이라고도 불린다 [WW94]. 모든 태스크들이 티켓(ticket), 패스(pass), 스트라이드 값을 각각 유지 및 갱신하고, 전체 태스크들 중에서 최소의 패스 값을 갖는 태스크를 선택하여 수행한다. 스트라이드

값은 전체 태스크들의 총 지분을 각 태스크의 지분으로 나눈 값이고, 초기 패스 값은 스트라이드와 동일한 값을 가진다. 스트라이드 알고리즘의 실행 순서는 WFQ와 동일한 결과를 보인다.

2.1.2 WF^{2}Q(worst-case fair weighted fair queueing) 알고리즘

기존의 WFQ 알고리즘은 모든 태스크들을 스케줄링 대상으로 고려하는 반면, WF^{2}Q 스케줄링 알고리즘 [BZ96]에서는 모든 태스크들을 스케줄링 대상으로 고려하지 않는다. WF^{2}Q 스케줄링 알고리즘에서는 다음 태스크로 선택될 수 있는 자격이 있는지의 여부를 나타내는 *적격*(*eligible*) 이란 개념을 도입하였다. 시스템 가상 시간과 태스크들의 가상 시간을 비교하여 *적격* 여부를 결정하고 *적격*의 태스크들을 대상으로 최소의 가상 시간을 가진 태스크를 선택한다. 예를 들면, 4개의 태스크 A, B, C, D=4:3:2:1의 지분 예에서 WF^{2}Q 스케줄링 알고리즘은 [ABCABACBAD]의 실행 순서를 갖는다. 이 알고리즘은 각 가상 시간과 시스템 가상 시간으로부터 *적격* 여부를 추가적으로 계산해야 하므로 부가적인 오버헤드가 있으나, WFQ 알고리즘보다 더 높은 공정성을 보장한다. 특히 음의 서비스 시간 오차는 -1, 양의 서비스 시간 오차는 1로 하한과 상한을 보장한다. WF^{2}Q 역시 스케줄링 오버헤드는 구현에 따라 $O(logN) \sim O(N)$으로 서버 시스템에서 확장성을 보장하기 어렵다.

2.2 라운드 로빈(round robin) 기반 스케줄링 알고리즘

전통적인 라운드 로빈 알고리즘은 타임 퀀텀(time quantum)이라는 일정한 단위 시간을 두고, 실행큐에 있는 태스크들에게 순서대로 단위 시간만큼씩 CPU 자원을 할당하는 알고리즘이다. 라운드 로빈 알고리즘 기반으로 동작하는 공정 스케줄링

알고리즘들 [Nag85, SV95, NVZ01]은 스케줄링 응답시간이 매우 빠르고 시간 복잡도가 $O(1)$ 이지만 지분의 범위가 크고 지분 분포가 편향되는 경우 서비스 시간 오차가 커진다는 단점이 있다. 또한 이러한 서비스 시간 오차는 태스크의 수가 커질수록 더 악화된다.

2.2.1 WRR(weighted round robin) 알고리즘

WRR 알고리즘은 공정 스케줄링에서 가장 단순한 방법으로 자신의 지분에 비례하여 시간할당량을 서비스하는 방식이다 [Nag85]. 단순히 지분에 비례하는 시간할당량이 주어지므로 장기적으로는 공정성을 잘 보장하는 것 같으나, 태스크 수 및 지분 분포에 따라 좋지 않은 공정성을 제공한다. 예를 들어, A, B, C, D의 4개 태스크가 각각 4:3:2:1이라는 지분을 갖는다면, WRR 스케줄링 알고리즘의 경우 [AAAABBBCCD]의 순서로 태스크가 수행된다. 이 때 서비스 시간 오차는 표 3.3과 같이 [-1.4, 2.4]에서 나타난다. 하지만, 지분의 비율이 400:300:200:100일 경우, 서비스 시간 오차는 -140에서 240까지 증가한다. WRR 알고리즘의 문제는 한 태스크가 모두 수행한 후에 다른 태스크들이 수행되므로 태스크 수가 증가하거나 또는 지분의 편향성이 커지면 이 오차는 더욱 커지는 것이다.

2.2.2 DRR(deficit round robin) 알고리즘

DRR에서도 각 태스크들은 지분에 비례하는 값으로 시간할당량이 주어지며 라운드 로빈 방식으로 동작한다. 태스크들은 자신에게 할당된 시간을 사용하지 못하고 남아있는 경우, 결손값(deficit value)으로 남아 있는 시간할당량을 유지하여 관리한다 [SV95]. 전체 태스크들은 라운드 로빈으로 동작하는데, 태스크의 결손 값이 1보다

크면 보상하여 해당 태스크를 한 번 더 수행한 후 결손 값을 1 감소시키고, 결손 값이 1보다 작은 경우 다음 태스크를 수행한다 [CCN+05]. 예를 들어, A, B, C, D의 4개 태스크가 각각 4:3:2:1이라는 지분을 갖는다면 [AAABBCDABC]의 실행 흐름을 보인다. 특히 DRR 스케줄링 알고리즘은 태스크들에게 부여되는 지분의 범위가 작은 경우 WRR보다 높은 공정성을 제공한다. 하지만 편향 지분 분포에서 태스크 수에 따라 서비스 시간 오차는 $O(N)$ 으로 증가하며 WRR과 유사한 공정성 오차를 보여준다.

2.2.3 VTRR(virtual-time round robin) 알고리즘

VTRR 스케줄링 알고리즘 [NVZ01]은 지분이 큰 순으로 정렬된 실행큐에서 라운드 로빈 방식으로 동작한다. VTRR에서는 WFQ에서 설명한 가상 종료 시간을 태스크별로 유지하며, 별도로 시스템 가상 시간을 QVT(queue virtual time)라는 개념으로 다음과 같이 유지한다.

$$QVT(t+q) = QVT(t) + \frac{q}{\sum_i \phi_i}$$

단, 여기에서 q는 태스크에게 주어지는 단위 시간할당량이다. 라운드 로빈으로 동작하면서 현재 태스크에 대해서 시스템 가상 시간을 고려하여 다음의 부등식을 계산한다.

$$VFT_i(t) - QVT(t+q) < \frac{q}{\phi_i}$$

위의 부등식이 참이면 VTRR 스케줄러는 실행큐에서 다음 태스크를 실행시키고, 참이 아니면 실행큐의 처음으로 돌아가 첫 번째 태스크를 선택한다. 라운드 로빈으로 동작하면서 태스크들이 받은 서비스 시간이 시스템 전체의 평균 서비스 시간

값을 넘지 않도록 유지하며 스케줄링을 수행한다. 또한 태스크마다 시간 카운터(time counter)를 두어 스케줄링 사이클 이내에서 지분 만큼만 수행될 수 있도록 보장한다. 여기서 스케줄링 사이클이란 태스크들의 모든 지분의 합을 의미한다. 예를 들어, A, B, C, D의 4개 태스크가 각각 4:3:2:1이라는 지분을 갖는다면 VTRR 스케줄링 알고리즘은 [ABCDABCABA]의 순서로 태스크가 수행된다. VTRR 스케줄링 알고리즘은 기존의 다른 라운드 로빈 기반 알고리즘들보다 좋은 공정성을 갖는다. 그러나 VTRR 역시 지분 분포가 편향된 환경에서 공정성을 유지하기 어렵다는 단점이 있다.

2.3 혼합(hybrid) 기반 스케줄링 알고리즘

혼합 기반 공정 스케줄링 알고리즘은 기존의 라운드 로빈 및 가상 시간 기반 스케줄링 알고리즘들의 장점을 최대로 절충하여 서비스 시간 오차와 스케줄링 오버헤드를 모두 최소화하고자 고안된 알고리즘이다 [RP03, CCN+05]. 보통 태스크들을 지분의 범위에 따라 G개의 그룹으로 나누고 그룹 간 스케줄링과 그룹 내 스케줄링으로 나누어 동작하는 2단계 스케줄링 방식으로 동작한다. 하지만, 혼합 기반 알고리즘들 역시 지분 분포가 편향되어 있는 경우 서비스 시간 오차는 증가할 수 있다.

2.3.1 STRR(stratified round robin) 알고리즘

STRR 알고리즘은 네트워크에서 패킷 스케줄러로서 제안되었다. STRR 알고리즘에서 플로우(flow)들의 지분율에 따라 플로우들을 2^{-k}에서 $2^{-(k-1)}$의 범위를 가진 클래스 계층 F_k으로 나눈다 [RP03]. 즉 유사한 범위에 있는 지분을 가진 플로우들을 하나의 그룹으로 구성한다. 그룹 별로 가상 시간이 관리되며, 매 스케줄링 시간에

그룹 간 스케줄링을 통해 가장 작은 그룹 가상 종료 시간 값을 가진 그룹이 선택된다. 이 후 선택된 그룹 내에서 DRR 스케줄링 알고리즘을 사용하여 다음에 전송할 플로우를 선택하는 2단계 방식이다. 따라서 스케줄링 함수의 시간 복잡도는 상수 시간 $O(G)$이며, 낮은 서비스 시간 오차를 가진다.

2.3.2 GR^3(group ratio round robin) 알고리즘

GR^3 알고리즘은 STRR 알고리즘을 개선하여 다중처리기에 적용하였다. 역시 지분의 범위에 따라 G개의 그룹으로 나누고 태스크들을 그룹에 할당한다. GR^3 [CCN+05]는 2^{k-1}에서 $2^k - 1$까지 범위의 지분을 가진 태스크들을 k 번째 그룹으로 나누는 방식으로 STRR과 유사하다. GR^3 그룹 간(inter-group) 스케줄링과 그룹 내(intra-group) 스케줄링으로 나누어 동작하는 2단계 스케줄링 방식으로 동작한다. GR^3 그룹 간(inter-group) 스케줄링을 위해 그룹 지분(group share)과 그룹의 누적 실행 시간을 별도로 관리한다. 그룹들은 지분이 큰 순서로 정렬되어 유지되며 태스크들도 각 그룹 내에서 지분 순서로 정렬되어 존재한다. 그룹 간 스케줄링 방식은 다음의 부등식에 따라 결정한다.

$$\frac{w_i(t)}{w_{i+1}(t)} > \frac{\phi_i}{\phi_{i+1}}$$

그룹 선택을 위해 라운드 로빈 방식으로 동작하며 부등식이 성립하면 다음 그룹을 선택하고, 성립하지 않으면 그룹 지분이 가장 큰 첫 번째 그룹으로 돌아간다. 그룹 내(intra-group) 스케줄링으로 선택된 그룹 내에서 태스크를 선택하기 위해 DRR [SV95] 알고리즘을 사용한다. 예를 들어, A, B, C, D의 4개 태스크가 각각 4:3:2:1이라는 지분을 갖는 경우, 각 태스크들은 [B, C], [A], [D]의 3개의 그룹으로

표 2.1: 기존 알고리즘들에 대한 서비스 시간 오차와 스케줄링 오버헤드(N=1,024)

공정 스케줄링		서비스 시간 오차		스케줄링 오버헤드
		랜덤 분포	편향 분포	
가상 시간 기반	WFQ	3	$5 \cdot 10^2$	$O(N)$ or $O(logN)$
	WF^2Q	2	2	
라운드 로빈 기반	WRR	$2 \cdot 10^3$	10^6	$O(1)$
	DRR	$1.5 \cdot 10^3$	10^6	
	VTRR	10^3	10^4	
혼합 기반	GR^3	5	$2 \cdot 10$	$O(G)$

나누어진다. 이때 그룹들의 그룹 지분 비율은 5:4:1이다. GR^3 스케줄링 알고리즘에 의해 [BACABABCAD]의 순서로 태스크가 수행된다. GR^3 스케줄링 알고리즘은 태스크들의 지분 범위가 증가하고 편향성이 커질수록 서비스 시간 오차는 증가한다.

2.4 기존 공정 스케줄링 알고리즘들의 비교 분석

다양한 지분 분포에서 기존 스케줄링 기법들의 공정성을 확인하기 위하여 각 태스크들에게 균일분포로 생성된 난수 값으로 지분을 부여하고 특정 개수의 태스크들의 지분이 10(%)부터 90(%)가 되도록 편향적으로 분포시켰다. 이 특정 개수의 태스크들도 정해진 차이를 갖도록 계층적으로 지분이 부여되었다. 공정성 분석을 위해 스케줄링 시점에 모든 태스크들의 서비스 시간 오차를 측정하였고 최소 서비스 시간 오차와 최대 서비스 시간 오차의 차이로 비교 평가하였다. 각 알고리즘들마다 최소/

최대 서비스 시간 오차 사이의 차이가 크기 때문에 오차들의 근사값으로 나타내었다. 각 알고리즘들의 서비스 시간 오차와 스케줄링 오버헤드를 정리한 결과는 표 2.1과 같다. 분석 결과, 기존 알고리즘들은 편향 지분 분포에서 서비스 시간 오차가 커짐을 확인하였다. 가상 시간 기반 스케줄링 알고리즘들은 높은 공정성을 보이는 반면 스케줄링 오버헤드는 구현에 따라 $O(logN) \sim O(N)$ 이므로 태스크 수가 작은 환경에서 높은 공정성을 요구하는 서버 시스템에 적합하다. 반면, 라운드 로빈 기반 알고리즘과 혼합 기반 알고리즘은 높은 수준의 효율성을 강조하는 대규모 서버 시스템에서 사용 가능하지만 지분 범위가 크고 편향된 환경에서는 높은 공정성은 기대 할 수 없다. 이 실험 결과는 편향된 지분 분포에서 기존 알고리즘들의 공정성이 좋지 않다는 것을 알 수 있다. 따라서 태스크가 많은 상황뿐만 아니라 편향 지분 분포도 고려하여 서버 시스템에 적합한 공정하고 효율적인 스케줄링 알고리즘이 필요하다.

제 3 장 태스크 만족 지수와 LZF 스케줄링 알고리즘

본 장에서는 가정하고 있는 시스템 모델과 태스크 만족 지수를 정의한다. 또한, 태스크 만족 지수 기반의 LZF 스케줄링 알고리즘을 제시하고 공정성과 효율성을 분석한다.

3.1 시스템 모델

본 절에서는 가정하는 시스템 모델을 설명한다. 본 논문에서는 M 개의 다중처리기가 있는 단일 서버 내에 N 개의 태스크($T = \{T_1, T_2, T_3, T_4 ... T_N\}$)가 존재한다고 가정한다. 또한 시스템 내에 실행 가능한 태스크의 수는 처리기의 수보다 크다고 가정한다 (즉 $N > M$). $N \leq M$ 이라면 태스크를 처리기에 고정하여 정적 할당 방식으로 쉽게 동작할 수 있기 때문이다. 처리기 자원은 일정한 시간할당량인 단위 시간(q) 만큼 태스크들에게 할당된다. 이는 태스크가 자원을 할당 받아 수행되는 기본 단위로 타임 퀀텀(time quantum)의 개념과 같다. 본 논문에서는 수식의 단순화를 위해 단위 시간(q)은 1로 고려한다. 선택된 태스크들은 단위 시간 동안 다른 태스크들의 선점없이 수행된고 가정한다. 각 태스크 T_i는 처리기 자원에 대한 서비스를 받기 위해 양의 정수 지분 ϕ_i를 가지고 있다. 또한 각 태스크의 전체 처리기 자원에 대한 기대값은 $\frac{\phi_i}{\sum_i \phi_i}$으로서 지분율 p_i로 표현할 수 있다. 따라서 공정 스케줄링 알고리즘

$\mathcal{F}$는 매 단위 시간마다 전체 처리기 자원에서 p_i 만큼 비례적으로 서비스 하려고 할 것이다. 본 논문에서 스케줄링 사이클 S은 태스크들의 지분의 합으로 정의된다. 예를 들면, 3:2:1의 태스크를 가진 세 개의 태스크가 존재할 때, 스케줄링 사이클은 6이다.

또한, 각 태스크 T_i들은 동시에 여러 처리기에서 동시에 수행할 수 없으므로, 지분율 p_i는 $1/M$보다 작아야 한다. 즉 ϕ_i는 다음을 만족할 때 실행 가능(feasible)하다고 한다.

$$\frac{\phi_i}{\sum_j \phi_j} \leq \frac{1}{M} \tag{3.1}$$

지분 재조정 과정을 통해 태스크들의 지분은 모든 시간에 대해 실행 가능하다고 가정한다.

본 논문에서는 어떤 공정 스케줄러 $\mathcal{F}$에 의한 스케줄링 결과의 공정성 평가를 위해 서비스 시간 오차(service time error)를 사용한다. 각 태스크 T_i에 대해 $w_i(s,t)$와 $s_i(s,t)$를 시간 구간 $[s,t]$ 동안 각각 실제로 서비스 받은 시간과 그 태스크가 서비스 받아야 할 시간이라고 하자. 이때 그 태스크 T_i의 서비스 시간 오차 $e_i(s,t)$는 $w_i(s,t)$ 과$s_i(s,t)$ 차이로서 계산된다. 즉 서비스 시간 오차는 수식 (3.2)과 같이 정의된다.

$$\begin{aligned} e_i(s,t) &= w_i(s,t) - s_i(s,t) \\ &= w_i(s,t) - \frac{\phi_i}{\sum_j \phi_j} \end{aligned} \tag{3.2}$$

즉 음의 서비스 시간 오차는 그 태스크가 어떤 주어진 시간에 서비스 받아야 할 시간보다 실제로 덜 받았음을 의미하며, 양의 서비스 시간 오차는 받아야 할 서비스

시간보다 더 많은 서비스를 받았음을 의미한다. 그러므로 서비스 시간 오차 값은 0에 가까울수록 완벽한 공정성에 가깝다. 일반적으로, 모든 태스크들에서 측정된 서비스 시간 오차중에서 최소/최대 서비스 시간 오차 또는 그 차이로서 평가되거나, 전체적인 평균 서비스 시간 오차로서 공정성을 정량화 한다. 본 논문에서는 최소/최대 서비스 시간 오차와 평균 서비스 시간 오차 모두 사용하여 제안 알고리즘과 기존 스케줄링 알고리즘들의 공정성을 평가한다.

3.2 태스크 만족 지수

GPS(Generalized Processor Sharing) 정책은 항상 모든 태스크들의 서비스 시간 오차가 0을 만족하는 이상적인 스케줄링 정책이다. 하지만, 이산적인 단위 시간을 사용하는 실제 시스템에 적용이 어렵기 때문에 기존의 공정 스케줄링 알고리즘들은 서비스 시간 오차를 최소로 하는 것에 목표를 두었고, GPS는 공정 스케줄링 참조 모델 및 평가 기준을 위해 사용된다. 기존 대부분의 알고리즘들은 효율적으로 과거 서비스를 기억하기 위하여 자신의 지분으로 정규화된 누적 실행 시간인 가상 시간(virtual time)에 기반을 두어 근사화하였다.

본 절에서는 가상 시간(virtual time) 대신 직접적으로 서비스 시간 오차(service time error)를 사용하는 태스크 만족 지수를 정의한다. 직접적으로 서비스 시간 오차(service time error)을 사용할 경우, 매 스케줄링 시간마다 모든 태스크들의 서비스 시간 오차를 다시 계산하는 비효율성이 존재하지만 평균 서비스 시간 오차 측면에서 기존 알고리즘 중 가장 좋은 공정성을 보여주기 때문이다.

본 논문에서는 새로운 평가 기준으로 다중처리기에서 태스크 만족 지수(satisfaction index, z)를 정의한다. M 개 다중처리기 상에서 n 번째 단위 시간에 평가되는

T_i의 만족 지수 $z_i^{(n)}$는 수식(3.3)과 같이 정의 된다.

$$z_i^{(n)} = z_i^{(n-1)} + I_i^{(n-1)} - M \cdot p_i \tag{3.3}$$

단, $z_i^{(0)} = 0$이고 $I_i^{(n)}$는 실행 여부에 따라 0 또는 1을 반환하는 함수이다. T_i가 n 번째 단위 시간에서 실행이 되면, $I_i^{(n)} = 1$이고, 그렇지 않으면 $I_i^{(n)} = 0$이다. 처리기가 M개인 다중처리기에서 T_i의 단위 시간당 실행 기대값은 $M \cdot p_i$이다. 그림3.1과 같이 태스크 T_i가 스케줄링 시간에 선택되지 않으면 태스크 만족 지수는 실행 기대값인 $M \cdot p_i$만큼 감소하는 반면, T_i가 선택되면 그 태스크의 만족 지수는 $1 - M \cdot p_i$만큼 증가한다. 즉 매 단위 시간마다 각 태스크들은 실행 기대값인 $M \cdot p_i$만큼 감소되고 실행되는 경우에만 1이 증가된다. z의 의미는 단위 시간당 실행 오차의 누적 결과로서 직관적으로 자원 할당량에 대한 태스크들의 만족 여부를 알 수 있다. 임의의 시간에 어떤 태스크의 z의 값이 음수라는 것은 태스크가 그 지분만큼 서비스를 받지 못해 불만족스럽다는 것을 의미하고 양수의 값은 그 반대를 의미한다. 단위 시간q는 1임을 고려하면, z가 -1이라는 것은 지분을 고려한 단위 시간당 실행 오차가 -1이라는 것으로 그 시간까지 그 태스크가 지분을 통해 받을 수 있는 시간보다 한 단위 시간만큼 서비스를 덜 받았다는 것을 의미한다. 또한, 매 단위 시간에서 모든 태스크들의 만족 지수의 합은 항상 0임을 정리 1과 같이 유도할 수 있다.

정리 1 *M개의 다중처리기상에서 매 스케줄링 시간 n에 대해 항상 $\sum_{i=1}^{N} z_i^{(n)} = 0$를 만족한다.*

증명 수식(3.3)의 만족 지수의 정의에 의해 모든 태스크들은 초기 만족 지수는 $z_i^{(0)} = 0$이므로 $\sum_{i=1}^{N} z_i^{(0)} = 0$을 만족한다. $\sum_{i=1}^{N} z_i^{(k)} = 0$이 성립한다고 가정할 때,

$\sum_{i=1}^{N} z_i^{(k+1)} = 0$ 이 만족함을 보인다.

k 번째 시간에서 M 개의 태스크 $a_i(1 \le i \le M)$ 에게 처리기가 할당되었다고 하고, 선택되지 않은 $N-M$ 개의 태스크들은 $b_i(1 \le i \le N-M)$ 이라 하자. 모든 태스크들에 대해 $\sum_i p_i = 1$ 이므로 다음을 만족한다.

$$\sum_{i=1}^{M} p_{a_i} + \sum_{i=1}^{N-M} p_{b_i} = 1$$

k 번째 시간에서 선택된 태스크 $a_i(1 \le i \le M)$ 들의 $k+1$ 번째 시간에서 만족 지수 합은 수식(3.3)에 의해 다음과 같이 계산된다.

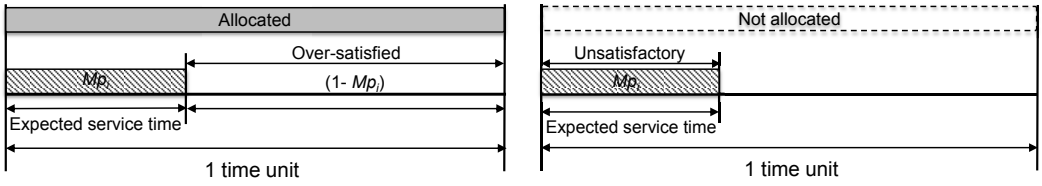
$$\sum_{i=1}^{M} z_{a_i}^{(k+1)} = \sum_{i=1}^{M} z_{a_i}^{(k)} + \sum_{i=1}^{M} (1 - M \cdot p_{a_i})$$

마찬가지로 k 번째 시간에서 선택되지 않은 태스크 $b_i(1 \le i \le N-M)$ 들의 만족 지수 합은 다음과 같다.

$$\sum_{i=1}^{N-M} z_{b_i}^{(k+1)} = \sum_{i=1}^{N-M} z_{b_i}^{(k)} + \sum_{i=1}^{N-M} (-M \cdot p_{b_i})$$

$\sum_{i=1}^{N} z_i^{(k)} = 0$ 이므로 $k+1$ 번째 시간에서 모든 태스크들의 만족 지수의 합은 다음과 같이 유도된다.

$$\begin{aligned}
\sum_{i=1}^{N} z_i^{(k+1)} &= \sum_{i=1}^{M} z_{a_i}^{(k)} + \sum_{i=1}^{N-M} z_{b_i}^{(k)} + \sum_{i=1}^{M} (1 - M \cdot p_{a_i}) + \sum_{i=1}^{N-M} (-M \cdot p_{b_i}) \\
&= 0 + M - M(\cdot \sum_{i=1}^{M} p_{a_i} + \sum_{i=1}^{N-M} p_{b_i}) \\
&= 0
\end{aligned}$$



(a) T_i 가 선택된 경우 만족 지수 (b) T_i 가 선택되지 않는 경우 만족 지수

그림 3.1: 지분율이 p_i 인 태스크 T_i의 실행 여부에 따른 만족 지수

따라서 모든 단위 시간에서 항상 $\sum_{i=1}^{N} z_i^{(n)} = 0$를 만족한다.

□

정리 1로부터 만약 어떤 스케줄러가 가장 작은 만족 지수의 값을 가진 태스크를 선택하려할 때, 만족 지수가 0보다 작은 태스크들만 스케줄링 대상으로 고려해도 충분하다는 것을 짐작할 수 있다. 또한 정리 2는 동일한 서비스 시간을 할당받은 두 태스크 중 지분이 더 큰 태스크의 만족 지수가 상대적으로 더 작음을 보인다.

정리 2 *$\phi_i > \phi_j$ 인 두 태스크 T_i와 T_j에 대해 $\sum_{t=0}^{t=n-1} I_i^{(t)} = \sum_{t=0}^{t=n-1} I_j^{(t)}$ 이 만족할 때, $z_i^{(n)} < z_j^{(n)}$ 이 성립한다.*

증명 $\phi_i > \phi_j$ 이므로 태스크 사이의 지분 비율은 $p_i > p_j$을 만족한다. 만족 지수의 정의 (수식(3.3)) 로부터 만족 지수는 다음과 같이 전개된다.

$$\begin{aligned} z_i^{(n)} &= z_i^{(n-1)} + I_i^{(n-1)} - M \cdot p_i \\ &= z_i^{(0)} + (I_i^{(0)} - M \cdot p_i) + (I_i^{(1)} - M \cdot p_i) + \cdots + (I_i^{(n-1)} - M \cdot p_i) \\ &= \sum_{t=0}^{t=n-1} I_i^{(t)} - n \cdot M \cdot p_i \end{aligned} \tag{3.4}$$

즉 수식 (3.4)은 n 번째 시간까지 만족 지수의 합이 $e_i(0,n)$과 같음을 의미한다.

마찬가지로 태스크 T_j의 만족 지수에 대해서 $z_j^{(n)} = \sum_{t=0}^{t=n-1} I_j^{(t)} - n \cdot M \cdot p_j$를 만족한다. 이로부터 $z_i^{(n)} - z_j^{(n)} < 0$임을 보인다.

$$\begin{aligned} z_i^{(n)} - z_j^{(n)} &= \sum_{t=0}^{t=n-1} I_i^{(t)} - n \cdot M \cdot p_i - (\sum_{t=0}^{t=n-1} I_j^{(t)} - n \cdot M \cdot p_j) \\ &= -n \cdot M(p_i - p_j) \\ &< 0 \end{aligned}$$

따라서, $\phi_i > \phi_j$인 두 태스크 T_i와 T_j에 대해 $\sum_{t=0}^{t=n-1} I_i^{(t)} = \sum_{t=0}^{t=n-1} I_j^{(t)}$이 만족할 때, 항상 $z_i^{(n)} < z_j^{(n)}$이 성립한다.

□

따라서 이 성질은 동일한 서비스 시간을 할당 받은 태스크들 중에서 지분이 가장 큰 태스크의 만족 지수가 가장 작다는 것을 의미한다. 또한 수식 (3.4)으로부터 만족 지수를 매번 갱신하지 않더라도 임의의 시점에서 태스크의 누적된 시간할당량 정보만으로 만족 지수를 계산할 수 있음을 의미한다. 본 논문에서 사용하는 기호 및 함수는 표 3.1과 같다.

3.3 태스크 만족 지수 기반 스케줄링 알고리즘

3.3.1 LZF 스케줄링 알고리즘

본 절에서는 태스크 만족 지수를 사용하는 그리디(greedy) 스케줄링 알고리즘인 LZF를 제시한다 [JKC+12, JJC+]. 태스크 만족 지수를 사용하는 가장 직관적인

표 3.1: 논문에서 사용하는 기호 및 함수

기호 및 함수	설명
N	실행큐에서 실행 가능한 태스크의 개수
M	다중처리기에서 처리기의 개수
T	실행 가능한 태스크 집합, $T = \{T_1, T_2, T_3, \cdots T_N\}$
ϕ_i	태스크 T_i에 부여된 지분
S	스케줄링 사이클
p_i	태스크 T_i의 자원 예약 비율 i.e., $p_i = \frac{\phi_i}{\sum_j \phi_j}$
$w_i(s,t)$	[s, t] 시간 동안 태스크 T_i가 받은 실제 서비스의 양
$s_i(s,t)$	[s, t] 시간 동안 태스크 T_i가 받아야 하는 서비스의 양
$e_i(s,t)$	[s, t] 시간 동안 태스크 T_i의 서비스 시간 오차
$I_i^{(n)}$	스케줄링 여부에 따라 0 or 1 반환하는 함수
	n 번째 단위 시간에서 태스크 T_i가 다음 태스크로 선택 될 때 $I_i^{(n)} = 1$
	선택되지 않으면, $I_i^{(n)} = 0$
$z_i^{(n)}$	n 번째 단위 시간에서 태스크 T_i의 만족 지수
$\hat{z}_i^{(n)}$	n 번째 단위 시간에서 태스크 T_i의 잠재 만족 지수
	$\hat{z}_i^{(n)} = z_i^{(n)}$ - $M \cdot p_i$
K	K-LZF 스케줄링에서 근사화 인자

방법은 가장 만족 지수가 작은 태스크를 선택하는 것이다. 단순한 스케줄링 선택 함수를 위해 잠재적인 만족 지수 $\hat{z}_i^{(n)}$를 정의한다. M 개의 다중처리기상에서 잠재 만족 지수는 n 번째 스케줄링 시간에 선택되지 않을 때, $(n+1)$ 번째 가질 수 있는 태스크의 만족 지수이며 다음과 같이 정의된다.

$$\hat{z}_i^{(n)} = z_i^{(n)} - M \cdot p_i \tag{3.5}$$

LZF의 선택 함수는 실행 가능한 태스크들 중 최소의 잠재 만족 지수를 선택함으로써 스케줄링 한다. $\hat{z}_i^{(n)}$ 값들이 동일한 경우에는 식별가능한 태스크 ID, 지분 등의 기준으로 임의로 선택한다.

$$D^{(n+1)} = arg\min_{Ti}\{\hat{z}_i^{(n)}\} \tag{3.6}$$

표 3.2는 네 개의 태스크 A, B, C, D의 지분 비율이 4:3:2:1인 경우 LZF에 대한 실행 흐름(S=10)을 보여준다. 이 예에서 LZF는 A, B, C, A, D, B, A, C, B, A의 순서로 태스크를 수행한다. 이 수행 결과는 표 3.3에서 확인 할 수 있듯이 최소 서비스 시간 오차 -0.6과 최대 서비스 시간 오차 0.6으로 오차 간 차이가 가장 작았다. LZF 알고리즘은 태스크들의 평균적인 오차를 나타내는 평균 서비스 시간 오차도 0.28로서 기존 스케줄링 알고리즘들에 비해 가장 작다.

3.3.2 LZF의 공정성 분석 및 효율적인 구현 방법

이 전 절에서 언급한 바와 같이, LZF 알고리즘은 매 단위 시간 마다 모든 태스크들의 만족 지수를 재계산(re-calculate)해야 하므로 시간 복잡도는 $O(N)$이다. 하지만,

표 3.2: LZF 스케줄링 알고리즘의 예 (A:B:C:D=4:3:2:1)

단위 시간	z_A	$\widehat{z}_A$	z_B	$\widehat{z}_B$	z_C	$\widehat{z}_C$	z_D	$\widehat{z}_D$	선택된 태스크
0	0	-0.4	0	-0.3	0	-0.2	0	-0.1	A
1	0.6	0.2	-0.3	-0.6	-0.2	-0.4	-0.1	-0.2	B
2	0.2	-0.2	0.4	0.1	-0.4	-0.6	-0.2	-0.3	C
3	-0.2	-0.6	0.1	-0.2	0.4	0.2	-0.3	-0.4	A
4	0.4	0	-0.2	-0.5	0.2	0	-0.4	-0.5	D
5	0	-0.4	-0.5	-0.8	0	-0.2	0.5	0.4	B
6	-0.4	-0.8	0.2	-0.1	-0.2	-0.4	0.4	0.3	A
7	0.2	-0.2	-0.1	-0.4	-0.4	-0.6	0.3	0.2	C
8	-0.2	-0.6	-0.4	-0.7	0.4	0.2	0.2	0.1	B
9	-0.6	-1	0.3	0	0.2	0	0.1	0	A

표 3.3: 기존 공정 스케줄링 알고리즘들의 예 (A:B:C:D=4:3:2:1)

알고리즘	실행 순서	서비스 시간 오차	평균 서비스 시간 오차
WRR	AAAABBBCCD	[-1.4, 2.4]	0.90
DRR	AAABBCDABC	[-1.0, 1.8]	0.62
VTRR	ABCDABCABA	[-0.8, 0.6]	0.35
WFQ	ABACBAABCD	[-0.9, 1.2]	0.44
WF^2Q	ABCABACBAD	[-0.9, 0.6]	0.38
GR^3	BACABABCAD	[-0.9, 0.9]	0.41
LZF	ABCADBACBA	[-0.6, 0.6]	0.28

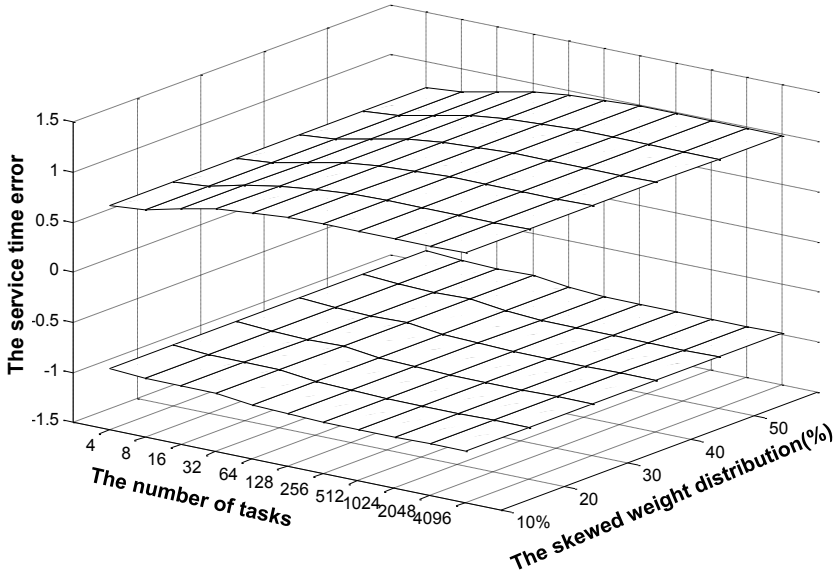


그림 3.2: 태스크 수와 편향 지분 분포에 따른 LZF 스케줄링 알고리즘의 공정성

LZF는 기존 알고리즘들 중 평균 서비스 시간 오차측면에서 가장 좋은 공정성을 보인다. 그림 3.2는 다양한 편향된 지분 분포 (N=4,096, D=1) 에서 LZF의 서비스 시간 오차를 보여준다. 이 예에서 LZF는 모든 경우에 대해 -1과 1 사이의 서비스 시간 오차를 보여주었으며, 평균 서비스 시간 오차측면에서 WF^2Q 보다도 더 좋은 공정성을 보였다. LZF 알고리즘의 높은 공정성은 본 논문에서 공정 스케줄러로서 선택한 이유다. 특히 LZF 알고리즘의 만족 지수의 상한은 다음과 같다.

정리 3 *LZF 알고리즘으로 스케줄링 할 경우 모든 i, n에 대해서 $z_i^{(n)} < 1$이 성립한다.*

증명 임의의 n번째 시간에서 모든 태스크들이 $z_i^{(n)} < 1$을 만족함을 귀납법으로 증명한다. 만족 지수 정의 수식 (3.3) 에 따라 모든 태스크들의 $z_i^{(0)} = 0$을 만족한다.

$n = 1$ 일 때, $z_i^{(1)} = -M \cdot p_i$ 이거나 $z_i^{(1)} = -M \cdot p_i + 1$이다. 모든 i에 대해서

$0 < p_i < 1$ 이고 $N > M$ 이므로 $z_i^{(1)} < 1$을 성립한다.

$n = k$ 일 때, $z_i^{(k)} < 1$ 가 성립한다고 가정하자. 또한 k 번째 최소 잠재 만족 지수가 $z_m^{(k)} - M \cdot p_m$ 이라고 할 때 모든 i에 대하여 $z_i^{(k)} - M \cdot p_i \geq z_m^{(k)} - M \cdot p_m$ 이 성립한다. 이 식을 모든 태스크들에 대해 더하면 다음이 성립한다.

$$\sum_{i}^{N} (z_i^{(k)} - M \cdot p_i) > N \cdot (z_m^{(k)} - M \cdot p_m)$$

이 때, $\sum_i p_i = 1$ 이고 정리 1에 의해 $\sum_{i=0}^{N} z_i^{(n)} = 0$가 만족하므로 다음과 같이 유도된다.

$$-1 > N \cdot (z_m^{(k)} - M \cdot p_m)$$

$$(z_m^{(k)} - M \cdot p_m) < -\frac{1}{N}$$

$n = k + 1$ 일 때, LZF 스케줄링 알고리즘은 k 번째에서 잠재 만족 지수가 $z_m^{(k)} - M \cdot p_m$의 잠재 만족 지수를 가진 태스크가 선택되므로 $z_m^{(k+1)} = z_m^{(k)} - M \cdot p_m + 1 = -\frac{M}{N} + 1 < 1$ 이다. 또한, 선택되지 않은 태스크들에 대해 가정에 의해 $z_i^{(k)} < 1$이므로 다음을 만족한다.

$$z_{i,i \neq m}^{(k+1)} = z_i^{(k)} - M \cdot p_i < 1$$

따라서 LZF 스케줄링 알고리즘은 모든 i, n에 대해서 $z_i^{(n)} < 1$을 만족한다.

□

이것은 수식 (3.4)에 의해 서비스 시간 오차의 상한이 1임을 의미한다. 즉 LZF 스케줄링 알고리즘은 임의의 시간에서 서비스 시간 오차가 1보다 큰 태스크가 존재하지 않는다는 것을 보장한다.

그림 3.2에서 보인 실험과 같이 실험적인 서비스 시간 오차의 하한은 -1으로 WF^2Q 알고리즘의 하한인 -1과 같았지만, 오히려 평균 서비스 시간 오차 측면에서 LZF 알고리즘이 WF^2Q보다 더 좋은 성능을 보였다. 이 사실은 실제 서버 시스템에서 LZF 알고리즘이 더 좋은 공정성을 보일 수 있는 근거이다. 또한 LZF 알고리즘은 태스크들의 가상 시간과 *적격*(*eligible*) 여부를 동시에 비교해야 하는 WF^2Q 알고리즘과 다르게 만족 지수의 한가지 평가 기준만으로 태스크들을 스케줄링 할 수 있는 장점이 있다.

LZF 스케줄링 알고리즘에서 최소의 잠재 만족 지수 $\hat{z}_i^{(n)}$를 효율적으로 탐색하기 위해 다음과 같이 다중리스트를 사용하여 비교 횟수를 줄이는 방법을 고려할 수 있다. 시스템 내 모든 태스크들이 다중리스트 L_K ($0 \leq K \leq \infty$)에 존재한다고 가정한다. L_K는 어느 주어진 시간에 K 단위 시간만큼 서비스 받은 태스크들의 집합이다. 그러므로 초기에 모든 태스크들은 실행된 적이 없으므로 첫 번째 리스트 L_0에 지분이 가장 큰 순서로 존재한다. 각 리스트에는 동일한 서비스 양을 할당 받은 태스크들만 유지 될 수 있도록 관리된다. L_{K-1}에 존재했었던 어떤 태스크가 실행되었다면 이 태스크는 인접리스트인 L_K 리스트로 이동한다. 정리 2에 의해 동일한 서비스 양을 할당 받은 태스크들 중에서 가장 첫 번째에 위치한 태스크의 잠재 만족 지수가 가장 작다. 이런 경우, 각 리스트 L_K로부터 첫 번째 태스크들의 잠재 만족 지수 $\hat{z}_i^{(n)}$ 만 계산 및 비교되어 가장 작은 $\hat{z}_i^{(n)}$를 찾을 수 있다. 이것은 LZF 스케줄링 알고리즘에서 모든 태스크들을 탐색하지 않고 일부 후보 태스크들의 비교만으로 최소의 $\hat{z}_i^{(n)}$를 찾게 하는 중요한 속성이다.

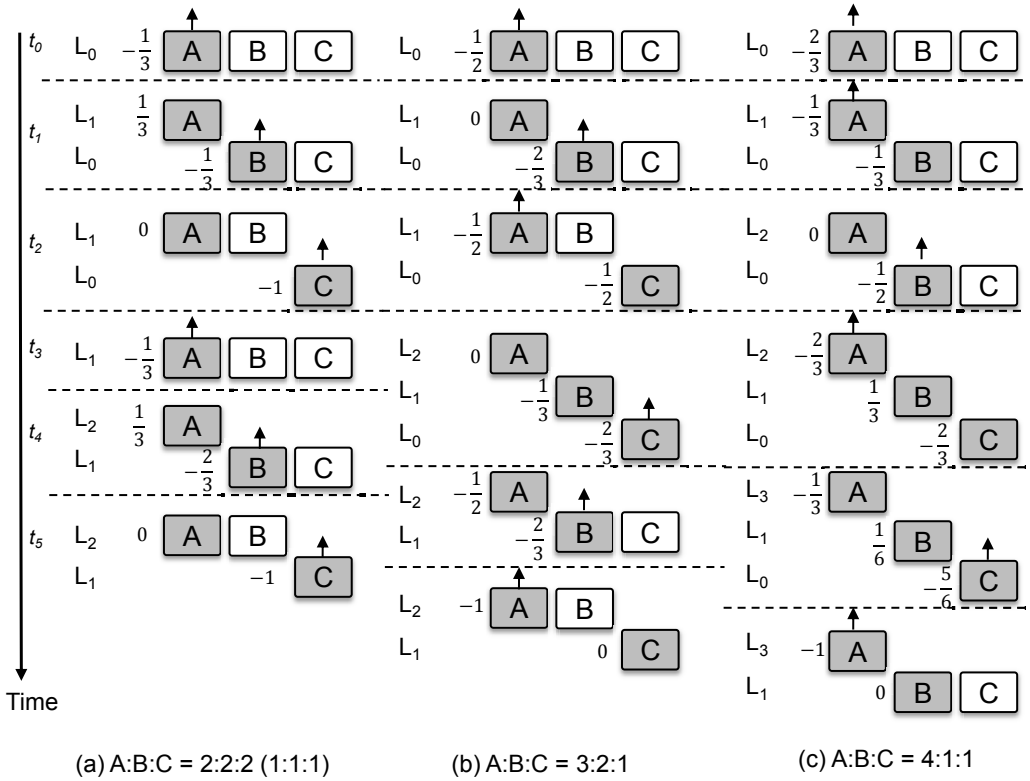


그림 3.3: 다중리스트를 사용하는 LZF 스케줄링 알고리즘의 실행 과정

그림 3.3은 다중리스트를 사용하여 효율적으로 구현한 LZF의 예이다. 세 개의 태스크 A, B, C는 각각 2:2:2, 3:2:1, 4:1:1의 지분 비율로 스케줄링 사이클만큼 실행될 때까지의 실행 순서를 나타낸다 (즉 $S = 6$). 초기에 모든 태스크들은 L_0에 존재하고, 비교할 필요 없이 $\widehat{z}_i^{(0)}$가 가장 작은 첫 번째 태스크를 실행한다. 실행 후, 선택된 태스크는 L_1에 이동한다. 이 다중리스트를 사용한 효율적인 구현 방법에 의한 태스크들의 실행 순서는 기존(naïve) LZF 스케줄링 알고리즘의 실행 순서와 정확히 동일하다. 이 예에서, 최소의 $\widehat{z}_i^{(n)}$를 찾기 위한 LZF 스케줄링 비용을 단지 탐색의 횟수만으로 계산한다면, 평균 비용은 각각 $(1+2+2+1+2+2)/6$, $(1+2+2+3+2+2)/6$ 및 $(1+2+2+3+3+2)/6$이 된다. 반면, 초기(naïve) LZF 알고리즘으로 최솟값을 찾기 위해서는 평균 3 번의 탐색이 필요하므로 다중리스트를 사용한 방법이 탐색 횟수 측면에서 효율적이라 할 수 있다. 또한, 그림 3.3 (c)와 같이 4:1:1의 비율의 지분 분포를 가진 태스크들의 탐색 횟수가 가장 크다는 사실을 관찰할 수 있는데, 이로부터 지수 분포가 편향될수록 효율적으로 최소의 $\widehat{z}_i^{(n)}$를 탐색하는 것이 어렵다는 사실을 알 수 있다.

이 다중리스트를 이용한 방법은 태스크의 다양한 지분 분포를 가진 서버 시스템에서 효율적으로 LZF를 구현이 가능함을 보인다. 아직 무한한 기억장소를 필요로 하는 문제가 있으므로 다음 장에서는 고정된 개수의 다중리스트를 사용함으로써 LZF를 근사화하는 방법을 소개한다.

제 4 장 서버 시스템을 위한 공정하고 효율적인 스케줄링 알고리즘

본 장에서는 서버 시스템에서 효율적으로 동작하기 위해 LZF를 근사화하는 K-LZF 스케줄링 알고리즘을 제안한다. K-LZF 스케줄링 알고리즘은 단일 실행큐로서 K 개의 서브리스트(sub-list)들로 구성된 연결리스트를 사용하여 각 서브리스트의 첫 번째 태스크들만 탐색할 수 있도록 연산들을 수행시켜 효율성을 향상시킨다. 만족 지수가 작은 태스크들이 후보 태스크가 될 수 있도록 유지하는 K-LZF 알고리즘의 3가지 연산들의 동작 방식을 설명하며 시간 복잡도를 분석한다.

4.1 K-LZF 스케줄링 알고리즘

4.1.1 자료구조

본 절에서는 LZF를 근사화하기 위한 K-LZF 스케줄링 알고리즘의 자료구조를 설명한다. K-LZF 알고리즘은 실행큐로서 연결리스트를 사용한다. 이 연결리스트는 실행흐름에 따라 K 개의 서브리스트(sub-list)로 나눠지며, $\textit{sift-list}_i (1 \leq i \leq K)$ 라고 정의한다. 그림 4.1는 K 개의 서브리스트로 구성된 연결리스트를 보여준다. 태스크들은 이 연결리스트 내에 지분이 큰 순서로 정렬되어 있으며, 일정 시점에 각 태스크들은 서브리스트 $\textit{sift-list}_1$ 부터 마지막 서브리스트 $\textit{sift-list}_K$ 사이에서 존재한다. 지분이 가장 큰 태스크는 항상 첫 번째 서브리스트 $\textit{sift-list}_1$ 에 존재한다. $i < j$ 일 때,

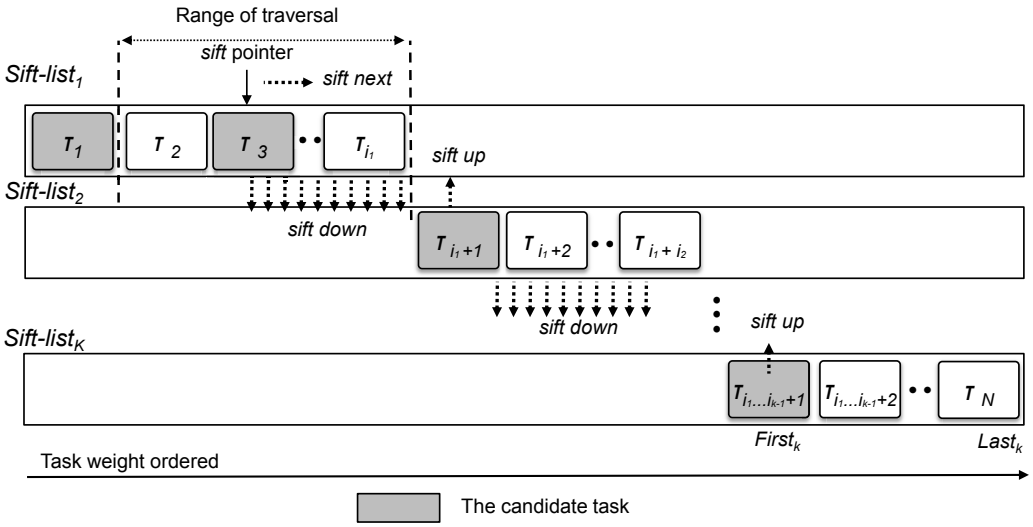


그림 4.1: K 개의 서브리스트들로 구성된 연결리스트 $\textit{sift-list}_i (1 \leq i \leq K)$

$\textit{sift-list}_i$는 $\textit{sift-list}_j$의 상위리스트라고 하고 $\textit{sift-list}_j$는 $\textit{sift-list}_i$의 하위리스트라고 한다. 그러므로 $\textit{sift-list}_i$내에 존재하는 태스크의 지분은 하위리스트 $\textit{sift-list}_j$내에 존재하는 어떠한 태스크들의 지분보다도 항상 크다. 여러 서브리스트들 사이에서 각 태스크들의 위치는 스케줄링 시점에 발생되는 *sift-next*, *sift-up* 및 *sift-down* 연산들에 의해 동적으로 재조정된다. 이들의 동작 방식은 4.1.2에서 자세히 설명한다. *sift-pointer*는 첫 번째 서브리스트 $\textit{sift-list}_1$에서만 순회하는 특별한 목적의 포인터이다. 만약, $\textit{sift-list}_1$내에 첫 번째 태스크만 존재한다면, 순회하지 않는다. *sift-pointer*는 무한 다중리스트를 사용하는 LZF 알고리즘과 다르게 K개의 서브리스트로 제한하면서 발생할 수 있는 문제를 해결하기 위해 도입되었다. *sift-pointer*는 편향된 지분 분포에서 효과적으로 공정성에 영향을 미친다.

그림 4.1에서 회색으로 표기된 태스크들, 즉 각 서브리스트 맨 앞에 위치한 태스크들과 *sift-pointer*가 가리키는 태스크는 스케줄링 선택 시 비교 대상이 되는 후보 태스크(candidate task)라고 정의한다. K-LZF는 LZF와 다르게 모든 태스크들을

비교하는 대신 이 후보 태스크들만 비교함으로써 효율성을 높였다. *sift-pointer*가 가리키는 한 개의 후보 태스크를 포함하여 각 서브리스트의 첫 번째 태스크들로부터 후보 태스크까지 총 $K+1$ 개의 후보 태스크가 존재한다. LZF 정책에 따라 스케줄링 시점에 이 후보리스트들의 잠재 만족 지수 $\hat{z}_i^{(n)}$를 계산하여 가장 작은 값을 가진 태스크를 다음 태스크로 선택한다. 후보 태스크들의 $\hat{z}^{(n)}$는 수식 (3.5)와 같이 $z_i^{(n)}$에서 $M \cdot p_i$를 빼면 계산된다. 이것은 태스크들의 탐색 횟수를 고려할 때 K-LZF 스케줄링 선택 함수의 시간 복잡도가 최대 $O(K)$를 가진다는 것을 의미한다.

4.1.2 동작 방식

본 절에서는 K-LZF 알고리즘의 연산들의 동작 방식들에 대해 설명한다. K 개의 서브리스트들로 구성된 연결리스트는 태스크들의 실행큐 역할을 한다. 4.1.1절에서 설명한 것과 같이 K-LZF 알고리즘은 각 서브리스트 *sift-list*$_i(1 \leq i \leq K)$ 로부터의 첫 번째 태스크들과 *sift-pointer*가 지정하는 후보 태스크들만 비교한다. 매 스케줄링 시점마다 태스크들의 만족 지수는 변하므로 이러한 변화를 효과적으로 반영해야 한다. 따라서 스케줄링 시점에 수행되는 연산들은 각 서브리스트 *sift-list*$_i$ 내에서 잠재 만족 지수가 가장 작은 태스크가 앞으로 오도록 유지할 수 있어야 한다. 그림 4.1과 같이 K-LZF 알고리즘은 *sift-next*, *sift-up* 및 *sift-down*의 세 가지 연산 중 하나의 연산을 수행하여 잠재 만족 지수가 작은 태스크들이 후보 태스크가 될 수 있도록 유지한다. 특히 수행되는 연산의 종류는 선택된 후보 태스크에 따라 달라진다. 다음은 세 가지 연산의 동작 방식과 호출 시점을 설명한다.

- *sift-next* 연산: 이 연산은 *sift-pointer*가 지정한 후보 태스크가 선택된 경우 수행된다. *sift-next*연산은 *sift-pointer*가 *sift-list*$_1$ 내에서 다음 태스크를 가리

키도록 *sift-pointer*를 이동시킨다. 만약 *sift-pointer*가 순회 범위의 마지막인 *sift-list*$_1$ 내의 마지막 태스크에 위치했다면 순회 범위 처음으로 다시 돌아온다.

- *sift-up* 연산: 각 서브리스트 *sift-list*$_i$ (단, $i \neq 1$)의 후보 태스크가 선택되면 *sift-up*연산이 수행된다. 그 후보 태스크는 상위 서브리스트인 *sift-list*$_{i-1}$로 이동된다. 그리고 나면, 자연스럽게 이전에 *sift-list*$_i$에서 두 번째로 위치하였던 태스크가 후보 태스크가 된다.

- *sift-down* 연산: 최상위 *sift-list*$_1$의 첫 번째 태스크가 선택되면 *sift-up*연산 대신 *sift-down*연산이 수행된다. 서브리스트 *sift-list*$_1$의 상위리스트는 없으므로 *sift-pointer*가 가리키는 후보 태스크부터 *sift-list*$_1$ 내의 모든 태스크를 하위리스트 *sift-list*$_2$로 이동시킨다. 마찬가지로, 이 연산은 *sift-list*$_{i-1}$의 태스크들을 *sift-list*$_i$로 변경한다. 비어있는 서브리스트가 존재할 때까지 연쇄적으로 이동한다. 존재하지 않는 경우, *sift-list*$_{K-1}$의 태스크들은 *sift-list*$_K$의 태스크 앞에 추가하여 결합된다. 그 후 *sift-pointer*는 *sift-list*$_1$의 순회 범위의 처음으로 조정된다.

스케줄링 시점에 선택된 태스크가 *sift-pointer* 포인터가 지정한 후보 태스크라면 *sift-next* 연산을 통해 *sift-pointer* 포인터를 순회 범위 내에서 다음 포인터로 이동시키므로 $O(1)$이다. 또한, *sift-up*연산은 선택된 태스크를 인접 상위리스트 이동시키므로 $O(1)$의 시간 복잡도를 가진다. 반면, *sift-down* 연산은 한번 수행 시 최대 K번 서브리스트를 순회하므로 시간 복잡도가 $O(K)$이다. 하지만, *sift-down* 연산은 매 스케줄링 시점마다 호출되는 것이 아니라 *sift-list*$_1$의 첫 번째 태스크가 다음 태스크로 스케줄링 될 때만 호출된다. 따라서 K-LZF 알고리즘의 시간 복잡도는 최대 $O(K)$이며 매 스케줄링 시점마다 $O(logN)$을 보이는 가상 시간 알고리즘들에 비해

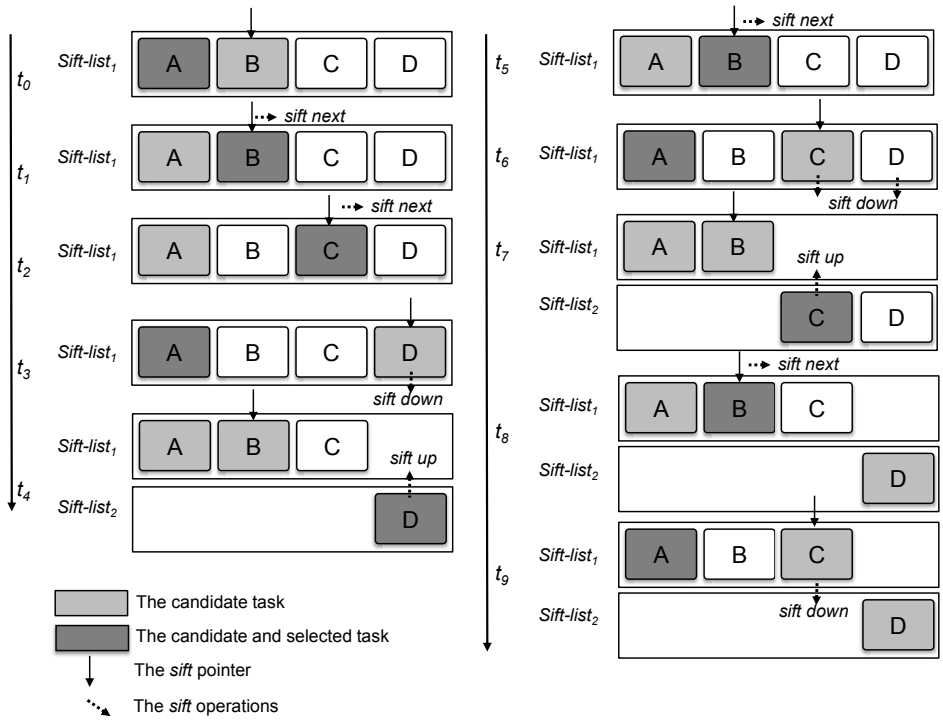


그림 4.2: K-LZF 스케줄링 알고리즘의 동작 예 (A:B:C:D=4:3:2:1)

효율적이다.

4.2 K-LZF 스케줄링 알고리즘의 동작 예

본 절에서는 K-LZF의 동작 과정을 구체적인 예와 함께 설명한다. 그림 4.2는 네 태스크 A, B, C, D 가 4:3:2:1의 지분 비율로 10 단위 시간 동안 서비스 받을 때의 실행 순서를 보여준다. 회색으로 표기된 태스크는 연결리스트 내 후보 태스크이고 검은색으로 표기된 태스크는 후보 태스크이자 그 시간에 K-LZF 스케줄링 정책으로 선택된 태스크를 나타낸다. t_0 일 때, 4개의 모든 태스크는 *sift-list*$_1$ 내에 존재한다.

초기에는 연결리스트 내에 오직 하나의 서브리스트 *sift-list*$_1$ 만 존재한다. 또한, *sift-pointer*는 순회 범위인 *sift-list*$_1$의 두 번째 태스크에 위치하고 있다. K-LZF는 초기에 잠재 만족 지수 성질에 의해 비교 없이 첫 번째 태스크를 선택하고 연산은 발생하지 않도록 하였다. t_1일 때, *sift-list*$_1$ 내 첫 번째 태스크인 A와 *sift-pointer*가 위치한 B가 후보 태스크이고 이 중 $\widehat{z}_B^{(1)}$ 값이 $\widehat{z}_A^{(1)}$보다 더 작으므로 태스크 B가 선택된다. *sift-pointer*가 지정한 태스크 B가 스케줄링 대상이 되었으므로, *sift-next* 연산이 수행되어 *sift-pointer*는 다음 태스크인 C를 가리키도록 이동한다. 마찬가지로, t_2에 후보 태스크는 A, C이고 *sift-pointer*에 의한 C가 선택되었으므로, t_2에도 *sift-next* 연산이 수행된다. t_3에서는 후보 태스크 A, D 중 *sift-list*$_1$ 내 첫 번째 태스크인 A가 스케줄링 대상이므로, *sift-down* 연산이 수행되며 *sift-pointer* 포인터는 바로 순회 범위 내 첫 번째 태스크인 B를 가리키도록 조정된다. *sift-down* 연산 후 *sift-list*$_2$에는 D가 포함된다. t_4에서는 *sift-list*$_1$, *sift-list*$_2$의 각각 첫 번째 태스크인 A, D 및 *sift-pointer*에 위치한 B가 후보 태스크이다. *sift-list*$_2$의 D가 선택되었으므로, *sift-up* 연산이 수행되어 D는 *sift-list*$_1$으로 이동된다. t_6에서의 *sift-down* 연산은 *sift-list*$_1$에 있는 C와 D를 하위 서브리스트로 모두 이동시킨다. 또한 t_9에서도 *sift-down* 연산이 수행되는데 이 때는 C가 이동하여 *sift-list*$_2$에서 D와 함께 존재하게 된다. 결과적으로 이 예에서 K-LZF 알고리즘은 A, B, C, A, D, B, A, C, B, A 순서로 실행시킨다. 이 실행 순서는 표 3.2에서 보인 LZF 스케줄링 알고리즘의 결과로 얻어진 실행 순서와 여전히 동일하며, 최소 서비스 시간 오차는 -0.6이고 최대 서비스 시간 오차 0.6이다.

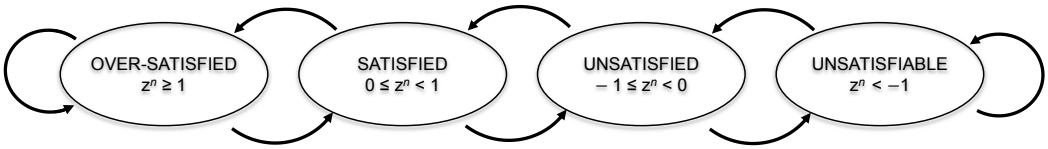


그림 4.3: K-LZF에서 만족 지수에 따른 태스크 상태

4.3 태스크 만족 지수에 따른 동적 정책

K-LZF의 세 가지 연산들은 각 서브리스트 내에서 실행 가능한 태스크들이 만족 지수 순으로 정렬되도록 유지하는 것을 목적으로 한다. 하지만, K-LZF는 *sift-down* 연산 과정에서 *sift-list*$_{K-1}$ 서브리스트가 *sift-list*$_K$와 결합되면서 만족 지수가 높은 태스크들이 후보 태스크가 될 수 있다. 이 결과는 마지막 서브리스트 *sift-list*$_K$ 내에서 잠재 만족 지수가 가장 낮은 첫 번째 태스크가 미래에 후보 태스크가 되지 못하기 때문에 공정성에 좋지 않은 영향을 미친다. 또한 마지막 서브리스트에서 항상 결합하는 경우 마지막 서브리스트에 위치한 태스크들에게 기아 (starvation) 문제가 발생할 수 있다. 이 문제를 해결하기 위해 잠재 만족 지수에 따라 태스크의 상태를 정의하고 각 상태에 따라 동적으로 연산들이 수행하도록 하여 성능을 개선한다.

그림4.3과 같이 K-LZF 알고리즘에서 태스크들은 만족 지수에 따라 OVERSATISFIED, SATISFIED, UNSATISFIED, 및 UNSATISFIABLE 4가지 만족 지수 상태를 정의하였다.

- OVERSATISFIED 상태: $\widehat{z}^{(n)} \geq 1$를 만족하는 태스크들의 상태
- SATISFIED 상태: $0 \leq \widehat{z}^{(n)} < 1$를 만족하는 태스크의 상태
- UNSATISFIED 상태: $-1 \leq \widehat{z}^{(n)} < 0$를 만족하는 태스크의 상태

- UNSATISFIABLE 상태: $\widehat{z}^{(n)} < -1$를 만족하는 태스크의 상태이며 한 번의 시간할당량으로도 SATISFIED 상태로 될 수 없는 상태

만족 지수가 높은 태스크들이 각 서브리스트의 첫 번째에서 위치하여 후보 태스크가 되는 것을 방지해기 위해 후보 태스크의 동적인 상태에 따라 *sift-down* 연산을 개선시킨다. 먼저, 첫 번째 태스크의 잠재 만족 지수가 가장 큰 서브리스트에서 *sift-down* 연산이 중지하여 서브리스트간 결합할 수 있도록 하였다. 이를 위해 최대의 잠재 만족 지수를 가진 태스크를 알아야 하는 데, 이는 서브리스트 중 최소의 잠재 만족 지수를 가진 태스크를 선택하는 과정에서 함께 계산할 수 있다. 이 과정에서 스케줄링 오버헤드는 증가하지만 *sift-down* 연산의 시간 복잡도 $O(K)$는 유지된다. 또한 4.1.2절에서 설명한 것처럼 *sift-down*연산은 *sift-list*$_1$의 첫 번째 후보 태스크가 스케줄러에 의해 선택되는 경우 수행된다. *sift-down* 연산이 호출될 시점에 *sift-pointer*가 지정하는 후보 태스크의 동적인 상태에 따라 *sift-down* 연산 여부를 결정한다. *sift-pointer*가 지정하는 후보 태스크의 상태가 OVERSATISFIED와 SATISFIED 상태라면 *sift-down*연산은 중지할 수 있다. 이것은 OVERSATISFIED와 SATISFIED 상태인 태스크들이 서브리스트의 후보 태스크가 되는 것을 방지할 수 있기 때문이다. 또한 이 동적인 방법을 통해 $O(K)$의 시간 복잡도를 가진 *sift-down* 연산의 전체 수행 횟수를 줄여 K-LZF 스케줄링 알고리즘의 연산 비용을 줄일 수 있다.

UNSATISFIABLE 상태의 만족 지수가 낮은 태스크들은 실행 후에도 각 서브리스트의 첫 번째에 유지하도록 한다. *sift-list*$_i$ (단, $i \neq 1$)의 후보 태스크들이 UNSATISFIABLE 상태이라면, 선택된 후에도 *sift-up* 연산을 수행시키지 않아 후보 태스크의 기회를 준다. *sift-list*$_i$ (단, $i \neq 1$)의 후보 태스크들이 UNSATISFIABLE

상태가 아닌 경우 선택되지 않더라도 *sift-up* 연산을 발생시킬 수 있다. 후보 태스크들의 이미 계산된 만족 지수만으로 상태를 확인할 수 있기 때문에 K-LZF 알고리즘은 태스크들의 상태 정보를 유지하기 위한 별도의 저장 공간은 필요 없다.

4.4 K 값 변화에 따른 K-LZF 알고리즘의 공정성 분석

LZF 알고리즘과 마찬가지로 K-LZF 스케줄링 알고리즘도 K 개의 *sift-list*들로 구성된 연결리스트 내에서 최소의 잠재 만족 지수를 가진 태스크를 선택하는 것을 목표로 한다. K-LZF 알고리즘은 K 값이 변함에 따라 후보 태스크의 수가 달라지므로 공정성에 영향을 받는다. K 값이 극단적으로 커져서 K 가 N 이 되면 K-LZF 알고리즘은 LZF 알고리즘의 성능을 보인다. 즉 태스크의 수에 상관없이 $O(1)$ 의 높은 공정성을 보이지만, N 개의 후보 태스크들을 모두 비교해야 하므로 스케줄링 함수는 $O(N)$ 의 시간 복잡도를 가지게 된다. 반면 K 값이 감소하여 1 이 되면, K-LZF 알고리즘은 *sift-next* 연산만 수행하여 첫 번째 태스크의 잠재 만족 지수와 비교만 하는 라운드 로빈 기반의 알고리즘의 성능을 갖는다. 이런 경우 K-LZF 알고리즘의 스케줄링 함수는 $O(1)$ 의 시간 복잡도를 가지고 효율적으로 동작하지만, 공정성은 태스크 수 (N) 가 증가할수록 악화될 수 있다.

K 값이 증가함에 따라 LZF 알고리즘에 얼마나 근접한 성능을 보이는 가를 평가하기 위해 $L(K)$ 함수를 정의한다. $L(K)$ 함수는 K 개의 후보 태스크만으로 최소 잠재 만족 지수를 가진 태스크를 찾을 수 있는 확률을 의미한다. 지분이 큰 순서로 정렬된 이 연결리스트 내에서 j 번째에 있는 태스크를 T_j 라고 하자 $(1 \leq \mathrm{j} \leq N)$. T_j 가 최소 잠재 만족 지수를 가질 때, K-LZF 스케줄러에 의해 선택되기 위해서는 이 태스크가 후보 태스크로 선정되어야 한다. $L(K)$ 함수는 각 태스크들이 최소 만족

Algorithm 4.1 K-LZF 스케줄링 알고리즘

-Initialization:

All of tasks are contained in *sift-list*[0] ordered by largest weights
uppermost ← the first task in *sift-list*[0]
$sift$ ← the second task in *sift-list*[0]
selected ← **nil**, *merged* ← **nil**
$siftz \leftarrow 0$, $minz \leftarrow 0$, $maxz \leftarrow 0$, $z[K] \leftarrow \{0\}$

-K-LZF scheduling:*sift-list*$[K]$
selected, *merged*← $sift$
$siftz$ ← **calZ:**$sift$
$minz$, $maxz \leftarrow siftz$
for all $0 \leq i < K$
    $first$ ← **getfirst**:*sift-list*[i]
    z[i] ← **calZ:**$first$
    if z[i] $< minz$ **then**
        selected ← $first$
        $minz \leftarrow z$[i]
    else if z[i] $> maxz$ **then**
        merged ← $first$
        $maxz \leftarrow z$[i]
    end if
end for

switch(*selected*)
case $sift$:
    perform *sift-next* for $sift$
case *uppermost*:
    $sift$ ← second task in *sift-list*[0]
    perform *sift-down* until *merged*
case $first$:
    perform *sift-up* for $first$
end switch
return *selected*

-getFirst: *sift-list*[i]
return the first task in *sift-list*[i]

-calcZ: T
return the potential satisfaction index of T

지수를 가질 때 후보 태스크로 선정될 확률의 합으로서 다음과 같이 정의된다.

$$\begin{aligned} L(K) &= \sum_{j=1}^{N} Pr(T_j \in C_K \mid T_j \text{ has the least } \widehat{z}) \cdot Pr(T_j \text{ has the least } \widehat{z}) \\ &= \sum_{j=1}^{N} f_j(K) \cdot p_j \end{aligned}$$

단, C_K 는 K 개의 후보 태스크 집합이고, $f_j(K)$ 는 T_j 가 최소 잠재 만족 지수를 가졌을 때 C_K 의 원소일 확률이다. 또한, 각 태스크가 최소 잠재 만족 지수를 가질 확률 $Pr(T_j \text{ has the least } \widehat{z})$ 은 지분에 비례하므로 지분 비율 p_j 로서 계산하였다. LZF 알고리즘인 경우, 후보 태스크가 N 개이므로 $L(N)$ 값을 계산해야 한다. 모든 j 에 대해 $f_j(N) = 1$ 을 만족하고 모든 지분 비율의 합 $\sum_{j=1}^{N} p_j$ 은 1이므로, $L(N) = 1$ 이다. 즉 K-LZF 알고리즘의 $L(K)$ 함수 값이 1에 가까울수록 LZF 알고리즘의 성능에 근사하다고 할 수 있다.

만약 K-LZF 알고리즘에서 N 개의 모든 태스크 중에서 임의로 K 개의 후보 태스크를 선택한다고 가정하면, $L(K)$ 함수는 다음과 같이 정해진다.

$$L(K) = \sum_{j=1}^{N} \frac{{}_{N-1}C_{K-1}}{{}_{N}C_{K}} \cdot p_j = \frac{K}{N} \sum_{j=1}^{N} p_j = \frac{K}{N}$$

따라서 N 개 중에서 K 개의 태스크를 후보 태스크로 임의로 선정하는 경우 K-

LZF 알고리즘은 확률적으로 K에 선형 비례하여 LZF에 근사함을 확인 할 수 있다. *sift-list*를 사용하는 경우 더 높은 $L(K)$ 값을 기대할 수 있다. 계산을 간단하게 하기 위해 *sift-up*, *sift-down* 연산만 고려하였고 이전 태스크인 T_{j-1}와의 위치 관계에 따라 태스크 T_j의 상태는 다음과 같이 정의한다.

- UP 상태: T_{j-1}와 T_j가 동일한 리스트에 위치한 경우 UP 상태라고 정의하고 상태 확률은 α_j 이다.
- DOWN 상태: T_{j-1}와 T_j가 태스크와 다른 리스트에 위치한 경우 DOWN 상태라고 정의하고 상태 확률은 β_j 이다.(단, $\alpha_j + \beta_j = 1$)

근사적인 $f_j(K)$ 함수을 계산하기 위해 오랜 시간 태스크들을 관찰 하였을 때 태스크 T_j의 각 상태 확률 α_j과 β_j는 *sift-up*과 *sift-down*을 연산들의 실행 비율로 고려하였다. $\alpha_j = \frac{p_j}{p_{j-1}+p_j}$ 이고 $\beta_j = \frac{p_{j-1}}{p_{j-1}+p_j}$으로 계산된다고 가정한다. 예를 들어 태스크 T_2가 UP 상태에 있는 경우는 T_2이 실행되어 *sift-up* 연산이 발생할 때이고, 태스크 T_2가 DOWN 상태에 존재하는 경우는 T_1가 실행되어 *sift-down* 연산이 호출될 때이다. 또한, 모든 태스크들에 대해 이 두 상태 확률은 같다고 가정한다. 이는 랜덤 지분 분포를 따르는 태스크들이 연결리스트 내에서 정렬되어 있다고 가정하였으므로 실행큐내에 태스크가 많아질수록 두 태스크들의 지분 비율은 거의 같아지기 때문이다 (즉, $p_{j-1} \approx p_j$).

K개의 *sift-list*를 사용하는 K-LZF 알고리즘에서는 K번째 까지 위치한 태스크들은 다른 이전 태스크들의 상태에 관계없이 후보 태스크가 될 수 있으므로 1이 된다. 반면 서브리스트의 개수인 K보다 더 큰 태스크들이 후보 태스크가 되기 위해서는 앞에 위치한 태스크들 중 DOWN 상태에 있는 태스크들이 $(K-1)$개 이하이어야

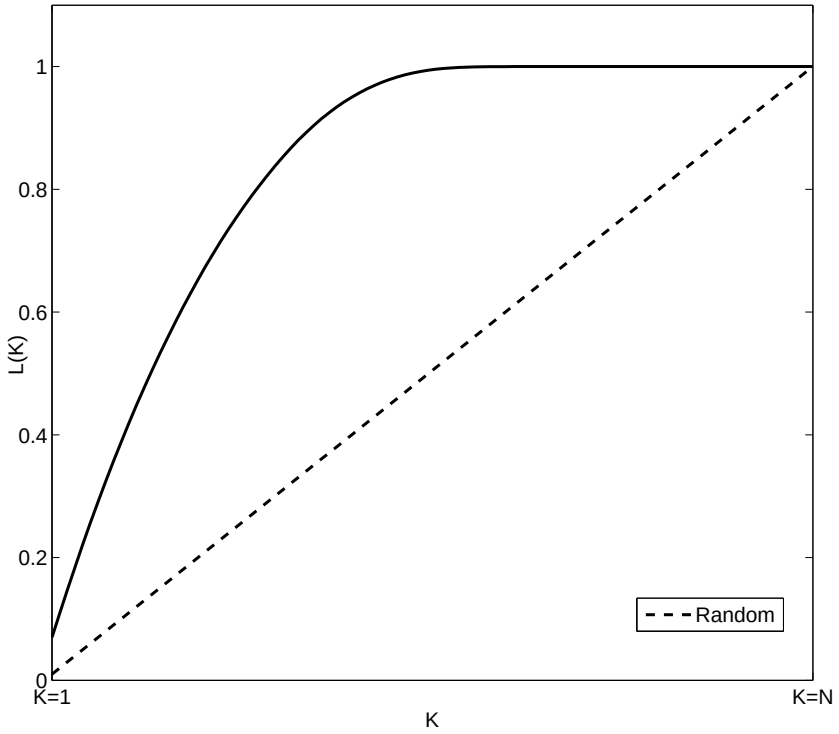


그림 4.4: K-LZF 알고리즘에서 K 값 변화에 따른 $L(K)$ 함수

한다. 이 중에서 최상위 서브리스트에 존재하는 첫 번째 태스크는 항상 후보 태스크이므로 자기 자신을 제외한 $(j-2)$ 개 중 $(K-2)$ 개까지만 DOWN 상태에 있어야 j 번째 태스크가 후보 태스크가 될 수 있다. 따라서 K 개의 *sift-list*를 이용하는 K-LZF 알고리즘에서 후보 태스크가 될 확률은 다음과 같다.

$$f_j(K) = \begin{cases} 1 & \text{if } j \leq K \\ 1 - \sum_{r=0}^{r=(K-2)} {}_{j-2}C_r \cdot \alpha^{(j-2)-r}\beta^r & \text{else if } K < j \leq N \end{cases}$$

그림 4.4는 K-LZF 알고리즘에서 K 값 변화에 따른 $L(K)$ 함수를 보여준다. 특히

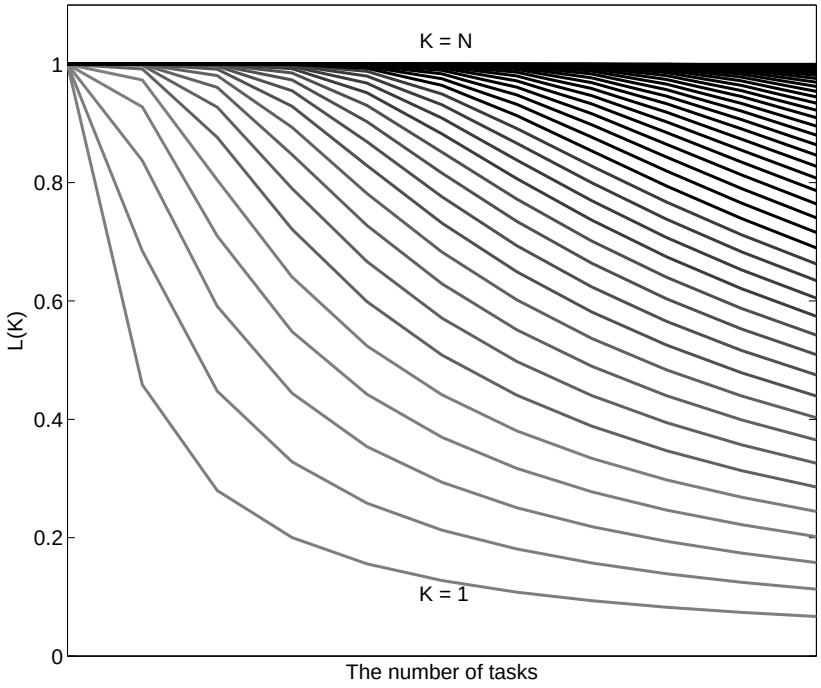


그림 4.5: K-LZF 알고리즘에서 태스크 수에 따른 K의 효과

비교를 위해 임의로 후보 태스크를 선택했을 때의 $\frac{K}{N}$를 함께 나타내었다. K 값이 증가함에 따라 $L(K)$ 함수는 K에 비례한다. 특히 K-LZF 알고리즘의 $L(K)$ 함수는 선형 비례보다는 더 높은 비율로 증가하며, $\frac{K}{N}$는 여유있는 하한을 의미한다.

LZF 알고리즘은 태스크의 수에 상관없이 항상 1을 갖는 반면 K-LZF 알고리즘은 태스크의 수가 다른 환경에서 $L(K)$함수에 대한 K의 효과가 달라질 수 있다. K-LZF 알고리즘에서 $K < N$인 경우, 태스크 수 N에 대해 $L(K)$는 감소 함수이다. 특히 K 값이 작을수록 감소하는 비율이 더욱 커져 그림 4.5와 같이 스펙트럼 형태로 나타난다. 물론 이 확률적인 $L(K)$ 함수 값은 전체 태스크 수(N)에 따라 달라진다.

제 5 장 성능 평가

본 장에서는 K-LZF 스케줄링 알고리즘의 성능 평가를 다룬다. 제안된 K-LZF는 대표적인 공정 스케줄링 알고리즘인 WFQ, WF^2Q, DRR, VTRR, GR^3들과 공정성과 효율성 측면에서 비교 분석되었다. 서버 시스템에서 발생 가능한 다양한 태스크 집합들의 시뮬레이션을 통해 수식 (3.2)에서 보인 최소/최대 서비스 시간과 평균 서비스 시간 오차를 측정하여 공정성을 분석하였다. 또한 AVOS 운영체제와 리눅스 운영체제에서 제안된 스케줄링 알고리즘의 평균 실행 시간을 측정하여 효율성에 대한 평가를 하였다.

5.1 실험 환경

성능 평가를 위해 두 가지 종류의 지분 분포를 가진 태스크 집합이 사용되었다. 첫 번째는 랜덤 지분 분포로 N개의 모든 태스크에게 일정 범위 $[1, 1024]$에서 균일분포를 따르는 난수 값으로 지분을 부여하는 방식이다. 그러므로 이 지분 분포에서 태스크의 수가 증가할수록 지분의 총합은 증가한다. 두 번째 고려한 지분 분포는 편향 지분 분포이다. 편향 지분 분포에서는 미리 정해진 D개의 태스크들이 전체 총합의 일정 비율$(\eta = 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8,$ 및 $0.9)$을 가지도록 분포시킨다. D개 태스크들도 편향적인 지분을 갖도록 특정 범위에서 균일분포의 난수 값을 생성하여 부여하였다. 먼저 태스크들 중 D개를 제외한 $N - D$개의 태스크에게 랜덤 지분 분포로 할당한다. 그 후 D개의 지분이 전체 지분 총합의 η비율 만큼 가지도록

분포시킨다. 따라서 $N - D$ 개 태스크들의 지분 합과 D 개의 태스크들의 지분 합의 비율은 1 - η : η 이 된다. 이러한 분포는 서버 시스템에서 흔히 발생할 수 있는 데몬 태스크들을 고려한 편향된 분포이다. 예를 들면, N=1,024, D=12 이고 η=0.3 이라면, 12개의 데몬 태스크가 전체 지분의 30%를 차지하고 1012개의 태스크가 나머지 70%의 지분을 갖는 편향된 지분을 갖는 분포를 설명한다. 이 때, η 가 0 이라면, 첫 번째 지분 분포인 랜덤 지분 분포와 동일하다.

모든 태스크들에 대해서 단위 시간마다 서비스 시간 오차를 측정하여 공정성 평가를 하였다. 공정성을 평가하기 위해 매 단위 시간 모든 태스크들의 서비스 시간 오차 중 최소 서비스 시간 오차와 최대 서비스 시간 오차를 측정하였다. 3장에서 설명한 것과 같이 최소 서비스 시간 오차는 음의 실수로 표현되며, 최대 서비스 시간 오차는 양의 실수로 표현된다. 이것은 N 개의 태스크 집합에서 그 최소 또는 최대 서비스 시간 오차를 가지는 태스크가 존재함을 의미하는 것으로 그 값들이 0 에 가까울수록 공정하다고 할 수 있다. 또한, 태스크 집합 전체의 공정성을 살펴보기 위해 평균 서비스 시간 오차를 분석하였다. 평균 서비스 시간 오차는 모든 태스크에 대한 단위 시간당 서비스 시간 오차 제곱합의 평균값을 구한 후 그 값의 제곱근을 취해 계산되었다. 특히 평균 서비스 시간 오차는 GR^3 와 K-LZF에 대해서 더 정확한 비교 분석을 위해 사용되었다.

다중처리기 기반 K-LZF 알고리즘의 성능을 분석하기 위해 1.2장에서 소개된 전역 실행큐 모델에 기반을 두어 시뮬레이션 프로그램에 구현되었다. 모든 실행 가능한 태스크들은 전역적인 공동 실행큐에 존재하고 사용 가능한 처리기가 있으면 K-LZF 알고리즘에 의해 그 처리기로 스케줄링 되는 것이다. 그러나 다중처리기 기반 시스템에서 어떤 태스크들도 여러 처리기에서 동시에 수행될 수 없다. 항상 태스크들은 여러 처리기 중 하나의 처리기에서만 수행되어야 한다. 어떤 태스크가

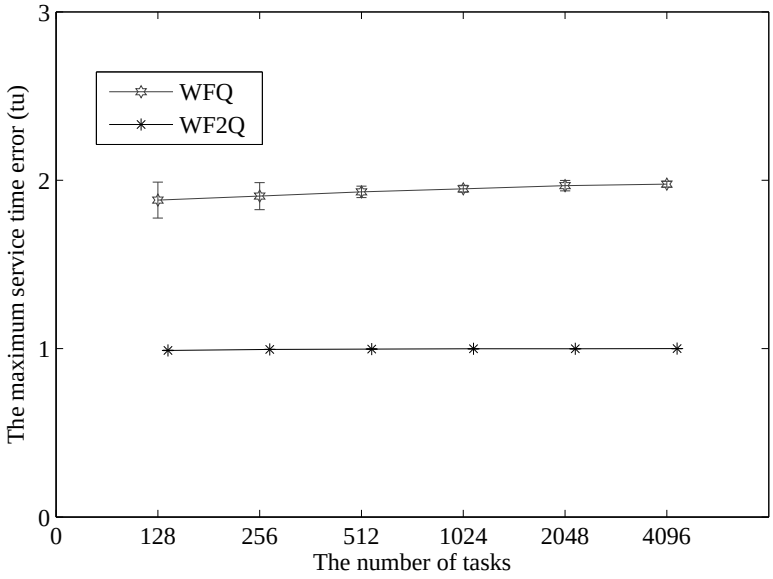
처리기에 이미 할당 된 후에도 다시 K-LZF 알고리즘에 의해 선택되었을 경우 이전에 할당된 처리기에서 계속 수행될 수 있도록 해야 한다. 이를 위해 어떤 처리기에 할당된 태스크 T_i가 다시 스케줄러에 의해 선택될 경우 선택된 횟수에 비례하여 처리기 자원을 연속적으로 사용할 수 있도록 할당된 처리기 정보와 시간할당량 정보 l_i를 관리한다.

다중처리기 기반 시뮬레이션에서 K-LZF는 다음과 같이 동작한다. 먼저 모든 처리기들은 사용 가능할 때마다 알고리즘 4.1에서 설명된 K-LZF 스케줄링 정책에 따라 태스크를 선택한다. 처음 선택된 태스크 T_i의 시간할당량 정보인 l_i의 값이 0이라면 1만큼 증가시키고 그 태스크를 처리기에 할당한다. 이 때, 선택된 태스크의 l_i 값이 0이 아니라면 T_i가 이미 다른 처리기에 할당된 상태이므로 l_i값을 1만큼 증가시키고 이미 할당된 처리기에서 계속 수행되도록 한다. 그 후 다른 태스크를 선택하기 위해 다시 K-LZF 스케줄링 알고리즘이 수행된다. 이 방식은 실행큐에서 실행 가능한 태스크의 수 N이 처리기의 수 M보다 커야 가능하다. 만약 태스크의 수가 처리기 수보다 작거나 같다면 태스크를 처리기에 고정하여 정적 할당 방식으로 동작할 수 있기 때문이다. 처리기가 실행을 마칠 때 실행된 태스크의 l_i의 값을 1만큼 감소시킨다. l_i값이 0이 아니라면 다시 그 태스크를 계속 실행시키고 0인 경우 다시 K-LZF 스케줄링 알고리즘을 호출하여 다른 태스크를 선택한다. 이 방식은 전역적으로 K-LZF 스케줄링에 의해 높은 공정성을 지원하면서 캐쉬의 효과도 기대할 수 있다.

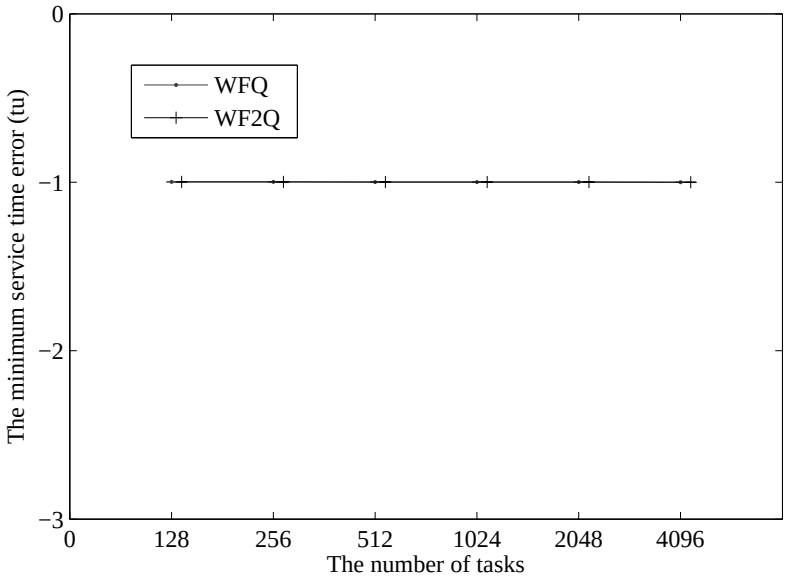
5.2 시뮬레이션 실험 결과

본 절에서는 시뮬레이션을 통해 K-LZF 알고리즘과 WFQ, WF^2Q, DRR, VTRR, 및 GR^3 알고리즘의 랜덤 지분 분포와 편향 지분 분포 환경에서 공정성을 평가한다. 첫 번째 실험에서 태스크 수에 따른 공정성을 평가하기 위해, 태스크 수를 128에서 4,096개까지 태스크를 증가시키며 서비스 시간 오차를 측정하였다. 5.1절에서 설명한 랜덤 지분 분포를 통해 태스크들에게 지분을 부여하였다.

그림 5.1~그림 5.3은 랜덤 지분 분포에서 WFQ, DRR, VTRR, GR^3 및 제안 알고리즘인 K-LZF(K=5)의 최소/최대 서비스 시간 오차를 보여준다. 그림 5.1에서와 같이 랜덤 지분 분포에서 WFQ의 서비스 시간 오차는 [-1, 2]의 범위를 가지며 태스크 수에 상관없이 일정한 서비스 시간 오차를 보여 높은 공정성을 보장하는 것을 알 수 있다. 마찬가지로 WF^2Q 알고리즘도 최소/최대 서비스 시간 오차가 [-1, 1] 범위에서 나타나는 것을 확인하였다. 그림 5.2는 랜덤 지분 분포에서 라운드 로빈 기반 알고리즘인 DRR, VTRR의 최소/최대 서비스 시간 오차를 보여준다. DRR의 서비스 시간 오차는 128개의 태스크를 동작했을 때, [-378, 552]의 범위를 가진다. 반면 태스크의 수가 4,096까지 증가하였을 때 서비스 시간 오차는 [-927, 1021]으로 최소/최대 서비스 시간 오차 간 차이가 커졌다. VTRR 알고리즘의 경우 태스크 수가 128개에서 4,096개로 증가할 때 [-26, 447]~[-30, 1016]으로 서비스 시간 오차 간 차이가 증가하였다. VTRR와 DRR 모두 태스크 수가 증가함에 따라 최소/최대 서비스 시간 오차 간 차이가 커졌다. 비록 라운드 로빈 기반 알고리즘들이 가상 시간 기반 알고리즘들에 비해 공정성이 악화되었지만, $O(1)$의 시간 복잡도로 스케줄링 할 수 있다는 장점이 있다. 반면 그림 5.3은 GR^3 알고리즘과 K-LZF 알고리즘이 태스크 수에 관계없이 항상 일정한 최소/최대 서비스 시간 오차를 가지는 것을 보

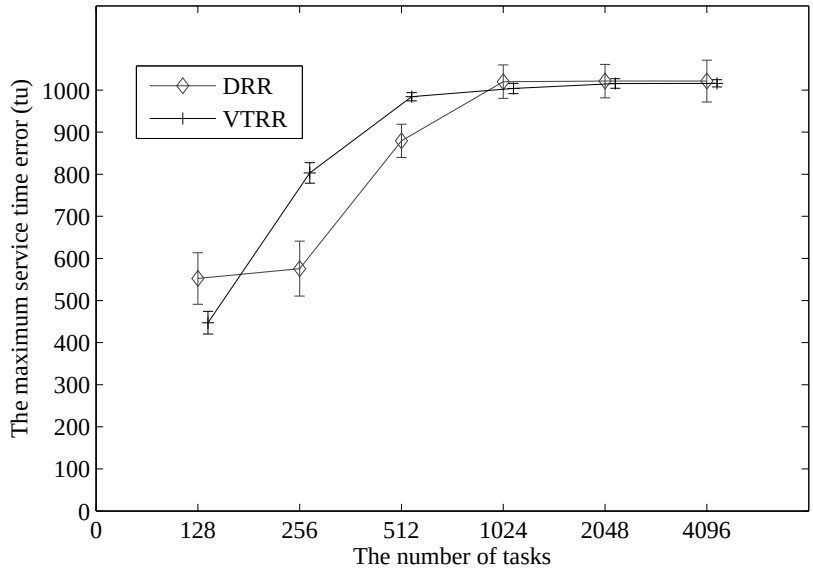


(a) 최대 서비스 시간 오차

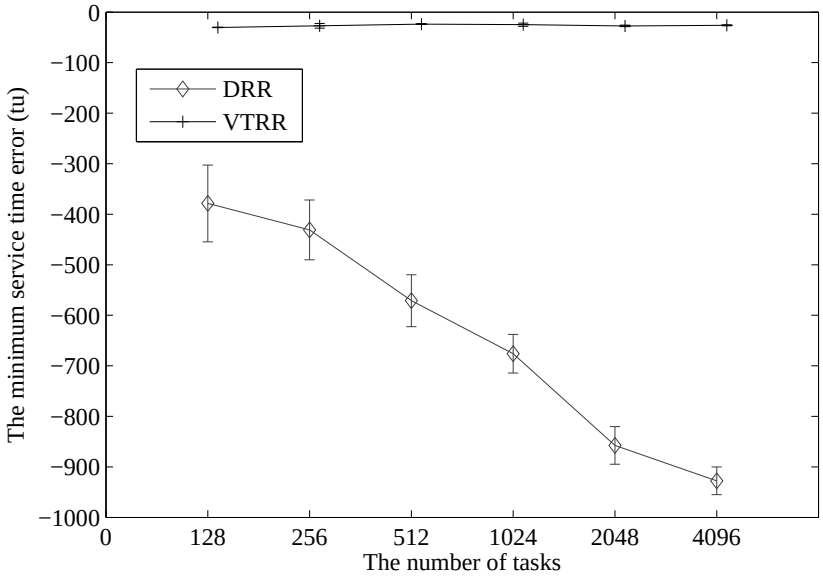


(b) 최소 서비스 시간 오차

그림 5.1: 랜덤 지분 분포에서 태스크 수에 따른 가상 시간 기반 알고리즘의 최소/최대 서비스 시간 오차(WFQ, WF^2Q)

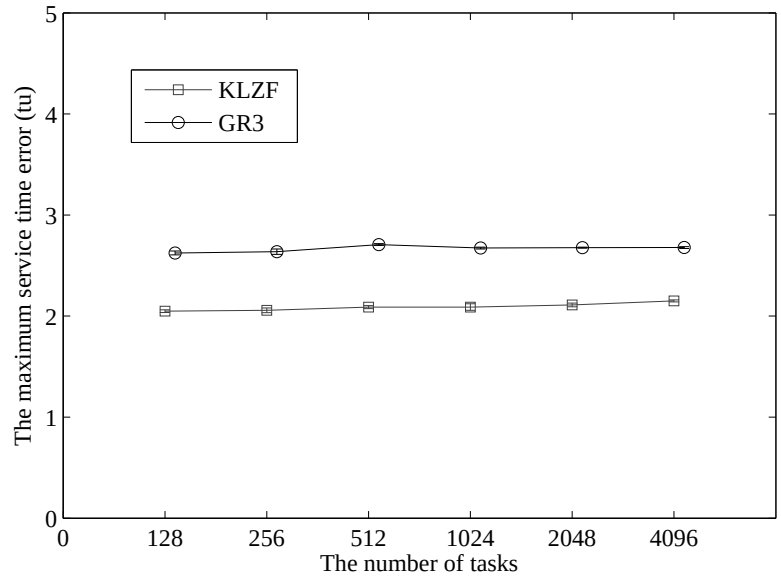


(a) 최대 서비스 시간 오차

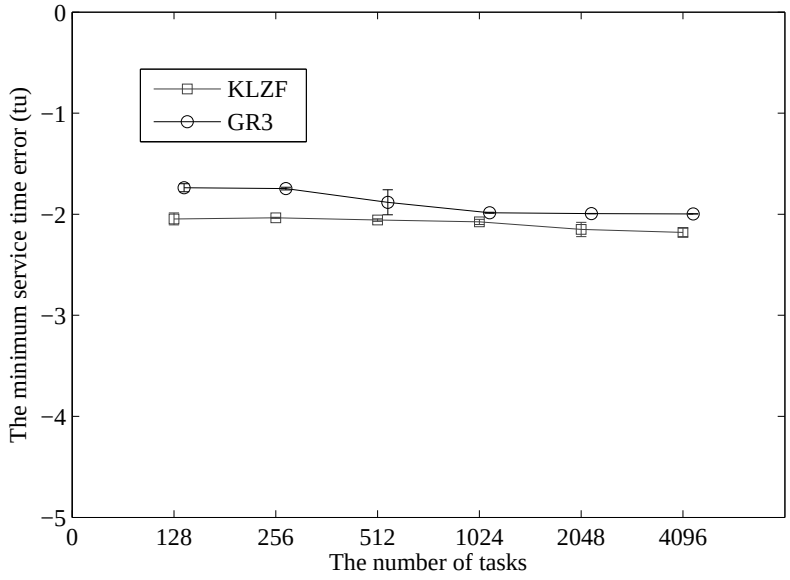


(b) 최소 서비스 시간 오차

그림 5.2: 랜덤 지분 분포에서 태스크 수에 따른 라운드 로빈 기반 알고리즘의 최소/최대 서비스 시간 오차 (DRR, VTRR)



(a) 최대 서비스 시간 오차



(b) 최소 서비스 시간 오차

그림 5.3: 랜덤 지분 분포에서 태스크 수에 따른 혼합 기반 알고리즘의 최소/최대 서비스 시간 오차 (GR^3, K-LZF)

표 5.1: 랜덤 지분 분포에서 GR^3와 K-LZF의 평균 서비스 시간 오차

태스크 수	GR^3의 평균 서비스 시간 오차	K-LZF의 평균 서비스 시간 오차
128	0.690635	0.453893
256	0.714095	0.458840
512	0.731051	0.467232
1024	0.739053	0.470186
2048	0.743701	0.472933
4096	0.743561	0.474712

여준다. GR^3 알고리즘의 서비스 시간 오차 범위는 [-1.73, 2.62]∼[-1.99, 2.67]이였다. 마찬가지로 제안된 K-LZF 알고리즘 역시 태스크 수가 증가하더라도 거의 변화 없는 [-2.03, 2.04]∼[-2.15, 2.17]의 서비스 시간 오차 범위를 보인다. GR^3 알고리즘이 최소 서비스 시간 오차측면에서 다소 좋은 공정성을 보였다.

하지만, K-LZF의 향상된 성능은 전체 태스크들에 대한 단위 시간당 평균 서비스 시간 오차를 통해 알 수 있다. 표 5.1는 랜덤 지분 분포에서 GR^3와 K-LZF 알고리즘의 평균 서비스 시간 오차를 보인다. K-LZF 알고리즘의 평균 서비스 시간 오차는 GR^3 알고리즘에서 구한 평균 서비스 시간 오차의 약 63%정도이다. 이 결과는 K-LZF 알고리즘이 LZF 알고리즘을 잘 근사하고 있음을 보여준다.

서버 시스템에서 D개의 데몬 태스크들이 편향된 지분을 가지고 있다고 가정하고 편향 지분 분포 상황에서 성능 평가를 수행하였다. 먼저 $N - D$개의 태스크에게 일정 범위 [1, 1024] 내에서 균일분포를 따르는 난수 값으로 지분을 부여한다. 나머지 D개의 태스크들은 전체 지분 총합의 일정 비율($\eta = 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8,$ 및 0.9)을 유지하도록 하였다. 즉 η가 커질수록 지분 분포의 편향성은 크다고

할 수 있다.

그림 5.4∼그림 5.12은 편향 지분 분포에서 N=256, 1,024 및 4,096 일 때 제안된 K-LZF(K=5)과 기존 알고리즘들의 서비스 시간 오차를 보여준다. 그림 5.4는 편향 지분 분포에서 N=256일 때 WFQ과 WF^{2}Q 알고리즘의 서비스 시간 오차를 보여준다. WF^{2}Q 알고리즘은 여전히 [-1, 1] 범위의 서비스 시간 오차를 보장한다. WFQ 알고리즘의 최대 서비스 시간 오차 값의 증가 폭이 크게 나타나 y축은 로그 단위로 나타내었다. 이전 논문에서 증명된 바와 같이 WFQ의 하한은 편향성이 증가하더라도 -1임을 확인하였다 [DKS89]. 반면, 최대 서비스 시간 오차는 17.72에서 155까지 증가하였다. 그림 5.7과 그림 5.10은 편향 지분 분포에서 각각 N=1,024와 4,096일 때 실험 결과들을 보여준다. N=1,024와 4,096일 때 실험한 결과들과 비교하면, 태스크 수에 비례하여 이 오차들의 범위는 더욱 커진다. N=1,024인 경우 η가 0.1에서 0.9까지 증가함에 따라 [-0.99, 69]에서 [-0.99, 602]까지 서비스 시간 오차 범위가 커짐을 확인할 수 있다. 이 오차의 범위는 N=4,096일 때, 서비스 시간 오차는 [-0.99, 312]에서 [-0.99, 2251]까지 증가하였다. 이 결과들은 WFQ 알고리즘은 최소 서비스 시간 오차는 하한을 보장하는 반면 최대 서비스 시간 오차가 지분들의 편향 정도에 따라 크게 증가할 수 있음을 의미한다. 또한 태스크 수가 증가하면, 이 오차는 더욱 커지는 것을 알 수 있다.

그림 5.5는 편향 지분 분포에서 N=256일 때 라운드 로빈 기반 알고리즘인 DRR과 VTRR의 최소/최대 서비스 시간 오차를 나타낸다. DRR 알고리즘의 최소/최대 서비스 시간 오차는 η=0.1일 때 [-473, 7454]에서 η=0.9 일 때 [-129420, 235919]까지 매우 크게 증가한다. 그림 5.8과 그림 5.11에서 확인 할 수 있듯이 태스크 수가 N=1,024와 N=4,096 일 때와 비교하면, DRR 알고리즘의 서비스 시간 오차는 더욱 커진다. VTRR 알고리즘은 DRR 알고리즘에 비해 상대적으로 서비스 시간 오차는

작았다. 하지만, VTRR 알고리즘 역시 N=256인 경우 편향 정도가 η=0.1일 때 서비스 시간 오차는 [-53, 641]에서 η=0.9 일 때 [-206, 962]으로 증가하였다. 뿐만 아니라 N=4,096에서 수행된 결과인 그림 5.11과 비교했을 때, VTRR 알고리즘에서 서비스 시간 오차들의 범위도 [-824, 16815]에서 [-2855, 94276]로 태스크 수가 증가함에 따라 서비스 시간 오차가 비례하여 매우 크게 증가하는 것을 알 수 있다. 이는 서버 시스템 내 태스크들의 지분이 편향되고 태스크 수가 증가할수록 라운드 로빈 기반 알고리즘들의 공정성 성능은 더욱 악화될 수 있음을 의미한다.

그림 5.3로부터 GR^3 알고리즘은 랜덤 지분 분포에서 태스크 수에 관계없이 일정한 공정성을 보장함을 보였다. 하지만, GR^3와 K-LZF 알고리즘의 성능 차이는 그림 5.6에서 확인할 수 있다. GR^3 알고리즘에서 서비스 시간 오차는 N=256일 때 η가 증가할수록 [-1.81, 2.59]에서 [-8.21, 10.61]까지 증가하였다. 또한 그림 5.9 및 그림 5.12를 통해 편향 지분 분포에서 태스크 수가 증가할수록 GR^3 알고리즘의 서비스 시간 오차는 더 증가한 것으로 확인 되었다. N=4,096일 때 GR^3 알고리즘의 서비스 시간 오차는 [-11.38, 12.09]까지 증가하였다. 반면, K-LZF 알고리즘의 서비스 시간 오차는 N=256일 때 η가 증가할수록 [-2.23, 2.12]~[-2.31, 2.15]이었다. 즉 K-LZF 알고리즘은 편향 정도와 무관하게 높은 공정성을 보였다. N=4,096까지 증가하여도 [-2.47, 2.41]~[-3.13, 2.90]으로 나타나 편향된 지분 분포에서 태스크 수가 증가하여도 K-LZF의 높은 공정성 성능은 유지되었다.

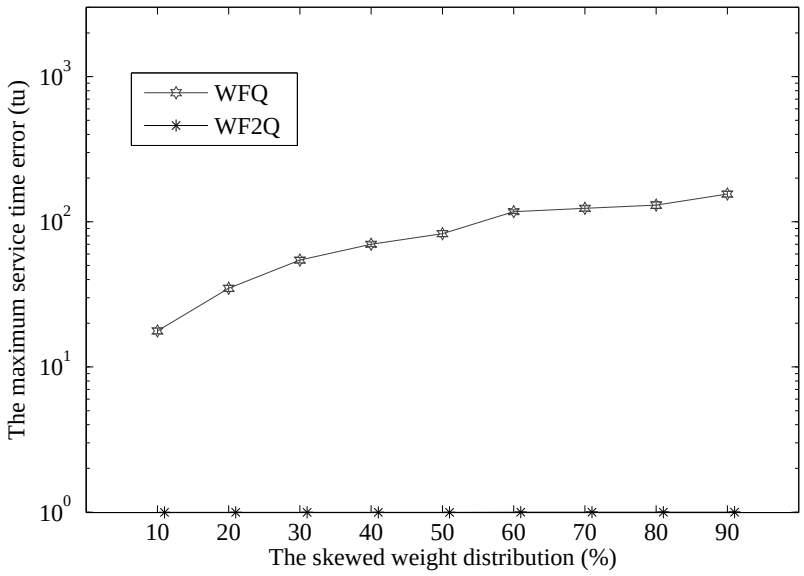
GR^3 알고리즘과 K-LZF 알고리즘의 성능 차이는 평균 서비스 시간 오차 기준으로 확연히 구분된다. 표 5.2에서 확인할 수 있듯이 편향 지분 분포(N=4,096)에서도 K-LZF 알고리즘이 상대적으로 더 작은 평균 서비스 시간 오차를 보여준다. K-LZF 알고리즘의 평균 서비스 시간 오차는 편향 정도에 따라 GR^3 알고리즘의 평균 서비스 시간 오차의 약 63% ~ 91%정도이다. 평균 서비스 시간 오차가 0 인

표 5.2: 편향 지분 분포에서 GR^{3}와 K-LZF의 평균 서비스 시간 오차(N=4,096)

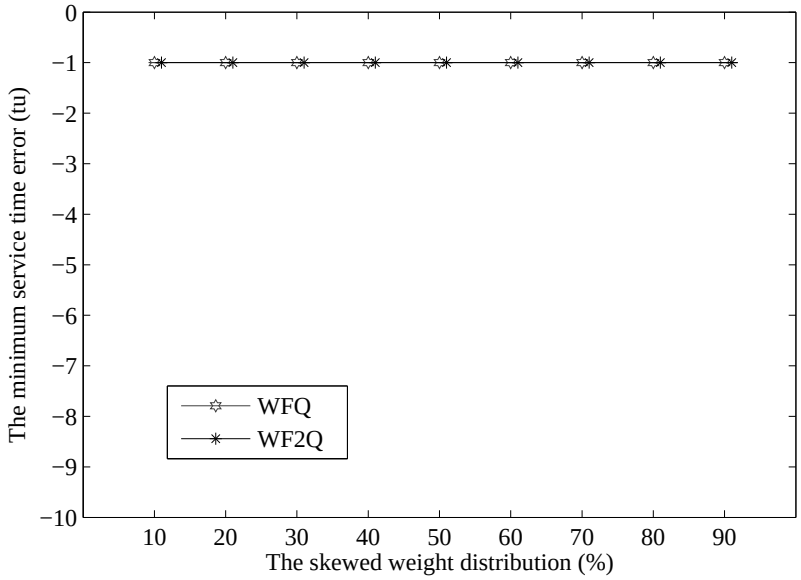
편향 정도η(%)	GR^{3}의 평균 서비스 시간 오차	K-LZF의 평균 서비스 시간 오차
10	0.733409	0.468291
20	0.742206	0.668214
30	0.742270	0.643615
40	0.742379	0.683592
50	0.742907	0.681548
60	0.742637	0.610709
70	0.742763	0.574993
80	0.743007	0.547383
90	0.743887	0.545899

이상적인 GPS과 비교했을 때 제안된 K-LZF 알고리즘이 더 높은 공정성을 보장할 수 있음을 의미한다. 이러한 결과들을 통해 다양한 종류의 태스크들과 데몬 태스크들이 혼재하더라도 K-LZF 알고리즘은 확장성을 제공하여 서버 시스템의 스케줄링 알고리즘으로 적합하면서도 우수한 성능을 제공한다는 결론을 내릴 수 있다.

또한, K-LZF 알고리즘에서 K 값의 변화를 통해 서버 시스템의 용도에 따라 공정성과 효율성의 성능을 조절할 수 있다. K-LZF 알고리즘에서 K 값에 따라 서브 리스트의 개수가 변한다. 서브 리스트 개수의 증가는 후보 태스크 수가 많아지는 동시에 비교 연산도 증가하므로 공정성과 효율성 사이에서 trade-off 효과를 준다. K-LZF 알고리즘의 시간 복잡도는 $O(K)$ 이므로 극단적으로 K 가 증가하여 태스크 수와 동일할 경우 K-LZF는 LZF 알고리즘이 된다. 따라서 K-LZF 알고리즘은 LZF의 높은 공정성을 보장할 수 있다. 반면 K 가 감소할 경우 공정성은 낮아질수 있지

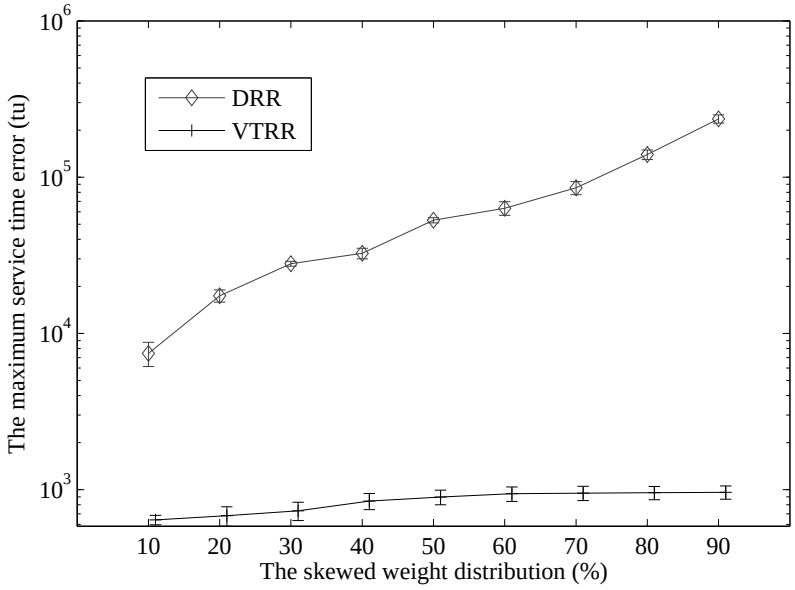


(a) 최대 서비스 시간 오차

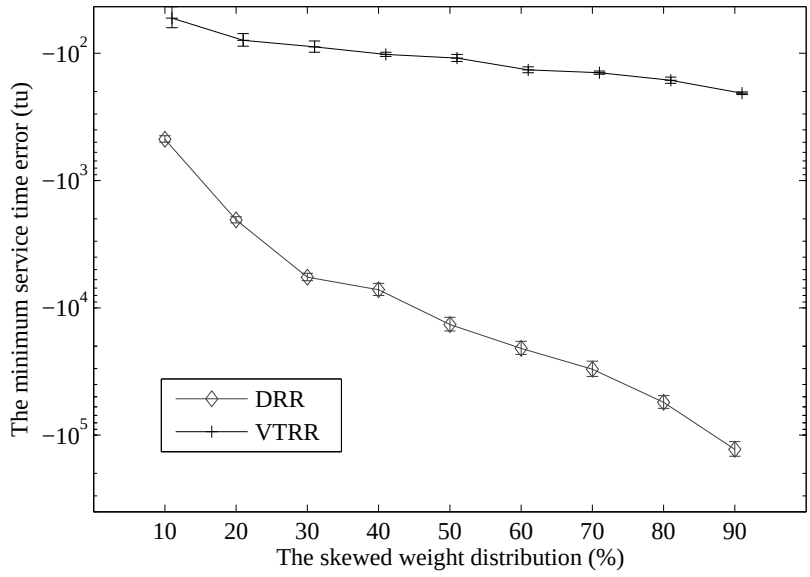


(b) 최소 서비스 시간 오차

그림 5.4: N=256 일 때 편향 지분 분포에서 가상 시간 기반 알고리즘의 최소/최대 서비스 시간 오차 (WFQ, WF^{2}Q)

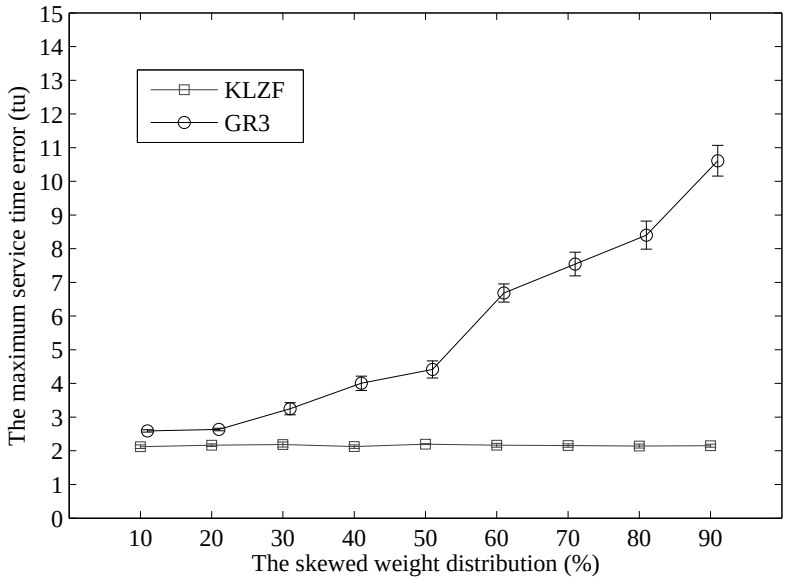


(a) 최대 서비스 시간 오차

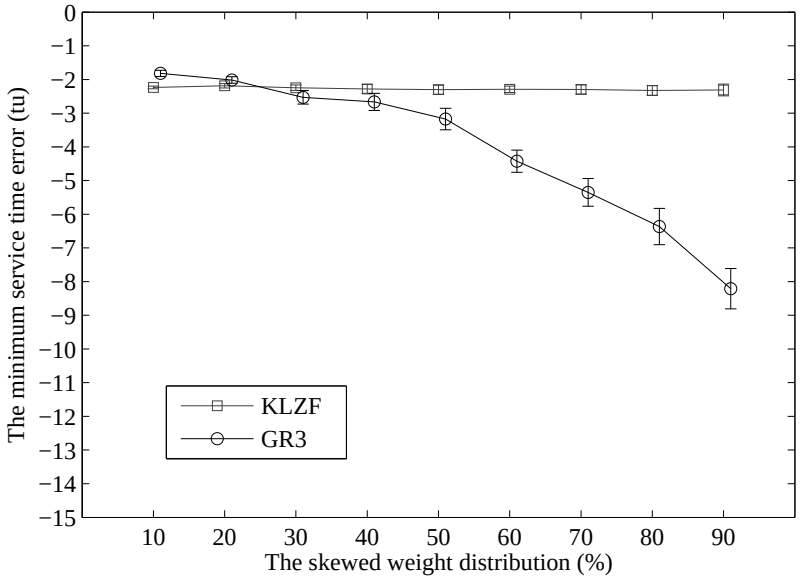


(b) 최소 서비스 시간 오차

그림 5.5: N=256 일 때 편향 지분 분포에서 라운드 로빈 기반 알고리즘의 최소/최대 서비스 시간 오차 (DRR, VTRR)

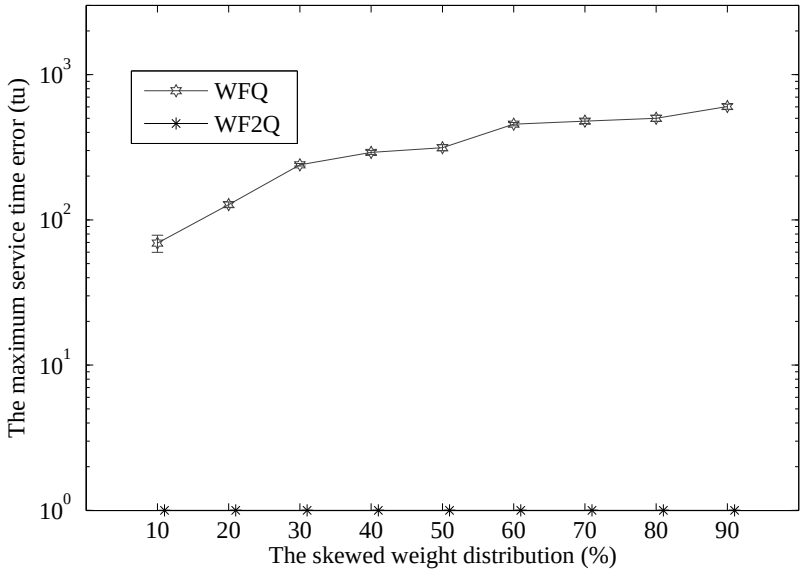


(a) 최대 서비스 시간 오차

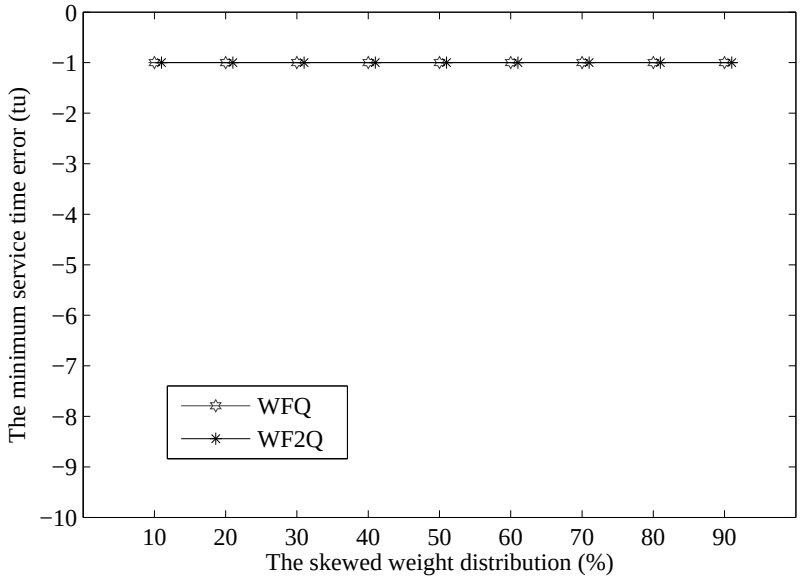


(b) 최소 서비스 시간 오차

그림 5.6: N=256 일 때 편향 지분 분포에서 혼합 기반 알고리즘의 최소/최대 서비스 시간 오차(GR^3, K-LZF)

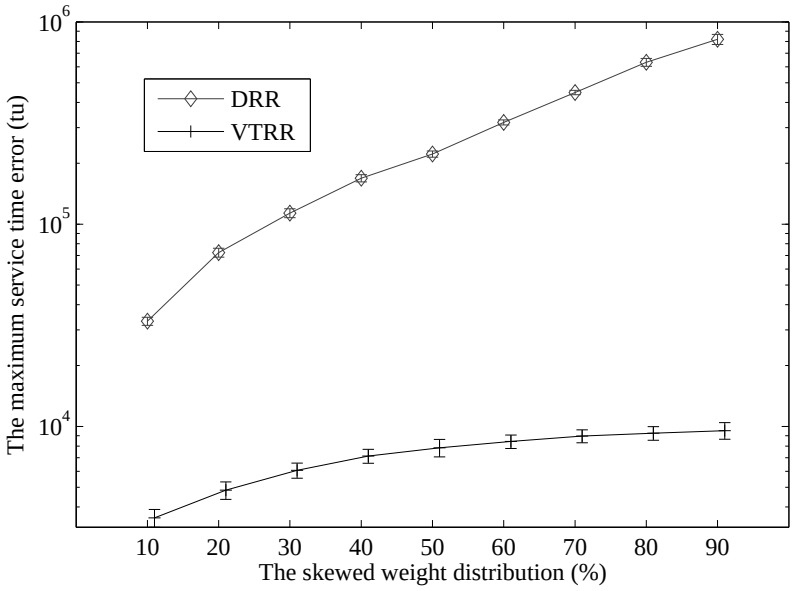


(a) 최대 서비스 시간 오차

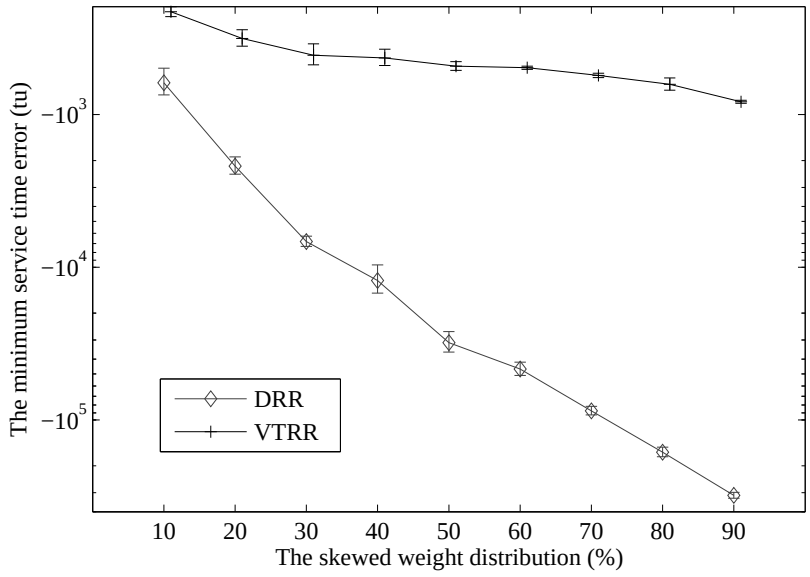


(b) 최소 서비스 시간 오차

그림 5.7: N=1,024 일 때 편향 지분 분포에서 가상 시간 기반 알고리즘의 최소/최대 서비스 시간 오차 (WFQ, WF^{2}Q)

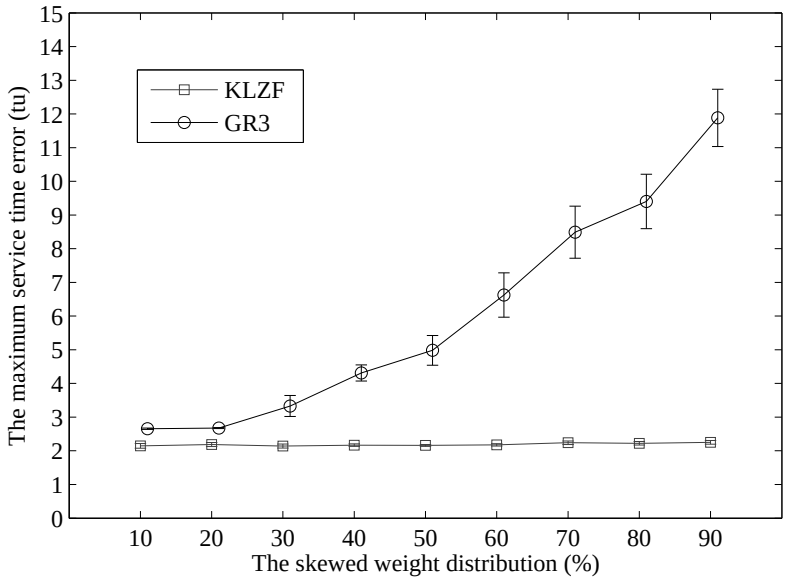


(a) 최대 서비스 시간 오차

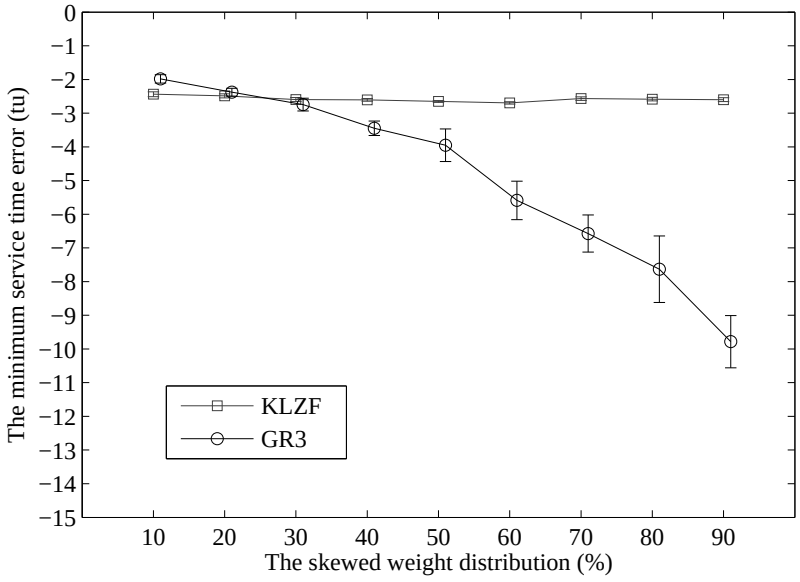


(b) 최소 서비스 시간 오차

그림 5.8: N=1,024 일 때 편향 지분 분포에서 라운드 로빈 기반 알고리즘의 최소/최대 서비스 시간 오차 (DRR, VTRR)

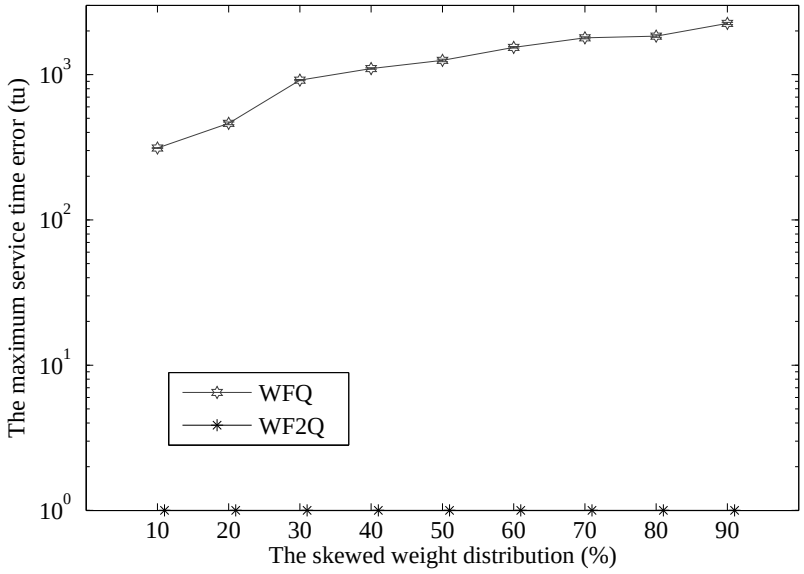


(a) 최대 서비스 시간 오차

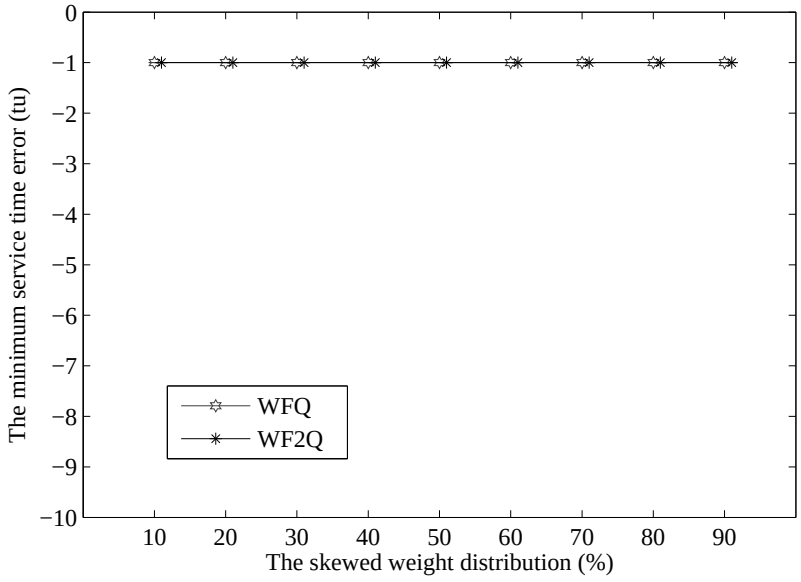


(b) 최소 서비스 시간 오차

그림 5.9: N=1,024 일 때 편향 지분 분포에서 혼합 기반 알고리즘의 최소/최대 서비스 시간 오차(GR^3, K-LZF)

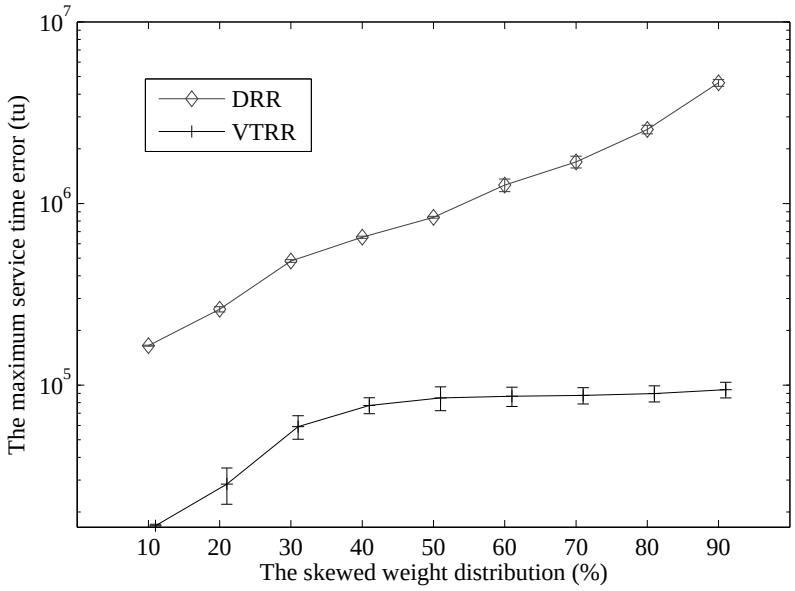


(a) 최대 서비스 시간 오차

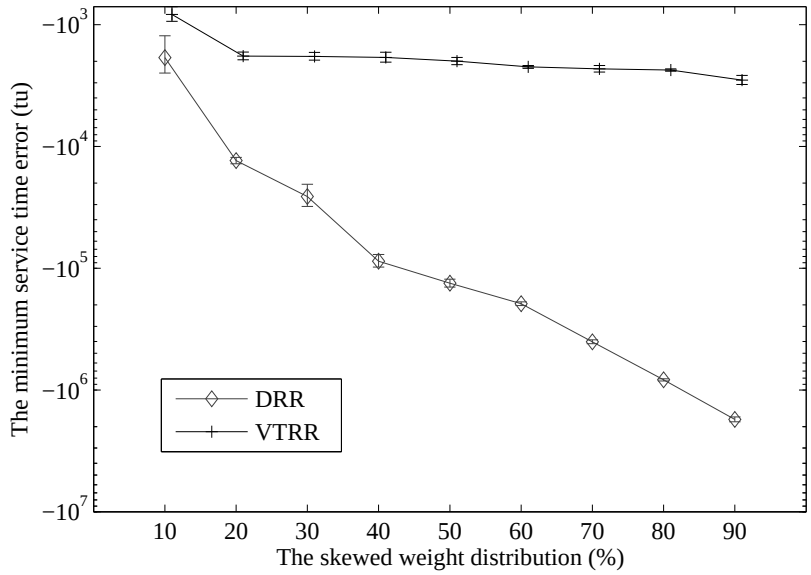


(b) 최소 서비스 시간 오차

그림 5.10: N=4,096 일 때 편향 지분 분포에서 가상 시간 기반 알고리즘의 최소/최대 서비스 시간 오차 (WFQ, WF^{2}Q)

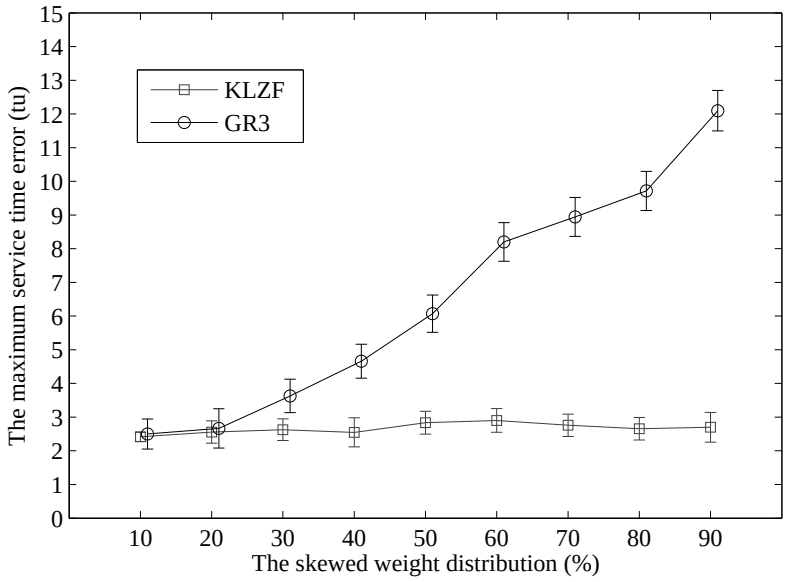


(a) 최대 서비스 시간 오차

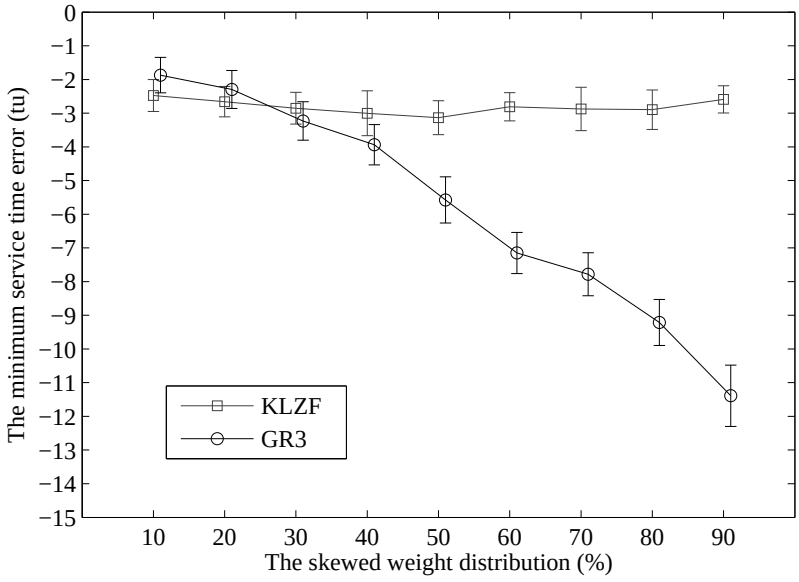


(b) 최소 서비스 시간 오차

그림 5.11: N=4,096 일 때 편향 지분 분포에서 라운드 로빈 기반 알고리즘의 최소/최대 서비스 시간 오차 (DRR, VTRR)



(a) 최대 서비스 시간 오차



(b) 최소 서비스 시간 오차

그림 5.12: N=4,096 일 때 편향 지분 분포에서 혼합 기반 알고리즘의 최소/최대 서비스 시간 오차 (GR^3, K-LZF)

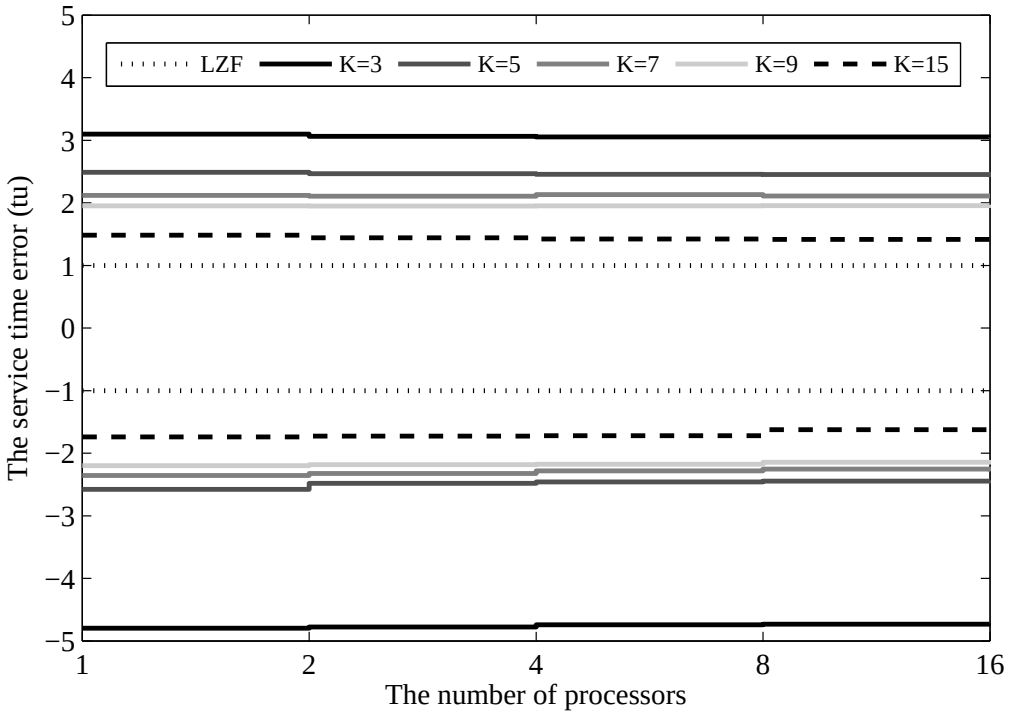


그림 5.13: 다중처리기에서 K 값 변화에 따른 최소 / 최대 서비스 시간 오차 비교

만 스케줄링 함수의 평균 실행 시간은 라운드 로빈 기반 공정 알고리즘의 성능에 근접하는 것을 기대할 수 있다.

K 값의 변화에 따라 K-LZF의 공정성 성능을 확인하기 위해 K=3, 5, 7, 9, 15에 대한 서비스 시간 오차를 평가하였다. 편향 지분 분포에서 η=0.1인 태스크 집합을 사용하였고, 태스크의 개수는 4,096이다. 특히 다중처리기 기반 K-LZF 알고리즘을 사용하여 처리기의 수(M=1, 2, 4, 8, 16)를 달리하여 시뮬레이션을 수행하였다. 공정성 비교를 위해 LZF 알고리즘의 서비스 시간 오차를 함께 나타내었다. 그림 5.13과 같이 K-LZF의 서비스 시간 오차는 K 값이 증가함에 따라 최소 서비스 시간 오차는 -1에 최대 서비스 시간 오차는 1에 근접함을 알 수 있다. M=1에서 K=3인

표 5.3: 다중처리기에서 K 값 변화에 따른 평균 서비스 시간 오차 비교

처리기 수	K=3	K=5	K=7	K=9	K=15
1	0.624373	0.468291	0.352073	0.329010	0.305846
2	0.439642	0.281567	0.250085	0.232690	0.216278
4	0.311910	0.199410	0.176446	0.164642	0.152902
8	0.221532	0.140770	0.124823	0.116661	0.107994
16	0.160921	0.099693	0.089071	0.082988	0.076308

경우 [-4.79, 3.09]이었고, K=15일 때 [-1.73, 1.48]까지 서비스 시간 오차가 감소하였다. M=16 에서 K 값의 변화에 따라, [-4.71, 3.04]에서 [-1.62, 1.41]까지 줄어들었다. 다중처리기에서 처리기의 수가 증가할수록 최소/최대 서비스 시간 오차 측면에서 처리기 수의 효과는 크지 않았다. 그러나 표 5.3에서 확인할 수 있듯이 평균적인 서비스 시간 오차 관점에서 다중처리기의 효과가 크다. 또한 처리기의 수가 증가할수록 전체적인 공정성이 향상될 뿐 아니라 K 값에 의한 성능도 개선됨을 확인할 수 있다.

5.3 AVOS 구현 실험 결과

본 절에서는 AVOS [1] 구현 실험을 통해 임베디드 서버 시스템에서 K-LZF 알고리즘의 효율성을 평가한다. 비교 분석을 위해 K-LZF 알고리즘 및 기존 공정 스케줄링 알고리즘인 WFQ, DRR, VTRR, GR^3 알고리즘을 AVOS 운영체제에 구현하였다. 실험을 위해 사용된 하드웨어는 ARM-CORTEX A8, 667Mhz 프로세서와

[1] AVOS는 오디오/비디오를 포함하는 동영상 응용 등과 같은 멀티미디어 서비스를 지원하기 위해 한국전자통신연구원(ETRI)에서 개발한 운영체제이다. 홈/카 서버 등의 저전력 서버에 적용하기 위해 멀티미디어 서비스를 위한 QoS 지원과 효율적인 자원 관리가 중요하다.

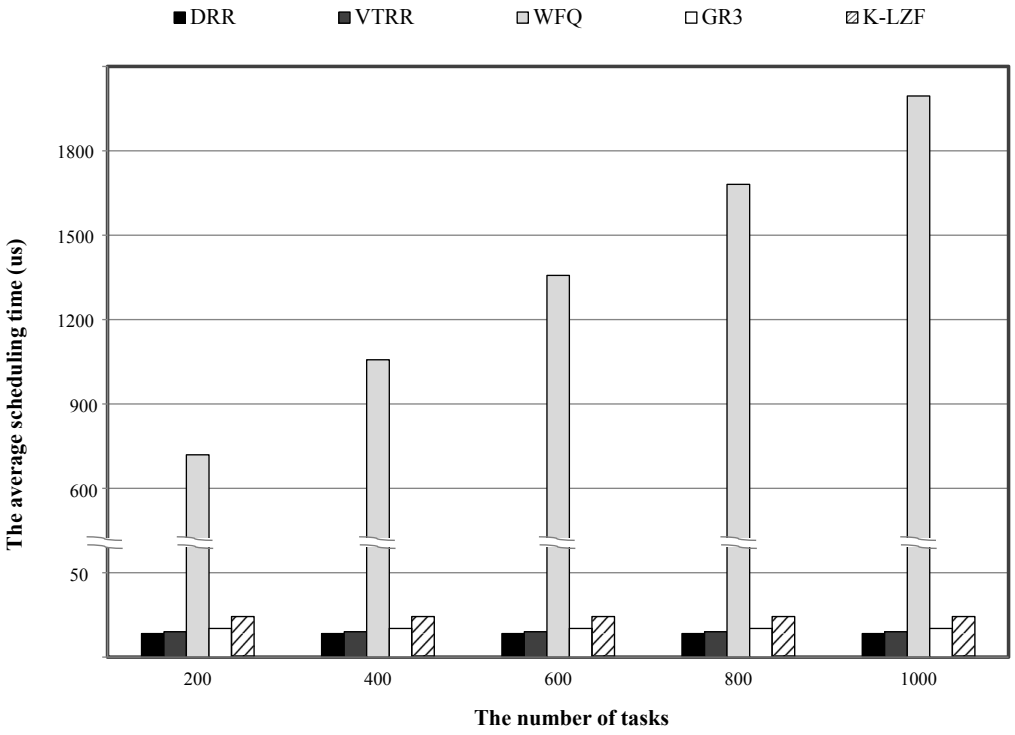


그림 5.14: AVOS 운영체제에서 스케줄링 함수들의 평균 실행 시간 비교

DDR2(128MByte)를 가지며, AVOS 운영체제가 이식된 하드웨어 플랫폼이다. 효율성을 평가하기 위해 태스크 수에 따른 스케줄링 함수의 평균 실행 시간을 측정한다. 실험을 위해 사용된 태스크들은 [1, 1024] 범위의 균일분포에서 생성된 난수 값으로 지분이 할당된다. AVOS에서 제공하는 $thread\_create$ 함수를 통해 태스크들을 생성하였고 200에서 1,000개까지 증가시키며 실험하였다. 평균 실행 시간을 계산하기 위해 스케줄링 함수의 실행 시간을 누적하여 측정하였으며 평균값으로 비교 평가되었다.

그림 5.14는 WFQ, DRR, VTRR, GR^3 및 제안 알고리즘인 K-LZF (K=5) 알고리즘들의 스케줄링 함수의 평균 실행 시간을 나타낸다. 상대적으로 실행 시간이 큰

표 5.4: AVOS 운영체제에서 K 값 변화에 따른 평균 실행 시간 비교

K 값	스케줄링 함수의 평균 실행 시간 (μs)
3	23
5	24
7	28
9	32
15	44

WFQ 알고리즘과 함께 비교하기 위해 y축은 로그 단위로 나타내었으며, 단위는 μs 이다. 태스크 수에 따라 스케줄링을 위한 평균 실행 시간은 큰 변화를 보였다. WFQ 알고리즘은 태스크 수에 비례해 스케줄링 시간이 길어진다는 문제점이 보였다. DRR, VTRR 알고리즘이 각각 평균 14 μs와 15 μs으로 빠른 스케줄링 시간을 보인다. 라운드 로빈 기반 알고리즘들의 시간 복잡도는 $O(1)$ 이므로, 태스크 수의 변화에 관계 없는 실행 시간을 보임을 알 수 있다. GR^{3}과 K-LZF 알고리즘은 평균 17μs와 24μs으로 라운드 로빈 기반 스케줄링 알고리즘들보다 약간 더 큰 스케줄링 시간을 가진다. 혼합 기반 알고리즘들은 그룹의 수 G 또는 서브리스트의 수 K에 비례하여 더 큰 상수를 갖기 때문이다. 하지만 GR^{3}와 K-LZF 알고리즘 역시 스케줄링 시간이 $O(G)$ 또는 $O(K)$ 이므로 태스크 수가 많은 상황에서도 스케줄링 시간에 영향을 주지 않는 다는 것을 확인 할 수 있다.

K-LZF 알고리즘에서 더 높은 수준의 공정성을 달성하기 위해 K 값을 크게 하는 경우 K-LZF의 성능에 영향을 준다. 표 5.4는 N=1,000일 때, K 값의 변화에 따라 스케줄링 시간이 어떻게 변화하는지 보여준다. K 값이 3∼15까지 증가할 때 스케줄링 시간은 약 [23, 44]까지 증가하였다. K 값이 스케줄링 대상을 선정하는 시간과

sift-down 연산 수행 시간에 영향을 주었기 때문이다. 이 사실은 K-LZF 알고리즘에서 K 값의 변화를 통해 서버 시스템의 용도에 맞는 공정성과 효율성의 성능을 조절할 수 있음을 의미한다.

5.4 LINUX 구현 실험 결과

본 절에서는 서버 시스템에서 많이 사용하는 범용 운영체제인 리눅스에서 K-LZF 알고리즘의 효율성을 평가한다. 제안된 알고리즘은 리눅스 스케줄링 알고리즘인 CFS 스케줄러와 비교되었다. 사용된 리눅스 커널 버전은 2.6.35이고 실험은 Intel 3.3 GHz 프로세서를 탑재한 PC에서 진행되었다. 태스크 수를 4096개까지 증가시켰으며 각 태스크의 지분은 랜덤 지분 분포를 따른다. 매 단위 시간에서 스케줄링 함수의 실행 시간을 누적값으로 측정하였으며 평균 실행 시간으로 평가하였다.

그림 5.15은 태스크 수가 증가함에 따라 K-LZF 알고리즘(K=5)와 CFS 알고리즘의 평균 실행 시간을 나타낸다. 태스크 수가 4096개까지 증가함에 따라 CFS 알고리즘의 평균 실행 시간은 0.088μs에서 0.369μs 까지 증가하였다. 이는 CFS 스케줄러는 최소 가상 시간을 가진 태스크들을 레드-블랙(red-black tree)를 사용하는데 스케줄링 시점마다 $O(logN)$의 복잡도를 가지기 때문이다. 반면, K-LZF 알고리즘은 태스크 수에 관계없이 최대 0.103μs의 평균 실행 시간을 보였다. N=4096 일 때, CFS 스케줄링 알고리즘의 약 27% 정도로 나타났다. 이는 K-LZF 스케줄링 알고리즘의 시간 복잡도가 $O(K)$이므로 K의 값에 의존하기 때문이다. 즉 K-LZF 알고리즘은 태스크가 많아지더라도 상수 시간의 스케줄링 시간을 보장하므로 서버 시스템에 적합함을 확인 할 수 있다.

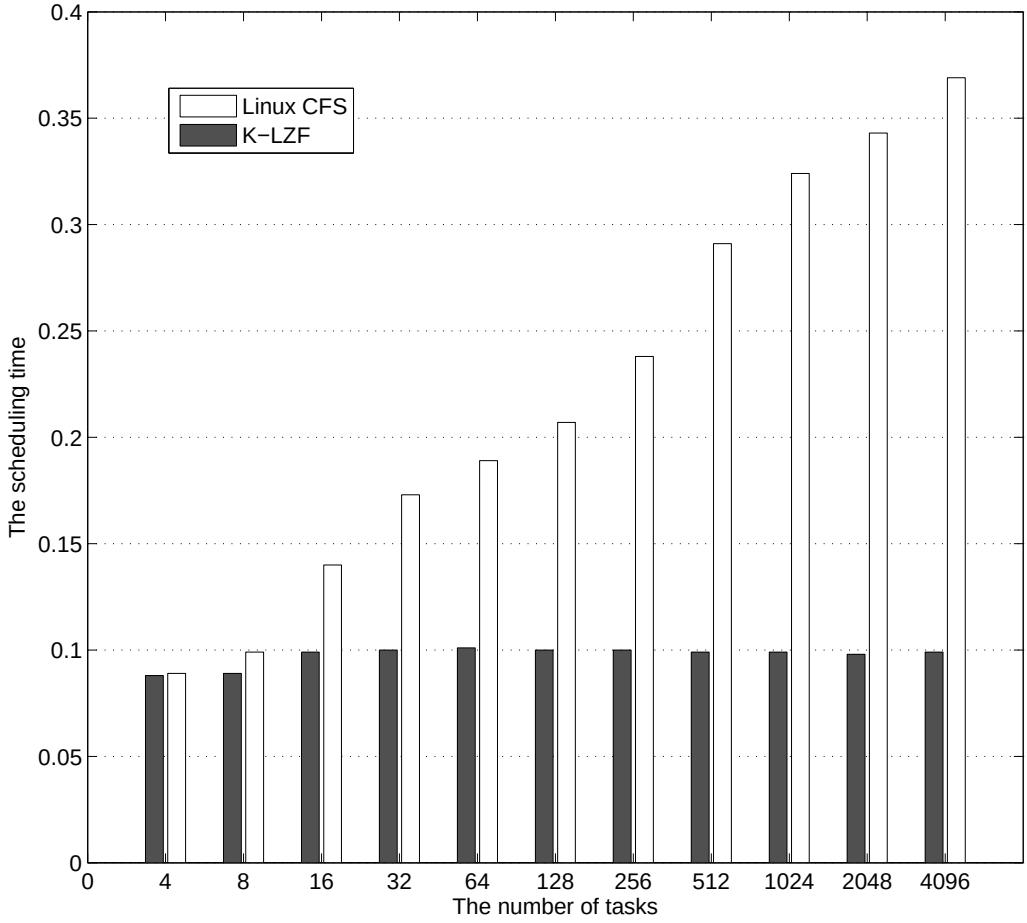


그림 5.15: LINUX 운영체제에서 CFS와 K-LZF 알고리즘의 평균 실행 시간 비교

제 6 장 결론

최근 서버 시스템들은 서비스 요구사항들이 다양화되고 복잡한 기능을 요구하면서 점차 더 높은 수준의 공정성과 효율성이 강조되고 있다. 서버 시스템은 일반적인 서비스뿐만 아니라 적절한 서비스 품질을 요구하는 멀티미디어 서비스, 적시에 자원을 요구하는 실시간성 서비스 등과 같이 다양한 특성을 가진 태스크들이 혼재하여 한정된 자원을 공정하게 할당하는 것이 더욱 중요해지고 있다. 동시에 서버 시스템은 많은 서비스 요청에도 안정적으로 지원할 수 있도록 높은 효율성과 확장성을 제공해야 한다. 공정하고 효율적인 자원 할당을 위해 많은 공정 스케줄링들이 연구되었다. WFQ나 WF^2Q와 같은 기존 가상 시간 기반 공정 스케줄링 알고리즘들은 높은 서비스 공정성을 보여주지만 스케줄링 함수의 시간 복잡도는 $O(logN)$으로 태스크 수가 증가할 경우 태스크 선택 시간이 증가한다. 반면 WRR, DRR 및 VTRR 라운드 로빈 기반 공정 스케줄링 알고리즘들은 $O(1)$의 높은 효율성을 보여주지만, 지분 범위가 커지거나 지분 분포가 편향될 경우 서비스 시간 오차가 증가하였다. 뿐만 아니라 태스크 수가 증가할 경우 그 오차는 더욱 커진다. GR^3와 같은 혼합 기반 공정 스케줄링 알고리즘들은 상수 시간의 시간 복잡도와 높은 공정성을 보여주었지만, 역시 편향 지분 분포에서 서로 다른 시간 요구사항을 가지는 다양한 태스크 집합에 대해 높은 서비스 공정성을 보여주지 못해 서버 시스템에 적합하지 않다.

본 논문은 지분 분포가 편향된 환경에서도 대규모 태스크들을 공정하게 서비스하면서 효율성을 향상시키기 위한 스케줄링 알고리즘인 K-LZF를 제안하였다. 서로

다른 시간 요구사항을 가지는 다양한 서비스들을 공정하게 처리하기 위해 태스크 만족 지수(z)를 사용하였다. 태스크 만족 지수는 단위 시간당 실행 오차의 누적값으로 기존 가상 시간과 다른 직관적인 스케줄링 평가 기준이다. 태스크 만족 지수를 기반으로 최소의 잠재 만족 지수를 가진 태스크를 선택하는 그리디 스케줄링 알고리즘인 LZF 알고리즘을 제시하고 분석하였다. 분석을 통해 LZF 스케줄링 알고리즘이 서비스 시간 오차의 상한을 보장함을 보였으며, 기존 스케줄링 알고리즘들에 비해 평균 서비스 시간 오차 측면에서 가장 높은 공정성을 보임을 확인하였다. K-LZF 스케줄링 알고리즘은 지분 분포가 편향된 환경에서도 높은 수준의 공정성을 보장하면서도 상수 시간 스케줄링 오버헤드를 갖도록 LZF 알고리즘을 근사화하였다. K-LZF 스케줄링 알고리즘은 만족 지수를 사용하는 LZF 정책을 사용하여 높은 공정성 보장의 기반을 마련하였다. K-LZF는 K 개의 리스트로 구성된 다중리스트 내에서 각 리스트의 첫 번째 태스크들만 탐색할 수 있도록 연산들을 수행시켜 효율성을 향상시켰다. 따라서 스케줄링 시점에 K-LZF 스케줄링 알고리즘의 시간 복잡도는 $O(K)$ 이다.

K-LZF 알고리즘에서 K 값의 변화를 통해 서버 시스템의 용도에 따라 공정성과 효율성 사이에서 성능을 조절할 수 있다. 시뮬레이션 분석 결과 K 값이 커질수록 LZF 알고리즘의 성능과 유사하고 K 값이 작을수록 라운드 로빈 기반 알고리즘의 성능에 근접함을 보였다. 또한 제안된 알고리즘은 지분 분포가 편향된 환경에서도 평균 서비스 시간 오차 측면에서 가장 좋은 공정성 성능을 보여주었다. AVOS 운영체제와 LINUX 운영체제에서 구현한 실험 결과에서 태스크 수에 상관없이 수행 시간은 $O(K)$의 상수 시간임을 확인하였다. 제안한 알고리즘은 지분 분포가 편향적인 환경에서 확장성 문제와 낮은 성능 문제를 동시에 해결하면서 서버 시스템에 적합한 공정 스케줄링 알고리즘이다.

본 논문에서 제안한 스케줄링 알고리즘은 공정하고 효율적인 자원 할당을 위하여 단일처리기 기반 임베디드 서버 시스템 또는 다중처리기 기반의 서버 시스템에서 모두 사용 가능하다. 뿐만 아니라 클라우드 서버 시스템에서 QoS보장을 위한 전역 스케줄링 알고리즘 또는 가상화 자원 스케줄링 알고리즘으로 활용하는 것도 향후 좋은 연구 주제이다.

참 고 문 헌

[AMB11] Abbas A Al-Muhsen and Radu F Babiceanu. Systems engineering approach to cpu scheduling for mobile multimedia systems. In *Proceedings of IEEE International Systems Conference (SysCon)*, pages 239–243. IEEE, 2011.

[AS] Syed Mutahar Aaqib and Lalitsen Sharma. Performance of web server architectures on single and multiple processor environments-a review.

[ASMH00] Shoukat Ali, Howard Jay Siegel, Muthucumaru Maheswaran, and Debra Hensgen. Task execution time modeling for heterogeneous computing systems. In *Proceedings of Heterogeneous Computing Workshop 2000 (HCW 2000)*, pages 185–199. IEEE, 2000.

[BZ96] Jon CR Bennett and Hui Zhang. Wf2q: worst-case fair weighted fair queueing. In *Proceedings of Fifteenth Annual Joint Conference of the IEEE Computer Societies(INFOCOM'96). Networking the Next Generation*, volume 1, pages 120–128. IEEE, 1996.

[CAGS00] Abhishek Chandra, Micah Adler, Pawan Goyal, and Prashant Shenoy. Surplus fair scheduling: A proportional-share cpu scheduling algorithm for symmetric multiprocessors. In *Proceedings of 4th Conference on Symposium on Operating System Design & Implementation*, Volume 4, pages 4–4. USENIX Association, 2000.

[CCN+05] Bogdan Caprita, Wong Chun Chan, Jason Nieh, Clifford Stein, and Haoqiang Zheng. Group ratio round-robin: O (1) proportional share scheduling for uniprocessor and multiprocessor systems. In *Proceedings of USENIX Annual Technical Conference*, pages 337–352, 2005.

[CNS06] Bogdan Caprita, Jason Nieh, and Clifford Stein. Grouped distributed queues: distributed queue, proportional share multiprocessor scheduling. In *Proceedings of Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 72–81. ACM, 2006.

[CRC13] Pramila Chawan, Vasundhara Rathod, and Monali Chim. Linux & windows operating systems. *Journal of Engineering Computers & Applied Sciences*, 2(6):15–20, 2013.

[DKS89] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review*, volume 19, pages 1–12. ACM, 1989.

[GVC96] Pawan Goyal, Harrick M Vin, and Haichen Chen. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *ACM SIGCOMM Computer Communication Review*, volume 26, pages 157–168. ACM, 1996.

[JJC+] Jinman Jung, Joonhyouk Jang, Yookun Cho, Jiman Hong, and Kuo. Tei-Wei. K-LZF: A fair and efficient scheduling algorithm for server systems. *Under preparation for submission.*

[JKC+12] Jinman Jung, Bongjae Kim, Yookun Cho, Wooseung Lee, Ahreum Kim, and Jiman Hong. A proportional share scheduling using task satisfaction index for quality of service guarantees. *Journal of the Korean Institute of Information Scientists and Engineers: Computing Practices and Letters*, 2012.

[KCK12] Sang Cheol Kim, Y Choi, and S Kim. Modeling and development of a large application on rtos. In *Proceedings of World Congress in Computer Science, Computer Engineering, and Applied Computing*, 2012.

[KLL97] Manhee Kim, Hyogun Lee, and Joonwon Lee. A proportional-share scheduler for multimedia applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, pages 484–491. IEEE, 1997.

[LBH09] Tong Li, Dan Baumberger, and Scott Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. *ACM Sigplan Notices*, 44(4):65, 2009.

[LPB13] Maxime Louvel, Alain Plantec, and Jean-Philippe Babau. Resource management for multimedia applications, distributed in open and heterogeneous home networks. *Journal of Systems Architecture*, 2013.

[Nag85] John Nagle. On packet switches with infinite storage. 1985.

[NAR+11] Klara Nahrstedt, Ahsan Arefin, Raoul Rivas, Pooja Agarwal, Zixia Huang, Wanmin Wu, and Zhenyu Yang. Qos and resource management in distributed interactive multimedia environments. *Multimedia Tools and Applications*, 51(1):99–132, 2011.

[NSM10] Mohammad R Nikseresht, Anil Somayaji, and Anil Maheshwari. Customer appeasement scheduling. *arXiv preprint arXiv:1012.3452*, 2010.

[NVZ01] Jason Nieh, Christopher Vaill, and Hua Zhong. Virtual-time round-robin: An o (1) proportional share scheduler. In *Proceedings of USENIX Annual Technical Conference*, pages 245–259, 2001.

[Pab09] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.

[RB07] Andreas Rosendahl and Goetz Botterweck. Mobile home automation-merging mobile value added services and home automation technologies. In *Proceedings of International Conference on the Management of Mobile Business (ICMB 2007)*, pages 31–31. IEEE, 2007.

[RJS00] John Regehr, Michael B Jones, and John A Stankovic. Operating system support for multimedia: The programming model matters. Technical report, Microsoft Research Technical Report MSR-TR-2000-89, 2000.

[RP03] Sriram Ramabhadran and Joseph Pasquale. Stratified round robin: A low complexity packet scheduler with bandwidth fairness and bounded

delay. In *Proceedings of Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 239–250. ACM, 2003.

[RSI09] Mark E Russinovich, David A Solomon, and Alex Ionescu. *Windows® Internals*. O'Reilly, 2009.

[SAWJ+96] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K Baruah, Johannes E Gehrke, and C Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 288–299. IEEE, 1996.

[SCK06] Arnab Sarkar, Partha P Chakrabarti, and Rajeev Kumar. Frame-based proportional round-robin. *IEEE Transactions on Computers*, 55(9):1121–1129, 2006.

[SV95] Madhavapeddi Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. *ACM SIGCOMM Computer Communication Review*, 25(4):231–242, 1995.

[SV98] Dimitrios Stiliadis and Anujan Varma. Efficient fair queueing algorithms for packet-switched networks. *IEEE/ACM Transactions on Networking*, 6(2):175–185, 1998.

[SWH+13] Chi-Sheng Shih, Jie-Wen Wei, Shih-Hao Hung, Joen Chen, and Norman Chang. Fairness scheduler for virtual machines on heterogonous multi-core platforms. *ACM SIGAPP Applied Computing Review*, 13(1):28–40, 2013.

[WW94] Carl A Waldspurger and William E Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of 1st USENIX conference on Operating Systems Design and Implementation*, page 1. USENIX Association, 1994.

[Zha91] Lixia Zhang. Virtualclock: a new traffic control algorithm for packet-switched networks. *ACM Transactions on Computer Systems (TOCS)*, 9(2):101–124, 1991.

Abstract

With the emergence of increasingly heterogeneous devices and networks, server systems are required to support a variety of services with different quality of service requirements. The degree of heterogeneity makes it more difficult to fairly and efficiently allocate resources based on the client's weight. Moreover, as server systems become larger, their performance can worsen significantly. However, most fair schedulers are not suitable for simultaneously providing scalability and robustness in server systems.

In this thesis, we present a fair and efficient scheduling algorithm for server systems. The proposed algorithm, called K-LZF, aims to achieve a high level of proportional fairness for the large number of heterogeneous tasks, with a constant scheduling overhead. The key idea behind K-LZF is to rearrange a set of tasks into a constant number (K) of lists and select only one candidate from the top of each list for comparison when the scheduler selects the next task. The parameter K can be used to adjust the balance between the proportional fairness and the scheduling overhead. The K-LZF algorithm was designed and implemented in the both AVOS and LINUX kernel. The results of the implementation and simulation study show that the K-LZF outperforms several existing scheduling algorithms with respect to scalability and robustness even when the degree of task heterogeneity becomes high.

Keywords: Server Systems, Resource Allocation, Task Scheduling, Fair Scheduling, Fairness, Scalability

Student Number: 2008-20975