13

멀티프로세서 스케줄링 (고급)

이 장에서는 **멀티프로세서 스케줄링**(**multiprocessor scheduling**)의 기본을 소개한다. 이 주제는 다소 고급에 속하기 때문에 병행성(concurrency) 주제를 어느 정도 깊게 공부한 이후에 다루는 것이 가장 좋다. 병행성은 이 책의 두 번째 주요 내용이다.

고사양 컴퓨터에만 존재했던 **멀티프로세서**(**multiprocessor**) 시스템은 일반적이 되었으며, 데스크톱 컴퓨터, 노트북, 심지어 모바일 장치에도 사용되고 있다. 여러 개의 CPU 코어가 하나의 칩에 내장된 **멀티코어**(**multicore**) 프로세서가 대중화의 근본 원인이다. 싱글코어 CPU의 성능 개선이 한계에 봉착하면서 멀티코어 기술이 각광을 받게 되었다. 우리는 다수의 CPU를 사용할 수 있게 되었다. 좋은 소식이지 않은가?

다중 CPU 시대가 오면서 많은 문제가 발생하였다. 가장 중요한 것은 전통적 응용 프로그램은 (예, 당신이 작성한 C 프로그램) 오직 하나의 CPU만 사용한다는 것이다. 더 많은 CPU를 추가해도 더 빨리 실행되지 않는다. 이 문제를 해결하려면 응용 프로그램을 **병렬**(**parallel**)로 실행되도록 다시 작성해야 한다. 보통 **쓰레드**를 이용한다. 쓰레드는 이 책의 두 번째 부분에서 매우 상세하게 설명된다. 멀티 쓰레드 응용 프로그램은 작업을 여러 CPU에 할당하며, 따라서 더 많은 수의 CPU가 주어지면 더 빠르게 실행된다.

응용 프로그램뿐 아니라 운영체제가 새로이 직면한 문제는 (놀랍지도 않게!) 멀티프로세서 스케줄링이다. 지금까지 우리는 단일프로세서 스케줄링의 많은 원칙들에 대해 논의하였다. 이 아이디어를 여러 CPU에서 동작하도록 어떻게 확장할 수 있을까? 우리가 해결해야 하는 새로운 문제는 무엇인가? 문제는 다음과 같다.

핵심 질문: 여러 CPU에 작업을 어떻게 스케줄 해야 하는가
운영체제는 어떻게 작업을 여러 CPU에 스케줄 해야 하는가? 어떤 새로운 문제가 등장하는가? 예전 기술을 적용할 것인가 아니면 새로운 아이디어가 필요한가?

13.1 배경: 멀티프로세서 구조

멀티프로세서 스케줄링에 대한 새로운 문제점을 이해하기 위해서는 단일 CPU 하드웨어와 멀티 CPU 하드웨어의 근본적인 차이에 대한 이해가 필요하다. 다수의 프로세서

여담: 고급 주제의 장들

고급 주제의 장을 제대로 이해하기 위해서는 책의 전반적인 내용에 대한 이해를 필요로 한다. 미리 알아야 할 선행 내용보다 먼저 나오는 것이 책의 구성상 더 적절한 경우도 많다. 예를 들어, **멀티프로세서 스케줄링**에 관한 이 장은 먼저 병행성에 관한 중간 부분을 읽은 후에 더 잘 이해가 될 것이다. 그러나 논리적으로는 가상화(일반적으로)와 CPU 스케줄링(구체적으로)에 관한 부분에 포함되는 것이 더 적합하다. 그러한 장들은 순서와 상관없이 먼저 읽을 것을 권한다. 이 경우에는 책의 두 번째 부분에 해당한다.

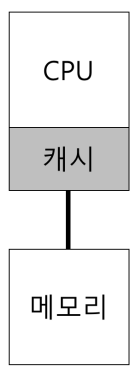
간에 데이터의 공유, 그리고 하드웨어 **캐시**의 사용 방식(예, 그림 13.1)에서 근본적인 차이가 발생한다. 우리는 이 문제를 개념적으로만 다루도록 하겠다. 구체적인 내용은 다른 과목 [CSG99], 특히 3, 4학년 혹은 대학원 수준의 컴퓨터 구조 수업에서 다룬다.

단일 CPU 시스템에는 **하드웨어 캐시** 계층이 존재한다. 이 캐시는 프로그램을 빠르게 실행하기 위해 존재한다. 캐시는 메인 메모리에서 *자주* 사용되는 데이터의 복사본을 저장하는 작고 빠른 메모리이다. 메인 메모리는 모든 데이터를 저장하지만 느리다. 자주 접근되는 데이터를 캐시에 임시로 가져다 둠으로써 시스템은 크고 느린 메모리를 빠른 메모리처럼 보이게 한다.

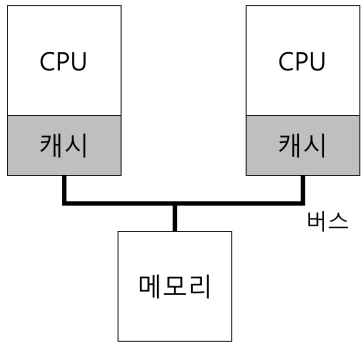
예를 들어, `load` 명령어를 수행하는 프로그램과 하나의 CPU만 있는 간단한 시스템을 생각해 보자. CPU는 작은 크기의 캐시(64 KB)와 큰 메인 메모리를 갖고 있다.

프로그램이 처음 이 `load` 명령어를 실행할 때, 데이터가 메인 메모리에 존재하므로 데이터를 가져오는 데 긴 시간이 소모된다(아마 수십 또는 수백 nano second). 데이터가 다시 사용될 것으로 예상한 프로세서는 읽은 데이터의 복사본을 CPU 캐시에 저장한다. 프로그램이 나중에 다시 같은 데이터를 가져오려고 하면, CPU는 우선 해당 데이터가 캐시에 존재하는지 검사한다. 캐시에 존재하기 때문에 데이터는 훨씬 빨리 접근되고(수 nano second) 프로그램은 빨리 실행한다.

캐시는 **지역성**(**locality**)에 기반한다. 지역성에는 **시간 지역성**(**temporal locality**)과 **공간 지역성**(**spatial locality**)의 두 종류가 있다. 시간적 지역성의 기본 아이디어는



〈그림 13.1〉 **캐시가 장착된 하나의 CPU**



〈그림 13.2〉 **메모리를 공유하는 캐시가 장착된 두 개의 CPU**

데이터가 한 번 접근되면 가까운 미래에 다시 접근되기 쉽다는 것이다. 루프에서 여러 번 반복해서 접근되는 변수 또는 명령어 자체를 상상해 보라. 공간적 지역성의 기본 아이디어는 프로그램이 주소 x의 데이터를 접근하면 x 주변의 데이터가 접근되기 쉽다는 것이다. 전체 배열을 차례대로 접근하는 프로그램 또는 순차적으로 실행되는 명령어들을 생각해 보자. 이런 유형의 지역성은 많은 프로그램에서 관측된다. 하드웨어 시스템은 캐시에 어떤 데이터를 저장할지 비교적 정확하게 추측을 할 수 있고, 캐시는 잘 작동한다.

자 이제부터 까다로운 부분: 그림 13.2에서 보는 바와 같이 하나의 시스템에 여러 프로세서가 존재하고 하나의 공유 메인 메모리가 있을 때 어떤 일이 일어나는가?

알려진 것처럼 멀티프로세서 시스템에서 캐시를 사용하는 것은 훨씬 복잡하다. 예를 들어, CPU 1에서 실행 중인 프로그램이 주소 A를(D 값을) 읽는다고 가정하자. 데이터가 CPU 1 캐시에 존재하지 않기 때문에 시스템은 메인 메모리로부터 데이터를 가져오고 값 D를 얻는다. 그런 후 프로그램은 주소 A의 값을 변경한다. 변경은 캐시에 존재하는 값만 D'으로 갱신한다. 메모리에 데이터를 쓰는 것은 시간이 오래 걸리므로 메인 메모리에 기록하는 것은 보통 나중에 한다. 운영체제가 프로그램의 실행을 중단하고 CPU 2로 이동하기로 결정한다고 가정하자. 프로그램은 주소 A의 값을 다시 읽는다. CPU 2의 캐시에는 그러한 데이터가 존재하지 않고 따라서 시스템은 메인 메모리에서 데이터를 가져온다. 이때 D'이 아니라 옛날 값인 D를 가져온다. 이런!

이 문제를 **캐시 일관성 문제**(**cache coherence**)라고 부른다. 이 문제의 해결에 관련된 방대한 내용을 잘 정리해 놓은 문헌이 있다 [SHW11]. 여기서는 모든 차이점을 생략하고 주요 쟁점에 대해서만 논의할 것이다. 더 많은 것을 알고 싶다면 컴퓨터 구조 수업을 수강하기 바란다.

기본적인 해결책은 하드웨어에 의해 제공된다. 하드웨어는 메모리 주소를 계속 감시하고, 항상 "제대로" 된 상황만 발생토록 시스템을 관리한다. 특히, 여러 개의 프로세스들이 하나의 메모리에 갱신할 때에는 항상 공유되도록 한다. 버스 기반 시스템에서는 **버스 스누핑**(**bus snooping**)이라는 오래된 기법을 사용한다 [Goo83]. 캐시는 자신과 메모리를 연결하는 버스의 통신 상황을 계속 모니터링한다. 캐시 데이터에 대한 변경이 발생하면, 자신의 복사본을 **무효화**(**invalidate**) 시키거나(즉, 자신의 캐시에서 삭제) **갱신**(새로운 값을 캐시에 기록) 한다. 나중 쓰기(write-back) 캐시는 메인 메모리에 쓰기

연산이 지연되기 때문에 캐시 일관성 유지 문제를 훨씬 복잡하게 만든다. 그러나 기본 기법이 어떻게 동작하는지는 대략 상상할 수 있을 것이다.

13.2 동기화를 잊지 마시오

일관성 유지에 대한 모든 일을 캐시가 담당한다면, 프로그램 또는 운영체제 자신은 공유 데이터를 접근할 때 걱정할 필요가 있을까? 불행하게도 대답은 예이고 이 책의 "병행성"에 관한 부분에서 자세하게 설명된다. 여기서 깊게 들어가지는 않지만 몇몇 기본 아이디어에 대해서는 간단하게 설명하겠다. 병행성에 관해 잘 알고 있다고 가정하고 진행할 것이다.

CPU들이 동일한 데이터 또는 구조체에 접근할 때(특히, 갱신), 올바른 연산 결과를 보장하기 위해 락과 같은 상호 배제를 보장하는 동기화 기법이 많이 사용된다. **락-프리(lock-free)** 데이터 구조 등의 다른 방식은 복잡할 뿐 아니라 특별한 경우에만 사용된다. 자세한 사항은 병행성에 관한 부분 중 교착상태 장을 참조하기 바란다. 예를 들어, 여러 CPU가 동시에 사용하는 공유 큐가 있다고 가정하자. 캐시의 일관성을 보장하는 프로토콜이 존재한다 하더라도 락이 없이는 항목의 추가나 삭제가 제대로 동작하지 않을 것이다. 구조체를 원자적으로 갱신하기 위해서는 락이 필요하다.

```
1  typedef struct __Node_t {
2      int value;
3      struct __Node_t *next;
4  } Node_t;
5
6  int List_Pop() {
7      Node_t *tmp = head;       // 이전 head를 기억...
8      int value = head->value; // ... 값도 기억
9      head = head->next;        // head 를 다음 포인터로 이동
10     free(tmp);                // 이전 head 해제
11     return value;             // head의 값을 반환
12 }
```

〈그림 13.3〉 **간단한 리스트의 삭제 코드**

연결 리스트에서 원소 하나를 삭제하는 코드를 생각해 보자. 코드가 그림 13.3에 나와 있다. 두 CPU의 쓰레드가 동시에 이 루틴으로 진입한다고 가정하자. 쓰레드 1이 첫 번째 행을 실행하면 `head`의 현재 값을 `tmp`에 저장한다. 그런 후 쓰레드 2가 첫 번째 행을 실행하면 역시 `head`의 같은 값을 `tmp`에 저장한다. `tmp`는 스택에 할당된다. 각 쓰레드는 자신만의 스택을 갖고 있다. 각 쓰레드는 동일한 헤드 원소를 제거하려고 한다. 제대로 된 상황에서는 각 쓰레드는 리스트의 첫 번째 원소를 한 번씩 제거해야 한다. 이러한 상황이 모든 문제를 (4행의 헤드 원소를 두 번 삭제하고 같은 데이터 값을 두 번 반환하는 등의) 야기한다.

물론 해결책은 **락(lock)**을 사용하여 올바르게 동작하도록 만드는 것이다. 이 경우 간단한 mutex를 할당하고 (예, `pthread_mutex_t m;`) 루틴의 시작에 `lock(&m)`, 마지막에 `unlock(&m)`을 추가하면 문제를 해결할 수 있다. 곧 보겠지만, 이러한 접근

방식이 문제가 없는 것은 아니다. 특히, 성능 측면에서 문제가 있다. CPU의 개수가 증가할수록 동기화된 자료 구조에 접근하는 연산은 매우 느리게 된다.

13.3 마지막 문제점: 캐시 친화성

멀티프로세서 캐시 스케줄러에서의 마지막 문제점은 **캐시 친화성(cache affinity)**이다. 개념은 간단하다. CPU에서 실행될 때 프로세스는 해당 CPU 캐시와 TLB에 상당한 양의 상태 정보를 올려 놓게 된다. 다음 번에 프로세스가 실행될 때 동일한 CPU에서 실행되는 것이 유리하다. 해당 CPU 캐시에 일부 정보가 이미 존재하고 있기 때문에 더 빨리 실행될 것이기 때문이다. 반면 프로세스가 매번 다른 CPU에서 실행되면 실행할 때마다 필요한 정보를 캐시에 다시 탑재해야만 하기 때문에 프로세스의 성능은 더 나빠질 것이다. 하드웨어의 캐시 일관성 프로토콜 덕분에 다른 CPU에서 실행되더라도 프로그램이 제대로 실행될 것이다. 멀티프로세서 스케줄러는 스케줄링 결정을 내릴 때 캐시 친화성을 고려해야 한다. 가능한 한 프로세스를 동일한 CPU에서 실행하려고 노력하는 방향으로 결정해야 한다.

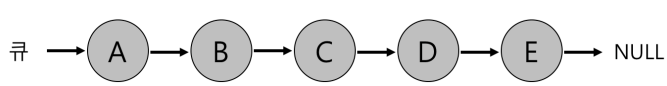
13.4 단일 큐 스케줄링

기본적인 지식을 학습했으므로 이제 멀티프로세서 시스템의 스케줄러 개발 방법에 대해 논의해 보자. 가장 기본적인 방식은 단일 프로세서 스케줄링의 기본 프레임워크를 그대로 사용 하는 것이다. 이러한 방식을 **단일 큐 멀티프로세서 스케줄링(single queue multiprocessor scheduling, SQMS)**이라고 부른다. 이 방식의 장점은 단순함이다. 기존 정책을 다수 CPU에서 동작하도록 하는 데는 많은 변경이 필요치 않다. 예를 들어, CPU가 2개라면 실행할 작업 두 개를 선택한다.

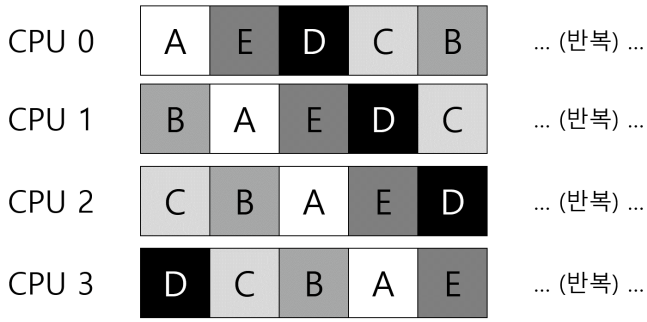
그러나 SQMS는 명백한 단점이 있다. 첫 번째 문제는 **확장성(scalability)** 결여이다. 스케줄러가 다수의 CPU에서 제대로 동작하게 하기 위해 코드에 일정 형태의 **락**을 삽입한다. 락은 SQMS 코드가 단일 큐를 접근할 때 (즉, 실행시킬 다음 작업을 찾을 때) 올바른 결과가 나오도록 한다.

불행히도 락은 성능을 크게 저하시킬 수 있고, 시스템의 CPU 개수가 증가할수록 더욱 그렇다 [And90]. 단일 락에 대한 경쟁이 증가할수록 시스템은 락에 점점 더 많은 시간을 소모하게 되고 실제 필요한 일에 쓰는 시간은 줄어들게 된다.

SQMS가 가지는 두 번째 주요 문제는 캐시 친화성이다. 예를 들어, 실행할 5개의 작업이 있고 (A, B, C, D, E) 4개의 프로세서가 있다고 가정하자. 스케줄링 큐는 다음과 같은 모양일 것이다.

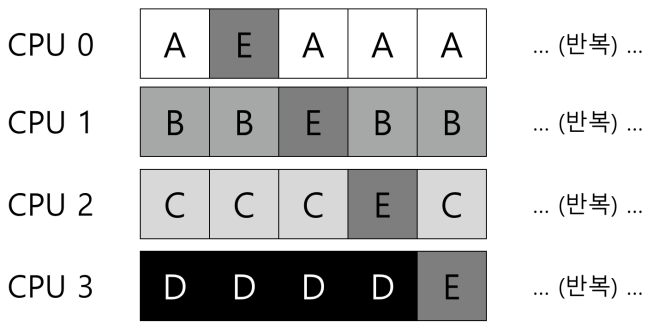


시간이 흐르면서 각 작업은 주어진 타임 슬라이스 동안 실행되고, 다음 작업이 선택되었다고 가정하자. 작업 스케줄은 다음과 같을 것이다.



각 CPU는 공유 큐에서 다음 작업을 선택하기 때문에 각 작업은 CPU를 옮겨 다니게 된다. 캐시 친화성 관점에서 보면 잘못된 선택을 하는 것이다.

이 문제의 해결을 위해 대부분의 SQMS 스케줄러는 가능한 한 프로세스가 동일한 CPU에서 재실행될 수 있도록 시도한다. 구체적으로, 특정 작업들에 대해서 캐시 친화성을 고려하여 스케줄링하고 다른 작업들은 오버헤드를 균등하게 하기 위해 여러 군데로 분산시키는 정책을 사용한다. 예를 들어 다음과 같은 동일한 5개의 작업에 대해 생각해 보자.



이 배열에서 A부터 D까지의 작업은 각각 자신의 프로세서에서 실행된다. 오직 E만이 하나의 프로세서에서 다른 프로세서로 **이동**한다. 그렇게 함으로써 대부분의 작업에게 친화성을 보존하고 있다. 다음에는 다른 작업을 이동시켜서 일종의 친화성에 대한 형평성도 추구할 수 있다. 그러나 이러한 기법은 구현이 복잡해질 수 있다.

SQMS 방식의 장단점을 볼 수 있다. 기존의 단일 CPU 스케줄러가 있다면 하나의 큐밖에 없기 때문에 구현이 간단하다. 그러나 동기화 오버헤드 때문에 이 방식은 확장성이 좋지 않고 캐시 친화성에 문제가 있다.

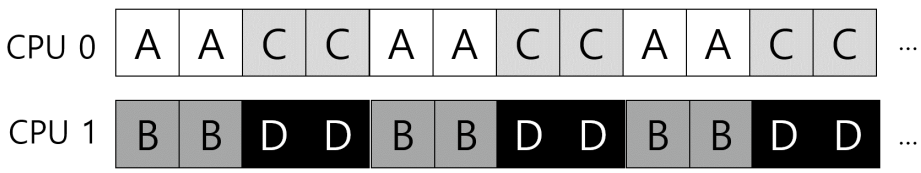
13.5 멀티 큐 스케줄링

단일 큐 스케줄러로 인한 문제 때문에 일부 시스템은 멀티 큐, 예를 들어, CPU마다 큐를 하나씩 둔다. 이 방식을 **멀티 큐 멀티프로세서 스케줄링**(**multi-queue multiprocessor scheduling**, **MQMS**)이라고 부른다.

MQMS에서 기본적인 스케줄링 프레임워크는 여러 개의 스케줄링 큐로 구성된다. 각 큐는 아마도 라운드 로빈 같은 특정 스케줄링 규칙을 따를 것이고 물론 어떤 스케줄링 기법도 사용 가능하다. 작업이 시스템에 들어가면 하나의 스케줄링 큐에 배치된다. 배치될 큐의 결정은 적당한 방법을 따른다 (예를 들어, 무작위로 할 수도 있고 또는 다른 큐보다 작은 수의 작업이 있는 큐로 배치할 수도 있다). 그 후에는 각각이 독립적으로 스케줄 되기 때문에 단일 큐 방식에서 보았던 정보의 공유 및 동기화 문제를 피할 수 있다.

예를 들어, 2개의 CPU(CPU 0과 CPU 1)와 A, B, C, D 네 개의 작업이 시스템에 존재한다고 가정하자. CPU는 각자 하나씩의 스케줄링 큐를 가지고 있으므로 운영체제는 각 작업을 배치할 큐를 결정해야 한다. 다음과 같은 결정을 내렸다고 하자.

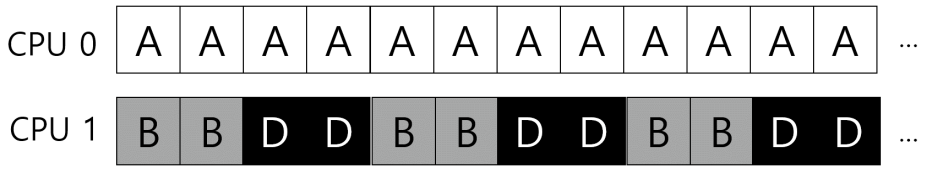
큐 스케줄링 정책에 따라 각 CPU는 현재 2개씩의 작업을 가지고 있다. 예를 들어, **라운드 로빈**의 경우 다음과 같은 스케줄이 생성될 수 있다.



MQMS가 SQMS에 비해 가지는 명확한 이점은 확장성이 좋다는 것이다. CPU 개수가 증가할수록, 큐의 개수도 증가하므로 락과 캐시 경합(cache contention)은 더 이상 문제가 되지 않는다. 또한, MQMS는 본질적으로 캐시 친화적이다. 작업이 같은 CPU에서 계속 실행되기 때문에 캐시에 저장된 내용을 재사용하는 이점을 얻게 된다.

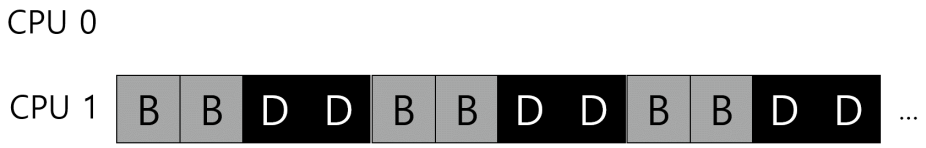
지금까지 주의를 집중해 왔다면 새로운 문제가 있다는 것을 알았을 것이다. 멀티 큐 기반 방식의 근본적인 문제는 **워크로드의 불균형(load imbalance)**이다. 앞의 예와 동일한 설정을 가정하자(4개의 작업, 2개의 CPU). 그러나 그 중 하나(예, C)가 종료되었다고 하자. 이제 다음과 같은 스케줄링 큐를 가진다.

우리는 각 큐에 라운드 로빈 정책을 실행한 경우, 다음과 같은 스케줄을 얻을 수 있다.



이 그림에서 알 수 있듯이, A가 B와 D보다 2배의 CPU를 차지하고 이는 우리가 원하는 결과가 아니다. 설상가상으로 A와 C가 모두 종료하여 B와 D만 남게 되었다고 가정하자. 스케줄링 큐는 다음과 같을 것이다.

결과 CPU 0은 유휴 상태가 된다! *(이 시점에 극적이고 불길한 음악 삽입)* 그래서 CPU 사용 타임라인은 슬프게도 다음 모양이 된다.



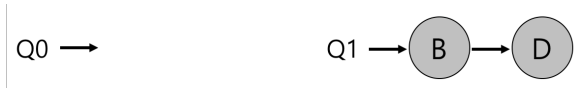
그렇다면 불쌍한 멀티 큐 멀티프로세서 스케줄러는 무엇을 해야 할까? 어떻게 하면 이 기분 나쁜 오버헤드 불균형 문제를 극복할 수 있고 나아가 Decepticon의 악의 세력을 물리칠 수 있을까[1]? 훌륭한 이 책과 관련 없는 질문을 중지하려면 어떻게 해야 하나?

핵심 질문: 워크로드 불균형에 대처하는 방법
원하는 스케줄링 목적을 더 잘 달성할 수 있기 위하여, 멀티 큐 멀티프로세서 스케줄러는 워크로드 불균형 문제를 어떻게 해결할 수 있을까?

이 질문에 대한 당연한 답은 작업을 이리저리로 이동시키는 것이다. 우리는 이 기술을 **이주**(**migration**)라고 부른다. 작업을 한 CPU에서 다른 CPU로 이주시킴으로써 워크로드 균형을 달성한다.

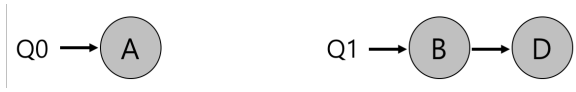
좀 더 명확하게 하기 위하여 몇 가지 예를 보도록 하자. 다시 한 번, 하나의 CPU는 유휴 상태이고 다른 CPU는 몇몇 작업을 가지고 있다고 하자.

1) 잘 알려지지 않은 사실은 사이버트론의 고향 행성이 나쁜 CPU 스케줄링 결정에 의해 파괴되었다는 것이다. 그리고 이제 이 얘기가 트랜스포머에 대한 처음이자 마지막이다. 물론, 이 점에 대해서는 진심으로 미안.

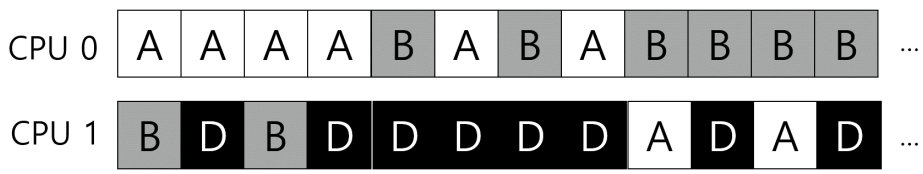


이 경우 어떻게 이주시켜야 하는지 이해하기 쉽다. 운영체제는 B 또는 D 중 하나를 CPU 0로 이동시킨다. 이 한 번의 작업 이주로 워크로드가 균형을 이루게 되고 모두가 행복하게 된다.

좀 더 까다로운 경우는 그 이전 예에서 발생한다. A는 혼자 CPU 0에 있었고 B와 D는 CPU 1에서 번갈아 실행되고 있었다.



이 경우 한 번의 이주만으로는 문제가 해결되지 않는다. 무엇을 해야 할까? 대답은 작업들을 지속적으로 이주시키는 것이다. 한 가지 가능한 해결책은 다음 타임라인에서 볼 수 있는 것처럼 계속해서 작업을 이주하는 것이다. 그림에서 처음에 A는 CPU 0에 혼자 있고 B와 D는 CPU 1에서 번갈아 실행되고 있다. 몇 번의 타임 슬라이스 후 B는 CPU 0로 이주하여 A와 경쟁한다. 그동안 D는 CPU 1에서 몇 개의 타임 슬라이스 동안 혼자 실행된다. 따라서 워크로드의 균형이 맞게 된다.



물론 가능한 많은 다른 이주 패턴이 존재한다. 그러나 지금의 더 까다로운 부분: 이주의 필요 여부를 어떻게 결정할까? 한 가지 기본적인 접근 방식은 **작업 훔치기(work stealing** [FLR88]) 라는 기술이다. 작업 훔치기에서는 작업의 개수가 낮은 (소스) 큐가 가끔 다른 (대상) 큐에 훨씬 많은 수의 작업이 있는지를 검사한다. 대상 큐가 소스 큐보다 더 가득 차 있다면 워크로드 균형을 맞추기 위해 소스는 대상에서 하나 이상의 작업을 가져온다.

물론 이러한 방식에는 자연스러운 문제가 있다. 큐를 너무 자주 검사하게 되면 높은 오버헤드로 확장성에 문제가 생기게 된다. 확장성은 멀티 큐 스케줄링의 가장 중요한 목적이다. 반면 다른 큐를 자주 검사하지 않으면 심각한 워크로드 불균형을 초래할 가능성이 있다. 시스템 정책 설계에 흔히 있는 일로서 적절한 값을 찾아내는 것은 마법의 영역이다.

13.6 Linux 멀티프로세서 스케줄러

Linux 커뮤니티에서는 멀티프로세서 스케줄러를 위한 단일화된 방식이 존재하지 않았다. 세 가지 스케줄러가 등장했다. O(1) 스케줄러, Completely Fair Scheduler(CFS) 및 BF 스케줄러 (BFS)가[2] 바로 그것이다. 언급한 스케줄러의 장점과 단점에 관한 훌륭한 개요를 보려면 Meehean의 학위 논문을 [Mee11] 참조하시오. 기본 아이디어의 일부만 요약한다.

O(1)과 CFS는 멀티 큐를, BFS는 단일 큐를 사용하기 때문에 두 방식 모두 실제 시스템에서 성공적으로 사용할 수 있다는 것을 보이고 있다. 물론, 이 스케줄러를 구분하는 다른 많은 분류 기준들이 존재한다. 예를 들어, O(1) 스케줄러는 우선순위 기반 스케줄러로서(전에 논의했던 MLFQ와 유사) 프로세스의 우선순위를 시간에 따라 변경하여 우선순위가 가장 높은 작업을 선택하여 다양한 목적을 만족시킨다. 특히, 상호작용을 가장 우선시 한다. 반면에 CFS는 결정론적(deteministic) 비례배분(proportional share) 방식이다(전에 논의했던 보폭 스케줄링에 가까움). BFS는 셋 중에서 유일한 단일 큐 방식이며 또한 비례배분 방식이다. 그러나 Earliest Eligible Virtual Deadline First(EEVDF) [SA96]라고 알려진 더 복잡한 방식에 기반을 둔다. 이 알고리즘들에 대해서는 스스로 학습하기 바란다. 이제는 스케줄링 알고리즘이 어떻게 동작하는지 이해할 수 있을 것이다.

13.7 요약

다양한 방식의 멀티프로세서 스케줄링을 살펴보았다. 단일 큐 방식(SQMS)은 구현이 용이하고 워크로드의 균형을 맞추기 용이하지만 많은 개수의 프로세서에 대한 확장성과 캐시 친화성이 좋지 못하다. 멀티 큐 방식(MQMS)은 확장성이 좋고 캐시 친화성을 잘 처리하지만 워크로드 불균형에 문제가 있고 구현이 복잡하다. 어떤 방식을 택하든지 쉬운 답은 없다. CPU 스케줄러는 사소한 코드 수정으로 시스템의 동작이 엄청나게 바뀌기 때문에 모든 경우에 다 잘 동작하는 범용 스케줄러를 구현하는 것은 매우 어렵다. 본인이 하는 작업이 가져올 결과를 정확히 이해하고 있거나, 괜찮은 보수를 받는다면 한 번 시도해 볼 만하다.

2) BF가 무엇을 나타내는지는 혼자 힘으로 찾아 보라. 미리 경고, 용기 없는 사람들을 위한 건 아니다.

참고 문헌

[And90] **"The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors"**
Thomas E. Anderson
IEEE TPDS Volume 1:1, January 1990
여러 기법들이 얼마나 확장성을 갖는지에 관한 고전적인 논문. 시스템과 네트워크 분야에서 매우 저명한 연구자인 Tom Anderson의 논문. 매우 훌륭한 운영체제 교재의 저자라는 것도 얘기해야겠다.

[Boy+10] **"An Analysis of Linux Scalability to Many Cores"**
Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, Robert Morris M. Frans Kaashoek, and Nickolai Zeldovich
OSDI '10, Vancouver, Canada, October 2010
멀티코어로 Linux를 확장시키는 어려움에 대한 훌륭한 최근의 논문.

[CSG99] **"Parallel Computer Architecture: A Hardware/Software Approach"**
David E. Culler, Jaswinder Pal Singh, and Anoop Gupta
Morgan Kaufmann, 1999
병렬 시스템과 알고리즘의 상세 내용이 담겨져 있는 보물같은 책. Mark Hill이 표지에 익살스럽게 밝힌 바와 같이 대부분의 연구 논문보다 많은 정보를 담고 있다.

[FLR88] **"The Implementation of the Cilk-5 Multithreaded Language"**
Matteo Frigo, Charles E. Leiserson, and Keith Randall
PLDI '98, Montreal, Canada, June 1998
Cilk는 경량 언어이자 병렬 프로그램을 작성하기 위한 런타임이고, 작업 훔치기 패러다임의 좋은 사례이다.

[Goo83] **"Using Cache Memory To Reduce Processor-Memory Traffic"**
James R. Goodman
ISCA '83, Stockholm, Sweden, June 1983
버스 스누핑 사용에 관한, 즉 버스에 나타나는 요청에 주의를 기울여서 캐시 일관성 프로토콜을 구현하는 방법에 관한 선구적인 논문. Goodman이 Wisconsin에서 행한 오랫동안의 연구는 영리함으로 가득 차 있으며 이 논문도 사례 중의 하나.

[Mee11] **"Towards Transparent CPU Scheduling"**
Joseph T. Meehean
Doctoral Dissertation at University of Wisconsin-Madison, 2011
최근 Linux의 멀티프로세서 스케줄링이 어떻게 동작하는지를 자세하게 다루는 학위 논문. 매우 대단함! 우리가 Joe의 공동 지도교수이기 때문에 약간 편향되었을 수 있다.

[SHW11] **"A Primer on Memory Consistency and Cache Coherence"**
Daniel J. Sorin, Mark D. Hill, and David A. Wood
Synthesis Lectures in Computer Architecture, Morgan and Claypool Publishers, May 2011
메모리 일관성과 멀티프로세서 캐싱에 관한 최고의 개요. 언급된 주제에 관해 더 많이 알고 싶어하는 사람들을 위한 필독 논문.

[SA96] **"Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation"**
Ion Stoica and Hussein Abdel-Wahab
Technical Report TR-95-22, Old Dominion University, 1996

이 멋진 스케줄링 아이디어에 관한 Ion Stoica의 기술 보고서. 그는 현재 U.C. Berkeley의 교수로 재직 중이며 네트워킹, 분산 시스템 및 다른 분야의 세계적인 권위자이다.