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

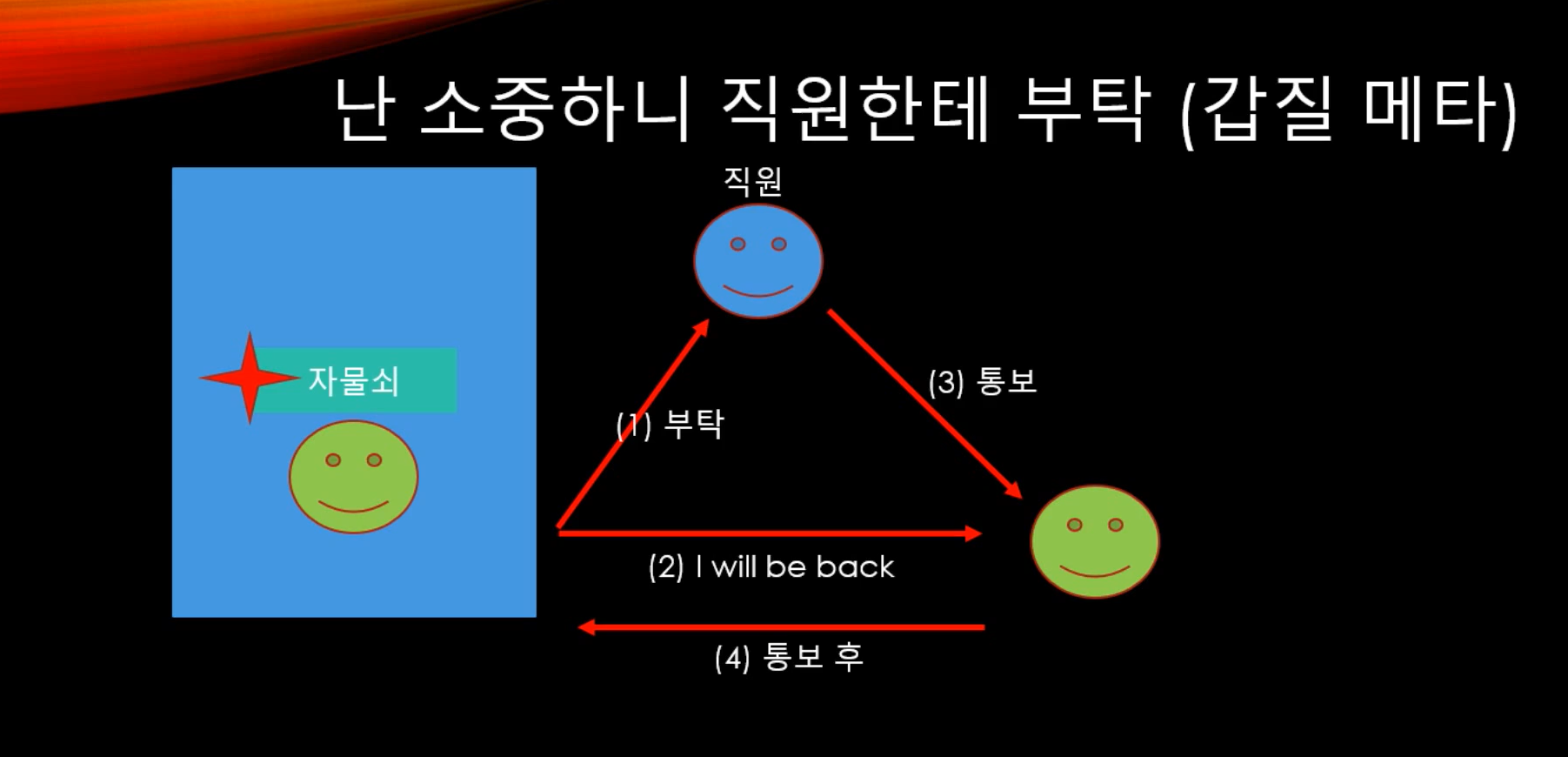
그다음 방법은

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();

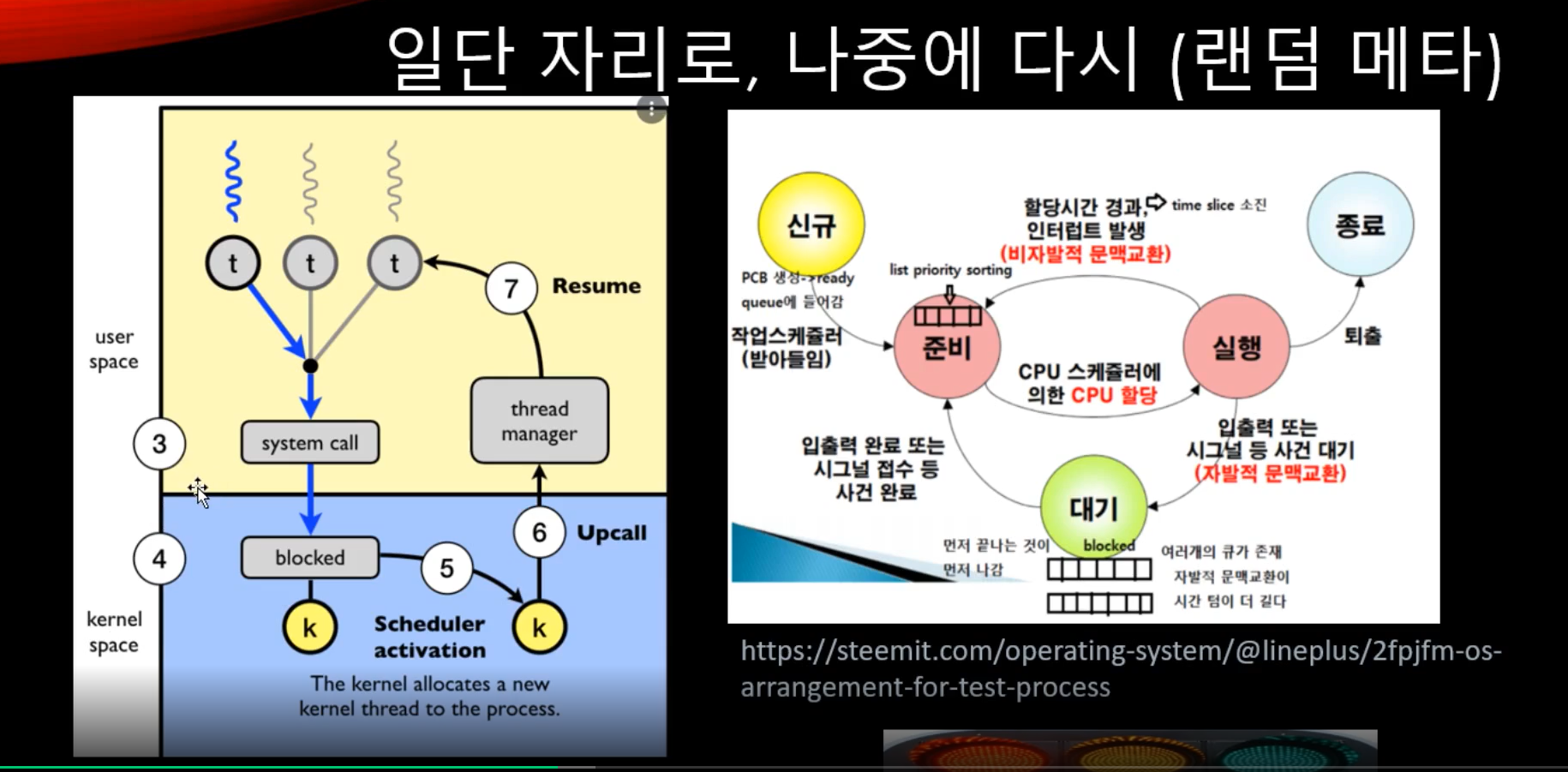
cout << sum << endl;

}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.

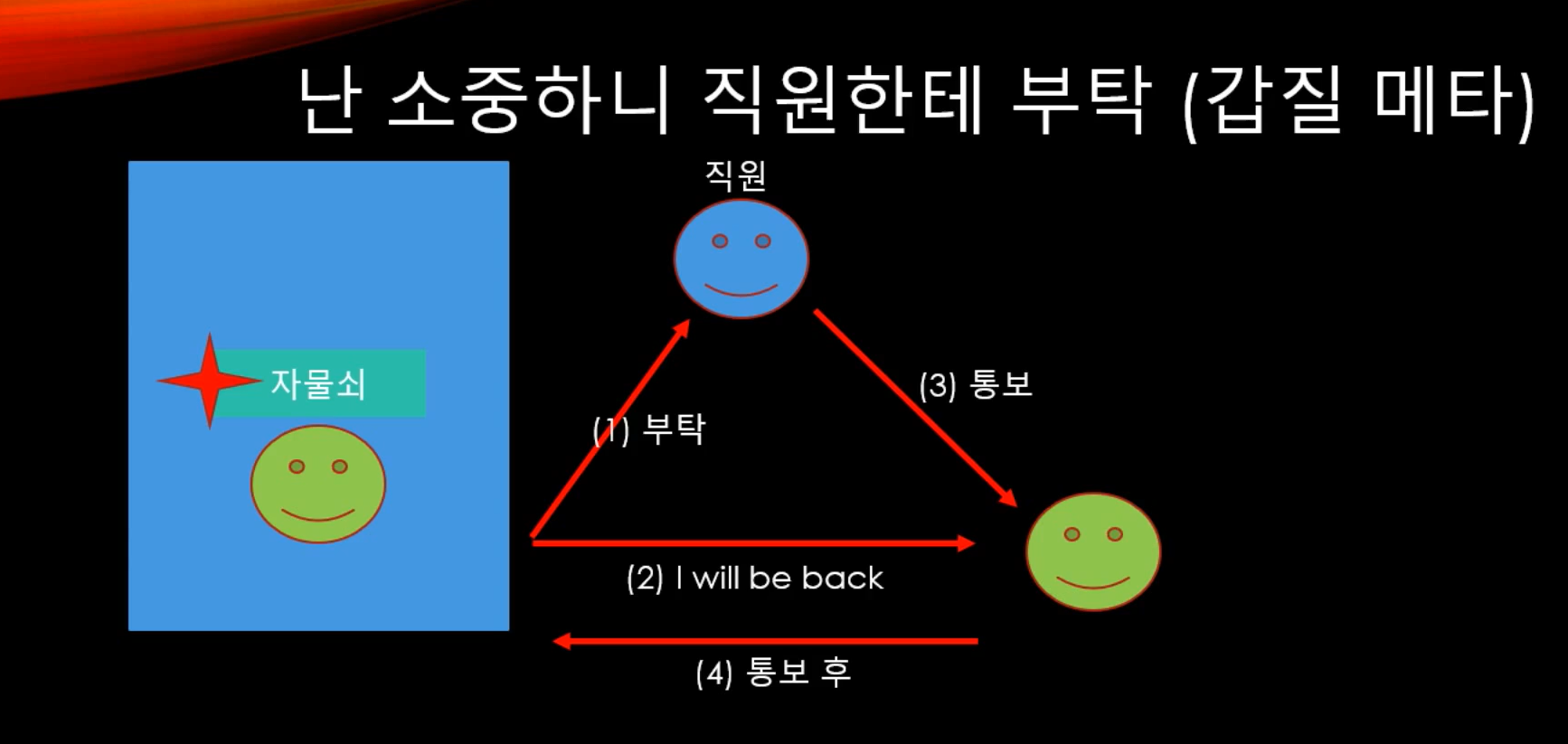


this\_thread::sleep\_for(0ms);

//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);

}