[REPORT]

■과 목 명: 운영체제 (SWE3004-42)
■담 당 교 수: 신동군 교수님
■제 출 일: 2022년 3월 18일
■학 과: 수학과
■학 번: 2016314786
■성 명: 김호진

Multiprocessor Scheduling

수학과 김호진 (2016314786)

이번 단원에서는 멀티프로세서 스케줄링의 기본 사항에 대해 소개한다. 수년간 컴퓨팅 스펙트럼의 high-end에만 존재해왔던 멀티프로세서 시스템은 시간이 지남에 따라 일반화되었고 데스크탑, 노트북, 심지어 모바일 기기에까지 보급되고 있다. 이렇게 멀티프로세서 시스템이 확산될 수 있게 된 원인은 여러 개의 CPU 코어가 단일 칩에 압축되어 있는 멀티 코어 프로세서의 증가에 있다.

물론 CPU가 하나 이상 존재하면 많은 어려움이 발생한다. 주된 문제는 일반적인 어플리케이션이 하나의 CPU만을 사용한다는 점에 있다. 이를 해결하기 위해서는 스레드를 사용하여 병렬로 실행되도록 어플리케이션을 다시 작성해야 한다. 멀티 스레드 어플리케이션은 작업을 여러 CPU로 분산시킬 수 있기 때문에 CPU 리소스가 더 많이 제공될수록 더 빠르게 실행된다.

어플리케이션을 넘어 운영체제에서 새롭게 발생한 문제는 멀티프로세서 스케줄링 문제이다. 앞서 단일 프로세서 스케줄링에 대한 여러 원칙에 대해 설명했다. 그렇다면 여러 개의 CPU에서도 작동할 수 있도록 해당 아이디어를 확장하기 위해서는 어떻게 해야 하며, 어떤 문제를 극복해야 할까? 이 질문에 답을 하기 위해서는 다중 CPU에서 작업들을 schedule 하는 방법을 찾아야 한다.

10.1 Background: Multiprocessor Architecture

멀티프로세서 스케줄링과 관련된 새로운 문제를 이해하기 위해서는 단일 CPU 하드웨어와 다중 CPU 하드웨어의 근본적인 차이점부터 이해해야 한다. 둘의 차이는 하드웨어 캐시의 사용과 여러 프로세서에서 데이터가 공유되는 방식으로부터 발생한다.

일반적으로 단일 CPU를 사용하는 시스템에서는 프로세서가 프로그램을 더 빠르게 실행할 수 있도록 도와주는 하드웨어-캐시 계층을 가진다. 캐시는 시스템의 메인 메모리에서 발견되는 주요(popular) 데이터의 복사본을 보관하는 작고 빠른 메모리이다. 반면 메인 메모리는 모든(all) 데이터를 저장하지만, 이러한 큰 메모리에 접근하는 속도가 매우 느리다. 시스템은 자주 액세스하는 데이터를 캐시에 보관함으로써 크고 느린 메인 메모리를 빠르게 보이도록 할 수 있다.

예를 들어, 메모리에서 값을 가져오는 explicit load instruction을 실행하는 프로그램과 하나의 CPU를 가진 간단한 시스템에 대해 생각해보자. 프로그램이 최초로 load를 실행하면 데이터는 메인 메모리 상에 있기 때문에 이를 가져오는 데 수십 나노 초에서 수백 나노 초 정도가 소요된다. 프로세서는 가져온 데이터가 재사용될 것이라 예상하고 데이터의 복사본을 CPU 캐시에 넣는다. 이후 프로그램이 동일한 데이터를 가져오면 CPU는 우선 캐시에서 해당 항목을 확인한다. 만약 캐시에서 데이터를 찾는다면 훨씬 더 적은 시간에 데이터를 가져올 수 있으므로 빠르게 프로그램을 실행시킬 수 있다.

캐시는 temporal locality와 spatial locality 개념에 기초한다. Temporal locality는 데이터를 액세스할 때 가까운 미래에 다시 액세스할 가능성이 높다는 점에 기반하며, Spatial locality는 프로그램이 주소 x의 데이터 항목에 접근할 경우 x 근처의 데이터 항목에도 접근할 수 있다는 점에 개념을 둔다. 많은 프로그램이 이러한 유형의 locality를 가지고 있으므로 하드웨어 시스템은 캐시에 저장할 데이터를 잘 추측한다.

그런데 단일 공유 메인 메모리를 가지는 단일 시스템에 여러 개의 프로세서가 있을 경우, 까다로운 부분이

생기게 된다. 여러 CPU 상에서 caching은 훨씬 더 복잡하게 이루어진다. 예를 들어 CPU 1에서 실행 중인 프로그램이 주소 A에서 데이터 항목(D)을 읽는다고 가정하자. CPU 1의 캐시에 데이터가 존재하지 않기 때문에 시스템은 메인 메모리에서 D를 가져온다. 이후 프로그램은 주소 A의 값을 수정하고 캐시에 새로운 값 D'을 업데이트한다. 다만, 메인 메모리에 데이터를 쓰는 것은 느리기 때문에 시스템은 일반적으로 추후에 그 일을 수행한다. 그런 다음 OS가 프로그램 실행을 중지하고 CPU 2로 이동하면 프로그램은 주소 A의 값을 다시 읽게 된다. CPU 2의 캐시에는 해당 데이터가 없으므로 시스템이 메인 메모리에서 값을 가져오면서 올바른 값 D' 대신 이전 값인 D를 가져오게 된다.

이러한 문제를 캐시 일관성(cache coherence) 문제라고 하며, 이에 대한 기본적인 해결책은 하드웨어에서 제공한다. 하드웨어는 메모리 액세스를 모니터링함으로써 기본적으로 올바른 일이 발생하고 단일 공유 메모리가 보존되도록 할 수 있다. Bus-based 시스템에서 해당 작업을 수행하기 위해서는 'bus snooping'이라는 기술을 사용한다. 각 캐시는 메인 메모리에 연결된 bus를 관찰함으로써 메모리 업데이트에 주의를 기울인다. 이때 CPU가 캐시에 저장된 데이터 항목의 업데이트를 확인하면 변경 사항을 알아차리고 캐시에서 제거하여 복사본을 무효화하거나 캐시에 새 값을 넣어 업데이트한다.

10.2 Don't Forget Synchronization

캐시가 일관성을 제공하기 위한 모든 작업을 수행한다고 하더라도, 프로그램(또는 OS 자체) 역시 공유 데이터에 액세스할 때 주의해야할 점이 있다. 업데이트와 같이 CPU 전체에서 공유 데이터 항목 또는 구조에 접근할 때 정확성을 보장하기 위해서는 mutual exclusion primitives(예: locks)가 사용되어야 한다. 만약 여러 CPU에서 동시에 액세스되는 shared queue에 locks이 없다면 일관성 프로토콜이 있더라도 queue에서 요소를 동시에 추가하거나 제거할 수 없다.

이를 좀 더 구체화하기 위해서 shared linked list에서 element를 제거하기 위해 사용되는 코드를 생각해보자. 두 CPU의 스레드가 동시에 이 루틴에 들어간다고 가정하고 스레드 1이 첫 번째 줄을 실행하게 되면, 이것은 tmp 변수에 저장된 헤드의 현재 값을 가지게 된다. 만약 스레드 2 역시 첫 번째 줄을 실행하면, 이 또한 private tmp 변수에 저장된 헤드의 동일한 값을 가진다. 그렇게 되면 각각의 스레드는 리스트의 헤드에서 요소를 제거하는 대신 동일한 헤드 요소를 제거하려고 시도하면서 모든 종류의 문제를 일으키게 된다.

해결책은 locking을 통해 이러한 루틴을 올바르게 만드는 것이다. 이 경우 simple mutex(예: pthread_mutex_t m;)를 할당한 다음 루틴의 시작 부분에 lock(&m)을 추가하고 마지막에 unlock(&m)을 추가하면 문제가 해결되며 코드가 원하는 대로 실행되도록 할 수 있다. 하지만 해당 방법의 경우 CPU 수가 증가하면 동기화된 공유 데이터 구조에 대한 접근이 매우 느려진다는 단점이 있다.

10.3 One Final Issue: Cache Affinity

마지막 문제는 캐시 선호도(cache affinity)로, 멀티프로세서 캐시 스케줄러 구현 시 발생한다. 이 개념은 간단하다. 프로세스가 특정 CPU에서 실행되면 CPU의 캐시(및 TLB)에 상당한 양의 state를 구축하게 된다. 이러한 상황에서 다음 번 프로세스를 실행했을 때 해당 CPU의 캐시에 state의 일부가 이미 존재한다면 더 빠르게 실행할 수 있으므로 같은 CPU 상에서 실행하는 것이 유리하다. 만약, 매번 다른 CPU에서 프로세스를 실행하게 되면 실행할 때마다 state를 reload해야 하므로 프로세스의 성능이 저하된다. 그러므로 멀티프로세서 스케줄러는 스케줄링 결정을 내릴 때 cache affinity를 고려해야 하며, 가능한 동일한 CPU에서 프로세스를 유지하고자 한다.

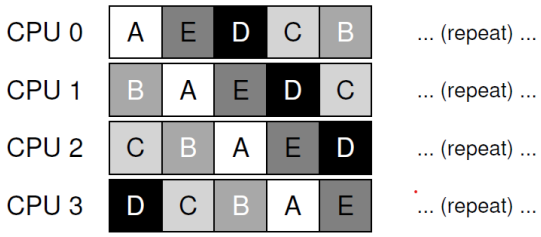
10.4 Single-Queue Scheduling

멀티프로세서 시스템을 위한 스케줄러 구현에서 가장 기본적인 접근 방식은 스케줄링해야 하는 모든 작업을 single queue에 넣어 단일 프로세서 스케줄링의 기본 프레임워크를 재사용하는 single queue multiprocessor scheduling(SQMS)이다. 해당 방식은 다음으로 실행하기에 가장 적합한 작업을 선택한다는 점에서 기존 정책을 유지하기 때문에 둘 이상의 CPU에서 작동하도록 조정하는데 많은 작업을 필요로 하지 않는다.

하지만 SQMS에는 분명한 단점도 존재한다. 첫 번째 문제는 확장성(scalability)의 부족이다. 스케줄러가 여러 CPU에서 올바르게 동작하도록 하기 위해 개발자들은 코드에 locking을 삽입한다. Locks은 SQMS 코드가 단일 큐에 액세스할 때 적절한 결과가 나타나도록 만든다. 그런데 locks의 경우, 시스템의 CPU 수가 증가하게 되면 성능을 크게 저하시킨다. single lock에 대한 경합(contention)이 증가하게 되면 시스템은 lock overhead에 더 많은 시간을 투자하게 되고, 이로 인해 시스템이 수행해야 할 작업에 투자하는 시간이 적어지게 된다.

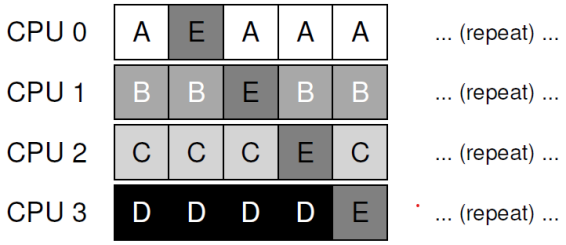
SQMS가 가지는 두 번째 주요 문제는 캐시 선호도(Cache affinity)다. 예를 들어, 실행할 작업 다섯 개(A, B, C, D, E)와 프로세서 네 개가 있다고 가정했을 때 스케줄링 queue는 다음과 같다.

시간이 지남에 따라 각 작업이 time slice에서 실행된 이후 다른 작업이 선택되었다면 CPU 간에 가능한 작업 스케줄은 다음과 같다.



각 CPU가 globally shared queue에서 실행할 다음 작업을 선택하면, 각 작업들은 CPU에서 다른 CPU로 이동하게 된다. 그로 인해 cache affinity 관점에서 말하는 것과는 정반대의 일이 수행된다.

이러한 문제를 해결하기 위해 대부분의 SQMS 스케줄러에는 프로세스가 동일한 CPU에서 실행될 가능성을 높이기 위한 일종의 affinity 메커니즘이 포함되어 있다. 해당 메커니즘은 작업에 affinity를 제공할 뿐만 아니라, balance load를 위해 몇몇 작업들을 이동시키기도 한다. 예를 들어, 다음과 같은 5개의 작업이 스케줄 되었다고 가정해 보자.

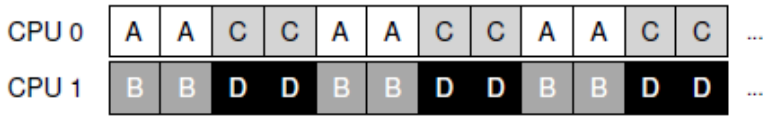


이 배열에서 작업 A부터 D는 프로세서 간 이동을 하지 않고 작업 E만 CPU에서 CPU로 migrating되므로 대부분의 affinity가 유지된다. 이 상황에서 다음번에 다른 작업을 migrate하기로 결정한다면 일종의 affinity fairness까지 달성할 수 있다. 그러나 이러한 계획을 실행하는 것은 복잡하다. 이처럼 SQMS 접근법에는 장단점이 존재한다. SQMS는 single queue를 가지는 기존의 단일 CPU 스케줄러를 이용하여 쉽게 구현할 수 있지만, 동기화 오버헤드로 인해 확장이 어려우며 cache affinity를 쉽게 유지할 수 없다.

10.5 Multi-Queue Scheduling

단일 queue 스케줄러에서 발생하는 문제로 인해 일부 시스템은 여러 개의 queue를 사용한다. 이러한 접근법을 multi-queue multiprocessor scheduling(MQMS)이라고 부르며, MQMS에서 기본 스케줄링 프레임워크는 다수의 스케줄링 queue로 구성된다. 각 queue는 라운드 로빈과 같은 특정 스케줄링 규칙을 따르지만, 다른 어떠한 알고리즘도 사용될 수 있다. 시스템에 작업이 들어가면 heuristic한 성질에 의해서 정확히 하나의 스케줄링 queue에 배치된다. 그 뒤 작업은 독립적으로 스케줄 되므로 단일 queue 접근법에서 발견되는 정보 공유 및 동기화 문제를 방지할 수 있다. 예를 들어 CPU가 2개(CPU 0 및 CPU 1)뿐이고 몇몇의 작업((A, B, C 및 D)이 시스템에 들어간다고 가정하자. 각 CPU에는 스케줄링 queue가 있기 때문에 OS는 각 작업을 배치할 queue를 결정해야 한다.

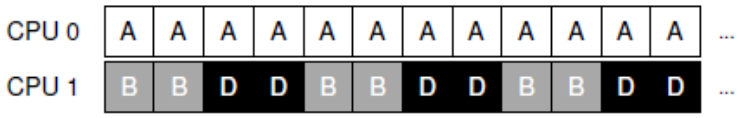
Queue 스케줄링 정책에 따라 실행할 작업을 결정할 때, 각 CPU에는 선택할 수 있는 작업이 두 개씩 존재한다. 예를 들어 라운드 로빈을 사용할 경우 시스템은 다음과 같이 스케줄을 생성할 수 있다.



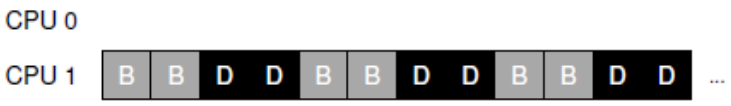
MQMS는 본질적으로 확장성을 가진다는 점에서 SQMS에 비해 뚜렷한 이점을 가진다. CPU의 수가 증가함에 따라 queue의 수도 증가하기 때문에 lock과 cache contention이 문제가 되지 않는다. 또한 MQMS는 cache affinity를 제공하므로 작업이 동일한 CPU에 유지되고 cached contents를 재사용할 수 있다.

하지만 깊게 들여다보면 multi-queue based approach의 근본적인 문제인 load imbalance가 발생한다는 사실을 알 수 있다. 위와 동일한 설정(4개의 작업, 2개의 CPU)에서 작업 중 하나(C)가 완료된다고 가정해 보자. 그러면 스케줄링 queue는 다음과 같다.

시스템의 각 대기열에서 라운드 로빈 정책을 실행하면 resulting schedule은 다음과 같이 표시된다.



A는 B와 D보다 2배 많은 CPU를 얻는데, 이는 원하는 결과가 아니다. 설상가상으로 A와 C가 모두 마무리되면서 시스템에 B와 D만 남는다고 가정했을 때 두 개의 스케줄링 queue와 resulting timeline은 다음과 같다.

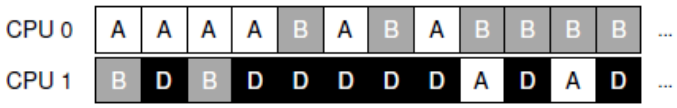


CPU 0은 idle이며, CPU 사용 타임라인은 매우 비효율적이게 된다. 그렇다면 이러한 다중 queue 멀티프로세서 스케줄러에서는 어떻게 load imbalance 문제를 극복할 수 있을까? 이 질문에 대한 분명한 답은 작업을 이동시키는 것으로, 이 기술을 migration이라고 한다. OS는 CPU 간의 작업을 migration함으로써 load balance를 달성할 수 있다. 명확한 설명을 위해서 몇 가지 예를 살펴보겠다. 앞서 보았던 것처럼 하나의 CPU는 idle이고 다른 CPU에는 일부 작업이 남아있는 상태라고 가정하자.

이 경우 필요한 migration은 쉽게 파악된다. OS는 B 또는 D 중 하나를 CPU 0으로 이동시키면 된다. 그리고 이러한 single job migration의 결과는 load balance를 이룬다.

A가 CPU 0에 홀로 존재하고 B와 D가 CPU 1에 번갈아 있는 더 까다로운 경우도 존재한다.

이 경우 single migration으로는 문제가 해결되지 않는다. 이를 해결하기 위해서는 하나 이상의 작업을 지속적으로 migration해야 하므로 작업이 계속해서 전환되어야 한다. 처음 그림에서 A는 CPU 0에 혼자 있고, B와 D는 CPU 1에 번갈아 배치되어 있다. 몇 개의 time slices 이후, B는 A와 경쟁하기 위해 CPU 0으로 이동하게 되고 D는 CPU 1에 홀로 남아 몇 개의 time slices를 즐기면서 load balance가 이루어진다.



물론, 이 밖에도 가능한 migration 패턴들이 다수 존재한다. 그러나 시스템이 migration을 제정하는 방법에 따라 까다로운 부분이 있을 수 있다. 기본적인 접근법 중 하나는 work stealing이라고 알려진 기술을 사용하는 것이다. Work stealing 접근 방식에서는 작업량이 적은 (source) queue가 다른 (target) queue를 지켜보면서 queue가 얼마나 가득 찼는지를 확인한다. Target queue가 source queue보다 더 꽉 차게 되면, source queue는 target queue에서 하나 이상의 작업을 "steal"하여 load balancing을 돕는다. 물론 이러한 접근법에도 문제가 있다. 다른 queue를 너무 자주 보면 높은 오버헤드가 발생하고 다중 큐 스케줄링 구현의 전체 목적이었던 scaling에 문제가 생기게 된다. 반면 다른 queue를 자주 보지 않으면 load imbalance가 심해지게 된다. 이처럼 시스템 정책 설계 과정에서 올바른 기준점을 찾는 것은 여전히 black art로 남아 있다.

10.6 Linux Multiprocessor Schedulers

흥미롭게도 리눅스 커뮤니티는 멀티프로세서 스케줄러를 구현하는 일반적인 솔루션이 존재하지 않는다. 다만, 시간이 지나면서 O(1) 스케줄러, Completely Fair Scheduler(CFS), BF 스케줄러(BFS)라는 세 가지 스케줄러가 생겨났다. O(1)과 CFS는 여러 개의 queue를 사용하는 반면, BFS는 단일 queue를 사용하여 두 접근 방식 모두 성공할 수 있음을 보였다. O(1) 스케줄러는 우선순위 기반 스케줄러로, 시간이 지남에 따라 프로세스의 우선순위를 변경하고 다양한 스케줄링 목표를 충족시키기 위해서 가장 높은 우선순위부터 스케줄링 한다. 반대로 CFS는 결정론적 비례적 점유율(deterministic proportional-share) 접근법을 가진다. BFS 역시 비례적 공유(proportional-share) 방식이기는 하지만, EEVDF(Earliest Eligible Virtual Deadline First)로 알려진 더 복잡한 체계를 기반으로 한다.

10.7 Summary

지금까지 멀티프로세서 스케줄링에 대한 다양한 접근 방식을 알아보았다. SQMS은 비교적 구현이 쉽고 load balancing을 잘 수행하지만, 많은 프로세서로의 확장이나 cache affinity 측면에서 본질적인 어려움을 가진다. MQMS는 확장성이 뛰어나고 cache affinity를 잘 처리하지만 load imbalance로 인해 문제가 발생하고 더욱 복잡하다. 어떤 접근 방식을 취하든 간단한 답은 없다. 또한, 작은 코드 변경으로 인해 큰 차이가 발생할 수 있기 때문에 범용 스케줄러를 구축하는 것은 여전히 어려운 작업으로 남아있다.

My Opinion

이번 과제를 통해 멀티프로세서 스케줄링의 기본 사항에 대해 알아보았다. 최근 나오는 CPU들을 보면 대부분 멀티 코어, 쿼드 코어, 헥사 코어 등으로 멀티 프로세서 시스템이 보편화되었음을 알 수 있다. 이렇게 멀티프로세서 시스템이 우리 주변에 자리 잡으면서 여러 개의 CPU 사이에서 어떻게 작업을 schedule 할 지 정하는 멀티프로세서 스케줄링이 중요한 문제가 되었다.

본문에서 언급되었듯이 멀티프로세서 시스템을 위한 스케줄러 구현은 single queue multiprocessor scheduling(SQMS)과 multi-queue multiprocessor scheduling(MQMS)을 통해 이루어진다. SQMS의 경우, 스케줄링해야 하는 모든 작업을 single queue에 넣어 단일 프로세서 스케줄링의 기본 프레임워크를 재사용하고 다음으로 실행하기에 가장 적합한 작업을 선택하는 기존의 정책을 유지하기 때문에 비교적 구현이 쉽고 load balancing을 잘 수행한다는 장점을 가진다. 하지만 SQMS는 다중 프로세서로의 확장이나 cache affinity 측면에서 본질적인 어려움을 가진다. 반면, MQMS는 확장성이 뛰어나고 cache affinity를 잘 처리하지만 구현이 복잡하고 load imbalance가 발생할 위험이 있다. 이를 통해서 알 수 있듯이 SQMS의 문제점을 보완하기 위해 MQMS가 등장하기는 했지만, 모든 경우에서 MQMS가 답이 되는 것은 아니다. 그러므로 상황에 맞는 적절한 방법의 스케줄 정책을 선택하는 것이 중요하다.

과제를 하는 도중 Migration의 기본적인 접근 방법으로 언급된 'work stealing'에 대한 궁금증이 생겼고 이에 대한 내용을 추가적으로 알아보았다. work stealing은 병렬 컴퓨팅(parallel computing) 환경에서 멀티 스레드를 활용한 스케줄링 전략으로 해당 알고리즘은 다음의 과정들을 통해 진행된다. 컴퓨터 시스템의 프로세서들은 각자의 work item queue를 가지고 작업을 수행하며, 각 work item은 순차적으로 수행되어야 하는 명령어들의 집합으로 구성된다. work stealing 알고리즘은 스레드를 일정한 수로 유지하면서 스레드마다 독립적인 작업 queue를 관리한다. 이 상황에서 하나의 스레드 큐가 비어지면 다른 스레드의 작업을 steal할 수 있도록 만들어 효율적인 작동을 유도한다.

이러한 work stealing을 사용하면 Thread Spawning 문제와 Race Condition 문제를 해결할 수 있다. Thread Spawning은 병렬 처리를 위한 스레드를 만들고 제거하는 작업으로 해당 작업은 멀티 스레드 컴퓨팅에 큰 과부하를 가져온다. Work-stealing 알고리즘은 미리 만들어 놓은 Thread Pool를 이용해 과부하 문제를 해결한다. Race Condition은 하나의 작업 queue에서 일정 수의 스레드가 작업들을 병렬 처리하는 경우에 발생하는 문제로 이로 인해 프로그램이 정상적으로 동작하지 않거나 성능 저하가 발생한다. 하지만 work stealing 알고리즘은 스레드마다 고유의 작업 queue를 가지고 있기 때문에 해당 문제가 발생하지 않는다.

본 과제를 수행하면서 단일 프로세서에서 멀티프로세서로 스케줄링 아이디어를 확장하는 과정에서 발생하는 문제들과 이를 해결하기 위한 다양한 접근법에 대해 알아볼 수 있었다. 시스템프로그램이나 컴퓨터구조개론 강의를 수강하면서 멀티프로세서를 이용하는 경우 캐시 일관성(cache coherence), 동기화(Synchronization), 캐시 선호도(cache affinity) 등의 문제가 발생하고 이를 고려해야 한다는 것을 알았지만 어떤 방식으로 해결해야하는 지에 대해서는 자세하게 배우지 못했다. 본문의 내용을 통해 이러한 궁금증을 해결할 수 있었고 다양한 예시들을 통해 SQMS와 MQMS의 동작을 알아보면서 각 정책이 가지는 장단점을 명확하게 파악할 수 있었다. 사실 이전까지는 MQMS가 SQMS에 비해 많은 queue를 이용하기 때문에 모든 면에서 좋을 것이라고 생각했는데 과제를 하면서 이러한 생각이 잘못되었음을 알게 되었다. 멀티프로세서 시스템이 주는 장점들은 명확하기 때문에 앞으로도 멀티프로세서 스케줄링에 관한 연구는 계속될 것이다. 과연 미래에는 SQMS와 MQMS의 장점을 모두 갖는 범용 스케줄러가 등장할 지 기대가 된다.