멀티쓰레드 프로그래밍

-현재 운영체제에서 병렬 프로그래밍의 (유일한) 구현수단

-하나의 프로세스 안에 여러 개의 쓰레드를 실행

-윈도우 리눅스 안드로이드 ios에서 기본으로 제공하는 유일한 병렬 프로그래밍 API

별도의 API라이브러리로는 GPGPU가 있음(GPU를 사용한 것)

-시분할을 통해 수백개의 프로세스를 번갈아 가면서 빠르게 실행 중

실행중인 프로세스의 상태를 강제로 준비(ready)로 변경가능

프로세스가 실행되고 자식 프로세스를 생성한다

부모 프로세스는 자식 프로세스를 별도의 가상 메모리에 실행시킴 부모프로세스는 자식프로세스의 주소공간은 분리되어 있어서 서로 메모리에 접근할 수 없다. 데이터를 공유하려면 별도의 통신이 필요

프로세스는 초기에 하나의 시작 스레드를 가짐

부모 스레드가 자식 스레드를 생성할 때 스택을 새로 만듦

스레드는 자신 고유의 스택을 가지고 있고 data code heap를 공유한다

스레드마다 독립적으로 스택을 가지지만 스레드에서 다른 스레드에 스택에 접근은 가능하다(주소만 알고있으면) 그러나 이것은 바람직하지 않음 , 스택에 있는 로컬변수들은 다른 스레드에서 간섭하지 않는 전제 하에 프로그래밍 되기 때문에 다른 스레드의 스택에 접근할 경우 오류가 발생할 수 있음

프로세스와 쓰레드의 차이점,

멀티프로세스보다 멀티스레드가 좋은 이유

1. 생성 오버헤드가 적다 (스택만 새로 할당하면 되기때문)

2. context switch overhead가 적다 (프로세스 간 스위칭에는 가상메모리 맵핑 바꾸고, TLB 리셋하고, 캐시 재활용도 안됨) 스레드 스위칭에는 가상메모리도 그대로 쓴다, 스레드간 통신이 간단

멀티스레드의 단점

1. 하나의 스레드에서 발생한 문제가 전체 프로세스를 멈추게 함

2. 디버깅이 어렵다(같은 인풋에도 결과가 계속 다를 수 있음)

멀티쓰레드의 사용 목적

1. 멀티코어 cpu에서 프로그램 성능 향상 O (성능향상이 되지않으면 필요가 없음, 지금도 충분히 성능이 잘 나오는데 굳이 멀티쓰레드를 사용할 이유는 없음, 싱글코어 컴퓨터에서는 멀티쓰레드가 성능향상이 없음)

2. 멀티 cpu 컴퓨터에서 프로그램 성능 향상 O (듀얼코어 cpu하나와 싱클코어 cpu가 두개있는 것은 프로그래머 입장에선 차이가 없다.)

3. 분산컴퓨터에서의 성능향상은 X (멀티쓰레드는 하나의 프로세스에서 실행되기 때문에 분산컴퓨터에서는 성능향상이 없다. 분산컴퓨터에서는 멀티프로세스를 써야함)

4. 프로그램을 모듈화 해서 알아보기 쉽게 하기 위함? X (멀티쓰레드가 알아보기 쉬울 순 있으나 그로 인해 오는 단점이 더 크기 때문에 부적절)

멀티코어 CPU: 한 개 이상의 코어로 구성된 cpu를 뜻함

멀티코어 프로세스를 만드는 이유

1. cpu의 성능을 올리기 위해

2. 클럭 속도를 높일 수 없다 (발열문제<물리법칙>)

3. 클럭 속도 말고 cpu의 속도를 올리는법 -> 아키텍쳐(캐시, 파이프라인, 예측분기, 동적수행 등등) 개선이 한계에 부딪힘

-> 남은 방법은 멀티 코어

멀티코어 cpu가 늦게 나온 이유

1. 프로그램을 새로 짜지 않으면 성능향상이 없다

2. 전혀 다른 알고리즘을 써야한다

3. 디버깅이 어렵다

4. 옛날부터 있었다 -> 과학 기술 계산용: 기상, QCD, 유체역학, N-body problem 등

수출 금지 품목(군사적 이유)

좋은 멀티쓰레드 프로그래밍이란 -> core의 개수에 비례해서 실행속도가 증가하는 것

리눅스에서 thread라는 개념이 없다

-모든 것은 process

-Pthread 라이브러리가 마치 thread가 존재하는 것처럼 보이게 해준다.

-code, data, 자원을 공유하는 process를 생성할 수 있다.

-다른 운영체제의 thread와 다른 것이 없다.

DATA RACE

Sum = sum + 2; 우리가 볼 때는 코드 한 줄이지만 어셈블러로 보면 어셈블러 코드 여러 줄 이고 그 여러 줄 사이에서 스레드가 컨텍스트 스위칭 될 수도 있음

\_asm add sum, 2;

이렇게 바꾸고 싱글 코어에서 실행하면 오류 없음

디스어셈블러에서 보면 어셈블러코드 한 줄로 끝나기 때문에 운영체제에서 인터럽트가 발생하지 않기 때문에 올바른 결과가 나온다

Lock과 Unlock사용

-mutex클래스 객체 생성후 lock(), unlock()

-같이 실행하면 안되는 코드는 같은 뮤텍스 객체를 사용해서 락을 걸어야한다. 쓰레드마다 서로 다른 뮤텍스 객체를 쓰면 뮤텍스끼리 동기화가 안되서 lock이 의미가 없음

-올바른 결과는 나온다 그러나, 속도가 느려짐 -> 멀티스레드의 의미가 퇴색

-lock은 한번에 하나의 쓰레드만 실행된다 그래서 병렬성이 감소 -> 성능향상 감소

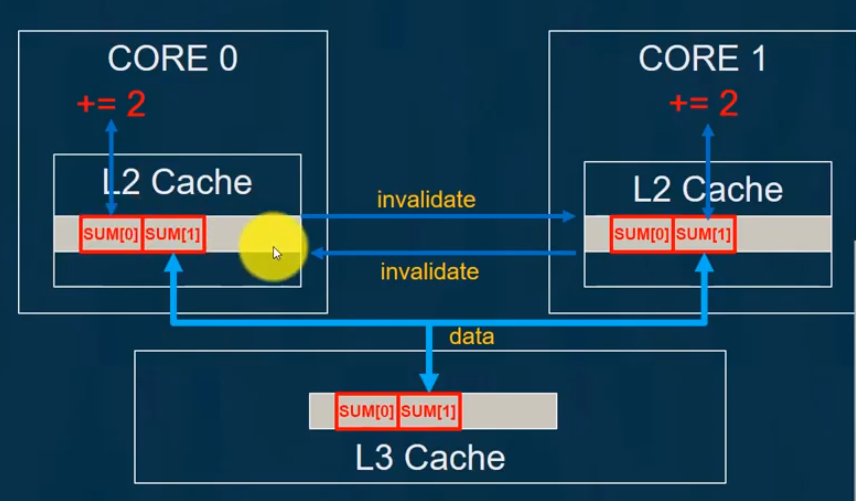
-lock()메소드 자체의 오버헤드가 크다

Atomic

atomic으로 변수를 선언하면 lock안써도됨

Cache Thrashing 캐시 쓰레싱

+=2 연산을 할 때 캐시에서 sum값을 읽어온다. 캐시를 읽을 때 바이트단위가 아니라 캐시라인 단위로 읽는다



~~코어에서 데이터레이스는 없지만 캐시라인단위로 캐시를 읽어서 결국 L3캐시에서 데이터를 가져와서 읽는다?~~

volatile int sum[MAX\_THREAD \* 64];

이렇게 해결할수있음

멀티스레드 프로그래밍의 종류

1. Heterogeneous 멀티쓰레딩

쓰레드마다 맡은 역할이 다르다.

다른 code part를 실행

쓰레드 간 Load Balancing이 힘들다

병렬성이 제한된다.

2. Homogeneous 멀티쓰레딩

Data/Event Driven 프로그래밍

모든 쓰레드는 Symmetric하다 – available 한 순서대로 input을 처리한다

자동적인 load balancing, 제한 없는 병렬성

작업 분배 Queue를 비롯한 일반적인 병렬 자료구조 필요

Volatile int \*a; -> 포인터 a가 가르키는 실제 값이 volatile

Int \* volatile a; -> 포인터변수 a 자체의 값이 volatile

임계영역

mutex lock은 systemcall로 구현하지 않고 알고리즘 + 특별한 cpu명령어로 구현한다

피터슨 알고리즘에서 오류가 생기는이유

PC에서의 메모리 접근이 atomic이 아니다

99.5%는 atomic이지만 0.5%정도는 atomic이 아니다

컴파일러 문제는 아님

메모리에 데이터를 쓴 순서대로 바뀌지 않음(메모리의 일관성 문제)

ex. 코드순서상 데이터 쓰기가 먼저고 그 다음이 데이터 읽기여도 데이터를 읽을 때 쓰기전 데이터를 읽을수도있음 즉 cpu가 명령어를 실행할 때 코드 순서대로 실행이안됨

write buffer: 캐시미스가 나서 메인메모리에 접근해야되는 상황이 되면 일단 버퍼에 넣고 다음 cpu는 다음명령을 수행한다. write buffer때문에 메모리업데이트 순서가 바뀌면서 일관성이 깨진다.

volatile키워드는 기계어로 변역되지 않음. 컴파일러에만 영향을 줌. cpu연산과는 관련이없음.

mfence 를 코르어디에 넣어야할지 알기힘듦

mfence 를 남발하면 성능이저하됌

atomic을 사용하면 코드사이에 mfence를 안넣어도되지만 성능이 더 안좋음

메모리에서 캐시로 데이터를 가져올때 캐시라인단위로 가져옴

x86 cpu에선 캐시라인크기는 64바이트

addr = addr / 64;

addr = addr \* 64;

이 코드로 데이터를 캐시라인 끝과 첫부분에 걸치게 만듬

그래서 캐시라인 두줄에 걸쳐서 데이터가 존재해서 캐시에서 메모리로 데이터를 전송할때

캐시라인 한줄씩 작업 => 총 두번작업이 일어나면서 atomic하지않게 됨

이러한 상황에 대한 대책

모든데이터를 char단위로 만듬 -> X

포인터변수를 쓸때 캐시라인에 걸치는지 확인을해야함

c++에서 atomic으로 선언한 변수들끼리만 메모리일관성이 생긴다.

atomic 변수는 다른 변수들과는 메모리 일관성이 깨질수있다.

over subscription - 사용가능한 코어개수보다 쓰레드를 더 많이 만들어서 lock을 얻은 스레드가 스케쥴링에서 제외될 경우 lock을 기다리는 모든 스레드가 공회전 하게된다.

무대기(wait free) - 모든 스레드의 메소드가 정해진 시간에 끝나는 것

무잠금(lock free) - 여러개의 스레드가 서로 간섭해서 딜레이가 생기는 상황이지만 멈추지않고 적어도 하나의 메소드는 작업을 마친다. (딜레이는 있지만 멈추지는 않음)

단점은

Atomic은 기본자료구조에만 쓸 수 있음(int char short 등)

CAS 이론

CAS알고리즘이 x86cpu에 70년대부터 있었음 -> lock unlock을 구현하기 위해 만들어짐

Cas는 모두 동시에 실행되지만 cas알고리즘 true를 받는건 한 스레드

cas구현방식

X86 cpu는 bus locking 방식

Arm cpu는 LL/SC 방식 (Load Linked / Store Conditional)

Armcpu방식이 더 좋음 ABA문제를 피할 수 있음