Chapter 10 - 멀티프로세서 스케줄링

지금은 멀티 코어 시대!

- 옛날과 달리 지금은 개인 컴퓨터에도 여러 개의 프로세서가 존재
- 따라서 그에 맞는 스케줄링 알고리즘의 설계가 필요
- 쓰레드와 하드웨어 간 데이터 공유, 캐시를 이용한 방식

캐시(Cache)

- 지역성(Locality)에 기반한 구조
- 시간 지역성(Temporal locality): 좀 전에 썼는데 이따도 쓸듯
- 공간 지역성(Spatial locality): 얘 썼으면 옆의 놈도 쓸듯
- 캐시 일관성 문제(Cache coherence): 캐시 갱신보다 메인 메모리 갱신이 느리기 때문에 한 CPU에서 갱신된 데이터가 두 번째 CPU에서 갱신되지 않고 사용될 가능성 존재 → 버스 스누핑(Bus snooping)으로 해결
- 하지만 그래도 운영체제는 캐시를 고려하더라도 공유 데이터 접근에 있어 주의해야 할 점이 존재하며 그때 사용하는 것이 락(Lock) 구조
- 캐시 친화성(Cache affinity): 아무리 캐시 일관성이 유지되더라도 다양한 곳에서 캐시를 사용하면 느려지는 것을 막을 수 없으며 멀티프로세서 스케줄링에서는 한 프로세스를 동일 CPU에서 최대한 실행하도록 해야 함

단일 큐 멀티프로세서 스케줄링(SQMS)

- CPU는 여러 개라도 큐는 한 개만 사용하는 방식
- 락 등을 통해 CPU 간에 발생하는 문제를 막으며 비교적 단순한 구현 가능
- 확장성(scalability) 결여 문제: 다수의 락은 시간 저하를 일으킴
- 캐시 친화성 문제: 큐에 맞추어 실행하다 보면 한 작업이 여러 프로세스를 타고 다닐 수 있으며 캐시 친화성이 하락 → 이를 해결하기 위한 구현은 복잡해짐

멀티 큐 멀티프로세서 스케줄링(MQMS)

- 위의 문제를 해결하기 위해 CPU마다 큐를 배치

- 확장성 결여 문제나 캐시 친화성 문제를 해결되었으나 워크로드 불균형(Load imbalance) 문제 발생 → 한쪽이 먼저 끝날지 처음에는 알 수 없으므로 나중에 가면 유휴 CPU가 생길 수 있음
- 이주(Migration)을 통해 작업을 다른 CPU로 옮김으로써 목적을 달성
- 작업 훔치기(Work stealing): 이주를 위 다른 큐의 현재 상태를 검사 → 오버헤드를 고려해 적절한 횟수의 조정 필요

실제 사례

- Linux에서는 O(1), CFS, BFS 스케줄러가 존재
- O(1)과 CFS는 멀티 큐를, BFS는 단일 큐를 사용
- O(1)은 우선순위 기반, CFS는 결정론적 비례배분
- 모든 곳에서 잘 동작하는 스케줄러를 구현하기는 어렵기 때문에 본인이 하는 작업이 무엇인지 인지하는 것이 중요