Report2-Multiprocessor Scheduling

Summary

Multicore 프로세서 출현으로 인해 Multiprocessor 시스템은 데스크탑 기기들 노트북뿐만 아니라 모바일 기기들에도 적용되고 있다.

여러 CPU를 사용해도 프로그램들이 하나의 CPU만 사용한다는 문제점이 있다. Multi-thread 어플리케이션들은 여러 CPU들에 일을 분배해 주어진 CPU 자원 내에서 더 빠르게 작동할 수 있게 thread를 사용하는 해결법이 있다.

Multiprocessor 구조

single CPU와 multi CPU 하드웨어 사이의 주요한 차이는 하드웨어 cache에 관련돼 있다.

single CPU 시스템에서는, 프로세서 속도를 높여주는 hardware cache 계층이 있다. Cache는 작고 빠르며, 메인 메모리에서 자주 사용되는 데이터의 복사본들을 가지고 있다. 반면에 메인 메모리는 모든 데이터를 가지고 있고, 접근은 느리다. 자주 사용되는 데이터를 cache에 유지하면 시스템이 크고 느린 메모리에 빨리 접근하는 것처럼 보이게 할 수 있다.

단순한 single CPU 시스템에서 메모리로부터 어떤 값을 가져오는 load instruction을 생각해보자. CPU는 작은 캐시와 큰 메인 메모리를 가지고 있다. 처음 load할 때, 데이터는 메인 메모리에 존재하기 때문에 값을 가져오는데 오래 걸린다. 데이터가 재사용될거라 생각해 프로세서는 이 데이터를 CPU 캐시에 복사해놓는다. 만약 프로그램이 후에 똑같은 데이터를 가져오려 한다면, CPU는 우선적으로 cache에 존재하는지 확인할 것이고 만약 존재한다면 더 빨리 데이터를 load할 수 있다. 이는 프로그램이 더 빨리 작동한다는 의미이다.

Cache는 locality라는 개념에 근거해 있다. Temporal locality, spatial locality의 두 가지 개념이 존재한다. Temporal locality는 어떤 데이터가 접근됐을 때, loop를 통해 접근하고 있는 것과 같이 가까운 미래에 다시 접근할 것 같은 경우이고 spatial locality는 프로그램이 x라는 주소에 있는 데이터에 접근할 때, x 주변에 있는 데이터도 접근할 것 같은 경우이다. Array를 순회하거나 순차적으로 실행되는 instruction이 그 예이다.

Multiple CPU의 캐싱은 더 복잡하다. 예를 들어, cpu1에서 실행되는 프로그램이 D라는 값의 데이터를 A라는 주소에서 읽었다. 데이터가 CPU1 캐시에 존재하지 않으므로 메인 메모리에서 값을 가져올 것이다. 그리고 프로그램은 A의 값을 변경하며 캐시에 변경된 값인 D'을 넣게 될 것이다. 메인 메모리에 값을 쓰는 것은 느리기 때문에 나중에 처리한다. 이제 OS는 프로그램을 종료하고 CPU2로 이동한다. 프로그램은 A에 있는 값을 읽어 들인다. CPU2의 캐시에는 아직 이 값이 없기 때문에 메인 메모리로 가 D'를 가져와야 맞지만 변경되기 전 값인 D를 가져오게 될 것이다. 이 문제를 cache coherence라고 한다.

기본적인 해결책은 하드웨어에 있다. 메모리 접근들을 모니터링 하면 하드웨어는 올바른 작업만 일어나게 할 수 있고, 공유 메모리에 대한 값들도 보존될 수 있다. 이를 할 수 있는 한가지 방법인 bus-based 시스템은 bus snooping이라는 오래된 테크닉을 사용하는 것이다. 각각의 캐시는 메인 메모리와 연결돼 있는 bus를 관찰하며 메모리 업데이트들을 지켜본다. CPU가 캐시에 존재하는 데이터들의 업데이트를 확인했을 때, 캐시에서 그 데이터를 지우며 저장된 복사본을 무효화하거

나, 새로운 데이터로 업데이트한다. Write-back 캐시는 더 복잡하다.

Synchronization

Coherence를 위해 캐시들이 이 모든 작업들을 한다 해도, 프로그램들이나 운영체제가 공유 데이터에 접근할 때 문제가 없는 것은 아니다.

공유된 데이터들이나 CPU 사이에 접근할 때, lock같은 mutual exclusion primitive들이 사용된다. 동시에 Multiple CPU로 접근하는 공유 queue가 있을 때, Lock 없이는 queue에서 동시에 요소들을 추가하거나 삭제하는 것은 coherence 프로토콜이 있다고 해도 예상한대로 작동하기 않을 것이다. 데이터 구조를 세세하게 업데이트하려면 lock이 필요할 것이다.

공유되는 linked list에서 하나의 원소를 삭제할 때 두개의 CPU가 동시에 실행된다.

```
6 int List_Pop() {
7 Node_t *tmp = head; // remember old head ...
8 int value = head->value; // ... and its value
9 head = head->next; // advance head to next pointer
10 free(tmp); // free old head
11 return value; // return value at head
12 }
```

Thread1은 줄6을 실행하고, tmp 변수에 head의 현재값을 저장한다. 그 후 thread2가 또 첫번째 줄을 실행하면, 똑같이 head의 값을 자신만의 tmp 변수에 저장한다. tmp는 stack에 저장되므로 각각의 thread는 각각의 저장공간을 갖고 있다. 그래서, 각 thread가 list의 헤드에서부터 원소를 삭제하는 것이 아니라, 두 thread 모두 같은 head element를 삭제하게 된다. 결국 10번 줄에서 head element를 두번 free한다거나, 같은 값을 두번 return한다거나 같은 문제들이 생긴다.

해결책은 locking이다. pthread_mutex_tm;과 같은 단순한 뮤텍스를 할당하고, lock(&m)을 루틴의 시작 부분에, unlock(&m)을 끝에 추가하면 해결된다. 코드는 예상대로 작동할 것이지만 synchronize된 공유 데이터로의 접근은 느려지는 문제가 있을 수 있다.

Cache affinity

마지막 한가지 이슈는 cache affinity이며 멀티프로세서 캐시 스케줄러를 만드는 데에 있다. 프로세스가 한 CPU에서 작동할 때, CPU의 캐시에서 상당한 용량의 state를 구축하기 때문에 다음 실행이 더 빨라질 수 있다. 하지만 만약 매번 다른 CPU에서 실행되면, 하드웨어의 캐시 coherence 프로토콜 덕분에 다른 CPU에서도 올바르게 돌긴 하겠지만, 매번 state를 불러와야 해 성능은 더 나쁠 것이다. 그래서 multiprocessor 스케줄러는 스케줄링할 때, 가능한 한 프로세스는 하나의 같은 CPU에서 돌게 cache affinity를 고려해야 한다.

Single-queue scheduling

Multiprocessor 시스템의 스케줄러를 만들 때 가장 기본적인 접근은 스케줄 돼야 하는 모든 작업들을 하나의 queue에 넣어 single processor 스케줄링의 기본 프레임워크를 재사용하는 것이다. 이는 single-queue multiprocessor scheduling(SQMS)이라고 불리며 단순하다. 다음에 어떤 작업을 실행할 지 정하고, 하나 이상의 CPU에서 실행될 수 있도록 이미 있는 체계를 수정하는 일은 그렇

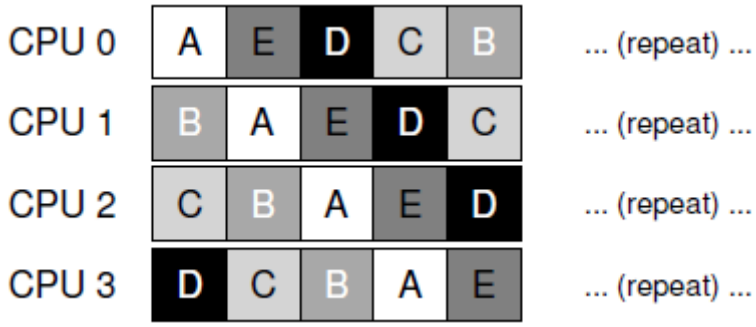
게 어렵지 않다. 하나의 확실한 단점은 scalability의 부족이다. 스케줄러가 여러 개의 CPU에서 잘 작동하게 하려면, 개발자들은 코드에 위에 설명한 것처럼 어떤 locking을 추가해야 한다. Lock은 SQMS 코드가 하나의 queue에 접근했을 때, 올바른 결과가 나오게 보장한다.

Lock은 하지만 CPU의 개수가 늘어날수록 성능을 크게 저하시킨다. Single lock이 많이 사용되면, 시스템은 lock overhead에 더 많은 시간을 쏟게 되고, 시스템이 해야 할 일들에 시간을 덜 쏟는다.

두번째 문제는 cache affinity이다. 예를 들어, 실행할 5개의 작업 A,B,C,D,E와 네 개의 프로세서가 있다고 생각해보자. 이 때, Scheduling queue는 아래와 같다.

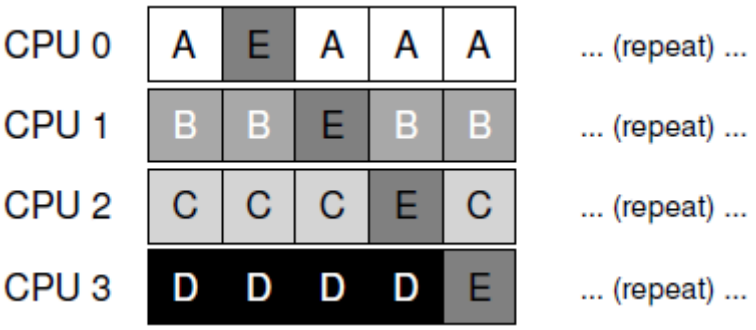
Queue → A → B → C → D → E → NULL

각각의 작업들이 time slice 단위로 돌면서 다음 작업이 선택된다고 가정했을 때, 가능한 CPU 사이의 스케줄이다.



각각의 CPU는 단순하게 공통의 queue에서 다음 작업을 고르기 때문에, 각 작업들은 다른 CPU들 사이에서 옮겨 다니게 된다. 이는 cache affinity의 관점에서 완전히 잘못된 결과이다.

이 문제를 다루기 위해 대부분의 SQMS 스케줄러들은 가능한 한 프로세서가 하나의 같은 CPU에서 실행될 수 있도록 affinity 메커니즘을 포함한다. 하지만 작업량을 알맞게 나누기 위해 작업들을 옮기기도 한다. 다음과 같은 상황을 생각해보자.



이런 배열에서, E를 제외한 다른 작업들은 CPU를 옮겨 다니지 않는다. E만 CPU 사이를 migrating한다. 대부분 affinity를 보존한다고 볼 수 있다. E가 아닌 다른 작업이 migrate하게 할 수도 있다. 하지만 이런 체계를 만드는 것은 조금 복잡할 수 있다.

그러므로 우리는 SQMS 접근이 장점과 단점이 있다고 볼 수 있다. 이미 single-CPU 스케줄러가 개발돼 있을 때, SQMS를 개발하는 것은 직관적이지만, synchronization 과부하 때문에 scale을 잘 하지 못하고 cache affinity를 손쉽게 보존할 수 없다.

Multi-queue scheduling

Single-queue 스케줄러들에 의해 생긴 문제들 때문에, 몇몇 시스템들은 하나의 CPU 당 하나의 queue를 갖는 multiple queue를 선택했다. 이 접근을 multi-queue multiprocessor scheduling, 줄여

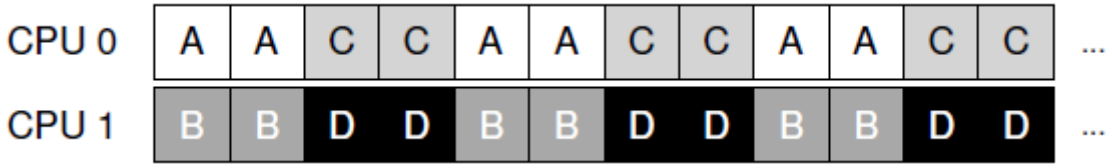
서 MQMS라고 한다.

MQMS에서 기본적인 스케줄링 프레임워크는 여러 개의 스케줄링 queue를 갖는다. 각 queue는 round robin같은 특정한 스케줄링 체계를 따른다. 한 작업이 이 시스템에 들어가면, 어떤 heuristic에 따라 하나의 스케줄링 queue에 위치된다. 그 후 각각 스케줄링 되며, single queue 접근에서 생긴 문제들이었던 정보 공유와 synchronization의 문제들을 피할 수 있다.

두 개의 CPU 시스템에 A,B,C,D 작업들이 들어왔다고 가정해보자. 각 CPU가 스케줄링 queue를 갖고 있을 때, OS는 어떤 queue에 어떤 작업을 넣을 지 결정해야한다. 아래와 같이 가능하다.

Q0 → A → C Q1 → B → D

Queue 스케줄링 원칙에 의거해서 각 CPU는 이제 두 가지의 작업 중 어떤 작업을 실행할 지 고를 수 있다. Round robin을 사용한다고 할 때, 시스템은 스케줄을 아래와 같이 정렬할 것이다.

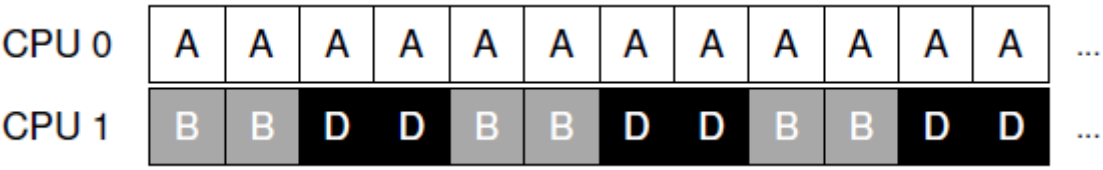


MQMS는 본질적으로 더 scalable하다는 SQMS와를 확연히 다른 장점이 있다. CPU의 숫자가 커지면, queue의 수도 늘어나서 lock과 cache의 작업량이 큰 문제는 아니다. 게다가, MQMS는 본질적으로 cache affinity를 제공한다. 작업이 같은 CPU에 항상 머무르고 캐시 데이터를 재사용한다는 이득을 확실히 취할 수 있다.

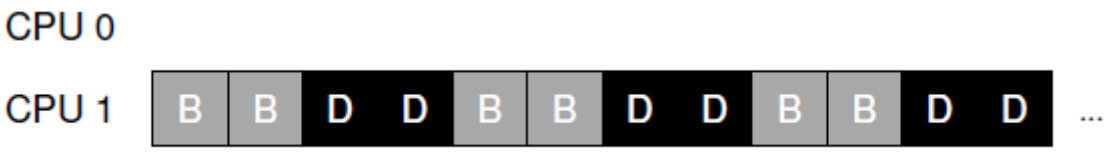
그러나, 자세히 보면 load unbalance라는 문제가 있다. 같은 상황에서, 예를 들어 작업 C가 끝났다면 스케줄링 queue가 아래와 같아진다.

Q0 → A Q1 → B → D

그 후, 우리가 이제 각각의 queue에 round robin 방식을 사용하면 아래와 같아진다.



그림에서 볼 수 있듯이, A가 B와 D보다 두배나 더 많은 CPU 자원을 사용하게 된다. 이는 원하던 결과가 아닐 뿐더러 A와 C 모두가 끝났다면, B와 D만이 시스템에 남게 된다. 그럼 두 scheduling queue와 타임라인은 아래와 같을 것이다.



CPU0은 일을 하지 않게 되고 CPU 사용 타임라인은 안 좋아진다.

확실한 load imbalance 해결법은 작업들이 옮겨 다니는 것이다. 이는 이전에 말했던 migration이다. 작업들을 CPU 들 간에 migrate 하면서, 올바른 작업량 균형을 얻어낼 수 있다.

하나의 CPU는 일을 안하고 다른 CPU만 작업하는 상황을 생각해 보자

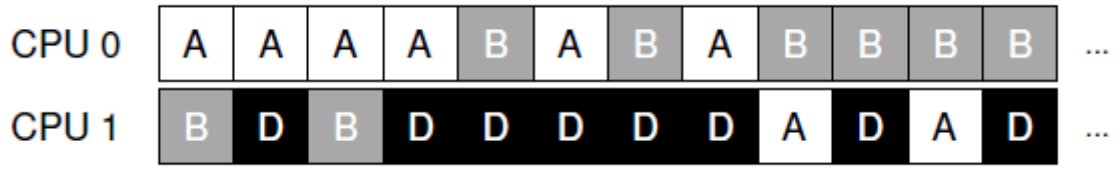
Q0 → Q1 → B → D

이 경우에 필요한 migration은 명확하다. OS는 B나 D 중 하나의 작업을 CPU0으로 옮겨야 한다. 이 한 번의 migration을 통해 작업량은 공정하게 분배된다.

한 까다로운 경우는 A만 CPU0에 남아있고 B와 D가 차례로 CPU1에서 실행되는 상황이다.

Q0 → A Q1 → B → D

이 경우에는 한 번의 migration이 해결해줄 수 없다. 이 때는 하나 이상의 작업들에 대한 연속적인 migration이 필요하다. 가능한 해결책은 아래 타임라인에서 보이는 바와 같이 계속 작업을 옮기는 것이다.



많은 다른 migration 규칙들 중 시스템은 어떻게 규칙을 정해야 할까?

하나의 기본적인 접근법은 work stealing 테크닉을 사용하는 것이다. 이 방식을 통해 작업이 적은 queue는 일정한 간격마다 다른 queue가 얼마나 차 있는 지를 확인한다. 만약 확인한 queue가 자신보다 더 차 있다면, 그 queue의 작업량을 균형있게 분배하기 위해서 한 개 이상의 작업을 steal 해온다.

물론, 그런 접근법도 문제점들이 있다. 다른 queue들을 너무 자주 보면, scaling하는 데에서의 문제나 높은 부하 같은 문제들이 생길 것이다. Multiple queue 스케줄링을 우선적으로 실행하는 전체적인 목적에 반하는 문제들이다. 하지만 또 너무 자주 다른 queue를 살펴보지 않게 되면, load 불균형에 시달리게 될 것이다. 올바른 threshold를 찾아내는 것이 중요하다.

Linux multiprocessor scheduler

리눅스에서 multiprocessor 스케줄러를 만드는 데에 있어 일반적인 해결법은 없다. 시간이 흐르며, 세 가지의 다른 스케줄러들이 개발 되었다. O(1) 스케줄러, Completely Fair Scheduler(CFS), BF Scheduler(BFS)가 있다. 각각의 장단점들은 Meehean의 논문에서 자세히 설명되어 있다.

O(1)과 CFS는 multiple queue를, BFS는 하나의 queue를 사용한다. 이는 두 가지의 접근법 모두 성공적이라는 것을 의미한다. 물론, 이 스케줄러들을 구분 짓는 다른 많은 부분들이 있다. 예를 들어, O(1) 스케줄러는 MLFQ와 비슷하게 priority-based 스케줄러이다. 시간이 흐르며 프로세스의 우선순위를 변경하고 다양한 스케줄링 목적들을 맞추기 위해 높은 우선순위의 프로세스부터 스케줄링한다. 상호작용에 특히 중점적이다. 반면에 CFS는 결정론적인 proportional-share 접근법이다. BFS는 유일한 single queue 접근법이며 이 또한 proportional-share 구조이지만, Earliest Eligible Virtual Deadline First(EEVDF)라는 훨씬 더 복잡한 체계에 기반하고 있다.

Summary

Multiprocessor 스케줄링에 대한 다양한 관점을 보았다. Single-queue 접근법인 SQMS은 만들기 더 직관적이고 부하균형을 잘 조절하지만, 많은 프로세서들의 scaling과 cache affinity 측면에서 문제가 있다. Multiple-queue 접근법인 MQMS는 더 잘 scale하고 cache affinity를 잘 보존하지만, 부하균형에 문제가 있고 더 복잡하다. 어떤 접근법을 사용하든 간단한 답은 없다. 코드의 작은 변

화만으로도 작업 내용에 큰 변화를 주기 때문에 일반적인 목적의 스케줄러를 짓는 것은 힘든 일이다. 내가 무엇을 하고 있는지 정확히 이해하고 있거나, 많은 보수를 받는다면 이 일을 연습해볼 만하다.

Takeaways

이 챕터는 multiprocessor 스케줄링에 관한 챕터였다. 우선 multiprocessor란 것은 여러 개의 CPU에 병렬적으로 작업을 처리해 작업 성능을 높이는 과정이다. 하지만 CPU는 메인 메모리로 접근하는 시간을 줄이기 위해 자주 사용되는 데이터들을 캐싱한다. 이 때문에 multiprocess가 실행되기에 조금 복잡해진다.

그저 단순하게 하나의 메인 메모리가 있고 두 개의 CPU가 똑 같은 작업을 할당 받는다면, 한 프로세스가 두 개의 CPU에서 각각 분배되어 2배로 성능이 높아지는 것을 기대할 수는 없다. 각 프로세스마다 캐싱하는 시점이 다르고 캐싱을 통해 메인 메모리의 데이터를 수정하지 않아도 되기 때문에 메인 메모리에 올바르지 않은 값이 존재할 수 있다. bus 시스템을 통해 캐싱 coherence를 얻어낼 수 있지만, 여러 CPU가 공유되는 하나의 메인 메모리에 접근할 때, 동시에 같은 요소를 추가하거나 삭제하게 되면 문제가 될 수 있다. 그래서 lock을 통해 이 문제를 해결할 수 있지만, 공유되는 데이터로의 접근이 느려지는 문제점이 발생한다. 또 한 가지 문제점은 cache affinity이다. CPU가 프로세스를 실행할 때, 캐시에 그에 맞는 state까지 구축한다. 하지만 매번 다른 CPU에서 실행되게 되면 다시 state를 구축해야해, 성능이 떨어져 보일 수 있다.

이를 해결하기 위해 single-queue scheduling, multi-queue scheduling이 있다. Single-queue scheduling은 말 그대로 여러 CPU가 하나의 queue를 이용해 스케줄링하는 것이다. 하지만 이는 scalability의 부족이라는 문제가 있고, 작업들이 여러 CPU들을 돌아다니며 실행되게 되면 결국 cache affinity의 관점에서 문제이다. 그래서 많은 SQMS 스케줄러들이 affinity 메커니즘을 포함한다. 예를 들면 다섯개의 작업 중 하나의 작업만 migrate하며 실행하는 방식으로 이를 해결한다.

Multi-queue scheduling은 하나의 CPU당 하나의 queue를 갖게 하는 것이다. 이는 확실히 cache affinity를 보장할 수 있고 캐시 데이터를 재사용한다는 이점이 있지만 한 CPU에서만 먼저 많은 작업들이 종료되게 되면, 한 CPU만 프로세스를 실행하며 자원의 낭비인 load unbalance 문제점이 발생한다. 이 해결책으로는 작업이 없는 CPU에 다른 CPU의 작업을 migrate해오는 것이다. 이에 가장 기본적인 방식으로는 work stealing 테크닉이 있다. 하지만 이 또한, 너무 자주 steal하려 하다보면, scaling에서나 높은 부하 같은 문제점이 있고, 너무 가끔 steal하려하다 보면 결국 load unbalance 문제가 나타난다. 적절한 threshold를 찾아야 한다.

리눅스의 multiprocessor scheduler는 O(1) 스케줄러, CFS, BFS 세 가지의 스케줄러가 있다. 세 가지 모두 성공적이라는 것은 모두 장단점이 존재한다는 뜻이다. 즉 일반적인 목적이 모두 부합하는 스케줄러를 설계하는 것은 매우 어려워 보인다.

Multiprocessing에 관한 모든 관점들에 대해 공부하고, 그에 따른 모든 장단점들까지 자세히 짚어볼 수 있는 단원이었다. 결론적으로 모든 문제에 일반적으로 알맞은 스케줄러가 존재하지 않는다는 점에서, 모든 알고리즘에 대해 정확히 이해하고 어떤 장단점이 있는지를 확실히 숙지해, 어떤 상황에서는 어떤 알고리즘의 테크닉을 사용해야 하는지 바로 알아낼 수 있는 능력이 multiprocessor scheduler를 구축할 때 가장 중요한 부분으로 보인다. 실행될 각각의 process들에 대한 정보도 좋은 기준이 될 수 있고, 각 CPU의 성능 차이도 중요한 것 같다. loop문이나 array를 보면 어떤 데이터가 자주 쓰일지, 또는 어떤 주소값이 순차적으로 순회될지 등등의 정보를 알 수 있기 때문에, 이런 코드 분석을 통해서도 캐싱이 어떻게 일어날지에 대해 조금은 미리 알아낼 수 있을 것이라 생각한다. 여러 스케줄러들을 분석하고 왜 이 상황에 이런 알고리즘의 스케줄러가 사용됐는지를 분석해보며 스케줄러에 대한 기본 지식을 쌓을 수 있었다.