Thread 를 만들자마자 바로 메인함수를 지정해줄 수 있다.

병렬적으로 쓰레드와 메인 동시에 실행

Main쓰레드가 먼저 종료되면 에러발생

t.join()쓰레드가 끝날때까지 대기해준다.

#include "pch.h"

#include <iostream>

#include "CorePch.h"

#include<thread>

// 공용적으로 사용하는 thread

void HelloThread() {

cout << " Hello Thread " << endl;

}

int main()

{

std::thread t(HelloThread); // 코드가 실행되면서 헬로쓰레드를 바로 시행

int32 count = t.hardware\_concurrency(); // CPU 코어 개수?

auto id = t.get\_id(); // 쓰레드마다 부여되는 id

//t.detach(); // t와 실질적으로 실행되는 쓰레드와의 연결고리를 끊어준다. std::thread 객체에서 실제 쓰레드를 분리한다.

// 분리를 하게 되며 t라는 애의 상태와 정보를 추출할 수 없기 때문에 활용하는 경우는 거의 없다.

//

if (t.joinable()) {

t.join();

}//연동된 쓰레드가 없다거나 하는것을 판별하는 ..

//t.join();

cout << " Hello Main" << endl;

cout << count << endl;

}

Vector로 여러 개의 쓰레드를 관리

#include "pch.h"

#include <iostream>

#include "CorePch.h"

#include<vector>

#include<thread>

// 공용적으로 사용하는 thread

void HelloThread() {

cout << " Hello Thread " << endl;

}

void HelloThread\_2(int32 num) {

cout << " num = " << num << endl;

}

int main()

{

vector<std::thread> v;

for (int i = 0; i < 10; ++i) {

v.push\_back(std::thread(HelloThread\_2, i));

}

for (int i = 0; i < 10; ++i) {

if (v[i].joinable()) {

v[i].join();

}

}

cout << " Hello Main" << endl;

}

멀티 쓰레드 환경에선 공유데이터가 항상 골치를 썩게 할 것 이다.

스택 같은 경우에는 자신의 영역을 따로 가지고 있다.

그러나 힙 이나 데이터 영역 같은 경우는 쓰레드끼리 서로 공유해서 사용하고 있는 것 들이기 때문에 전역으로 만든 즉 데이터 영역에 올라가 있는 데이터라고 하면 두 쓰레드에서 서로 경합을 해서 서로 건들이고 있는 상황이다. 따라서 C++이 필요한 이유는 어떤 영역을 건드리고 있는지를 확인해야 한다.

공유데이터에 접근하는 순서가 정해져있지 않기 때문에 서로 데이터를 사용하려고 접근하여 충돌하는 오류가 발생한다.

데이터를 동시에 다루는 문제들을 동기화라는 용어로 표현한다.

순서가 보장이 되어야 한다.

여러가지 동기화 버전중 atomic한 버전이 있다.

All or Nothing

#include "pch.h"

#include <iostream>

#include "CorePch.h"

#include<thread>

#include<atomic>

// atomic atom(원자) : All - Or - Nothing

// DB

// A라는 유저 인벤에서 집행검이라는 아이템을 빼고 B라는 유저 인벤에 집행검을 추가한다.

// 교환이라고 볼수 있다. 반드시 atomic하게 한번에 일어나야 한다.

atomic<int32> sum = 0;

void Add()

{

for (int32 i = 0; i < 1000000; ++i) {

sum.fetch\_add(1);

//int32 eax = sum;

//eax = eax + 1;

//sum = eax;

}

}

void Sub()

{

for (int32 i = 0; i < 1000000; ++i) {

sum.fetch\_add(-1);

//int32 eax = sum;

//eax = eax - 1;

//sum = eax;

}

}

int main()

{

Add();

Sub();

cout << sum << endl;

std::thread t1(Add);

std::thread t2(Sub);

t1.join();

t2.join();

cout << sum << endl;

}

LOCK

#include "pch.h"

#include <iostream>

#include "CorePch.h"

#include<thread>

#include<atomic>

#include<vector>

#include<mutex>

vector<int32> v;

// Mutual Exclusive (상호 배타적)

mutex m;

//RAII ( Resource Acquisition Is Initialization)

template<class T>

class LockGuard {

public:

LockGuard(T& m)

{

-mutex = &m;

\_mutex->lock();

}

~LockGuard() { \_mutex->unlock(); }

private:

T\* \_mutex;

};

void Push() {

std::lock\_guard<std::mutex> lockGuard(m);

for (int32 i = 0; i < 10000; ++i) {

// 자물쇠 잠그기

//LockGuard<std::mutex> lockGuard(m);

/\* std::unique\_lock<std::mutex> uniqueLock(m,std::defer\_lock);

uniqueLock.lock();\*/

//m.lock();

v.push\_back(i);

if (i == 5000) {

//m.unlock();

break;

}

// 자물쇠 풀기

//m.unlock();

}

}

int main()

{

v.reserve(20000);

std::thread t1(Push);

std::thread t2(Push);

t1.join();

t2.join();

cout << v.size() << endl;

}

DeadLock

LOCKguard를 사용한다고 해서 모든 데드락들이 사라지진 않는다.

언락을 하지않는 상황 말고도 다른 데드락 현상이 발생하는데 실제 현업에서도 많이 발생한다.

Null 포인터 크래쉬가 많이 발생한다.

락이 풀릴때까지 무작정 그 앞에서 기다리는 방법 , (스핀락)

그 앞에서 나오면 바로 사용할 수 있다는 장점이 있지만 기다리기만 해야하는 단점 ,

스핀락 면접에서 자주나오는 질문

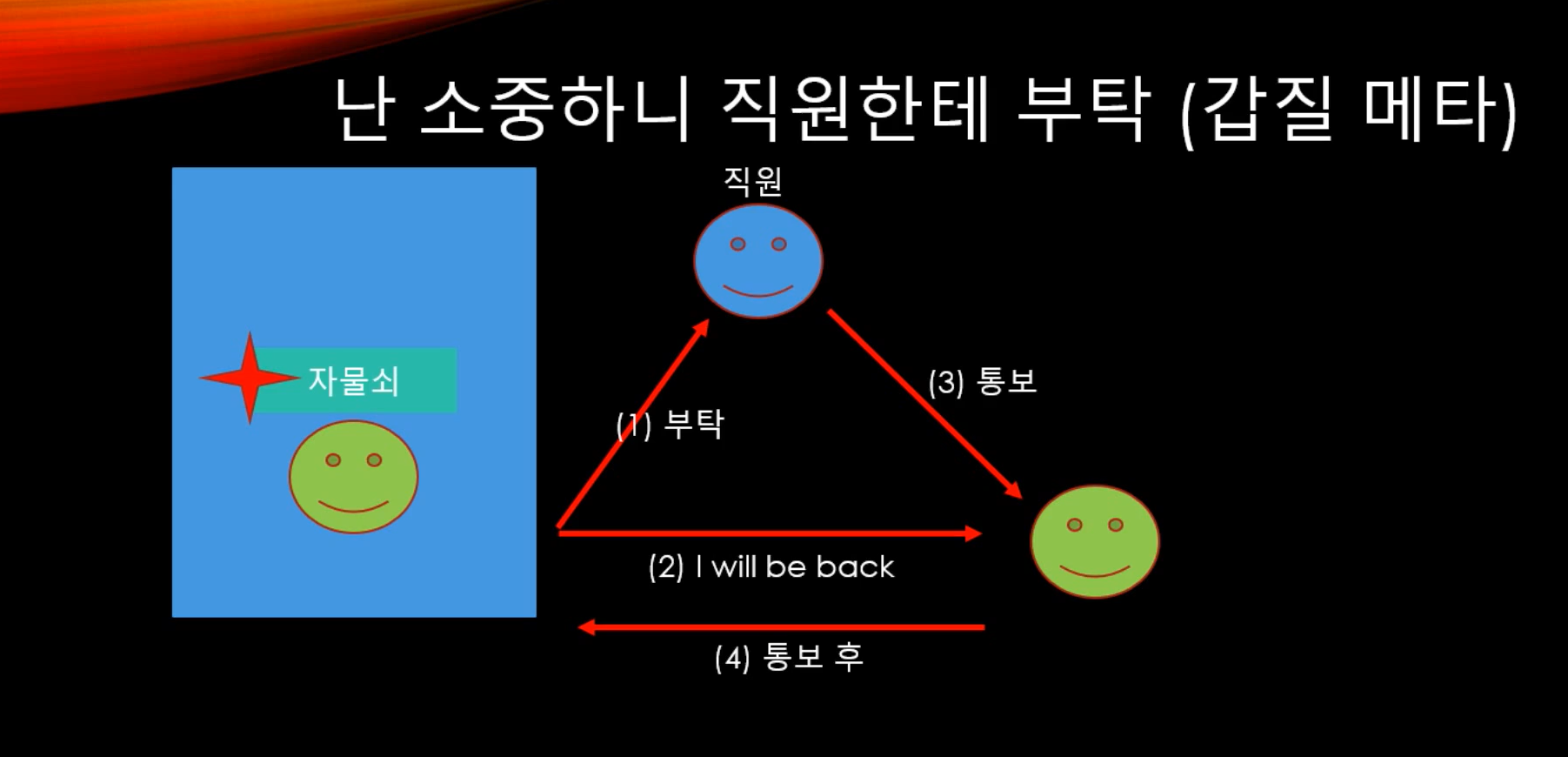
그다음 방법은

일단은 자리로 되돌아가고 나중에 다시 오겠다. 는 방식 ( 랜덤방식)

확실성이 없고 운에 맡긴다는 단점 , 만약 되돌아갔는데 10초만 기다려도 차지할수 있ㄸ너것을 그걸 못기다려서 되돌아간다면 그사이에 다른 사람이 들어갈 수 있는 그런 낭비가 발생한다.

굉장히 미묘하게 엇갈릴 수 있다. 기다리는 시간은 없지만 왔다갔다 하면서 시간을 효율적으로 사용하는 것 같지만 운에 맡기는 것이기 때문에 …

다음 방법은



자리가 나면 누군가가 와서 알려주는

단점 다른 사람의 리소스를 그만큼 사용해야한다.

컨텍스트 트위칭 ,

유저에서 관리자모드로 돌아가는 ..

기다리다가 자리로 돌아가는 이런 상황들

생각보다 부하를 많이 잡아먹는다. 실행하기 위한 부가적인 정보들이 많이 레지스터에 들어가게 되는데 다른 레지스터들을 불러워서 복원을 시켜주고 등등

스핀락

#include "pch.h"

#include <iostream>

#include "CorePch.h"

#include<thread>

#include<atomic>

#include<vector>

#include<mutex>

#include"AccountManager.h"

#include"UserManager.h"

class SpinLock

{

public:

//lock을 거는

void lock()

{

bool expected = false;

bool desired = true;

//CAS(Compare - And -Swap\_

//CAS 의사코드

/\* if (\_locked == expected) {

expected = \_locked;

\_locked = desired;

return true;

}

else

{

expected = \_locked;

return false;

}\*/

while (\_locked.compare\_exchange\_strong(expected, desired) == false)

{

expected = false;

}

}

void unlock()

{

//lock을 푸는

//\_locked = false;

\_locked.store(false);

}

private:

atomic<bool> \_locked = false;

};

int32 sum = 0;

mutex m;

SpinLock spinLock;

void Add() {

for (int32 i = 0; i < 1000; ++i)

{

lock\_guard<SpinLock>guard(spinLock);

sum++;

}

}

void Sub() {

for (int32 i = 0; i < 1000; ++i)

{

lock\_guard<SpinLock>guard(spinLock);

sum--;

}

}

int main()

{

thread t1(Add);

thread t2(Sub);

t1.join();

t2.join();

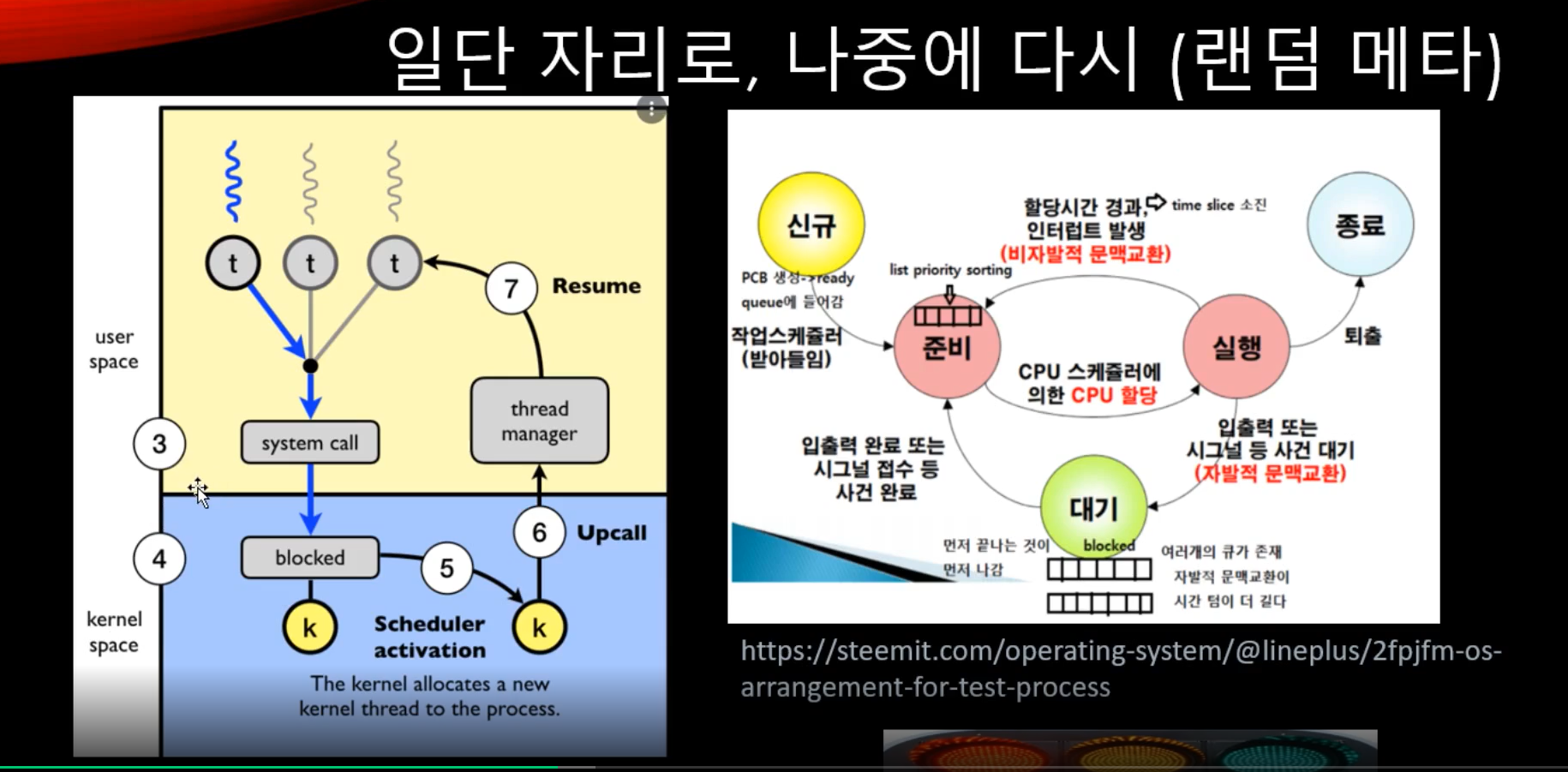
cout << sum << endl;

}존버메타

공용으로 화장실을 이용하고 먼저들어가는사람이 자물쇠를 잠궈버리는 상황, 경쟁을 하고 있는 상황이라고 가정을 해보자 여기서 문제는 화장실에 들어가는 행동이랑 들어가서 자물쇠를 걸어버리는 행동은 사실 한번에 일어나야 말이되는 상황이다 하지만 지금은 그런 제한이 존재하지 않기 때문에 두명이 동시에 화장실을 입장해서 서로 승리자라고 주장하면서 같이 문을 잠궈버리는 상황이 발생한다 우리눈에는 한번에 일어나는것처럼 보이지만 실제로 컴퓨터는 어셈블리어로 변경하여 단계를 나누어 실행한다. 따라서 이런 상태가 발생한다. 결국에는 이렇게 문제가 되는 상황은 자물쇠를 잠구는것과 화장실에 들어가는 것이 쪼개지만 안된다. 이를 아토믹하게 일어나야한다라고 말할수 있다.

락을 구현할 때 대기 하는 두번째 방법인 랜덤메타.에 대해 알아볼 것이다.

지난시간에는 대부분의 기능을 구현했기 때문에 오늘은 단순하고 sleep계열의 함수를 호출하기만 하면 된다. Sleep을 한다는 것 자체가 운영체제의 스케쥴링과 밀접한 연관이 있기 때문에 이거에 대해서만 몇가지 언급을 해야한다.

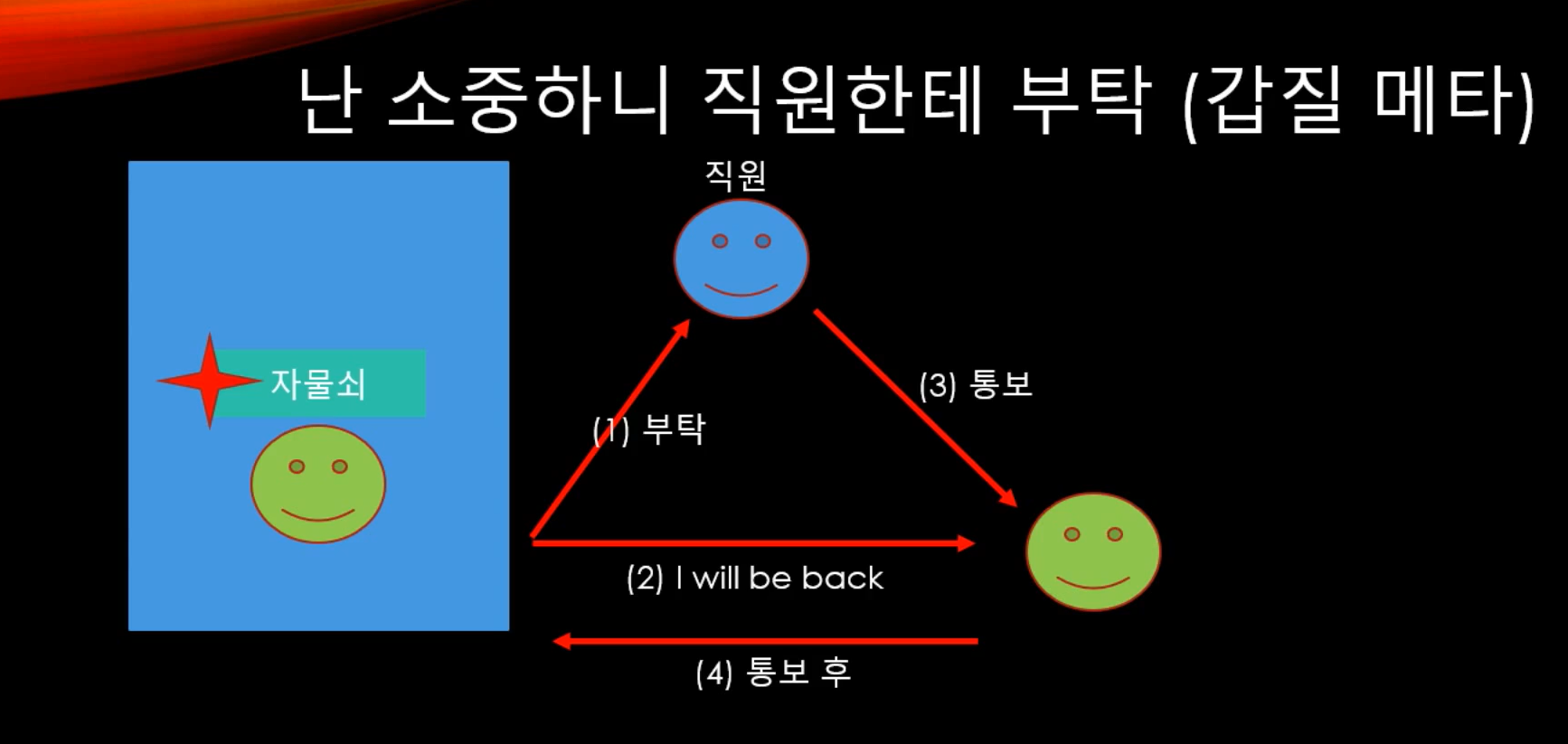


this\_thread::sleep\_for(0ms);

//this\_thread::yield();

갑질메타 EVENT

Lock에서만 사용되지 않고 많은 부분에서 사용 중요\*\*



여기서의 직원은 관리자를 얘기한다. 관리자가 우리 대신해서 순서보장을 해달라 요청을 하는 것

관리자가 심판의 역할을 맡게되는건데 이벤트를 사용하고 만들 때 C#기준으로는 auto reset event와 manual reset event 였는데 c++에선 코드에 사용에 따라 갈린다. 상태에 따라가지고 진행이 가능한지 안한지 여부를 판단할 수 있다 .

다시 살펴보면 서로 순서를 정해야 하는 상황에서 안에 있는 애가 언제 나올지 모르고 문앞에서 계속 기다리고 있기도 뭐하니까 중재를 요청한다 관리자에게 나는 자리에 있을태니까 나오면 나를 깨워줘!

#include "pch.h"

#include <iostream>

#include "CorePch.h"

#include<thread>

#include<atomic>

#include<vector>

#include<mutex>

#include<windows.h>

mutex m;

queue<int32> q;

HANDLE handle;

void Producer()

{

while (true) {

{

unique\_lock<mutex> lock(m);

q.push(100);

}

::SetEvent(handle); // signal상태로 바꿔주세요 true를 대입하는 느낌

this\_thread::sleep\_for(10000ms);

}

}

void Consumer()

{

while (true)

{

::WaitForSingleObject(handle, INFINITE);

//NON - signal

unique\_lock<mutex> lock(m);

if (q.empty() == false)

{

int32 data = q.front();

q.pop();

cout << data << endl;

}

}

}

int main()

{

//커널 오브젝트

// Usage count

// Signal ( 파란불 ) / Non\_signal ( 빨간불 ) << bool

// Auto // Manual << bool

handle = ::CreateEvent(NULL/\*보안속성\*/, FALSE/\*bManualReset\*/, FALSE/\*bInitialState\*/, NULL);

thread t1(Producer);

thread t2(Consumer);

t1.join();

t2.join();

::CloseHandle(handle);

}

Condition Variable

임의로 모든애들에게 동작하지 X

Mutex와 짝지어서 동작한다.

Event와 굉장히 비슷하다.

#include "pch.h"

#include <iostream>

#include "CorePch.h"

#include<thread>

#include<atomic>

#include<vector>

#include<mutex>

#include<windows.h>

mutex m;

queue<int32> q;

HANDLE handle;

//참고 ) CV는 user-Level Object ( 커널 오브젝트 X )

condition\_variable cv;

void Producer()

{

while (true) {

// 1) Lock을 잡고

// 2) 공유 변수 값을 수정

// 3) Lock을 풀고

// 4) 조건변수를 통해 다른 쓰레드에게 공지

{

unique\_lock<mutex> lock(m);

q.push(100);

}

cv.notify\_one(); // wait중인 쓰레드가 있으면 딱 1개를 깨운다.

//this\_thread::sleep\_for(10000ms);

}

}

void Consumer()

{

while (true)

{

unique\_lock<mutex> lock(m);

cv.wait(lock, []() {return q.empty() == false; });

// 1) Lock 을 잡고

// 2) 조건 확인

// - 만족 => 빠져나와서 이어서 코드를 진행한다.

// - 불만족 => Lock을 풀어주고 대기 상태

// 그런데 notify\_one을 했으면 항상 조건식을 만족하는 것 아닐까?

// Spurious WakeUp (가짜 기상? )

//while (q.empty() == false)

{

int32 data = q.front();

q.pop();

cout << data << endl;

cout << q.size() << endl;

}

}

}

int main()

{

//커널 오브젝트

// Usage count

// Signal ( 파란불 ) / Non\_signal ( 빨간불 ) << bool

// Auto // Manual << bool

handle = ::CreateEvent(NULL/\*보안속성\*/, FALSE/\*bManualReset\*/, FALSE/\*bInitialState\*/, NULL);

thread t1(Producer);

thread t2(Consumer);

t1.join();

t2.join();

::CloseHandle(handle);

}

\*\* 동기, 비동기 차이

데이터 처리모델

* 동기
  + 데이터의 요청과 처리가 한 자리에서 동시에 발생하는 것
* 사용자가 데이터를 서버에게 요청한다면 그 서버가 데이터 요청에 따른 응답을 사용자에게 다시 리턴해주기 전까지 사용자는 다른 활동을 할 수 없으며 기다려야만합니다.
* **ex)**  
  브라우저를 실행시키는 시간이 약 10분이 소모된다고 가정했을시, 브라우저가 실행되는10분이라는 시간 동안 사용자는 컴퓨터💻의 다른 프로그램들을동작시키지 못하며 브라우저가 켜지는 그 순간만을 계속 기다려야한다는 것입니다. 이 시간이 10분이든 100분이든 관계없이 **한 개의 데이터 요청에 대한 서버의 응답이 이루어질 때까지 계속 대기해야만합니다.**
* 비동기
  + 동시에 일어나지 않는 다는 의미
* 서버에게 데이터를 요청한 후 요청에 따른 응답을 계속 기다리지 않아도되며 다른 외부 활동을 수행하여도되고 서버에게 다른 요청사항을 보내도 상관없습니다

Future

경우에 따라서 가벼운 상황에서 순서를 보장해서 다른애한테 이벤트를 알려줘야 하는 단발성 이벤트가 발생할 수 있다. 이를 Future를 사용한다.

활용도가 떨어지는 것 같긴 하다.

#include "pch.h"

#include <iostream>

#include "CorePch.h"

#include<thread>

#include<atomic>

#include<vector>

#include<mutex>

#include<windows.h>

#include<future>

//공용데이터

int64 result;

int64 Calculate() {

int64 sum = 0;

for (int32 i = 0; i < 100000; ++i)

sum += i;

return sum;

}

// 여기서 100만이 아니라 100처럼 간단한 것이라면? 쓰레들르 만들어서 하는것은 너무 무거운 방법 프로그램의 효율성이 떨어진다

// 비동기방식

// 이럴때 future를 사용한다.

int main()

{

// 동기 ( synchronous) 실행

int64 sum = Calculate();

cout << sum << endl;

thread t(Calculate);

//todo

t.join();

}

#include "pch.h"

#include <iostream>

#include "CorePch.h"

#include<thread>

#include<atomic>

#include<vector>

#include<mutex>

#include<windows.h>

#include<future>

//공용데이터

int64 result;

int64 Calculate() {

int64 sum = 0;

for (int32 i = 0; i < 100000; ++i)

sum += i;

return sum;

}

// 여기서 100만이 아니라 100처럼 간단한 것이라면? 쓰레들르 만들어서 하는것은 너무 무거운 방법 프로그램의 효율성이 떨어진다

// 비동기방식

// 이럴때 future를 사용한다.

int main()

{

// 동기 ( synchronous) 실행

int64 sum = Calculate();

cout << sum << endl;

//std::future

{

// 비동기 방식

// 3가지 옵션

// 1) deferred -> lazy evaluation 지연해서 실행하세요

// 2) async -> 별도의 쓰레드를 만들어서 실행하세요

// 3) defered | async -> 알아서 골라주세요

std::future<int64> future = std::async(std::launch::async, Calculate);

//결과물을 당장받을필요가 없다고 가정

// TODO

int64 sum = future.get(); // 결과물이 이제서야 필요하다.

}

//실제로 데이터 로딩을 할 때 사용

}

if (status == future\_status::ready)

{

//잠시 살펴보는 기능

}

class Knight

{

public:

int64 GetHP() { return 100; }

};

Knight knight;

std::future<int64> future2 = std::async(std::launch::async, &Knight::GetHP, knight); // Knight.getHP() 멤버함수도 호출가능 하지만 인스턴스가 꼭 필요

------------------------

Future 최종

#include "pch.h"

#include <iostream>

#include "CorePch.h"

#include<thread>

#include<atomic>

#include<vector>

#include<mutex>

#include<windows.h>

#include<future>

//공용데이터

int64 result;

int64 Calculate() {

int64 sum = 0;

for (int32 i = 0; i < 100000; ++i)

sum += i;

return sum;

}

void PromiseWorker(std::promise<string>&& promise)

{

promise.set\_value("Secret Message");

}

void TaskWorker(std::packaged\_task < int64(void)> && task)

{

task();

}

int main()

{

// 동기 ( synchronous) 실행

int64 sum = Calculate();

cout << sum << endl;

//std::future

{

// 비동기 방식

// 3가지 옵션

// 1) deferred -> lazy evaluation 지연해서 실행하세요

// 2) async -> 별도의 쓰레드를 만들어서 실행하세요

// 3) defered | async -> 알아서 골라주세요

// 언젠가 미래에 결과물을 뱉어줄거야!

std::future<int64> future = std::async(std::launch::async, Calculate);

//결과물을 당장받을필요가 없다고 가정

// TODO

int64 sum = future.get(); // 결과물이 이제서야 필요하다.

}

//std::promise

{

// 미래(std::future) 에 결과물을 반환해줄꺼라 약속(std::promise) 해줘~ (계약서?)

std::promise<string> promise;

std::future<string> future = promise.get\_future();

thread t(PromiseWorker, std::move(promise));

string message = future.get();

cout << message << endl;

t.join();

}

//std::packaged\_task

{

std::packaged\_task<int64(void)>task(Calculate);

std::future<int64> future = task.get\_future();

std::thread t(TaskWorker, std::move(task));

int64 sum = future.get();

cout << sum << endl;

t.join();

}

// mutex, condition\_variable 까지 가지 않고 단순한 애들을 처리할 수 있는

// 특히나, 한 번 발생하는 이벤트에 유용하다

// 닭잡는데 소잡는 칼을 쓸 필요 없다. }

컴퓨터 구조 원리

캐시와 파이프라인

왜 캐시가 필요한지 알아보자

컴퓨터의 두뇌는 CPU

메모리는 RAM어딘가에 올라와 있는 상태 접근을 해서 사용할 때 마다 CPU에서 접근을 해야하ㄴ는데

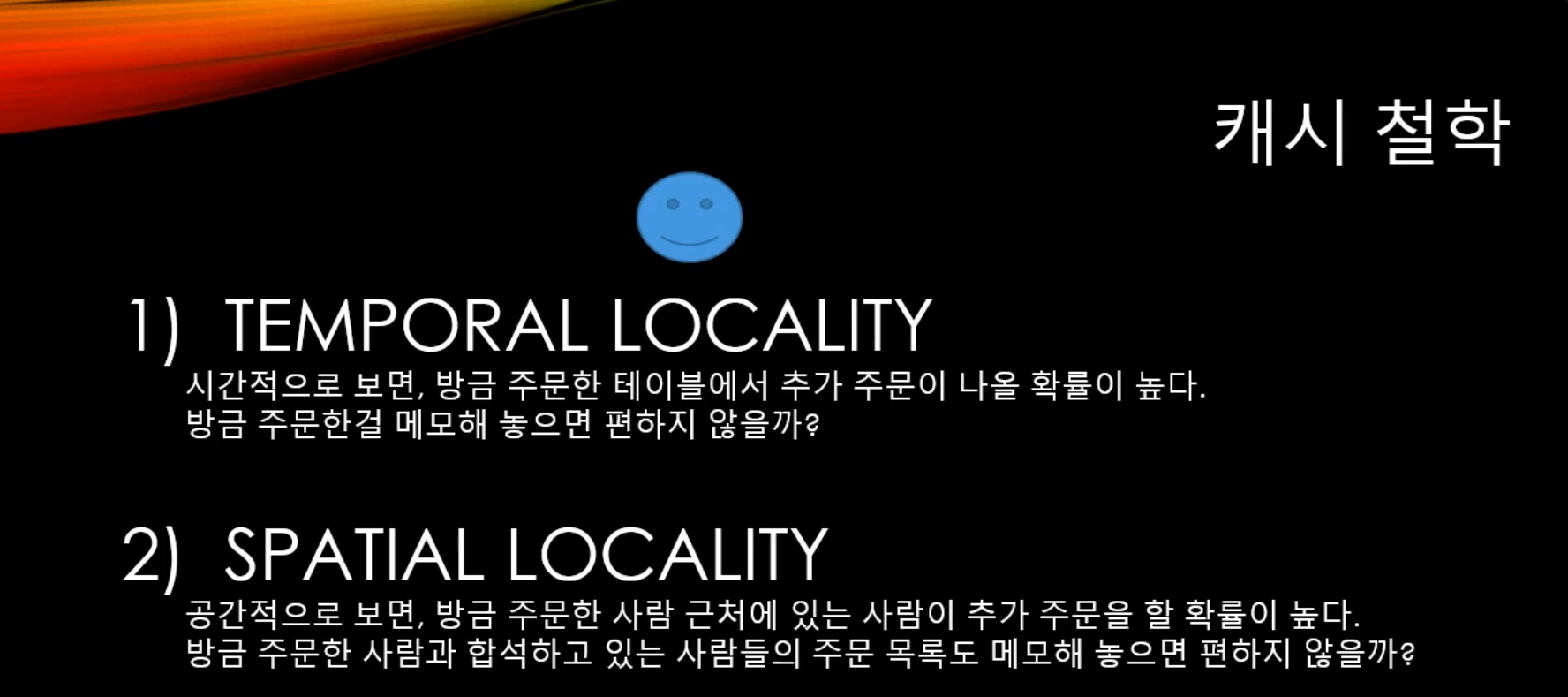
캐시를 도입하여 말그대로 임시저장소이다. 영구적으로 유지되지않고 일종의 메모장

실질적으로 CPu 의 ALU(연산) 그 근저에 임시적으로 저장할 수 있는 캐시장치가 있다. 하나만 존재하는게 아니라 피라미드 구조로 되어있다. 레지스터 L1캐시 L2 캐시



캐시를 뒤져서 없으면 그제서야 RAM에 가서 데이터를 꺼내 온다.

사용빈도가 적은 놈을 지우고 새로운놈으로 램에서 덮어씌운다



캐시 실습

#include "pch.h"

#include <iostream>

#include "CorePch.h"

#include<thread>

#include<atomic>

#include<vector>

#include<mutex>

#include<windows.h>

#include<future>

int32 buffer[10000][10000];

int main()

{

memset(buffer, 0, sizeof(buffer));

{

uint64 start = GetTickCount64();

int64 sum = 0;

for (int32 i = 0; i < 10000; i++)

for (int32 j = 0; j < 10000; j++)

sum += buffer[i][j];

uint64 end = GetTickCount64();

cout << "Elapsed Tick" << (end - start) << endl;

}

{

uint64 start = GetTickCount64();

int64 sum = 0;

for (int32 i = 0; i < 10000; i++)

for (int32 j = 0; j < 10000; j++)

sum += buffer[j][i];

uint64 end = GetTickCount64();

cout << "Elapsed Tick" << (end - start) << endl;

}

}

CPU 파이프라인

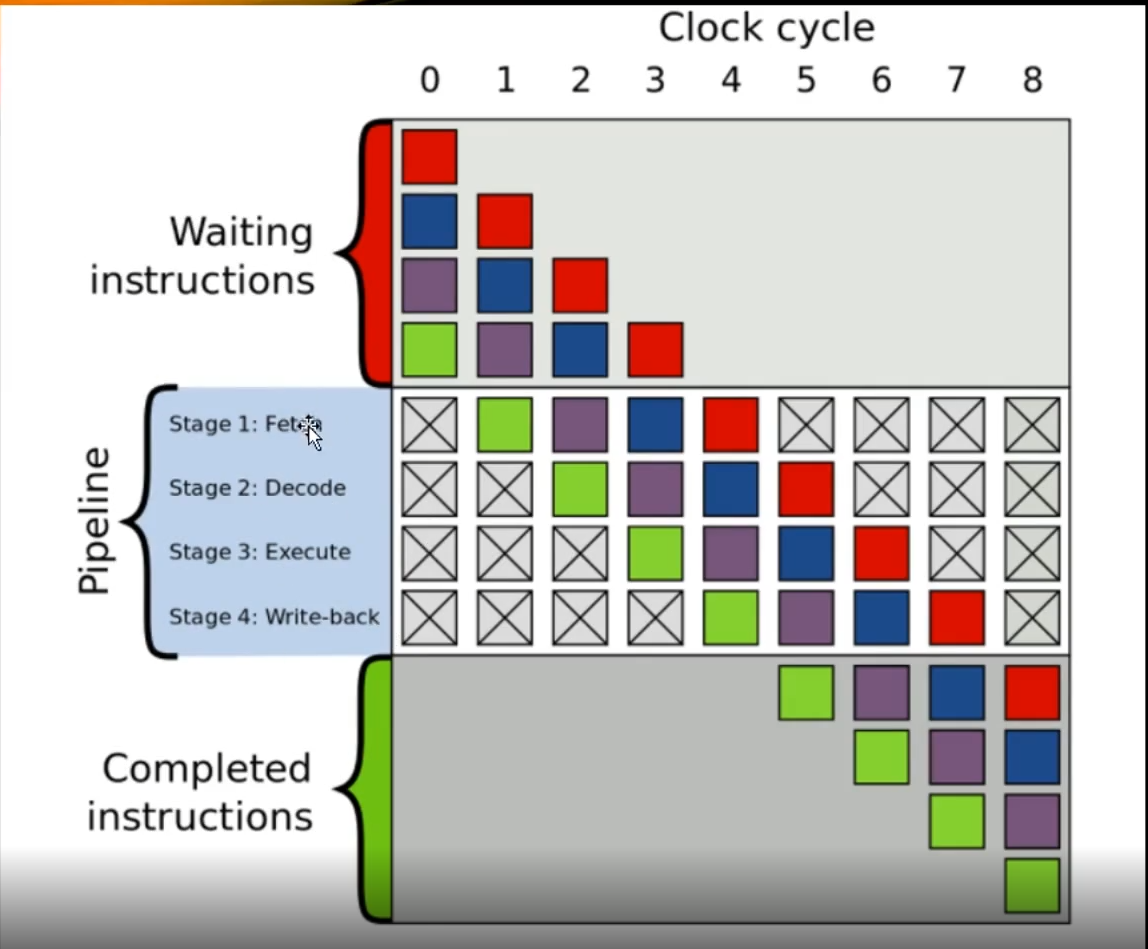


빨래를 해야한다고 가정하자 하나만 하는 것이 아니라 바구니가 여러 개 도착한상태 이걸 빨래를 해야하는데 여러가지 단계로 이루어져있다. 왼쪽 세탁기 오른쪽 건조기로가정

스탭을 밟아야하는데 세탁기를 돌리고 건조기를 돌리고 꺼내서 다리미질 그다음 옷을 정리해서 진열

그런 작업까지 하고있는 그림이다. 그런데 4단계를 걸쳐서 해야지만 최종적으로 빨래가 완성이 되는건데 그렇다고 해서 이것을 곧이곧대로 순서대로 해야하긴 하지만 첫번쨰 빨래가 완료될때까지 다른 것은 시작도 안하면 낭비가 발생하게 된다. 첫번째가 건조기에 들어가면 두번째가 세탁기 이런식으로 밀리면서 해야한다. 일반적으로 생각해보면 이런상황이 있다고 가정했을 때 노는 애가 없이 열심히 일을 할 것 이다. 가장 효율적으로 동작을 한다고 볼 수 있다.

병렬적으로 진행하고 있다고 보인다 하지만 문제는 모든 빨래의 양이 같지 않고 빨래마다 걸리는 시간이 각각 다른상황이 있다고 가정하면 곧이곧대로 받은 순서대로 실행하기 보다는 순서를 뒤바꿔서 시간의 효율을 챙겨야한다. 결국에 우리 입장에서는 모든 빨래를 해야하는 입장이지 순서가 없다면은 굳이 받은 순서대로 해야한다는 이유는 없기 때문이다.

CPU에서도 똑 같은 문제가 발생한다. 어떤 명령어를 실행할 때 여러 개의 단계을 거쳐서 실행된다. 

가져오고 해석하고 실행하고 가져다주고 이런식으로 여러가지로 나누어져있는데 이를 CPU 파이프라인이라고 한다.

단일 쓰레드 기준으로 만 생각을 했고 코드의 변화가 없기 때문에 지금까지 눈치를 채지 못했따는 이야기

멀티쓰레드에서는 뒤바꿔주는 것을 따라서 나비효과가 발생한다.

#include "pch.h"

#include <iostream>

#include "CorePch.h"

#include<thread>

#include<atomic>

#include<vector>

#include<mutex>

#include<windows.h>

#include<future>

//가시성 , 코드 재배치

int32 x = 0;

int32 y = 0;

int32 r1 = 0;

int32 r2= 0;

volatile bool ready;

void Thread\_t1()

{

while (!ready)

;

y = 1; //Store y

r1 = x; // Load x

}

void Thread\_t2()

{

while (!ready)

;

x = 1; // Store x

r2 = y; // Load y

}

int main()

{

int32 count = 0;

while (true)

{

ready = false;

count++;

x = y = r1 = r2 = 0;

thread t1(Thread\_t1);

thread t2(Thread\_t2);

ready = true;

t1.join();

t2.join();

if (r1 == 0 && r2 == 0)

break;

}

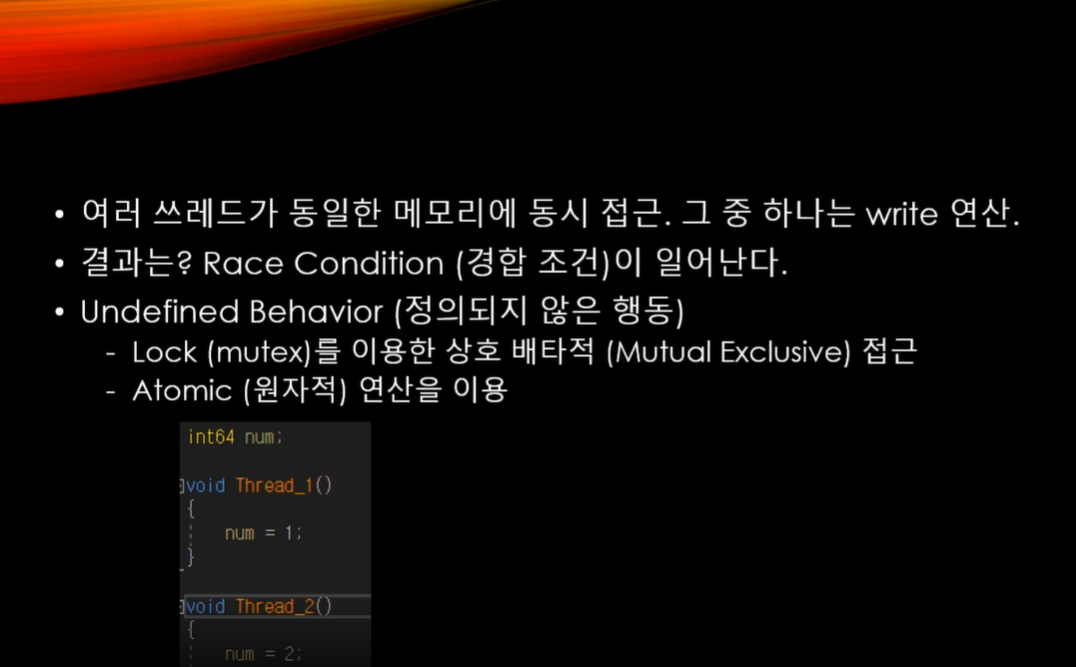
cout << count << " 번 만에 빠져나옴 " << endl;

}

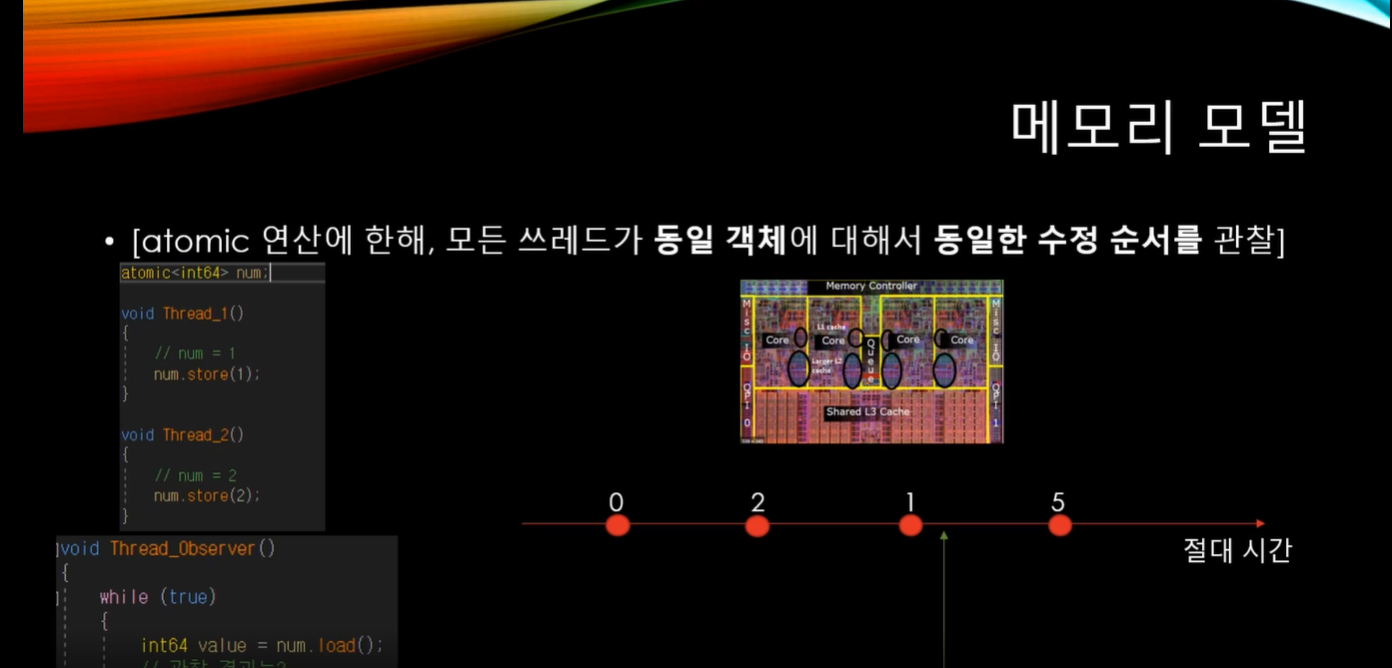
메모리 모델

프로그램 세계는 우리가 보이지 않는 곳에서 CPU최적화 작업을 위해 여러가지 작업을 한다.

싱글쓰레드 환경에선 아무런 문제가 되지 않았고 보이지도 않았다.



내용을 고치는 순간 문제가 발생



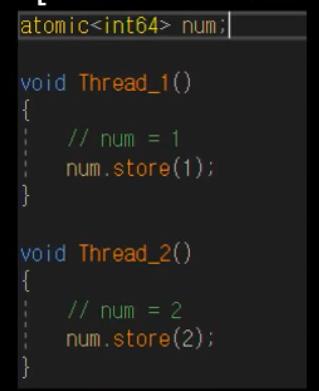
여기서 얘기하고 있는 동일 객체 동일한 수정순서가 중요

당장 현재 어떤 객체가 있는지 보장할 순 없지만 동일한 순서를 보장할 수 있다

Atomic

시간의 흐름을 거스를 순 없다.

즉 관찰한 이후 그 값을 과거의 값을 관찰할 수 없다?

여기서 얘기하는 원자적 연산은 더 이상 쪼개질 수 없는 연산을 얘기하는 것 CPU가 한번에 처리할 수 있는 것을 원자적 연산이라고 한다. 이렇게 한번에 일어나는 일들은 원자적 연산이라고 할 수 있다.

#include "pch.h"

#include <iostream>

#include "CorePch.h"

#include<thread>

#include<atomic>

#include<vector>

#include<mutex>

#include<windows.h>

#include<future>

atomic<bool> flag = false;

int main()

{

//flag = true;

flag.store(true,memory\_order\_seq\_cst);

//bool val = flag;

bool val = flag.load(memory\_order\_seq\_cst);

//이전 flag 값을 prev에 넣고 flag 값을 수정

{/\*

bool prev = flag;

flag = true;\*/

bool prev = flag.exchange(true);

}

//CAS (Compare-And-Swap) 조건부 수정

{

bool expected = false;

bool desired = true;

flag.compare\_exchange\_strong(expected,desired);

}

}

#include "pch.h"

#include <iostream>

#include "CorePch.h"

#include<thread>

#include<atomic>

#include<vector>

#include<mutex>

#include<windows.h>

#include<future>

atomic<bool> ready;

int32 value;

void Producer()

{

value = 10;

ready.store(true, memory\_order::memory\_order\_seq\_cst);

}

void Consumer()

{

while (ready.load(memory\_order::memory\_order\_seq\_cst) == false)

;

cout << value << endl;

}

int main()

{

ready = false;

value = 0;

thread t1(Producer);

thread t2(Consumer);

t1.join();

t2.join();

// Memory Model ( 정책 )

// 1) Sequentially Consistent ( Seq\_cst)

// 2) Acquire-Release(consume,acquire,release,acq\_rel)

// 3) Relaxed(relaxed)

// 1) seq\_cast (가장 엄격 = 컴파일러 최적화 여지 적음 = 직관적 )

// 가시성 문제 바로 해결 ! 코드 재배치 바로 해결 !

//

// 2) acquire-release

// 딱 중간 !

// release 명령 이전의 메모리 명령들이, 해당 명령 이후로 재배치 되는 것을 금지

// 그리고 acquire로 같은 변수를 읽은 쓰레드가 있다면

// release 이전의 명령들이 -> acquire하는 순간에 관찰 가능 (가시성 보장)

//

// 3) relaxed (자유롭다 = 컴파일러 최적화 여지 많음 = 직관적이지 않음 )

// 너무나도 자유롭다 !

// 코드 재배치도 멋대로 가능! 가시성 해결 NO!

// 가장 기본 조건 ( 동일 객체에 대한 동일 관전 순서만 보장 )

// 거의 사용하지 않음

// 인텔, AMD 의 경우 애당초 순차적 일관성을 보장을 해서

// seq\_cst를 써도 별다른 부하가 없음

}

결론 다양한 메모리 모델이 있고 우리는 첫번째 모델인 엄격한 seq\_cst로 두고 호출을 해줄 것이다.

메모리 모델, 메모리 정책에대해 배워봤다. 나중에 면접용?

Thread Local Storage

TLS는 굉장히 단순한 파트

쉽고 어렵고를 떠나서 나중에 굉장히 유용하게 사용할 것

TLS 쓰레드마다 가지고 있는 독립적으로 별도의 공간을 가지고 있는 것



#include "pch.h"

#include <iostream>

#include "CorePch.h"

#include<thread>

#include<atomic>

#include<vector>

#include<mutex>

#include<windows.h>

#include<future>

// Thread local storage

thread\_local int32 LThreadID = 0;

void ThreadMain(int32 threadID) {

LThreadID = threadID;

while (true)

{

cout << " Hi ! I am Thread " << LThreadID << endl;

this\_thread::sleep\_for(1s);

}

}

int main()

{

vector<thread> threads;

for (int32 i = 0; i < 10; ++i)

{

int32 threadID = i + 1;

threads.push\_back(thread(ThreadMain, threadID));

}

for (thread& t : threads)

t.join();

}

지금까지 배운 내용들을 복습하는 내용으로 LOCK 기반의 스택 자료구조를 만들어 볼 것이다.

한쪽 쓰레드에선 계속 데이터를 밀어넣음 한쪽에선 계속 꺼내옴

void Push() {

while (true) {

int32 value = uie(dre);

q.Push(value);

this\_thread::sleep\_for(10ms);

}

}

void Pop()

{

while (true) {

int32 data = 0;

if (q.TryPop(OUT data))

cout << data << endl;

}

}

int main()

{

thread t1(Push);

thread t2(Pop);

thread t3(Pop);

t1.join();

t2.join();

t3.join();

}

멀티쓰레드환경에선 empty부분을 나누지 않고 pop을 할 때 동시에 해준다.

ConcurrentStack.h

#pragma once

#include<mutex>

template<class T>

class LockStack

{

public:

LockStack() {}

LockStack(const LockStack&) = delete;

LockStack& operator=(const LockStack&) = delete;

void Push(T value)

{

lock\_guard<mutex> lock(\_mutex);

\_stack.push(std::move(value));

\_condVar.notify\_one();

}

bool TryPop(T& value)

{

lock\_guard<mutex> lock(\_mutex);

if (\_stack.empty())

return false;

// empty => top -> pop

value = std::move(\_stack.top());

\_stack.pop();

return true;

}

void WaitPop(T& value)

{

unique\_lock<mutex> lock(\_mutex);

\_condVar.wait(lock, [this] {return \_stack.empty() == false; });

value = std::move(\_stack.top());

\_stack.pop();

}

//bool Empty()

//{

// lock\_guard<mutex> lock(\_mutex);

// return \_stack.empty();

//}

private:

stack<T> \_stack;

mutex \_mutex;

condition\_variable \_condVar;

};

ConcurrentQueue.h

#pragma once

#include<mutex>

template<class T>

class LockQueue

{

public:

LockQueue() {}

LockQueue(const LockQueue&) = delete;

LockQueue& operator=(const LockQueue&) = delete;

void Push(T value)

{

lock\_guard<mutex> lock(\_mutex);

\_queue.push(std::move(value));

\_condVar.notify\_one();

}

bool TryPop(T& value)

{

lock\_guard<mutex> lock(\_mutex);

if (\_queue.empty())

return false;

value = std::move(\_queue.front());

\_queue.pop();

return true;

}

void WaitPop(T& value)

{

unique\_lock<mutex> lock(\_mutex);

\_condVar.wait(lock, [this] {return \_queue.empty() == false; });

value = std::move(\_queue.front());

\_queue.pop();

}

private:

queue<T> \_queue;

mutex \_mutex;

condition\_variable \_condVar;

};

기존의 stack과 queue를 맵핑해서 사용

Lock-Free Stack #1

락 프리 스택 락을 사용하지 않는 스택

호불호가 갈린다. 복잡한 반면 성능테스트는 생각보다 효율이 좋지않다 멋있는 방법이지만 실질적으로 많은 혼동을 유발한다. lock없이 돌아가서 성능이 빠를거 같지만 아님 별차이가 없음

락프리 스택같은경우에 기존의 컨테이너를 맵핑하는 게 아닌 처음부터 다시 만들어야 한다

일단은 Node 기반으로 바꾸어 주고 여기에 데이터가 있고 노드는 다음 노드를 가리킨다.

값을 받고 저장하는 생성자

프리스택에선 atomic<Node\*> \_head 를 가지고 있다 .

결국에는 리스트와 마찬가지로 // [][][][][] 이런식으로 데이터가 존재 //[head]가 가리키고 있는 부분으로 밀어넣으면 됨

또한 이부분을 찍어서뽑아내고 날리는 식으로 작업을 해주면 된다.

Void Push(const T& value)

여기에서 우리는 값을 넣어 준다는 것은 새로운 노드를 만들어 넣고 연결을 해주어야 한다.

이를 3스탭으로 나눌 수 있다.

1. 새 노드를 만든다 .
2. 새 노드의 next = head
3. Head = 새 노드

락 프리 라고 하더라도 경합없이 일감을 해결할 순 없다. 말그대로 락만 사용하지 않을 뿐

1. Head 읽기
2. Head -> next 읽기
3. Head = head->next
4. Data 추출해서 반환
5. 추출한 노드를 삭제

첫단계부터 난관에 부딪힌다 헤드를 읽고 있는데 여러명이 동시에 사용하고 있는 것이기 때문이다. 결론부터 말하면 compare\_exchange\_weak를 사용해야한다

내가 접근하고있는데 다른애가 접근하고 있으면 어쩌지를 끊임없이 생각해야한다.

Bool TryPop(T& value)

{

}

이미 삭제된 메모리에 접근을 하는 상황이 발생할 수 있다

누군가가 똑 같은 메모리에 접근을 할 때 삭제를 해버리면 문제가 발생하기 때문에 여러가지 알고리즘 중 선택을 하나를 해야한다. 다양한방법이 있다

lockfreestack

template<class T>

class LockFreeStack

{

// 랩핑 ㄴㄴ 첨부터 노드기반으로 만들어야함

struct Node

{

Node(const T& value) : data(value) {}

T data;

Node\* next;

};

public:

// 새로운 노드를 만들어 준다음 연결해준다.

// 1 ) 새 노드를 만들고

// 2 ) 새 노드의 next = head

// 3 ) head = 새 노드

// [][][][][]

// [head]

void Push(const T& value)

{

Node\* node = new Node(value);

node->next = \_head;

/\* if (\_head == node->next)

{

\_head = node;

return true;

}

else

{

node->next = \_head;

return false;

}\*/

while (\_head.compare\_exchange\_weak(node->next, node) == false) {

//node -> next = \_head;

}

// 이 사이에 새치기 당하면?

//\_head = node;

//락프리에서도 대기상황이 없는 건 아니다.

}

// 1 ) head 읽기

// 2 ) head -> next 읽기

// 3 ) head = head->next

// 4) data 추출해서 반환

// 5) 추출한 노드를 삭제

// [ ] [ ] [ ] [ ] [ ]

// [head]

bool TryPop(T& value)

{

Node\* oldHead = \_head;

//if (\_head == oldHead)

//{

// \_head = oldHead->next;

// return true;

//}

//else

//{

// oldHead = \_head;

// return false;

//}

while (oldHead && \_head.compare\_exchange\_weak(oldHead, oldHead->next) ==false)

{

//oldHead = \_head;

}

if (oldHead == nullptr) //데이터가 없는 상황

return false;

//Exception X

// 복사를 하는 이순간에 메모리 고갈로 인해 문제가 일어날수 있는 환경은 우리는 신경쓰지 않는다.

// 게임에서는 복사하는 과정에서 터지면 터지게 냅두고 버그를 잡는것이 훨씬 좋다.

value = oldHead->data;

//잠시 삭제 보류

//delete oldHead; // 여기서 문제 발생

return true;

}

private:

// [][][][][]

// [head]

atomic<Node\*> \_head;

};

다른 쓰레드가 사용할 지 도 모르는 메모리를 삭제한다는 큰 문제점이 생긴다.

스택 정책상 넣는건 여러명이 넣을수도 하지만 빼는건 한명만 뺄수 있게 해도 상관없지만 자료구조에서 항상 그대로만 로직하기엔 말이안됨

사용하지 않을때에만 삭제하면 된다. 가장 쉽게 찾을 수 있는 방법은 팝카운트를 계속 추적

첫번째 방법은 pop 을 실행중인 쓰레드의 개수를 파악

template<class T>

class LockFreeStack

{

struct Node

{

Node(const T& value) : data(value), next(nullptr) {}

T data;

Node\* next;

};

public:

void Push(const T& value)

{

Node\* node = new Node(value);

node->next = \_head;

while (\_head.compare\_exchange\_weak(node->next, node) == false) {

}

}

bool TryPop(T& value)

{

++\_popCount;

Node\* oldHead = \_head;

while (oldHead && \_head.compare\_exchange\_weak(oldHead, oldHead->next) == false)

{

}

if (oldHead == nullptr) {

--\_popCount;

return false;

}

value = oldHead->data;

return true;

}

// 1) 데이터 분리

// 2) Count 체크

// 3) 나 혼자면 체크

void TryDelete(Node\* oldHead)

{

// 나 외에 누가 있는가?

if (\_popCount == 1)

{

// 나 혼자네?

// 이왕 혼자인거, 삭제 예약된 다른 데이터들도 삭제 해보자

Node\* node = \_pendingList.exchange(nullptr);

if (--\_popCount == 0)

{

// 끼어든 애가 없음 -> 삭제 진행

// 이제와서 끼어들어도 , 어차피 데이터는 분리해둔 상태 ~!

DeleteNodes(node);

}

else if(node)

{

// 누가 끼어들었으니 다시 갖다 놓자

ChainPendingNodeList(node);

}

// 내 데이터는 삭제

delete oldHead;

}

else

{

// 누가 있네 ? 그럼 지금 삭제 하지 않고 삭제 예약만

ChainPendingNode(oldHead);

--\_popCount;

}

}

void ChainPendingNodeList(Node\* first, Node\* last)

{

last->next = \_pendingList;

while (\_pendingList.compare\_exchange\_weak(last->next, first) == false)

{

}

}

void ChainPendingNodeList(Node\* node)

{

Node\* last = node;

while (last->next)

last = last->next;

ChainPendingNodeList(node, last);

}

void ChainPendingNode(Node\* node)

{

ChainPendingNodeList(node, node);

}

static void DeleteNodes(Node\* node)

{

while (node)

{

Node\* next = node->next;

delete node;

node = next;

}

}

private:

atomic<Node\*> \_head;

atomic<uint32> \_popCount = 0; // pop을 실행중인 쓰레드 개수

atomic<Node\*> \_pendingList; // 삭제 되어야 할 노드들(첫번째 노드)

};

지난번에 팝카운트를 따라가는 방법 누군가 사용 중이면 삭제하지 못하게;

C#이랑 자바같은경우에는 GC가 있기 때문에 계속 돌면서 메모리가 해제되는걸 알아서 체크하기 때문에 신경쓸 필요가 없다. 이 얘기는 즉 생포인터를 사용하지 않고 스마트포인터를 쓰면 해결할 수 있지 않을까? 어느정도 일리가 있는 소리이다.

락프리 스택을 만들 때 팝카운트를 추적을 하는 방법을 했었는데 오늘 사용할 방법은

노드의 레퍼런스를 카운트 하여 한다.

template<class T>

class LockFreeStack

{

struct Node;

struct CountedNodePtr

{ // 포인터와 참조횟수를 아토믹하게 관리하겠다는 구조체를 직접 생성

int32 externalCount = 0;

Node\* ptr = nullptr;

};

struct Node

{

Node(const T& value) : data(make\_shared<T>(value)) {}

shared\_ptr<T> data;

atomic<int32> internalCount = 0;

CountedNodePtr next;

};

public:

void Push(const T& value)

{

CountedNodePtr node;

node.ptr = new Node(value);

node.externalCount = 1;

// [ ! ]

node.ptr->next = \_head;

while (\_head.compare\_exchange\_weak(node.ptr->next, node) == false)

{

}

}

shared\_ptr<T> TryPop()

{

CountedNodePtr oldHead = \_head;

while (true)

{

// 참조권 획득 (externalCount를 현 시점 기준 +1 한 애가 이김)

IncreaseHeadCount(oldHead);

// 최소한 externalCount >= 2일테니 삭제 X (안전하게 접근할 수 있는)

Node\* ptr = oldHead.ptr;

//데이터 없음

if (ptr == nullptr)

return shared\_ptr<T>();

// 소유권 획득 (ptr->next로 head를 바꿔치기 한 애가 이김)

if (\_head.compare\_exchange\_strong(oldHead, ptr->next))

{

shared\_ptr<T>res;

res.swap(ptr->data);

// external : 1 -> 2(+1) -> 4(나+1 남+2)

// internal : 1 -> 0

// 나 말고 또 누가 있는가?

const int32 countIncrease = oldHead.externalCount - 2;

if (ptr->internalCount.fetch\_add(countIncrease) == -countIncrease)

delete ptr;

return res;

}

else if(ptr->internalCount.fetch\_sub(1) == 1)

{

// 참조권은 얻었으나, 소유권은 실패 -> 뒷수습은 내가 한다.

delete ptr;

}

}

}

private:

void IncreaseHeadCount(CountedNodePtr& oldCounter)

{

while (true)

{

CountedNodePtr newCounter = oldCounter;

newCounter.externalCount++;

if (\_head.compare\_exchange\_strong(oldCounter, newCounter))

{

oldCounter.externalCount = newCounter.externalCount;

break;

}

}

}

private:

atomic<CountedNodePtr> \_head;

};