**<6.26>**

**선점형 - 비선점형 OS**

**커널스레드 – 선점형**

유저스레드 – 비선점형

**우리는 커널스레드(선점형)을 사용한다.**

어떤OS든 유저스레드가 내장되어있음. ( 최근에 표준으로 추가되었다. )

(유저모드 동기화/ 커널모드 동기화)는 다른개념이다.

커널스레드는 커널모드가 들어가기 때문에 유저모드단계에서 돌수있는 유저스레드보다 성능이 떨어진다?

유저스레드는 애초에 병렬로 돌수가 없음.

(단순히 동시에 두개의 코드가 도는것처럼 흉내를 내는 것뿐)

**따라서 이는 비교대상이 아니다.**

**Thread**

**쓰레드란 일반적으로 자신의 Call stack과 CPU상태(state)를 갖는 소프트웨어 수행경로를 의미한다.**

**쓰레드는 프로세스 문맥(context)상에서 동작한다.**

프로세스의 문맥에는 코드와 데이터가 존재하고, 스레드가 수행되는 주소공간이 정의되어있다.

**윈도우 같은 경우는 스레드가 코드의 수행단위가 되기때문에 스레드가 없는 코드의 단위는 존재할 수 없다. 지금의 모든OS는 이렇게 발전이 되어왔음.**

(과거 유닉스,리눅스는 프로세스가 수행의 단위였기때문에 스케쥴링도 프로세스 단위.)

스케쥴링에 대해 구글링할때, 자바기준으로 작성된 자료가 많으므로 주의해서 볼 것. 크게 다르진 않지만 약간의 용어와 개념이 다르다.

**지금의 윈도우OS는 모두 쓰레드가 핵심**이므로, 스레드를 한개가진 프로세스보다 스레드를 3개가진 프로세스가 더 많이 될 것이다. (프로세스간 구분이 아닌 스레드간 스케쥴링)

(스레드마다 1MB의 스택이 확보된다.)



처음에 메모리에 대해 배울때 위와같은 그림의 형태를 많이 접한다.

이는 개념상 정리한것이며, 스택은 여기저기서 만들어지고 힙도 띄엄띄엄 생기는 등의형태를 취한다.

**코드와 데이터는 하나로 존재하며 프로세스 기준으로 가상메모리 테이블이 존재하고, 가상메모리 테이블 안에는 코드와 데이터가 전역적으로 하나가 존재할 것.**

**기본 힙프로세스가 있고 우리가 힙을 만들면 얼마든지 만들어진다.**

**스택역시 쓰레드를 만들때마다 계속해서 생겨나고, 스택은 누군가 접근하면 안되는 영역이므로 1MB예약을 잡아놓고 들어가게된다.**

**CPU의 상태는 결국 모든 코드를 어셈블리로 받는것이며, CPU가 코드를 처리한다는 것은 레지스터 기준이다. 스택,실행위치,가상메모리테이블 모든것이 레지스터에 들어있는 정보이며 CPU는 이 레지스터를 기준으로 모든 코드를 실행시킨다.**

**결국 CPU에 들어있는 레지스터 상태가 스레드의 상태와 같다는 얘기가 된다.**

특정 쓰레드가 컨텍스트 스위칭이되어 코어를 잡고 돌기시작했을때, 얘가 돌고있는 모든 레지스터를 CPU에 반영하는 것이다.

**이 반영되는 정보는 코드영역, 어디까지 돌았는지, 스택의위치는 어딘지, 가상함수 테이블은 어디인지, 어디까지 연산했는 지 등 필요한 모든 정보가 레지스터에 그대로 반영된다.**

이때 해당 쓰레드를 중지하고 B쓰레드를 실행시킬때, 현재 사용하는 모든 CPU레지스터를 쓰레드컨텍스트 블록에 백업하고 메모리에 저장한 후에 B쓰레드를 돌리게 된다.

**기본적으로 컨텍스트 스위칭이란 작업은 CPU의 레지스터 상태를 온전히 백업하고 복구시키는 작업이다.**

**Multi-Thread**

**멀티스레드 프로그램은 일반적으로 [수행시간, 사용자 응답성, 프로그램 구조] 중 적어도 하나 이상은 싱글스레드 프로그램보다 좋아야 한다.**

위 항목에 해당사항이 없다면 멀티스레딩을 할 이유가 없음.

**우리는 거의 대부분이 더 빠른 수행시간을 위해 멀티스레드 프로그래밍을 한다.**

(기본적으로 멀티쓰레딩은 기본적으로 동기화이슈가 많이 터지기 때문에 쉬운 작업이 아니다.)

**프로그램 구조 간결**

대표적으로 게임클라이언트의 WinMain이 있다.

WinMain에 있는 메인스레드에서 윈도우 메시지를 처리하고,(PeekMassage활용) 메시지가 없다면 우리의 로직을 돌리는 형태를 취했다.

이구조를 좀더 적절하게 고치기위해 **멀티 쓰레드를 활용하여 메인쓰레드는 윈도우 메시지를 Peek이아닌 GetMassage로 처리하고, 게임로직은 다로 쓰레드를 만들어서 돌리는 형태.**

또는 모니터링을 위한 출력(printf)같은 경우도 출력을 담당하는 쓰레드를 만드는 구조로 갈수도 있을것이다.

이러한 경우 쓰레드를 하나 더 만든것이기 때문에 당연히 성능은 떨어질 것이나, 구조를 간결하게 하기 위해서라면 시도해 봄직하다.

**사용자 응답성**

**쓰레드로 비동기적인 처리를 유도시켜 Block되는 상태를 막아보자는 입장의 접근이다.**

Connect를 Block Socket으로 호출할 때, block되는 경우이는 서버가 죽은 느낌이다.

**이러한 상황에서 connect를 다른 쓰레드로 뺀다면 실질적인 작업은 해당 쓰레드안에서 처리하고,결과에 대한 피드백을 주도록 설계하는 것이다.**

이경우 사용자입장에서는 피드백이 있기때문에 이상함을 느끼지않는다.

**하지만 이 역시 코드가 더 복잡해지고 수행시간이 떨어질수있는 단점이 존재한다.**

**수행시간 감소(병렬화의 증가)**

멀티쓰레드 구조는 병렬화구조의 증가로 수행시간을 감소시켜 성능향상을 노리는것이 일반적이다.

**작업 A,B,C는 결국은 서로 전혀 연관성이 없다는 전제이다.**

**결국 같은 메모리,같은 자료구조에 접근하는 작업이라면 동기화를 할 수밖에 없고 이경우 병렬화가 제대로 되지않으므로 의미가 없다.**

멀티쓰레드를 설계하는 일반적인 경우,

**서버 분산처리를 할 때 스케일 아웃같은 느낌으로 똑같은 일을 하는 쓰레드를 여러 개를 만들어 병렬처리로 성능을 높일 수 있을 것이고,**

**또한 기능자체를 분리시키는 방법이 있을 것이다.**

**(네트워크 IO만을 전담하는 쓰레드, AI를 전담하는 쓰레드, 실제 업데이트 로직 쓰레드 등..)**

우리가 진행한 TCP Fight같은 경우도 어떤 유저가 움직일때마다 Sector에 접근을 계속해서 하고있다. 다른유저에 대한 검색이 계속되며, 이런작업들을 병렬로 여러쓰레드가 동시에 들어간다고 한다면 결국은 동기화를 걸어야만하므로 멀티쓰레드 자체가 큰 의미가 없어진다.

**MMORPG같은 경우 유저간의 상호작용이 매우 많아 서로간에 심하게 얽혀있기 때문에 메인쓰레드를 멀티쓰레드로 바꾸기 매우 난해하다.**

**때문에 아직까지도 메인업데이트 로직을 싱글스레드로 가는곳이 많다.**

(물론 잘설계된 멀티쓰레드 서버도 존재한다. 이후 언급)

**병렬스레드의 극대화**

**해당 쓰레드가 block이 걸려야만 하는 상황일 경우. (대표적으로 I/O가 있음)**

**채팅의 경우**

채팅프로그램에서, 키보드 입력을 받으면서 수신을 하기위해 멀티쓰레드로 키보드입력은 입력대로 받고 recv는 recv대로 한다면 따로 처리가 가능해진다.

**파일 입/출력의 경우**

파일 입/출력 같은 경우도, 파일읽기가 너무 느리기 때문에 클라이언트 데이터 로딩을 모두 멀티쓰레드로 간다. 파일이 매우 클 경우 로딩이 오래걸리기 때문에 로딩하는 쓰레드를 따로 만들고, 메인에서는 요청하는 형태.

**음악재생 스트리밍의 경우**

(서버개발자의 경우 살면서 음악을 건드릴 일은 없을것이나..)

음악이나 사운드는 링버퍼를 만들고, 사운드 소스코드, 드라이버단에서 이 링버퍼를 순회하면서 재생하는 개념이므로, 링버퍼를 가리키는 커서의 다음위치에 미리미리 데이터를 준비시켜야 한다. (웨이브는 보통 4~50MB가 나오고, 버퍼에 한계가 있으므로 한번에 로딩을 할 수가 없음)

그래서 이걸 한방에 버퍼에 올릴수가없기떄문에 링버퍼에서 뱅글뱅글 재생시키고 있는것이다.

이러한 형태일 경우 매프레임 이것을 작업해야하므로, 해당 작업을 다른 쓰레드로 빼는 것.

**이는 성능의 향상을 노려볼수도있겠지만 구조적인 간결함을 노린 상황이다.**

**안정성 향상**

안정성은 약간 애매한 부분이다.

**우리가 돌리던 A쓰레드가 멈췄을 때 다른쓰레드는 멈추지 않게끔 하자는 것이 목적이나, 이는 의미없는 행위이다.**

쓰레드가 멈춘다는것은 무한루프에 빠지거나 데드락에 걸리는 경우를 뜻하는데, 이러한 경우는 애초부터 나오면 안되는 경우.

(DeadLock : 교착상태를 뜻하며, 두개이상의작업이 서로 상대방의 작업이 끝나기만을 기다리고있기 때문에 결과적으로 아무것도 완료되지 못하는 상태를 가리킨다.)

**만약에 오류가 나서 죽는경우 프로세스 자체가 죽기때문에 안전성을 확보하고 싶다면 프로세스 단위로 가야한다.**

**호모지니어스방식에서의 스레드 안정성**

호모지니어스에서, 스레드에서 문제가 생기는 경우 덤프빼고 해당 스레드를 중지시키는 경우,

**여러 스레드가 같은 범주의 일을 하지만(ex MainThread) 모두가 똑같은 일을 하는것은 아니기 때문에 해당 문제를 만나는 스레드만 죽게 될것이다.**

(만약 해당스레드가 특정일만을 한다면 모두 죽을것이므로 의미없음)

**StateFull도 이와같은 방식으로, 특정기능에 문제가있다면 해당기능만 문제가 발생하고 다른 기능들은 잘 돌아갈것이다.**

이 같은 경우 안정성이란말이 얼추 맞는다.

**물리서버 Active – Standby**

서버가 죽었을때를 대비해서 stand\_by형태로 서버를 따로 하나 더두는 것이 논의거리로 나온적이 있었다.

**가용성을 100%유지하는것이 모든 서버 서비스 운영자들의 목표이기 때문에, 실제 물리서버에서는 모든 서버를 운영할때 Active-Standby 형태를 취해서, Active서버가 죽으면 Standby서버가 바로 대처할수 있게끔 한다.**(물리적으로 두개를 쌍으로 물고감)

(고가용성 : HA(Hight Avaliability 이란 서버와 네트워크 프로그램등의 정보 시스템이 상당히 오랜기간 동안 지속적으로 정상 운영이 가능한 성질을 말한다. 가용성이 높다는 의미로, 고장나지않음을 의미한다.)

여기서의 목적은 서버가 **물리적으로 죽을 때**를 얘기하고 있으므로(ex 블루스크린, 디스크가 나간경우, 파워가 갑자기 꺼지는 경우 등) **미리 똑같이 셋팅해둔 서버를 백업용으로 놔두고 Active문제가 터지면 바로 전환시킬 수 있게끔 하자는 얘기이다.**

**이를 프로세스 단계로 적용시켜 데이터는 실시간으로 동기화시키고, Active프로세스가 죽을경우 클라쪽에서 접속한 유저들을 모두 connect를 새로해서 그대로 이어가게끔 하는것이 목적.**

**프로세스에서의 Active – Standby**

**이는 프로세스의 크래쉬, 프로세스의 오류를 죽어나가는것을 대비한느것으로 물리서버와는 다른 얘기이다.**

(HA를위해서는 원래 성능을 포기하므로, 이때 떨어지는 성능은 감안해서라도)

**Active-Standby서버는 완전히 같은 코드로 돌고있기 때문에 프로세스가 죽으면 준비된 다른프로세스도 죽을것이므로, 전혀 의미가없다.**

(그렇다고 같은 서버를 다른코드로 만든다는것은 더 말이안된다.)

**결론적으로, 애초에 서버자체를 물리적으로 두대를 둔다는것은 하드웨어를 얘기하는것이고 프로세스 다운을 대비하여 프로세스를 하나 더만들어주는것은 아무런 의미가 없다.**

퍼블리셔쪽에서 서버가 죽으면 다시 실행되게 해달라는 요구를 하기는 한다. 이때는 클라쪽에서 어느정도 작업을 거쳐 바로 이어서 connect를 하므로, 사용자입장에서는 잠깐의 렉이 있었던것을 느끼고 넘어간다.

(하지만 이것도 좋은 선택은 아니다.)

**Dump**

**덤프는 예외가 생겼을때 레지스터, 메모리 등을 복사해서 따로 빼는 개념이다.**

**현재는 VS가 기능이 매우 좋기때문에 pdb파일만 존재하면 실제 디버그모드에서 break pointer를 건것처럼 확인이 가능하다.**

**(단, 스택이 깨지는 경우는 레지스터와 디스어셈블러에 의지하여 판단해야 할것)**

**Windbg**

MS에서 제공하는 디버깅 프로그램이다.

텍스트 명령어로 하나씩 보내야하기 때문에 사용하기 불편하다.

**pdb가 새로만들어져 맵핑이 안된경우(버전이 변경되어 맵핑이 불가능함)와 같은 예외상황에서 Windbg를 사용하여 억지로 메모리에 올려 텍스트로 볼때 종종 사용한다.**

현재에서는 Windbg를 사용하는 일은 잘없을 것.

(단, 스택이 깨지는 경우는 레지스터와 디스어셈블러에 의지하여 판단해야 할것)

**쓰레드를 사용하면 안되는 경우**

**1. 두개의 쓰레드가 명확하게 독립적이지 않은 경우**

**2. 쓰레드를 다루는데 발생한 부하가 쓰레드에 수행되는 실제 작업보다 더 클 경우**

(구조가 간결해지는 경우라면 시도해 볼만 함)

쓰레드 스케쥴링 시간보다 할일이 더 없다는것은 거의 할일이 없는 쓰레드이다.

**작업쓰레드를 몇개로 만드냐에 따라서 테스트 수치가 다르게 나오는데, 이를 가지고 해당 서버/하드웨어 환경에서 적당한 것을 찾아내야 할것이다.**

(이 수치를 기준으로 +-를 하다보면 어느순간에 성능이 튈때가 있다. 이때가 딱 적절한것)

이후 쓰레드 스케쥴링에 대한 방식과 개념을 다 이해하고 난다면 감이 대충 올것이다.

**우리는 싱글스레드로 가되 너무 부담되는경우 쓰레드로 빼는 형태를 취할 것이다.**

**최대한 쓰레드를 적게쓰는 방향을 고민해야하며, 어떻게 나눌것인지부터 고민하지않는다.**

(또한, 공유자원을 빈번하게 접근하는 구조라면 병렬로 분산하는것 자체를 고민을 해봐야 하는 상황이다.)

**쓰레드 동기화 이슈**

단순하게 변수하나를 ++하는것도 동기화를 장담하지 못한다.

언어차원에서는 한줄이겠지만 어셈블리입장에서는 또 다르기 때문에, 단순하게 변수하나를 ++하는것도 동기화를 장담하지 못한다.

((

어셈블리 명령어는 최소단위 오퍼레이션하나로 1:1로 대응되지는 않는다.

**어셈블리 내부에 repeater(반복)일 경우, 도중에 OS가 개입하면 반복을 중단시킬수 있다.**

**mov, cmp, lea 와 같은 일반적인명령어들은 CPU가 처리해야되는 최소단위.**

이명령어 하나가 수행되는 동안에 컨텍스트 스위칭이 유도될수는없다.

어쨌든 한번들어왔으면 마무리 되어야 한다.

))

때문에 단순한 덧셈뺄셈 과정도 동기화가 필요하다.

**메모리/자료구조에 두개의 쓰레드가 동시에 들어가는 상황이면 무조건 동기화를 걸어서 하나의 쓰레드가 독점적으로 접근할수있게끔 설계해야한다.**

**락이 걸린 임계영역에 쓰레드가 접근하면 내 쓰레드를 Block시키고, 해당동기화 객체가 signal올올 때지 기다린다**. 이후 signal이 오게되면 다시 깨어나서(스케쥴링 당해서) 컨텍스트 스위칭에 들어가기 때문에.

**결국 동기화를 걸어서 락을 건다는 것은 내가 진입하고 다른쓰레드는 들어오지 못하게 막는 것이므로, 컨텍스트 스위칭을 유도하는것이다.**

(쓰레드를 두개 만들어서 동기화를 걸고 한번씩 왔다갔다 하는것은 스케쥴링에 대한 부담이 훨씬 늘어나기 때문에 이는 잘못된 설계이다.)

**CreateThread**

**윈도우에서 쓰레드를 만들수있는것은 Createthread밖에 없다.** (윈도우API. 가장 원초적임)

**메모리할당은 VirtualAlloc. 파일오픈은 CreateFile.**

**Beginthreadex(include processh) cppthread(include thread) 역시 내부에서 Createthread하는 것으로, 표준으로 플랫폼에 맞춰서 랩핑한것.**

우리는 지금 CRL를 쓰고있는 입장이다. (srand(rand()), strtok을 사용할때 CRL내부에서 할당한 메모리가 있음을 알수있었다.)

**CreateThread는 윈도우한테 쓰레드를 요청한것으로, 언어(CRL)이 눈치채지 못한 상태로 쓰레드가 만들어 지는것이기 때문에, 런타임 라이브러리는 준비를 마치지 못하게 된다.**

**반대로 \_beginthreadex로 쓰레드를 만들면 런타임 라이브러리한테 요청한것이기 때문에 준비가 된다는 것이다.**

이러한 이유로 C++표준에서 제공하는 함수인 \_beginthreadex를 권장한다.

(아예 C의 연산자와 윈도우API로만 모든 코드를 짤것이라면 상관이 없다. 또한, 지금에 이르러서는 사실 에러가 발생하지않는다. 라이브러리도 바보가 아니기 때문에. 그래도 혹시 모르니까 권장하는 방식을 사용한다.)

**Process - Thread**

**프로세스란 정보를 보관해주는 구조체**(쓰레드가 실행되는 환경을 제공.)

(실제로 구조체로 선언이 되어있다)

**프로세스가 어떤일을 하기 위해서는 쓰레드가 포함되어 있어야 하며, 프로세스는 메인쓰레드가 종료되면 종료된다.**

**쓰레드는 프로세스안에서 자신만의 스택과 운영체제에 의해서 스케쥴링되는 CPU시간을 할당받는 실행의 단위.**

**쓰레드를 포함하고있는 프로세스의 주소공간 Context(문맥)안에서 수행되고, 하나의 프로세스안에 존재하는 모든스레드는 메모리의 같은 주소에있는 데이터의 동일한 값을 얻을수있다.**

**전역변수는 당연히 공유되고, 스택도 같은 가상메모리 테이블에 있으므로 공유된다.**

(스택의경우 접근할 경로가 제공되지않았을뿐, 메모리 입장에서는 완전히 같다. 스택에있는 포인터를 공유해서 접근을 시도한다면 가능해질것. 하나의 가상메모리 테이블안에서는 각자 다른스레드에 속한 스택이라고 하더라도 접근이 가능하다.)

**스택메모리는 같은 가상메모리안에서 구역을 나눈것일뿐, 쓰레드에 완벽하게 독립적인 구조가 아니다.**

**우선순위와 라운드로빈**

**쓰레드가 타임슬라이스 할당량을 모두 사용하거나 자진해서 CPU사용권을 내려놓으면(I/O작업, Sleep사용등으로 block상태에 들어가는경우) 운영체제는 레디상태인 다음 쓰레드를 수행시킨다.**

**우선순위는 프로세스와 쓰레드의 우선순위를 합산하여 결정되며, 우선순위가 높은 프로세스 또는 쓰레드를 최우선적으로 실행시킨다.**

만약 해당 쓰레드가 block상태로 돌입되지않는다면 우선순위 낮은쓰레드들은 기회조차 얻지못할 것이다.

작업관리자에서 오른클릭하면 우선순위가 변경하는게 나온다.



만약에 우선순위 높은 프로세스가 CPU 100%를 점유하는 무한루프를 돌고있다면, 우선순위가 낮은 프로세스는 전혀 실행되지않을것. (우선순위가 높은 프로세스에서 IO작업을 한다면 우선순위가 낮은 프로세스에 기회가 올 수도있음)

마우스도 움직이지않음.

**논페이지드 풀 메모리**

가상화된 메모리를 사용하지않고 항시 메모리에 상주하는 메모리.

커널모드에서 작동하며, 커널및 장치 드라이버는 시스템이 페이지폴트를 다룰 수 없는 상황에서 접근할 수 있는 데이터를 저장하기위해 사용된다.

프로세스 및 스레드, 커널 및 객체, 뮤텍스, 세마포어와 같은 동기화객체, 파일개체로 표시되는 파일참조 I/O요청 패킷(IRP)이 포함된다.

**페이지 메모리**

가상화된 메모리로 사용이 가능. 보통 레지스트리 키 및 기타 레지스트리 데이터 구조에 대한 참조를 저장할 때 사용된다.

섹션이라고 하는 메모리맵핑 파일을 나타내는 데이터구조도 페이징풀.

**타임슬라이스(퀀텀타임)**

**타임슬라이스는 OS마다 복잡한 공식을 토대로 산출되며, 정확하게 수치로 표현할수없다.**

(일반적으로 15m/s ~ 20m/s로 나오지만, 단답형으로 상수값으로 나온다는 없음. 수치계산이 난해하다.)

**윈도우 서버OS는 타임슬라이스가 일반클라이언트보다 2배 길다.**

**서버는 대부분 기능이 한정적이기 때문에 여러 개 돌리지않기 때문에 취한형태이다.**

작업관리자에서 쓰레드가 수천개가 돌고있음을 확인할수 있는데, 타임슬라이스가 20m/s라면 이는 매우 긴시간이다.

거의 대부분의 작업에 I/O작업이 들어가기 때문에 보통은 퀀텀타임을 다쓰는경우는 거의없음.



또한 일반적으로 서버를 만든다고하면 (쓰레드를 최대한 안쓴다고는하지만) **결국 IOCP로 비동기 I/O로 하게된다면 열댓개이상의 쓰레드가 사용되고, 기능별 로직과 DB저장 같은것 까지 염두하면 2,30개정도가 나올것이다.**

(메인컨텐츠를 주도하는 쓰레드는 싱글쓰레드형태일것)

**우리가 짜는 대부분의 코드에는 연산자로만 이루어진 코드는 거의 없다. 연산자로 이루어진것은 짧은시간내에 끝나며, 대부분은 I/O작업에 시간을 할애한다.**

(키보드 입출력, 네트워크 입출력, 파일 입출력 등등)

**I/O작업이 들어가는 경우 쓰레드는 타임슬라이스를 포기하고 block상태에 들어간다.**

(우리가 가져다쓰는 대부분의 함수들은 타임슬라이스를 포기하고 있으며, 타임슬라이스를 온전히사용하고 싶다면 어떤함수도 호출하면 안될 것)

우리가 호출하는 함수들은 알게모르게 타임슬라이스를 포기하고있다. (ex printf)

사실 작업관리자에서 실행중인 쓰레드현황은, 거의대부분 block걸려서 뭔가를 기다리고있는 쓰레드들이다.

싱글코어 하드웨어에서 프로그램을 하나 짠게 아니라면, CPU를 100% 온전히 먹는 프로그램을 짜기 힘들다. (I/O작업이 들어가면 CPU사용률이 떨어지기 때문)

**때문에 타임슬라이스를 온전히 모두 사용하는 경우는 많지않다.**

(Sleep의 경우도 쓰레드를 block시키는 것이기 떄문에 타임슬라이스를 포기하는 형태)

**컨텍스트 스위칭**

결국 컨텍스트 스위칭은 CPU 레지스터를 갈아엎는 작업이다.

**따라서 같은 프로세스에 있는 쓰레드를 컨텍스트 스위칭한다면 가상메모리 테이블에 없는것이기때문에 훨씬 더 가벼울 것이다.**

**다른프로세스에 있는 쓰레드를 컨텍스트 스위칭하려면 가상메모리 테이블을 전부 갈아엎어야하므로, 캐시히트율이 매우 떨어질 것.**

**TCB(Thread Context block)은 쓰레드정보가 담긴 구조체 데이터 덩어리이고,**

**(레지스터 등..)**

**PCB(Process Context block)은 쓰레드들의 핸들정보와 가상메모리 테이블 주소공간이 들어가는 요소일 뿐이다.**

**interrupt**

거의대부분 OS구현은 인터럽트로 끝난다.

((

이전에 타이머 인터럽트 0번을 언급한 적이 있었다. (찾아볼 것)

메인보드 하드웨어 차원에서 인터럽트를 주기적으로 쏘고있으며, 이걸 기준으로 시간을 갱신하고 쓰레드 스케쥴링도 해당 인터럽트를 받아서 처리한다.

))

**CPU자체에 인터럽트를 받아들일수있는 공간이 존재하며, 하드웨어를 꽂으면 무조건 인터럽트 번호를 부여해야 한다.**

번호가 부여됐다면 특정하드웨어가 인터럽트 신호를 보낼때 즉시 빠져나갈수 있을 것.

**[ 제어판 - 장치관리자 – 보기 - 연결별 리소스 - 인터럽트 ]**



**인터럽트번호는 절대 충돌하지않으며, 인터럽트 번호를 부여한 다음에는 인터럽트 번호 신호마다 핸들러를 등록한다.**

(DOS시절에는 키보드 입력받을때 키보드 인터럽트에 내 함수 포인터 핸들러를 등록시켜, 키보드 인터럽트가 발생되면 내코드로 치고들어온 형태였다.)

**랜카드도 랜드라이브에 있는 특정코드가 핸들러에 등록하는것이고, 이후 네트워크 어댑터에서 인터럽트 신호가 발생되면 CPU가 수행되던걸 중단하고 해당 인터럽트에 대한 Proc가 호출된다.**

(CPU의 설계자체가 명령하나를 수행할때마다 인터럽트명령이 있는지 확인함)

**인터럽트 신호를 받고 Proc로 튀어나가는것은 컨텍스트 스위칭이 아니며, 그냥 CPU가 호출하여 돌고있던 쓰레드가 멈춘것이다.**

(따라서 내부에서 백업이 다 이뤄져야 할것)

**최대한 레지스터를 안쓸것이지만 어쩔수없이 썼다면 내부에서 백업했다가 다시 복원시킨다. 이것은 인터럽트신호를 발생한쪽 핸들러에서 다 알아서한다.**

(Syscall(커널모드 전환)역시 인터럽트이다. 커널에 있는 어떤 코드를 수행하기위해 인터럽트 번호를 꽂아넣고 커널모드로 전환 후 인터럽트 신호를 받아서 수행되는 구조.)

**스케쥴러는 타이머 인터럽트로 지금 Running상태인 쓰레드가 할당된 타임슬라이스를 모두 사용했는지 확인하면서 컨텍스트 스위칭 여부를 결정하는 것이다.**

(이때 타이머 인터럽트 발생주기를 1로 낮춰놓으면 쓰레드 스케쥴링이 더 정교하게 돌것이다.)

**그리고나서 타임슬라이스를 다썼다고 판단이 되면 디스패쳐라는 인터럽트를 발생시켜서, Running상태이던 쓰레드를 빼고 어느쪽 ReadyQ로 들어갈지 결정하여 넣어준다.**

**인터럽트 핸들러의 시간허용범위**

**어떤드라이버 인터럽트 핸들러에서 해서는 안될것들을 해서 문제가되거나, 시간을 너무 많이 소비하는 등 이러한 문제들을 일컬어 커널패닉이라고 한다.**

**현재는 MS에서 드라이버 코드를 모두 검증하여, 문제없는 것만 허용받아 돌아가게된다.**

**ReadyQ**

**모든 코어에는 ReadyQ가 존재하며(이는 과거형태로, 현재는 바뀌었다), 우선순위별로 쓰레드가 관리되고있으며 스핀락을쓰거나 락프리형태로 되어있다.**

(OS커널에서는 행위가 너무 작기때무에 락을거는 행위가 없다.)

**이때 디스패처가 Running상태의 쓰레드를 빼서 어떤 코어에 배정하는것에 따라 속도가 달라질 수있다.**

**만약 Running상태의 쓰레드를 돌고있던 코어가 아닌 전혀다른 코어ReadyQ에 넣어버렸다면 캐시미스가 날 가능성이 높아질 것이다.**

MS쪽에서 공개하는 커널자료 문서상에서는 모든쓰레드가 만들어질때 CPU에 대한 번호가 있다고나오는데, **우리도 이 번호(선호도)를 API함수로 지정할수 있다.**

**선호도를 0번넣고 돌린다면 100%먹게될 것.**

문서상에서는 처음에 쓰레드가 만들어질 때부터 하이퍼쓰레딩이라면 홀수 기준으로, 그다음에 짝수기준으로 쓰레드가 생성이 될떄마다 선호도가 부여가 된다는 언급이 존재하는데 이말대로라면 **선호도라는것은 한번 배정받은 곳에서 거의 무조건 깨어나야만 한다.**

**때문에 해당쓰레드는 하나의 코어에서만 써야하는데.. 실제로 이렇게 돌지는않는다.**

무한루프를 만들어놓고 실행시키면 CPU사용량이 일정하게 분포된다.

이 얘기는 스케쥴러가 CPU를 돌린다는얘기인데, 문서상 공개한 것으로는 여기까지이므로 더이상은 아는바가 없음.

**하이퍼 스레딩**

**하이퍼 쓰레딩이란, 코어는 하나이지만 논리적으로 코어가 두개인 모양새로 만들어주는 것이다.** 하지만 실제로 하나의 코어에 두개의 쓰레드가 돌지는 못하므로, **코어안에 파이프라인 구조상 비는공간이 생기거나 캐시미스가 나는경우 치고들어가는 모양새**가 된다.

**중간중간 비는공백에 다른쓰레드거를 계속 끼워넣는것으로, 실제 듀얼코어와 하이퍼쓰레딩 코어랑은 당연하게도 성능차이가 난다.**

**(항상 미친듯이 도는 쓰레드를 코어개수랑 정확하게 일치시켜놓으면 컨텍스트 스위칭이 일어나지않고 항상 그 쓰레드에서 깨어난다.)**

**현재 ReadyQ**

**윈도우 10부터는 구조가 약간바뀌었는데, 지금까지는 레디큐가 코어별로 존재했는데 이제는 CPU코어들을 그룹으로 관리하여 그룹단위 ReadyQ가 나왔다.**

이는 코어가 배정된 쓰레드는 다른코어에 여유가 있어도 마냥 기다릴수밖에 없는 비효율적인 구조를 개선하기 위해서, 코어를 그룹으로 묶어 CPU전체 코어개수에 비례하게 그룹이 잡히는 형태를 취해 **ReadyQ가 해당 그룹단위안에서 코어에 들어갈수 있는 모양새로 커널이 변경됨.**

스케쥴러라는 입장이 멀티코어에서 마치 매니저 역할로서 하나만 존재하고 얘가 CPU를 관리하는것인가, 아니면 CPU마다 스케쥴러를 관리하는 것인가?

**스케쥴러는 애초에 관리적인 목적으로 존재하는애가 아니고, 결국 하나의 타이머 인터럽트를 통해 수행될 뿐이다.**

**CPU선호도**

CPU선호도 같은 경우는 우리는 건드리지 않을것이나, 건드려도 큰 상관이 없다.

**CPU선호도를 조작해서 가장큰 성능을 낼수있는것은 지금 코어 개수보다 쓰레드가 적을때이다.**

코어개수에 딱맞거나 적다면 선호도를 조작해서 컨텍스트 스위칭이 최대한 적게 발생하도록 만들수 있으나, 지금코어개수보다 쓰레드가 훨씬많다면 선호도를 조작했다고해도 이득보는상황이 쉽지않다.

**우리가 만드는 서버환경은 쓰레드가 훨씬 많을 것이므로, 선호도 조작은 애매한 부분이다.**

**기아현상**

**우선순위가 높은 스레드가 block당하지않고 계속해서 돌고있을 때 낮은 우선순위의 스레드가 레디큐에서 무한정 기다리게되는 현상**

MS의 ‘밸런스 셋 매니저’는 CPU에 ReadyQ에 있는 스레드들 중, 4초이상 대기중인애들에 대해 우선순위를 높여주는 방식으로 방안을 찾고자한다.

단, ReadyQ를 모두 뒤질수는 없기 떄문에 열 몇개의 단위로 검사한다고 한다.

**<6.29>**

**프로세스/쓰레드 우선순위**

우선순위를 건드리는것은 개발자 마다 다르기 때문에, 향후 회사정책이나 사수를 따라갈것.

**건드려도 의미없다는 주장**

CPU가 부담이되서 특정부분의 우선순위를 높인다는것은 이미 밸런스가 깨졌다고 본다.

쓰레드를 설계하는데 중요하지 않은것은 없다.

**시스템이 알아서 우선순위를 변경하는 경우**

ex)특정 윈도우창 활성화

**우선순위가 자체적으로 변경되는 경우**

I/O와같은작업으로 인해 Block되어 Signal을 기다리는 상황.

이는 Send recv도 마찬가지로, 해당쓰레드가 신호를 받아서 깨어날때 일시적으로 OS가 알아서 우선순위를 상승시킨다.

(우리가 서버에 사용하는 쓰레드만 우선순위를 다같이 높인다면 어떨것인가? 한번 적용해볼 것)

**Blocked Thread 와 성능의 관계**

**블락되는 스레드는 시스템성능에 영향을 주지 않고, ReadyQ에 몇개의 쓰레드가 들어있든 성능은 달라지지않는다.**

**큐형태로 대기하고있는쓰레드 하나만 신경쓰고있을것이기 떄문에 뒤에있는것은 신경쓰지않는다.**

**Sleep역시 몇개의 쓰레드를 Sleep시키든, 긴시간으로 Sleep을 시키든 가장 빨리 깨어날 쓰레드만보는것이기 때문에 성능과는 관계없다.**

(단 쓰레드가 많아지면 많아질수록 리소스는 많이 사용해야 할것.)

**쓰레드의 우선순위가 같다면 동기화 객체를 대상으로 두개의 쓰레드가 기다리고 있는경우, 어떤 쓰레드가 깨어날지 예측할 수 없다.**

**타임슬라이스**

시스템이 boot되어 운영체제가 수행되면, 운영체제는 가장먼저 컴퓨터의 하드웨어가 특정 주기를 가지고 인터럽트를 발생시키도록 타이머(timer)를 지정한다.**(대부분 15.6m/s)**

**스케쥴러는 스레드를 수행시키기 전에 남은 시간 할당량을 나타내는 TCB(Thread control block)항목을 쓰레드가 수행하기 위해서 허가받은 값으로 초기화한다.(보통은 20m/s).**

**스케쥴러가 이 스레드에게 CPU제어권을 넘겨주면 쓰레드는 다음 타이머 인터럽트가 발생할 때까지 CPU를 전적으로 사용하게 된다.**

**타이머 인터럽트**

**타이머 인터럽트가 발생하면, 하드웨어는 현재 수행중인 쓰레드를 중지시키고 CPU의 제어권을 스케줄러에게 넘겨주게 된다**. (컨텍스트 스위칭X)

**여기서 스케쥴러는 타이머 인터럽트 사이에서 경과된 시간을 타임슬라이스에 반영하지않는다.**

(따라서 타임슬라이스가 남았다면 다시금 쓰레드가 수행할수 있도록하지만, 타임슬라이스 할당량이 남아있지 않다면 스케쥴러는 수행중인 쓰레드를 선점하고 같은 우선순위를 가진 쓰레드들의 ReadyQ의 끝에 현재 쓰레드를 넣게된다.)

**timebeginperiod(1)로 타이머의 해상도를 1m/s로 맞추게되면, 이전과 비교해서 훨씬 정교하게 돌것**이며(15.6m/s -> 1m/s) **그만큼 CPU사용량도 많아지기 때문에 배터리사용량/전력 소모량도 늘어나게 될것이다.**

**Sleep과 같은 함수들을 정교하게 제어하기위해 우리는 반드시 timebeginperiod(1)를 선언하고 갈것이다.**

**타임슬라이스가 끝날때마다 스케쥴러가 수행되므로, 타임슬라이스가 20m/s라고 할때 초당 50번의 스케쥴러가 수행이 될것이다.**

**이때 인터럽트가 발생되어 수행한 코드는 내 타임슬라이스에서 차감시키기때문에, 타임슬라이스에는 영향을 주지않는다.**

인터럽트는 수시로 발생하기때문에(네트워크 송수신, 파일입출력, 키보드입력 등..) 스케쥴러는 인터럽트 발생으로 인한 시간을 제외하여 타임슬라이스에 대한 시간을 보장한다.

때문에 절대시간은 인터럽트가 많이 발생되면 발생될수록 늘어나는 꼴이된다.

**Atomic Operation**

**최소단위 연산을 뜻하며, atomic operation은 인터럽트될 수없는 연산을 의미한다.**

타임슬라이스와 선점을 설명하면서 운영체제가 컨텍스트 스위칭이 발생했는지 판단하기 위해서 주기적으로 CPU의 컨트롤을 얻는 타이머 인터럽트를 사용하는데, 이런 **인터럽트는 CPU에 의해서 정상적인 과정을 거쳐 수행되고있는 명령어(instruction)에 대해 비동기적으로 발생한다.**

**즉 운영체제는 프로그래머가 제작한 프로그램의 코드가 실행되는 동안에 인터럽트가 발생한 시점에서 수행하고있는 Operation(연산)을 중단시키고 언제나 CPU의 제어권을 획득할 수 있음을 의미한다**.

**하지만 cmp, jmp, mov같은 최소단위연산은 절대 도중에 중단시킬수없다.**

명령어repeater같은 경우 CPU차원에서의 반복문이기 때문에 해당 명령어를 수행도중 인터럽트 신호가 온다면 컨텍스트 스위칭이 발생할수있다.

**이때 구조체끼리 대입할때 64비트크기일 경우 repeter mov, byte ptr로 byte단위로 카피하는 코드가 만들어진다.** **이를 다른쪽에서 변수를 얻는다면 카피되던 중간의 값을 가지고 올 수 있으므로, 안전하지않다.**

**Cach Line에 걸쳐졌을 경우**

명령어차원에서는 최소단위연산이라도, 메모리를 긁어오는 행위는 두단계로 나뉘어지기 때문에 쪼개지는 형태가 되므로 주의해야 한다.

**캐시라인이 틀어졌을경우, API차원에서 보장하는 안전한 atomic연산이 보장되지않는다. 따라서 반드시 캐시라인이 맞는다는 전제를 안고가야 할것.**

Rock-free를 할때 이를 대상으로 간다.

Rock-free는 rock을 걸지않고 가는것이기 때문에 항시 이 문제가 따라온다.

**std::atomic (class)**

atomic클래스는 정수형 또는 포인터타입에 대해 산술연산을 atomic하게 수행할 수 있도록 해주는 템플릿 클래스이다. (+, -, and, or, xor등)

C++11에서 표준으로 들어왔으며, 이전에 volatile변수와 interlocked계열 함수를 일일히 사용하는 번거로움을 없앨 수 있다. (내부적으로 interlocked 함수로 래핑되어있음)

**쓰레드 동기화**

**Mutex**

(생략)

추가적으로, C++에 std::mutex가 등장한다.

이는 SRWLock을 쓰면서 스레드아이디를 확인하는 개념을 넣었다.

내부는 크리티컬섹션과 흡사.

**DeadLock**

**서로 원하는 리소스가 상대방에게 할당되어 있기 때문에 이 두 프로세스는 무한정 기다리게 되는 상태를 뜻한다.**

**(설계를 잘못한 경우, 동기화객체가 남발된경우 컨텐츠 로직을 추가하면서 발생되는 경우)**

이전VS에서는 DeadLock이 발생했을 경우 디버깅도구가 잘 제공되지않아 잡기가 힘들어서 로그를 남겨 누가 중첩으로 걸었는지 찾았으나,

**현재는 디버깅도구에서 쓰레드가 나와서 해당쓰레드가 어떤작업을 하고있는지 모두 보여주기 때문에 데드락이 나오는 경우 해당 상황만 잡는다면 즉시 알 수 있다.**

데드락은 크래시가 나는것이 아니기 때문에 서버는 죽지않고 잘 도는것 같으나 유저입장에서는 특정 기능이 안되는것으로, 관리자 입장에서는 알아차리기 힘들 수 있다.

**데드락의 유무를 판단하는 모니터링방법도 존재하지 않기 때문에 따로 코드를 만들어서 어떤스레드의 하트비트를 나오는대로 확인하는 등의 방법을 사용한다.**

**DeadLock의 필요조건**

**1. 다수의 자원에 대한 소유와 대기(Hold and wait of multiple resources).**

적어도 하나의 쓰레드는 자원에 대한 소유권을 갖고있어야 하고, 다른쓰레드에 의해서 소유된 공유자원을 획득하려고 해야한다. (멈춰야된다는 얘기)

**2. 상호배제(Mutual exclusion)**

자원의 소유권은 반드시 한번에 하나의 쓰레드에게만 허용되어야한다. 쓰레드는 같은 자원에 대한 소유권을 공유할수없다.

소유권이 공유된다거나 동시에 접근이된다면 당연히 데드락은 존재할리없다.

**3. 무선점(no preemption).**

운영체제가 다른쓰레드에게 공유자원에 대한 소유권을 주기위해서 이미 공유자원을 소유하고있는 스레드의 소유권을 빼앗을수 없다.

소유권을 뺏거나 뭐 이런행위를 한다면 데드락이 존재할리 없을것.

**4. 순환대기(Circular wait).**

둘이상의 쓰레드가 순환 대기를 발생시키는 형태로 자원에 대해서 대기하고 있어야 한다. 교차로 접근하는 상황을 얘기하는것.

**위 4개의 조건이 모두 부합해야 DeadLock이 발생한다.**

**Thread Life Cycle**

스레드의 라이프 사이클은 다음과 같다.

**1. CreateThread함수를 호출한 결과로 스레드 커널 객체가 생성된다.**

(단, mainThread의 경우 CreateProcss함수를 호출할때 생성되므로, 커널오브젝트가 자동적으로 만들어진다.)

**2. 운영체제가 새로운 쓰레드의 커널 객체를 초기화한다.**

**쓰레드가 사용할 스택할당, 쓰레드가 Running/block를 넘나드는 과정에서 스케쥴러가 쓰레드의 수행상태를 저장할 TCB, context구조를 생성하는 작업**이 포함된다.

생성된 쓰레드에는 default priority는 THREAD\_PRIORITY\_NORMAL이다.

**3. 쓰레드는 새로운 쓰레드의 시작Proc에서 수행을 시작한다.**

주쓰레드의 경우에 시작 프로시저는 main이나 WinMain이 된다. 다른쓰레드의 경우에는 이 지점이 쓰레드가 생성도니 지점이 된다.

**4.스레드가 수행된다**

**5.쓰레드가 종료코드와 함께 종료된다.**

모든쓰레드 함수는 return int형으로 되어있으며, 반환값은 큰 의미는 없음

**6. 쓰레드의 종료코드가 쓰레드의 커널 객체에 저장되며 쓰레드의 사용횟수 카운터값이 감소된다.**

Usage Count가 0이아니면 커널 오브젝트는 삭제되지않고, 이쓰레드의 유효한 핸들을 갖고있는 경우에는 쓰레드의 종료코드를 얻을 수 있다.

UsageCount가 0이 될경우 쓰레드커널 오브젝트도 삭제된다.

**해당 쓰레드가 파괴되는것과 커널오브젝트가 소멸되는것은 다른얘기.**

쓰레드를 생성후 closehandle로 생성하자마자 핸들을 반환하는 경우가 존재하는데, 이는 쓰레드를 종료 시키는 것이 아니라 UsageCount를 미리 차감해놓고 진행하는 것이다.

우리는 쓰레드에대한 상태를 확인해야되기때문에 보관하는 형태로 간다.

**스레드 생성**

모든 쓰레드 생성함수는 수행을 시작할 지점에 대항하는 함수의 포인터를 얻으며, 이 함수를 (thread entry procedure)라고 부른다.

이함수 포인터는 THREAD\_START\_ROUTINE로,

DWORD WINAPI ThreadStartFunc(LPVOID lpThreadParameter);

의 형태를 띄고있다.

CreateThread를 호출시키면서 매개변수로 해당함수의 주소값을 던지면 해당함수를 시작으로 만들어질것이다.

이떄 파라미터 dwCreationFlags로 **CREATE\_SUSPENDED** 를 주게되면 대기카운트 1의값을 갖고 Blocked상태로 생성되며, ResumeThread함수의 호출에 의해 수행을 시작하게 된다.

**굳이 Blocked로 만들 이유가 없으므로 대부분그냥 0을 전달해 바로 돌게끔하여 사용한다.**

**Thread ID, Thread HANDLE**

**쓰레드ID는 시스템상에서 부여되는 쓰레드의 번호이며,**

**(중복되지않으나 재생성에대해 유니크하지않음.로그/디버깅시 스레드ID를 확인한다.)**

**쓰레드의핸들은 쓰레드커널오브젝트를 접근하기위한 수단.**

굳이 생성시점에 보관하지않아도 상관없음

**GetCurrentThreadID**

**자기자신의 쓰레드ID를 반환한다.**

필요에 따라 로그에 많이 활용하게 된다.

**GetCurrentThread**

커널오브젝트의 핸들은 함부러 얻을수없다.

커널오브젝트의 핸들을 추가로 확보하기위해서는 핸들을 복사한다는 개념으로 가야하기 때문에, DuplicateHandle이나.. 아주복잡한 작업이 진행되어야 한다. 단순히 핸들을 임의적으로 얻는다는것은 존재하지않고 존재해서도 안된다.

(듀플리케이트 핸들을 통한 핸들복사는 via에서 확인할 것. 쓸일은 없음)

**따라서 GetCurrentThread는 반환한 핸들은 자기자신을 가리키는 가상핸들을 리턴한다.**

내스레드 안쪽에서 나의 핸들을 요구하는것이 의미가 없을 뿐더러, 진짜 핸들을 얻게되면 UC카운터가 또 증가해버리게 된다.

**따라서 ID와 달리 어떤쓰레드에서든 GetCurrentThread를 호출하든 같은 값이 나올 것.**

**쓰레드의 종료**

**종료는 Signal/Non-Signal로 판단한다. 이제부터 소개하는 스레드 종료함수는 금지한다.**

void ExitThread(

DWORD dwExitCode

);

해당 스레드 안에서 즉시 자기자신의 스레드를 종료시킨다.

BOOL TerminateThread(

HANDLE hThread,

DWORD dwExitCode

);

외부에서 특정 스레드를 강제로 종료시킨다.

종료의 순간은 스레드 자기자신이 결정하고 하던일에 대한 정리를 해야하므로(소멸자),

**바깥에서 파괴한다는 것 자체가 말이되지 않는 상황이다.**

**따라서 위 두 함수는 절대로 사용해서는 안되며, 스레드에게 종료를 요청하고 해당스레드가 스스로 자기를 정리하고 종료해야만 한다.**

**단, 어떤 사유로 인해서 스레드에게 종료를 요청했는데도 종류가 안되는 경우, 강제로 파괴하거나 무시하고 넘어간다. (어차피 르프로세스가 종료되면 알아서 종료된다.)**

**STILL\_ACTIVE**

모든쓰레드는 ExitCode가 존재해서, GetExitCode를 호출했을때 STILL\_ACTIVE를 반환한다면 종료가 아직 안된것이고, 종료인자가 반환된다면 종료된것으로 판단할 수 있다.

**단, STILL\_ACTIVE는 define된 상수값이므로, 쓰레드가 STILL\_ACTIVE를 반환하면서 종료되었다면 받는쪽에서 구분할수 없으므로 100%안전성을 보장해주지 못한다.**

(우리는 Signal을 대상으로 스레드를 기다린다.)

**endthreadex**

\_beginthreadex와 대응되는 함수로서, 런타임라이브러리에서 제공하는 함수이기때문에 그나마안전하다 라는 뜻이다. 쓰면안되는것은 마찬가지.

**지역객체에 대한 소멸권한은 그 함수자체에 하드코딩이 된것이기 때문에 동적으로 쓰레드를 종료하거나 동적으로 프로세스를 종료하는 코드는 모두 금지.**

어차피 끌건데 왜 꼭 이런 방식을 선택해야 하는지 의문이 들 수도 있다. 한가지 예를들자면,

**1.**

예전에 우리가만든 list의경우 iterator를 단순히 뱉어내기만 했으나,

**디버그모드에서의 STL컨테이너는 나를 참조하고있는 iterator가 누구인지 기억하고있다가, 해당 iterator가 소멸될때 컨테이너 쪽에서도 삭제한다.**

**이때 list컨테이너가 소멸자가 호출이되는데 참조하는 itereator가 있다면 문제가 있다고 여긴다.**

**결과적으로 우리가 endthreadex나 exit같은 걸 어딘가에서 런타임상황에서 호출시킨다면 여기저기 널려있는 iterator호출자가 소멸되지않고**, **실제 전역에 선언했던 컨테이너가 소멸될때 오류가 날 가능성이 있다.**

**2.**

어쩄든 메인스레드가 리턴되면 다른스레드도 종료가 될것인데, 이때 다른스레드가 살아있는 상황에서 먼저 메인스레드가 리턴이되어 종료절차를 밟는다면 전역객체에 대한 소멸자가 호출이 된다. **이때 다른스레드에서 전역객체를 참조하는 코드가 존재한다면 예외 발생.**

이는 실제 오류를 찾아가서 콜스택을 확인하면 이상한곳이 나와있고, 메인쪽에 있던 스레드는 내 코드가 아니므로 보이지도 않는다.

전역객체 쪽에있는 다른쪽 스레드에서는 엉뚱한곳에 예외가 던져져있다.

**이는 단순하게 예외참조 오류가 아니라 런타임라이브러리예외가 던져지기 때문에 디버깅이 굉장히 난해해진다.(어디가 문제인지 조차 파악이 안되는 코드에서 예외가 뜬것이기 때문)**

**SuspendThread 와 ResumeThread**

DWORD SuspendThread(

HANDLE hThread

);

대기카운트를 1증가 시킨다.

DWORD ResumeThread(

HANDLE hThread

);

대기카운트를 1차감 시킨다.

위와같은 두 함수도 용도자체는 자기자신이 될수도 있지만 외부에서 쓰레드를 제어하는 형태이므로, 쓰지않는다.

일반 동기화객체를 사용한 블락이나, Sleep을 사용한 블락이랑은 다른얘기다.

**블락이 걸리는상황**

**(I/O를 기다리는것, 동기화객체를 기다리는것, 특정 커널오브젝트 시그널을 기다리는것, 타이머를 기다리는것, 서스펜드를 기다리는것.)**

**Sleep**

Sleep(0)은 자신이 가진 타임슬라이스를 포기하고 ReadyQ에 다시 들어가는것으로, 간혹 사용할일이 생긴다.

**1. 락을걸기엔 너무멀고 락을걸지않고 기다리기에는 불필요한 경우**

(Sleep(1)은 Blocked까지 빠지기 0에비해 상당히 멀다.)

**2. 일부러 컨텍스트 스위칭유도해서 버그를 만들어 테스트해야 하는 환경에서도 쓸일이 생긴다.**

((

지금까지 만든 모든프로그램은 무한루프를 돌면서 Update()를 돌렸다.

**어차피 지금상황에서는 해당일밖에 할수 없으므로 이렇게 간것인데, 게임같은 경우는 이와같이 가지않는다.**

(WinMain일을 하는조건과 쉬는조건으로 정해져있다.)

TCP Fight클라이언트에서 쉬는조건은 목표하는m/s보다 훨씬빨리 처리가되어 시간이 남았을경우이며, 아무생각없는 루프를 돌지않았다.

**서버역시 모든 스레드는 할일이 없을때 쉬고 할일이 있으면 돌아야 한다.**

**대부분 일반적인 클라프로그램은 키보드입력을 기다리거나 어떤조건에 의해서 멈춰있는 상황이 나, Select모델의 서버는 polling방식(주기적으로 확인하는 방식)이기 때문에 이를 해결하지 못한다. (때문에 Select모델서버의 경우 코어하나를 모두 사용할 것)**

**타임인자에 시간을 넣어 블락되는 대기시간을 주는 방법이 있겠지만, 우리는 64개 이상을 커버해야 하는 입장이기 때문에 완전히 해결하지 못한다.**

**따라서 할일이 있는지없는지도 모르는것이 Select의단점으로 꼽힌다.**

**나중에 다른 소켓모델을 사용할경우, 아무이유없이 루프를 도는것은 존재해서는 안된다.**

게임 메인루프 목표치의 프레임이 나올것이고 거기에 Sleep()을 넣을것이다.

**스레드 인자 (구분자로 사용)**



파라미터에 3을 넣었다.(이자리는 스레드에 전달될 변수에 대한 포인터자리.)

**해당 파라미터는 사용하고싶은대로 사용하며, 거의대부분은 구분자를 넣는다.**

스레드를 용도가 정해진 목적으로 분리해서 간다고하면 상관없겠지만, **스레드풀을 사용할때 스레드마다 구분이 되지않으므로 이를 구분하는 구분자로 사용하는것.**

**rand()에서의 활용하는 예**

srand시드값은 라이브러리 내부의 getptd구조체에 저장되고, 이구조체는 스레드마다 존재한다.

따라서 main문에서 srand를 할경우 시드값이 모두 동일해지기때문에, 스레드별로 가야한다.

이때 파라미터를 구분자로 활용하는것.

**또다른 방법은 스레드ID를 사용하거나, 스택메모리 주소를 사용한다.**

(프로세스안에서는 스택의 저장소는 중복될수없으므로, 해당함수의 지역변수 주소를 사용하면 될것이다.)

**std::thread**

std::thread 같은 경우는 멤버함수 여러가지 쓰레드가 나온다.

쓰레드 관련 동기화 객체 API들과 런타임 라이브리에서 제공하는 \_beginthreadex로가고, 손에 익은다음에 std::thread를 사용하는것을 권장한다.

단 std::thread를 사용하는 경우 내부적으로 어떻게 돌아가는지 정확히 파악하고 써야할 것이다.(f11로 들어가게되면 소스코드가 나오기때문에 충분히 파악할수 있을 것)

\_beginthread는 너무 간략화되서 ex를 사용하며, 인자는 CreateThread랑 완전히 같다.

**exit, exitprocess, exitThread**

**exitprocess()는 해당 프로세스를 종료시키는 API함수로, 당연히 우리가 쓸 대상이 아니다.**

**프로세스 자체를 억지로 종료시킬경우 전역이나 static의 소멸자가 호출되지않으며 지역객체역시 종료의 기회를 얻지못한다.**

(지역객체역시 소멸자는 선언된 블락이 닫힐때 호출되기 때문)

**exit(); 은 exitprocess를 런타임 라이브러리에서 랩핑한것으로, 전역/지역 객체의 소멸자 호출 정리작업이 들어간다.**

(싱글톤, 전역객체의 경우 소멸자 호출이 되지않으므로 억지로 유도하기위해 atexit을 사용한다. exit를 호출시키면 atexit에 등록되있는 것들과 등록된 소멸자 호출들이 진행이 되고나서 메인쓰레드도 종료된다. (이 작업들은 main함수가 리턴되고 난 다음에 진행된다.))

(atexit : 이전에 싱글톤에서 언급. 인자로 함수포인터를 넣으면 해당 프로세스가 종료될때 자동으로 호출이 된다. 큐방식으로 포인터를 등록시켜놓으면 해당 런타임 라이브러리가 메인쓰레드 종료하고나서 거기 등록된함수들을 연속으로 등록시켜주면서 연속으로 꺼진다.)

**우리가 메인문을 리턴을 하게되면 메인쓰레드가 바로 꺼지는게 아니라 런타임라이브러리에 메인쓰레드로 넘어가서 정리작업 절차를 진행하고 메인쓰레드가 꺼지고 프로세스가 꺼지는형태.**

((

**우리의 메인문은, 메인프로세스의 메인쓰레드의 최초의 쓰레드 프로시져가 아니다.**

실제쓰레드 프로시져는 런타임라이브러리 내부에 들어있고 몇개의 내부를 거쳐 메인을 호출시키는 것이다.

))

**따라서 exit을 호출시킬 경우 전역변수, 객체를 정리는 하지만 역시 런타임에서 결정하는 코드이므로, 지역객체 소멸에 대한것은 소멸의 기회를 얻지 못하므로, 사용해서는 안된다.**

exitthread의 경우역시 해당쓰레드에는 종료절차를 거치겠지만 런타임라이브러리에서 제공되는것이 아니기 때문에 사용금지.

**스레드 동기화**

인터락, 크리티컬섹션, 뮤텍스, 세마포어, 이벤트, srw 락 등이 존재한다.

윈도우10 Core 부터 새로추가된 변수 동기화객체가 있음.

아직 제대로활용을 안해본 상태..

**Intorlocked 정의**

**인터락은 동기화 함수지만 객체가 아니므로 동기화객체가 아니다.**

**인터락계열 함수들은 여러쓰레드에서 공유되는 변수를 대입/+-연산,값교체등과 같은 행동에 대한 원자성을 보장해주는 매크로 함수이다**

내부에서 CPU차원에서의 동기화인 lockincrement명령어를 사용한다. (atomic operation보장)

메모리차원에서 똑같은 지점을 지정한다면 동기화가 걸리고, CPU가 해당일을 수행하면서 진행되는 동안은 잠기게 된다.

**OS차원의 기다림과 CPU차원에서의 기다림**

**동기화객체를 사용한다면 OS차원 기다림**

**인터락 함수의 경우는 CPU차원에서의 기다림**

(인터락함수는 동기화객체를 사용하지않기 때문에 CPU차원에서의 기다림이다.)

**OS차원에서의 기다림은, OS가 해당스레드를 block시키고 signal이 발생하면 깨우는 형태.**

**CPU차원에서의 기다림은, 이는 OS차원에서 그냥 명령어 하나만 수행되는 기다림이다.**

**CPU차원에서의 기다림은 그냥 명령어가 느린것으로, 컨텍스트 스위칭이 발생하는 형태가 아니다.단순히 해당 메모리에 접근하기위한 기다림.** (타임슬라이스가 늘어나는 것은 아니다.)

CPU차원에서 명령을 수행하는데, 명령이 끝나지않으므로 잠깐 느린것처럼 OS가 작동한다.

또한 인터럽트가 발생되지않기 때문에 타임슬라이스가 끝나지않고, 할당된 시간이 초과했다고 하더라도 이 명령이 끝나야 인터럽트 처리가 가능하다.

**멀티스레드 테스트**

**두개이상의 코어에서 동시에 도는 상황을 스레드 동기화 이슈라고 말하며, 싱글코어일 경우 본인만이 사용할 것이기 때문에 스레드 동기화에 대한 의미가 없다. 당연히 같은 코어를 얘기한다.**

때문에 멀티쓰레드 프로그래밍은 듀얼코어 이상에서 테스트해야 한다.

**코어가 많아질수록 스레드이슈가 더 많이 발생될 것이다.**

**Interlocked 특성**

**인터락도 내부에서 속도가 떨어지는것이기 때문에 남발하게되면 성능이 저하된다.**

가볍게 쓰지만 남발하는 경우 체감되는 성능저하가 있을것.

인터락시리즈는 Increment, Decrement외에도 엄청나게 많다.

**인터락관련된 모든함수의 파라미터로 들어가는 변수의 조건은 경계에 서야한다.**

**시작점주소값이 무조건4(64시리즈의 경우는 8의경계)에 경계에 서야하며, 이것이 지켜지지않을 경우 인터락을 보장하지않는다.** (지금은 에러는 나진않지만 보장해주지않음)

때문에 Inlock64시리즈는 무조건 8byte경계에 서야한다.

**일반적으로 그냥 사용한다면 문제가 되지않으나(멤버변수, 구조체, 지역변수 다 이렇게 선언이 되고 있기 때문에) 포인터 조작을 하거나 어떤 사유로 인해 틀어진경우 원자성이 보장되지않는다.**

**인터락시리즈 함수중에는 128bit짜리도 존재하는데, 이것은 억지로 16바이트 경계에 세워야한다.** (우리가 선언할수있는 변수의 최대크기는 8바이트(64bit)이고, 이 변수는 8의 배수경계에 서기 때문에)

**8byte짜리 두개를 합쳐 16바이트변수를 만드는데, align을 이용하여 16바이트 경계에 서도록한다.**(alignmalloc을 사용하여 16을 전달하면 16경계에 세워진다.)

**InterlockedIncrement**

InterlockedIncrement의 경우, 증가된 크기를 리턴해준다. (모든 플렛폼에서 보장되는것은 아니나, 우리의 플랫폼인 WINPAI는 100%보장하므로 믿고 간다.)

**어떤 변수를 인터락을 사용하여 안전하게 1증가했다고 가정한다. 이때 증가시킨 값을 비교를 통해 어떤 행위를 하고자 할때(비교, 출력 등) 이 값이 내가 증가시킨 값이라는 보장이 없다.**

(이사이에 다른쪽에서 또 증가시켰을 수도 있기 때문에)

내가 증가시킨값이 정확하게 떨어지는것을 알고싶다면, 이때 리턴값을 사용한다.

**증가나 감소는 원자성으 보존하긴하지만 값을 알지못하면 쓸모가 없기 때문에 리턴값은 굉장히 많이 사용된다.**

**InterlockedExchange**

LONG InterlockedExchange(

LONG volatile \*Target,

LONG Value

);

**어떤 변수를 인터락으로 증가시키고 있을때, 다른곳에서 대입을 하는 행위는 문제가 되지않는다.**단순히 어떤 변수에 값을 대입하는것은 안전한 행위다. (예외로 캐시라인이 틀어졌을 경우가 있을수 있겠지만, 인터락은 어차피 캐시라인이 틀어졌다면 보장받지 못함)

대입은 그냥 대입일뿐이고, increment는 단지 atomic한 증가를 보장해주는것이다.

어떠한 변수를 안전하게 쓴다고해야해서 interlock을 도배해야되는것은 아니다.

**interlockedexchange의 리턴값은 리턴값이 원래 저장되어있던 값이 나오기때문에, 이값을 활용한다. 단지 대입을 한다고한다면 아무런 특징이 없음.**

InterlockedIncrement와 interlockexchange가 같이 돌기에는 약간 애매한 부분이있다.

멀티스레드 환경에서는 Send를 세션당 1회로 제한한다. 누군가 샌드를 하고있다면 기다리지않고 그냥 패스.

만약 양쪽스레드에서, 특정 flag를 확인하여 코드진행유무를 판단한다고 한다면 이 flag를 동시에 확인하는 경우 한스레드만이 들어가는것이 보장되지않으므로 atomic한 보장이 필요하다.

이때 interlockedexchange를 확인하여 리턴되는값(이전값)을 확인하는 방식으로 사용한다.

**InterlockedCompareExchange**

LONG InterlockedCompareExchange(

LONG volatile \*Destination,

LONG ExChange,

LONG Comperand

);

**Interlockedexchange는 어떤 임의의값을 넣으면서 그전값이 뭐냐는것을 확인하는 것이다.**

flag를 확인하는 경우 (TRUE,FALSE 두종류만이 들어가는 경우)만 사용할수있음.

하지만 비교할 값자체도 임의의 값인 경우에는 CompareExchange를 사용하면 된다.

교체할값이 비교값과 같다면 바꿔주고, 같지않다면 바꿔주지않는행위를 원자성을 보장해준다.

이것은 CPU가 제어해줘야 할수있는것이며, 언어가 할수있는것이 아니므로 API함수이다.

우리가 쓰는 x86아키텍쳐에는 다 명령어로 존재한다.(옛날부터 만들어져있었음)

때문에 lockcmpchg(comparechange),lockcmp, lock이라는 명령어가 존재하고있음.

당연히 성능은 떨어진다.

**waitonaddress**

BOOL WaitOnAddress(

volatile VOID \*Address,

PVOID CompareAddress,

SIZE\_T AddressSize,

DWORD dwMilliseconds

);

**어떤변수의 주소값을 기다리고, 해당 주소값이 변경되는경우 깨어난다.**

아직 작동원리를 파악하지못했으며, 얘로인해 동기화 객체에 대한 개념이 윈도우 내부에서 완전히바뀌었다.

동기화객체쪽은 ms에서도 성능을 높이기위해 개량작업을 계속해서 하기때문에, 자주바뀐다.

대표적으로 CriticalSection이 윈도우7에서 윈도우10 Core 으로 넘어오면서 완전히 바뀌었다.

**<7.1>**

**스레드 디버깅**

VS에서 디버깅시, 스레드창에서 특정스레드를 중지시켜 돌지못하게 할수있음.

밧줄꼬인표시가 나기도하는데, 이는 다른스레드가 이구간에 있다는 것을 뜻함.

병렬스택 창에서 스레드가 어떻게 어떤식으로 돌고있는지 알수있다.

**Critical Section**

**크리티컬 섹션은 경합이 발생되면 커널모드로의 전환이 일어나지만 기본은 유저모드이다.**

(스레드가 블락당한다는것은 커널모드에서만 진행이 가능하다. 유저모드에서는 방법X)

**Mutex나 Semaphore는 커널오브젝트이므로 단순히 확인하는 과정 절차도 커널모드전환이 일어나며, 이 차이는 성능면으로봤을때 굉장히 큰 차이**

**이다.**

**따라서 우리는 유저모드 객체를 사용한다.**

(세마포어 뮤텍스를 쓰는케이스는 0이다.)

**CRITICAL\_SECTION cs;**

**InitializeCriticalSection(&cs);**

**EnterCriticalSection(&cs);**

**//동기화**

**LeaveCriticalSection(&cs);**

**DeleteCriticalSection(&cs);**

InitalizeCriticalSection으로 초기화하는 것은 큰 의미는 없음. 해당 구조체를 0으로 만드는 느낌으로 진행한다.

**EnterCriticalSection은 리턴값이 없고, 시간초과 매개변수가 없고 임계영역에 들어가려는 스레드는 임계영역이 사용가능해질때까지 무기한으로 block당하며, 풀수있는 방법이 없다. (데드락이 발생할 수 있는 이유)**

**임계영역은 같은 스레드에 의해 재귀적으로 계속해서 획득할 수 있다.**

**단, 임계 영역은 내부적인 카운터를 갖고있기 때문에 임계영역을 획득한 횟수만큼 반드시 해제(relase)해야 한다.. 재귀적으로 허용하지만 짝은 맞춰야함.**

**크리티컬섹션 구조체 내부에 스레드 아이디가 보관이되고, 해당 스레드안에서는 재귀적으로 EnterCriticalSection 호출을 허용해주는 형태이다.**

**지금은 체감이 안될수있으나, 이는 개발자에게 굉장히 편리한 기능이다.**

(SRWLock같은 경우 이와같은 형태가 빠져있기 때문에 성능이 더 좋으나, 자기자신을 중복하여 획득하려고 한다면 데드락에 걸릴 수 있다)

((

재귀적인 상황.

보통 동기화객체는 함수내부로 포함시킨다.

출력함수 내부에서 LOCK – 출력 – UNLOCK 같은 형태.

이때 맵순회(내부에서 락) 중에 플레이어가 죽는다면 PlayerDelete()를 호출시키는데, 이 내부에서 또 맵에대한 Lock을 걸어야 한다.

이경우 SRWLock을 사용하면 데드락이므로, 구조를 뒤집어 엎던가 크리티컬 섹션을 사용해야 한다.

))

TryEnterCriticalSection은 지금 내가 들어갈수 있냐없냐를 시도하는 함수이다.

**임계영역을 blocked없이 획득한 경우나 자기자신이 이미 임계영역에 들어가 있는 경우 TRUE를 리턴하고, 다른 스레드가 이미 임계영역을 소유한 경우에는 이 함수가 FALSE를 리턴한다.**

이 함수가 임계영역에 들어가는데 사용될 경우 TryEnterCriticalSection함수가 TRUE를 리턴하는 경우에만 조건적으로 LeaveCriticalSection함수를 호출할 수 있을것.

TryEnterCriticalSection은 사용하지않는다.

우리가 뭔가의 동기화 객체에 접근을 한다는것 이 일을 해야 하기 위함인데, 누가 쓰고있다면 기다려야한다.

**성능을 높이기 위해 병렬처리를 극대화시킨 복잡한 설계를 한다면 오히려 성능이 떨어지거나 유지보수가 힘들어 지기때문에, 스레드프로그래밍은 항상 간단하게 설계해야 한다.**

**DeleteCriticalSection();**

CriticalSection은 개발자가 선언했지만 명식적으로 Delet도 호출시켜야 한다.

**예전에는 크리티컬섹션 구조체안에 현재 들어온 스레드 ID, 카운트값, 세마포어핸들이 들어간다. (실제로는 이벤트 객체)**

이는 겉모양은 유저모드지만 안에서 커널오브젝트가 만들어진 것으로, 유저모드에서 동작할 수 있는 동기화객체는 존재하지않았다.

**하지만 윈도우 10 Core부터는 세마포어가 아닌 완벽하게 유저모드에서 존재하는 동기화객체가 새로 개발이 된상태**

(이벤트 커널오브젝트는 커널오브젝트 동기화객체의 제일 밑바탕이 된다. 세마포어, 뮤텍스역시 이벤트가 바탕이다.)

**EnterCriticalSection**

**EnterCriticalSection에서 실제로 하는일은 Interlocked을 체크하는 함수를 한번 호출하고 바로 빠지는 형태이다.**

**인터락 관련함수로 비트체크 flag를 올려놓고 카운트를 차감하며 빠지는것.**

(카운트는 마이너스로 들어가게된다.)

**따라서 경합이 발생하지않았을 경우 InterlockedIncrement를 하는것과 성능이 비슷하다.**

**하지만 함수 내부에서 체크했는데 스레드ID가 다른경우(진입한 스레드와 구조체 내부에 저장된 스레드), EnterCriticalSection내부에서 CreateEvent로 커널동기화 오브젝트를 생성하고 이를 대상으로 blocked이 들어가고 Leave하면 서로 signal하면서 깨어나는 것이다.**

**이로인해 유추할수 있는 것은, DeleteCriticalSection은 내부에서 동기화객체가 생성됐다면 이를 해제해야 되기 때문이에 호출시키는 것으로, 경합이 없는경우 의미없는 행위가 된다.**

**또한 설계상 동기화 객체가 무수히 많이 들어있다고 경합이없다면 성능이 떨어지지않으므로, 경합의 발생횟수가 중요하다.**

**현시점에서는 안해도 되는함수가 되어버렸다.**

**크리티컬 섹션에는 내부에 세마포어, 그내부엔 CreateEvenet..**

**모든 동기화객체의 기본은 Event.**

**KidEvent의 등장**

지금은 객체자체가 새로운개념의 동기화 객체가 나왔다.

**이전 구조에서는 누군가 Enter에 성공을 하면 세마포어라는 커널오브젝트 이벤트를 만들었다.**

이때 커널오브젝트 자체는 생성을 실패할수있는 가능성이 존재하고, 이는 동기화객체의 구조가 결함이 있다는 것을 뜻한다.

(Windwos 10 Core 패치 이후 완전히 구조가 바뀌어서 CreateKidEvent라는 API가 있었는데Waitonaddress로 대체된듯..거의 같은개념.)

**KidEvent는 생성의 개념이 아니라 이미 만들어진 객체를 가져다 쓰는 개념이다.**

**CriticalSection동기화의 개념이 완전히 유저오브젝트로 넘어온것이다.**

이 이상은 정보가 거의 공개된것이 없기때문에 파악하기 힘듬.

**이 개념이 개발되면서 기존의 유저 오브젝트는 이제 완전히 유저오브젝트 개념으로 바뀌었다.**

**내부객체는 세마포어로 존재하지만 실제 이벤트는 생성이 실패할수없는 유저 이벤트로 이루어져 있다.**

**SRWLock**

**SleepReadWrite lock은 우리가 가장 많이 사용하게 될 유저동기화 객체이다**.

**SRWLock을 주력으로 사용하되, DeadLock이 발생했는데 도저히 개선이 되지않을 경우에 Critical Section으로 전환하는 것을 권장한다.**

SRWLock은 Read용 Lock과 Write용 Lock이 분리가 된다.

(CriticalSection은 온전히 한곳만을 진입이 가능하다.)

단점은 스레드를 확인하는 코드가 빠지게되므로, 재귀적인 형태에 락이 걸린다.

**Share(:공유) 모드 락은 share에 접근이 가능해지고,**

**exculsive(:독점) 락은 모든 접근이 차단된다.**

SRWLOCK SrwLock;

void **Initialize**SRWLock(PSRWLOCK SRWLock);

Initialize는 단순히 값을 초기화 하기 때문에 초기화에 대응하는 Release는 존재하지않는다.

Acquire - Release라는 개념으로 들어간다.

**진입형태**

void **Acquire**SRWLockExclusive(PSRWLOCK SRWLock); //독점접근.

void **Acquire**SRWLockShared(PSRWLOCK SRWLock); //Shared끼리 접근허용

**해제**

void **Release**SRWLockExclusive(PSRWLOCK SRWLock);

void **Release**SRWLockShared(PSRWLOCK SRWLock);

**Try**

bool **TryAcquire**SRWLockExclusive(PSRWLOCK SRWLock);

bool **TryAcquire**SRWLockShared(PSRWLOCK SRWLock);

Try는 CriticalSection과 마찬가지로 잘사용하지않는다.

해당 SRWLock으로 기준으로 비트체크를 통해 빠지는 형태이다. (스레드 아이디 체크없음)

때문에 코드가 간결해지며, 누군가 쓰고있다면 해당 주소기준으로 동기화신호를 기다리는것.

**Shared는 동시에 접근이 가능하므로 (읽기 전용)동시에 순회할수있음**

**Exclusive는 독점접근으로, 읽기쓰기 모두 불가능하다.**

**Acquire를 같은곳에서 두번하게된다면 그 즉시 데드락에 빠지게 된다.**

**Spinlock**

**스레드 동기화기법은 동기화객체를 사용하는것과, 스핀락 두가지가 있다.**

SpinLock은 내가 획득하려는 이 접근이 굉장히 빠른 시간안에 획득할수있을것 같을때 사용한다.

**락이 걸린 상태일때 내 타임슬라이스 안에 락이 해제가 될것같을때 사용.**

동기화 객체를 사용하여 스레드가 블락된다면, 내 타임슬라이스를 포기하고 block으로 빠진다음 다시 신호를받아 ReadyQ로 들어가서 순서를 기다려야 한다.

이는 Sleep(0)을 사용한다고 해도 마찬가지이다. (컨텍스트 스위칭)

**CriticalSection, SRWLock의 유저동기화 객체들은 스핀락이 기본적으로 내장되어있어서, 굉장히 짧은시간동안 시도(SpinLock)을 해보고 그다음에 block당하러 가게된다.**

(SpinLock의 개념은 자원을 획득할 수 있을 때 까지 타임슬라이스를 소모하면서 루프를 도는것. Cpu점유함.)

**SpinLock은 InterlockExchange와 변수를 통해 임의로 만들 수 있으나, 결국 Loop를 도는형태이므로 때문에 장시간 하게된다면 오히려 역효과가 나게된다.**

**따라서 우리가 임의로 스핀락을 만들어 사용하는것은 절대 금지.**

(지금 사용하는방식은 일반적인 동기화객체와 스핀락이 결합이 된것이다.)

**SpinLock, LockFree vs 스레드 동기화객체**

스레드 동기화객체를 주로 사용하는경우, 아무리 동접자가 높아지고 서버가 버벅인다고 하더라도 CPU점유율이 잘 높아지지 않는 현상을 보이는 반면.

스핀락, 락프리로 도배하는경우 CPU사용률 자체는 높아지지만 처리속도가 좀더 빠를수도있다.

(이는 무조건적으로 락프리가 빠르다고 말할 수는 없음)

**InitializeCriticalSectionAndSpinCount**

BOOL InitializeCriticalSectionAndSpinCount(

LPCRITICAL\_SECTION lpCriticalSection,

DWORD dwSpinCount

);

크리티컬섹션에서 SpinLock의 Count를 조절할 수 있는 함수이다.

해당 Count회수만큼 시도하고 Block으로 빠지게 될것.

(SRWlock은 1024번에 고정치였는데 지금 코드상에서는 55번으로 바뀐것을 확인할 수 있음)

**Pause명령어**

WINAPI는 yeildProcesser()가 있고(via에서 등장) 이를 호출하면 pause로 인자에 어셈으로 들어간다.

사실은 pause는 하는일이없다. 그냥 명령하나를 버리는것이다.

NOP이라는 어셈블리 명령어는 보통 점프테이블에서 사용되는데, 사실상 공백과 같은 어떤 operation(연산)도 없는 명령어이다.

**pause는 이러한 NOP과 동급이며, 하이퍼스레딩에서 일부러 명령어 하나를 쉬는 명령어이다.**

(yeildProcesser를 호출하면 인라인 어셈블리 pause가 호출된다.)

최근 하이퍼스레딩 CPU가 나오는 이후 SpinLock을 구현할때는 항상 pause를 넣으며, 거의 정석처럼 받아들이고 있다. (MS가 이렇게 사용하므로 사실상 검증은 끝난것이라고 봐도 무방)

**SpinLock에 Pause명령어를 넣는이유**

SpinLock을 사용한다는 것은 CPU코어하나를 모두 점유하고 가겠다는 뜻이다.

**이때 가장 이상적인 상황은 CPU코어 2개일때, SpinLock을 도는 코드가 한쪽에서 루프를 돌고있고, 이에 대응되는 자원을 다른쪽에서 사용하는 상황이다.**

이 자원을 사용하는 곳에서 금방 작업이 끝날것 같음을 SpinLock을 통해 기다리고 있는 상황.

**이때 만약 SpinLock을 도는데 이에 대한 자원을 활용하는 어떤 스레드가 Block당한 상태거나 ReadyQ에서 대기하고 있는 상황이라면 이는 아무리 기다려도 끝나지 않을 것.**

떄문에 pause같은 경우 내가 혼자서 미친듯이 도는것은 너무 불필요한 행위이므로, 하이퍼스레딩에서는 한번씩 다른쪽으로 전환을 시킨다.

**하이퍼스레딩**

텍스트, 스크린샷, 필기구이(가) 표시된 사진

자동 생성된 설명

하이퍼 스레딩은 캐시미스 등으로인해 여유가 생길 때 파이프라인에서 명령어 로드/패치 메모리적재 등과 같은 과정을 번갈아가면서 시도하는 것.

pause가 하는일은 명령은 없고 이번타임 쉬는 것으로, 일부러 명령하나 버려서 하이퍼스레딩이 옆스레드가 돌게끔 하는 것.

때문에 하이퍼 스레딩이 지원되지않는다면 pause가 의미없는 행위가 될 것이다.

**만약에 A스레드옆에 B스레드가 A스레드자원을 쓰는 스레드로 같이 배정이 되었다면 좋은 효과를 볼수 있을 것이고, 다른코어에 배정된다면 어차피 루프를 도는것이므로 전혀 다른 스레드의 일을 시키면서 돌게 되는것.**

(OS입장에서는 하이퍼스레딩 CPU는 2개의 CPU로 본다.)

**하이퍼스레딩이 지원되는 환경에서 SpinLock을 돈다면, 혼자서 점유율을 모두 차지하면서 기다리지않고 다른쪽 코어에 한번씩 기회를 주는 것.**

**WaitOnAddress**

BOOL WaitOnAddress(

volatile VOID \*Address,

PVOID CompareAddress,

SIZE\_T AddressSize,

DWORD dwMilliseconds

);

최근에 나온 API로, 지정된 주소의 값이 변경될때까지 기다린다.

실제적인 작동방식이 공개된게 없다.

**SRWLock사용시 주의할 점**

**루프안에서 동기화를 위해 (Lock - unLock - Lock)와같은 형태의 코드가 나왔을 경우.**

(이와같은 코드가 나올일은 없으나 예제를위해 가정)

**이때 A스레드가 SRWLock을 Release하면 대기하던 스레드B는 깨지않고 다시금 A스레드가 Lock된다.**

**커널 오브젝트를 쓴다고하면 (ReleaseMutex 등) 커널모드전환이 되면서 정말로 대기중인스레드를 를 깨우면서 오겠지만, SRWLock은 유저객체로 변경이되면서 이러한 과정들이 생략되어버렸다.**(커널모드로의 전환이 없다는것은 block중인 스레드를 깨울수 없음)

이 구조대로라면 누군가 들어가서 깨우는 느낌이아니고 혼자서 깨어나는 방법밖에 없는데.. (유저모드 동기화객체가)완전히 파악이 안된상태.

위와같은 방식으로 코드를 짤일은 없으나 염두해두고 있을 것.

**KeyEvent**

크리티컬 섹션에서 동기화 이슈가 발생하여, 내부적으로 커널오브젝트를 생성하려고했는데 실패한다면 어떻게 될것인가?

이를 위해 나온 개념이 KidEvent이다.

**윈도우커널 분석에 KidEvent라는 개념이 새로나왔다**.

이걸로인해서 완벽한 유저모드의 동기화객체 구현이 됐다고 언급만 되어있을 뿐 이것에대한 알고리즘 작동방식을 찾지 못하는상황.

누군가를 leave등으로 Lock을 풀었을때 대기중인 스레드를 관리하지않고 있다면 어떤식으로 스레드가 깨어날 것인가?

시스템모니터링을 가보면 컨텍스트스위칭 횟수도 모니터링이된다.

특정스레드의 컨텍스트 스위칭 카운팅을 볼수있는데, 이를 통해 SRWLock을 풀때마다 컨텍스트스위칭이 어떤변화가 있는가 살펴보니 해제하는 순간에 대기중인 스레드가 같이 깨어났다.(이 사항은 검증된 자료가 아닌 추측이므로 참고만 할 것.)

(Event의 경우 SetEvent로 signal을 주면 대기중인 스레드를 특정지어 진입하는 상황으로, 이러한 자신을 기다리고 있는 스레드 정보들을 리스트로 커널에서 관리하는 형태)

이상하니까 그냥 의문만 가지고 넘어간다..

스레드가 블락되려면 커널모드로의 전환이 일어나야하는데.. 유저모드에서의 이벤트 작동이 말이되나..? 이것도 잘 모르겠음.

**WaitForMultipleObjects**

WaitForMultipleObjects는 여러 개의 커널오브젝트를 대상으로 한번에 기다린다.

**Singal/Non-Signal은 동기화에 사용되기도하고, 또한 프로세스 핸들을 가지고 프로세스 종료여부를 판단할때도 사용이 가능하다.**

(프로세스/스레드 종료시 Signal.)

**GetExitThread의 반환값이 STILL\_ACTIVE인지 확인하는것으로 스레드의 종료 여부를 알 수도 있겠지만,**

**안전하고 정확한 방법은 스레드의 핸들을 가지고 Signal을 기다리는 것이다.**

**Event**

프로세스나 스레드는 목적이 다른것이 있고, 걔에 대한 정보를 참조하는 객체로서 존재하는 것이다.

이벤트는 오직 동기화를 위해 나온 개념이다.

(뮤택스, 세마포어, 파일은 모두 점유해서 사용한다면 NonSignal, 소유권을 풀어준다면 Signal발생. 예를들어 파일이 입출력이 완료됐다면 Signal이 오게되는 것.)

**모든 커널 오브젝트들은 다 Signal과 Non-Signal이 존재하고, 우리는 Event객체에서 이를 활용하여 직접 동기화를 제어할 것이다.**

SetEvent, ResetEvent를 통해 직접적으로 Signal의 유무를 바꿔 스레드를 제어한다.

**WaitForSingleObject**

DWORD WaitForSingleObject(

HANDLE hHandle,

DWORD dwMilliseconds

);

매개변수로 핸들을 전달하고 인자로 들어간 시간만큼 대기한다음 리턴한다. 만약 그전에 Signal이 오면 리턴한다.

**리턴값**

**리턴되었다는 것은 실패든 성공이든 반응이왔다는 뜻이다.**

**WAIT\_FAILED**

실패한 상황으로, 사실상 오브젝트가 아닌 상황밖에 없다.

**WAIT\_OBJECT\_0**

내가 기다리는 Signal이 있었다는 뜻.

**WAIT\_TIMEOUT**

타임아웃(지정된 시간 초과)

**WAIT\_ABANDONED**

뮤텍스를 소유한 스레드가 뮤텍스의 소유권을 해제하지않고 종료됐거나 강제적으로 소유권이 포기가 된 상태를 뜻한다.

**WaitForMultipleObjects**

DWORD WaitForMultipleObjects(

DWORD nCount,

const HANDLE \*lpHandles,

BOOL bWaitAll,

DWORD dwMilliseconds

);

핸들의 포인터로 배열이 들어가며, 개수가 필요하다.

해당 배열에 있는 핸들을 대상으로 signal을 확인한다.

**bWaitAll에 TRUE인 경우 모든핸들이 signal이 되어야 리턴되는것이고,**

**FALSE인 경우 하나라도 signal이 올경우 리턴이 된다.**

특이점은, WaitALL을 false로 했을때 하나의 객체 signal만 확인해준다는 것이다.

따라서 아래와같은 반환값을 통해 어떤 객체에 signal이 왔는지 확인한다.

**WAIT\_OBJECT\_0** ~ ( **WAIT\_OBJECT\_0** + *nCount* – 1)

(첫번째 Index가 0이므로 -1을 해줘야 한다.)

Mutex, Semapore, Event모두 위 두함수를 사용하지만 우리는 Mutex와 Semapore는 사용하지 않는다.

(7.1 스레드 프로그래밍 1단계 과제)

**Mutex**

**뮤텍스는 커널객체이며, 시스템에 있는 자원의 소유권을 서로 배타적으로 획득할 수 있도록 한다. 그러므로 임계영역과는 달리 뮤텍스는 서로 다른 프로세스가 스레드에 접근할수 있다**.

서버쪽에서는 딱히 쓸일이 없으나 클라쪽에서는 중복 실행(클라이언트 여러개를 실행하는 행위)을 막을때 사용하는 케이스가 있다.

**CreateMutex는 이름을 지정할수 있으므로, 특정이름으로 지정해놓고 클라가 실행할때마다 해당 이름의 뮤텍스가 존재하는가 확인하는 방식이다.**

(이것도 프로세스간의 동기화느낌으로 볼수있음.)

HANDLE CreateMutexA(

LPSECURITY\_ATTRIBUTES lpMutexAttributes,

BOOL bInitialOwner,

LPCSTR lpName

);

**SECURITY\_SECURITY\_ATTRIBUTES - NULL**

두번째 매개변수는 생성함과 동시에 나의 소유권을 허용하는지를 결정한다.

**TRUE - 생성과 동시에 나에 대한 소유권이 생긴다.**

**FALSE - Mutex의 생성만 이루어진것**

세번째 인자에 이름을 지정하면 다른 프로세스에서 접근이 가능해진다.

CreateMutex호출이 성공하면 뮤텍스 객체의 핸들이 리턴되는데, 이때 Mutex의 이름을 지정했고 같은 이름의 Mutex가 존재한다면 함수호출은 성공되고 GetLastErorr의 반환값으로

**ERROR\_ALREADY\_EXISTS**를 리턴하게 된다.

따라서 CreateMutex호출 후 핸들이 NULL인 경우 GetLastError()를 통해 이미 같은 이름의 Mutex가 존재하는지 확인하는 형태로 클라쪽에서 써먹게된다.

**누군가 이 Mutex의 소유권을 획득했다고 한다면 Non-Signal이므로, Block걸리고 사용하던곳에서 소유권을 포기할때 Waitfor..함수로 signal을 기다리던 쪽에서 깨어난다.**

**이후 Signal을 받아 깨어나면서 즉시 내가 획득하여 non-siganl상태가 되어 리턴된다.**

(만약 아무도 기다리고 있지 않았다면 계속 signal상태로 대기하고 있었을 것.)

**커널객체 동기화 방식은 유저모드의 동기화 방식과는 많이 다르며, 복잡하기 때문에 이를 통해 어떤 자원을 독점적으로 접근하다는 코드가 깔끔하게 떨어지지않는다.**

모든스레드는 할일이 있으면 하고 없으면 쉬어야 한다.

파일입출력의 경우, 내부에서 어떤 I/O를 대상으로 자동으로 blocked가 된다면 (우리가 지금까지 쓴 모든 경우. Select포함) 어떤일을 지정하면 그 일이 수행될수 있을때 알아서 깨기때문에 문제될게 없었다.

그런데 어떤 로직을 짜고 그 스레드에서 일이 주어졌을때 깨어나야하고, 일을 마치고 블락당해야 하는 형태를 깔끔하게 구현할 방법이 없으므로, WaitFor시리즈 함수를 통해 이역할을 담당한다.

(Event의 경우 SetEvenet, ResetEvent로 우리가 제어한다)

**세마포어**

세마포어는 복수 접근이 가능하다.

모든 동기화객체는 획득한 곳(스레드)에서 락을 풀어야 하는데, 세마포어 같은 경우는 다른쪽에서 푸는 것이 가능하다.

이는 락을 거는 느낌보다는 접근카운트를 차단하는 느낌이다.

**ThreadPool**

스레드 풀을 사용하지않는다면 스레드를 1회용으로 사용하여 스레드의 생성/파괴가 빈번해진다.

(스레드 풀은 Event로 만들어 우리가 직접 제어한다.)

따라서 스레드를 미리 생성하여 평소에는 Block상태로 쉬고있다가 일을 해야할때 일을하고, 일이 마무리되면 알아서 Block당하러 가는 것.

결국 크게 루프를 돌면서 WaitforSingleObject를 통해 Signal이 왔을때 깨어나서 일을 하는 구조.

앞으로 만들 모든 스레드는 이러한 구조가 된다.

**Mutex와 Semapore**

우리가 사용할 일은 없지만 면접에서 자주 나오는 질문이므로, 개념과 장단점, 차이점은 반드시 알아 둘것.

또한 추가로, 크리티컬섹션과 SrwLock에 대한 언급이 된다면 매우 좋을 것.

**<7.3>**

**방향성**

**방향성이 있다면 동기화객체를 사용하지않아도 무방하다.**

한쪽에서는 false를 확인해서 뭔가 작업을하고, 완료되면 true.

다른쪽에서는 무조건 false로만 바꿔주는 형태.

우리가 진행한 예제의 g\_shutdown같은 경우도 마찬가지.

TRUE로 바꿀수있는것은 메인스레드밖에 없고 나머지는 계속 확인만 할것이기 떄문에.

**Event**

Win32 Event 객체는 가장 일반적인 형태의 (스레드간 그리고 프로세스간)동기화객체이다.

**Mutex, Semaphore는 signal/non-signal의 방법이 운영체제가 만들어놓은 것으로 정해져있지만, Event는 직접 우리가 제어하는 것이다.**

(가장기본적인 동기화 객체베이스가 이벤트이기 때문에 Mutex, semaphore 역시 내부에서 이벤트를 사용한다.)

이벤트 객체의 기본적인 개념은 조건을 만족시킬때까지 non-signal상태로 대기하며, signal이 올 경우 signal상태로 변한다. 따라서 Mutex/Semaphore보다 훨씬 유연하게 동기화작업을 할수 있을 것.

이벤트는 독점접근 개념보다는 신호의 개념으로 사용한다.

**CreateEvent**

HANDLE CreateEventA(

LPSECURITY\_ATTRIBUTES lpEventAttributes,

BOOL bManualReset,

BOOL bInitialState,

LPCSTR lpName

);

**SetEvent로 Signal을 주고, RestEvent로 non-signal상태를 만든다.**

**LPSECURITY\_ATTRIBUTES lpEventAttributes,**

기본적으로 NULL. 사용할일이 없다.

**BOOL bManualReset,**

**AutoReset은 누군가가 signal을 받아 깨어남과 동시에 non-signal상태로 진입한다.**

(Mutex와 같은 형태이다.)

**ManualReset은 누군가 signal을 받아 깨어나더라도 signal상태를 유지한다.**

**BOOL bInitialState,**

처음부터 signal을 줄것인지에 대한 여부.

TRUE일경우 signal, FALSE일 경우 Non-signal.

**LPCSTR lpName**

이름을 지정한다.

커널 오브젝트 이므로 이름을 지정하고, 같은 이름의 이벤트 객체가 이미 존재하는 경우 GetLastError의 반환값으로 ERROR\_ALREADY\_EXISTS 를 리턴한다.

**이름을 가진 이벤트를 지정했고 같은 이벤트 객체가 이미 존재하는경우 세번째 매개변수인 InitialState매개변수는 무시된다.**

우리는 어차피 커널객체로서 event를 사용하지 않고, 단독적인 프로세스안에서 쓰는 목적으로 쓰는것이기 때문에 큰 의미는 없음.

**SetEvent**

BOOL SetEvent(

HANDLE hEvent

);

SetEvent는 이벤트 객체를 signal상태로 만든다.

Susepended와 Resume의 경우 Count가 0이되어야지만 blocked에서 빠져나와 가동되었다.

**하지만 Event는 Count개념이 아닌 singal이기 때문에 SetEvent를 두번호출했다고 하더라도 signal상태로 변경되는것이 전부이다.**

(SetEvent의 성공여부는 잘 판단하지않음)

**AutoResetEvent에서 SetEvent가 호출된 경우, 해당 이벤트를 기다리고있던 스레드가 있었다면 그중 하나가 해제되고 그와 동시에 자동으로 리셋되어 Non-Signal 상태로 변한다.**

**(단, 대기하고있는 스레드가 하나도 없었다면 이벤트는 이 이벤트를 기다리는 스레드가 생길때까지 signal된 상태로 남게된다.)**

**ManualResetEvent에 이 함수가 출되었을 때 이 이벤트를 대기하고있는 모든 스레드가 대기 상태에서 해제되며, 이벤트 객체는 ResetEvent함수를 이용해서 명시적으로 non-signal로 바꿔줄때까지 signal된 상태로 남는다.**

**ResetEvent**

BOOL ResetEvent(

HANDLE hEvent

);

**ResetEvent는 원하는 시점에 Non-signal상태로 만드는 함수이다.**

**AutoReset같은 경우는 자동으로 signal이 풀리는 옵션을 가졌기 때문에 ResetEvent를 쓸경우 버그를 만들어낼수있다.**

**때문에 ManualReset(수동리셋)의 경우에만 ResetEvent를 사용한다.**

**Event를 사용하는 경우 - 스레드 풀**

워커스레드 입장에서는 언제 일이생길지 모르기때문에, 마냥 기다려야 한다.

**CPU점유율을 차지하면서 계속해서 if문으로 일이왔는지 체크하는 falling방식의 경우 절대 존재해서는 안되기 때문에, 이때 event를 사용한다.**

**이때 manualReset-mode인 경우 SetEvent를 호출해도 깨어난 스레드는 다시 non-signal되지않는다. 따라서 할일을 마치고 ResetEvent를 호출시켜 non-signal상태로 직접 만들어줘야 한다.**

**Event를 통해 srwlock이나 CriticalSection을 만드는것도 가능하지만, 이는 굉장히 번거로운 방법이다. 어차피 크리티컬 섹션도 내부에서는 이런방식. 쓰기편하게 래핑한것뿐이다.**

(누군가가 지금 CriticalSection의 EnterCriticalSection으로 들어왔는데 사용중인경우, 해당 객체에 대해 wait한다. 그리고 leave할떄 signal을 주면서 나를 기다리고 있던 스레드를 하나 깨우는 방식으로 래핑되어 있다.)

**1. 스레드 풀의 형태**

일이 생겼을때 비동기로 처리하기위해 워커스레드를 만들었다고 가정하자.

이때 워커스레드가 하나일 경우 SetEvent로 signal을 줘도 깨울스레드가 없기 떄문에, **n개를 만들어 n개 모두 blocked상태로 일을 기다리고 SetEvent를 통해 signal을 줄때마다 n개중 하나가 신호를 받고 일을 시작하는 상황을 만들고자 한다.**

이는 동시에 병렬로 돌아가는 상황으로, 스레드풀의 형태이다.

(스레드 풀의 가장 기본적인 방법은 Event를 사용하는것)

**2. AutoReset? or ManualReset?**

로직상으로 본다면 n개의 스레드가 waitForSingleObject로 신호를 기다리고 있다.

**(ManualReset-Mode상태에서 SetEvent를 주면 5개의 스레드가 모두 깨어나서 ResetEvent를 해주지않으면 계속해서 signal상태로 대기할것이므로, AutoReset-Mode로 간다.)**

SetEvent 시 대기중인 스레드중 하나만을 깨우고, 나머지 스레드는 깨우지 않는다. 여기서 또 SetEvent를 하면 다음스레드가 일을 진행, 그 다음 또 SetEvent하면 다음스레드가 일을 진행..

**3. 모든스레드가 일을 진행중일때**

이렇게 모든 스레드가 일을 진행하는 상황일때, 다시금 SetEvent를 호출하면 signal상태가 유지된다.(남은 스레드가 없기 때문에)**(??)**

그리고 누군가가 일을 끝내고 돌아왔을때 WaitforSingleObject를 호출하는 순간 signal을 확인하고 Non-Signal로 바꾸면서 바로 다시 작업을 시작할 것이다.

**ManualReset / AutoReset의 쓰임새**

**위와같은 가정에서 알수있듯이 두개이상의 스레드를 대상으로 하나씩만 깨우는 스레드 풀개념으로 간다면 무조건 AutoReset-Mode로 가야한다.**

(일반적인 경우 AutoReset은 잘 사용하지않는다.)

**이와달리 ManualReset은 의도적으로 대기중인 모드스레드를 깨우고자 할때 사용한다.**

**ManualReset은 보통 종료할때 사용하고, AutoReset은 Jop을 던지거나 깨울때 사용한다.**

(이렇게 두가지로 사용하는 스레드의 경우 두개의 이벤트를 기다리는 형태가 될것.)

**ManualReset / AutoReset 개념정리**

전역에 Event객체가 있고, n개의 스레드가 WaitFor...로 Event객체가 Signal되기를 기다린다.

Thread

{

while(1)

{

WaitFor...(Event);

//로직

}  
}

main문에서 SetEvent를 한다.

AutoReset Mode - 기다리던 n개의 스레드중 1개만 깨어난다.

(RestEvent는 사용하지않음. 이벤트객체는 스레드를 하나깨우고, 다시 Non-signaled상태가 된다)

ManualReset Mode - 기다리던 n개의 스레드가 모두 깨어난다.

Reset해줘야만 Event객체가 다시 Non-signaled상태로 돌아간다.

스레드가 시그널, 논시그널 상태가 되는것과

이벤트객체가 시그널, 논시그널되는것은 다른얘기.

스레드 시그널상태 - 현재 Running중.

스레드 논시그널상태 - 현재 Blocked상태

이벤트객체 시그널 - 이벤트객체를 기다리던 스레드가 깨어날수있음.

이벤트객체 논시그널 - 이벤트 객체를 기다리던 스레드가 깨어나지않음.

**스레드 종료 방법**

**위에 소개한것처럼 ManualRestMode의 signal을 활용한 방법이 주로사용되지만, 반드시 이러한 방법을 고집할 필요는 없다**.

종료를 jop으로 던지는 방법도 존재한다.

**Event의 다른 사용처**

스레드 풀외에도 IOCP, windows Server등에서 사용된다.

WINAPI에도 스레드풀이 제공되는데, 내부는 IOCP로 만들어져 있다.

여기서 IOCP는, 소켓모델이 아니라 스레드풀을 말한다. 따라서 IOCP를 스레드풀로 쓰기도한다.

**SetEvent호출시 깨어나는 순서**

SetEvent를 호출했을때, 우리는 이전에 블락당했던 스레드가 다시 깨어나는 것을 바란다.

(방금 사용했던 스레드를 다시 깨운다고 하여 캐시히트가 높아지거나 컨텍스트 스위칭이 낮아지는것은 아니다. 조금이나마 성능향상의 가능성을 높이기 위함)

**하지만 Event는 자신을 기다리고있는 스레드를 리스트로 관리하고 있기 때문에, 철저히 순서를 지켜 Wait한 순서 그대로 나온다.**

(WaitForSingleObject할때마다 이 이벤트뒤에다가 스레드정보를 push,

SetEvent로 signal이 올 경우 가장 첫번째 스레드를 깨운다.)

개발자 입장에서는 안타까울 따름..

(이 순서를 믿고 의존적인 코딩을 하면안되지만, 이 단점을 보완하기위한 장치가 있어야한다.)

이를 개선한게 IOCP. 최근에 썼던 스레드를 다시 깨울수있게끔 제어하는 것이 IOCP.

IOCP는 스레풀이다.

**PulseEvent**

BOOL PulseEvent(

HANDLE hEvent

);

그냥 줬다 다시 뺐는것이다.

ManualReset / AutoReset에 따라 작동방식이 완전히 다르다.



**ManualReset에서의 PulseEvent는,**

**지금 signal을 기다리는 스레드를 모두 깨우고 Non-signal.**

**AutoReset인 경우 PulseEvent는,**

**이벤트를 signal상태로 만들고 singal을 기다리던 스레드 중의 하나를 깨운다. 만약 signal을 기다리는 스레드가 없었다면 Non-signal.**

**결국은 둘다 지금 대기하고있던 스레드를 잠깐 깨우는 역할이다.**

한번 signal줘서 깨웠다가 signal상태로 유지되지 않는다.

(manual은 모든스레드, auto는 하나의 스레드)

**대기중인 스레드에게만 신호를 줘서 어떤 작업을 하는것이 의미가 없기 때문에 애매한 함수이다. 거의 쓸일이 없을 것.**

**Leave(Release) 시 주의할점**

Enter/Leave로 감싼코드에서 분기문이 나오게 될 경우 leave가 늘어나게 될것이다.

이는 코드가 난잡해지고 DeadLock의 확률도 증가하기 때문에 해결해야 하는 상황이다.

**C의 방법으로는 do while를 GOTO문처럼 사용하여 바깥으로 나가면 일괄적으로 leave시켜주는것이 깄겠고**

**C++ 의 방법으로는 클래스의 생성/소멸자의 호출시기를 이용하여 해결하는 방법이다.**

스코프가 닫힐때 소멸자가 호출될 것이므로, 여기에 leave를 심어두는것.

이때 리턴하기전까지 leave하지않으므로 별도의 스코프로 묶어줘야 한다.

이경우 생성/소멸자 호출되는것이기 때문에(함수호출 증가) 성능은 떨어진다.

(소멸자는 인라인처리가 안되기 때문.)

(추가로 가상함수는 인라인처리되지않는다.)

**그러나 안전성을 이유로 이러한 방식을 택하는 것을 권장한다.**

**C++이 계속 발전되면서 생기는 고민**

C++11 등 여러가지 래핑된 기능들이 나오고 있다.

최적화컴파일로 어느정도 성능을 잡아줄수있겠거니 했으나 C++의 여러가지 최신기능(스마트포인터 )을 쓰게되면 최적화컴파일을 아무리해도 성능이 절반도 나오지않는다.

(차라리 C#이 나을정도)

점점더 안전하고 편하게 코딩하게끔 발전되고있지만 성능은 점점 떨어지고 있다.

최적화 컴파일을 하지않을 경우 Select모델보다도 못한 상황이 되어버리고..

이부분은 고민이 필요한 부분.

**Cache miss**

**어떤데이터를 바꾸고자 할때, 내 코어에 있는 캐시메모리는 내가직접 변경하고 들어가지만 다른 코어에 같은 캐시라인에 올려져있는 형태라면 이를 모두 전파시킬 수 없다.**

**다른 코어에 전파시키는것이 더 낭비이므로, 해당 캐시라인에 대해 캐시무효화를 선언한다.**

위와같은 이유로 캐시미스가 난다.

**Cache Line**

어떤 변수가 캐시라인에 걸쳐져있을 경우 두번에 걸쳐 읽어오기 때문에 성능이 떨어진다.

얼마나 성능이 떨어지는 직접 확인해본다.

(생략. 따로해볼것)

**데이터 관점에서의 동기화 객체.**

**동기화객체는 요소마다 존재해야 하는 것을 뜻한다. PlayerList, ItemList, ObjectList… 모두 개별적으로 존재.**

**컨텐츠 관점에서의 동기화 객체**

**Item판매, 퀘스트확인 등 중첩을 계속 걸어야 되는 경우를 말한다.**

**이는 DeadLock이 나올수 있기 때문에 오히려 좋지않은 상황이다.**

무조건 데이터 단위로 동기화객체를 만드는것도 좋지않지만, 너무 세분화 하는 경우에는 관리가 힘들어질 것이다.

**따라서 그룹으로 잘 분리된다면 지역별로 Lock을 거는 등의 방법을 사용한다. 이는 설계에 따라 다르므로 예시들기가 애매한 부분이 있음.**

데이터요소 하나하나 분리하는 식으로 접근하면 게임같이 컨텐츠가 복잡한 상황에서는 적합한 상황이아니다.

==========================================================================

**7.3 과제**

깨우기 위한 목적으로 ThreadEvent와, 종료를위한 exitEvent두개의 이벤트를 만든다.

다수의 WorkerThread는 무한대기하면서 ThreadEvent에 의해 깨어난다.

데이터 STL List를 둔다.

- 과제  
  
        
DATA : STL List < 숫자 >  
  
  
**MainThread**  
  
        특정 키가 눌리면 SaveThread 를 깨운다.  
        특정 키가 눌리면 모든 스레드를 종료 시킴.  
        모든 스레드가 종료 되었는지 확인 후 꺼짐.

MainThread는 단순하게 컨트롤역할이다.

MainThread는 단순히 Loop를 돌게되면 코어하나를 다 먹고가기 때문에, 존재해서는 안되는 상황이다.

**지금 상황이라면 block이 걸리는 키보드Input함수를 써도 무방하다. (루프를 돌릴 이유가 전혀 없기 때문)**

**만약 MainThread에서 주기적으로 뭔가 할일이 있다고한다면 천천히 돌려야한다. (ex Sleep(1))**

**PrintThread**  
  
        1초마다 List 의 데이터를 출력  
  
        1-2-59-20-39-9-192-85-2-385-282-2-3- ....

**DeleteThread**  
  
          333ms 마다 List 의 끝 데이터를 삭제  
  
  
**WorkerThread x 3**  
  
          1초마다 List 에 rand 값 삽입.  
  
  
**SaveThread**  
  
          메인스레드의 이벤트에 의해 깨어나며  
          깨어나면 List 의 내용을 파일 TXT 로 저장한다.  
          (2회 이상 저장시 기존 파일을 지우고 저장한다)  
  
          조건 : Sleep 을 쓰지 않음.

w모드로 저장하면 된다.

파일저장을 하는이유.

읽기는 읽기인데 시간이 많이 걸리는 읽기를 요구한것이다.

과제에서는 어떻게하면 더 좋은 구성이될까 고민을 하는 것을 권장한다.

단순히 구현하는 것은 아무의미없음.

추가적으로 Sleep은 사용할 이유가 없다. 언급한 상황들을 만족하려고 한다면 Sleep이 들어갈 자리가 없을 것.

=============================================================================

이제부터 할내용은. 테스트가 잘안되므로 이론으로 암기해야 하는 내용이다. 지금까지 언급한 내용도 아니다.

동기화객체를 잘 사용했다면 메모리베리어 까지 염두할 상황은 거의없다.

**Meltdown**

Meltdown(:붕괴) 버그란, 커널메모리에 접근되는 것을 말한다.

이는 절대로 가능해서는 안되지만 Out-of-order 에 의하여 가능해진다.

**Out-of-order**

**Out-of-ordering이란 명령어재배치를 말한다.**

지금 CPU의 발전은 단순하게 클럭을 높인다고 끝나지않는다. 현재 수준의 클럭이 나오는 이유는 명령어 재배치가 들어갔기 때문이다.

**어셈블리 코드상에서 mov, cmp, lea, add 모든 명령어들이 그 순서대로 실행되지않는다.**

**이 순서를 지킨다면 비는 공간이 많아진다. 캐시미스가 났을 경우 마냥 기다려야하며, 하이퍼스레딩이라고 하더라도 둘다 캐시미스가 난다면 마냥 기다려야 한다.**

**이러한 단점을, 명령어를 처리하는 과정에서 밑에있는 코드를 더 빨리 처리할수 있다고 한다면 먼저 처리하는 것으로 상쇄시킨다.**

(마치 최적화 컴파일이 최적화하는 것과 비슷하다. OutofOrdering은 CPU차원에서 이러한 일들을 하고있는 것이다.)

**분기예측**

**다음 실행될 조건문이 어떤 곳으로 분기할 것인지를 확실히 알기전에 미리 추측하는 CPU 기술이다.**

CPU는 OutofOrder이외에도 분기예측도 들어간다.

미리 추측하여 분기를 판단하여, 예측에 성공했다면 굉장한 성능의 효과를 볼것이고, 실패한다면 원래하던대로 돌아간다.

**OutofOrdering으로 인한 Meltdown**

**그중에 OutofOrdering같은 경우는 명령어의 순서가 바뀜으로인해서 아직 해당 명령어가 처리되어서는 안되는데 CPU차원에서 먼저 명령어를 처리하는것이다.**

이로인해 커널메모리에 접근이 가능한 Meltdown이 생기는 것.

(이를 구현하는 것은 아직 본인도 아직 구현해보지 못함. 샘플코드를봐도 잘 모름.)

당연히 CPU는 어떤데이터와 연관성이 있는 코드의 순서를 뒤바꾸지는 않는다.

**CPU가 봤을 때 전혀 연관성이 없어보이는 것을 바꾸기 때문에, 우리입장에서는 사실 왠만해서는 걱정할 필요가 없다.**

**하지만 아무상관없는 메모리를 바꿨음에도 다른쪽 스레드에서는 변화가 생길수도있다. 이로인해 미묘한 동기화 이슈가 발생할수도 있다는 얘기가 된다.**

보통은 동기화이슈가 될만한 로직에는 동기화로 Lock을 걸기 때문에 사실 신경안써도 되는 부분이다. 경험하기조차 쉽지않음.

(인터락을 사용해도 메모리 베리어가 쳐지기때문에 경험할일 거의없음)

**Memory Barrier(메모리 베리어)**

**위와 같이 OutofOrder로 Meltdown이 발생하는 것을 막기위해 MemoryBarrier이 등장한다.**

(면접시 물어보는 경우가 있으므로 반드시 기억해둘 것.)

**MemoryBarrier는 명령어 재배치를 무효화시키는 것으로, 마치 컴파일러 최적화입장에서의 volatile과 같다.**

**\_mm\_mfence()**

#include <intrin.h>

\_mm\_mfence();



이 펜스를 기준으로 위와 아래는 섞지 말라는 고전적인 memory barrier명령어이다.

어셈블리 명령어 mfence가 존재하는 것을 확인할 수 있다.

이 명령어를 기준으로 위와 아래를 섞지 않겠다는 명령어 이다.

(위는 위끼리, 아래는 아래끼리 섞일수는 있음)

**\_\_faststorefence()**

\_\_faststorefence()(x64)



뒤에 다시 언급한다.

**\_ReadWriteBarrier()**

\_ReadWriteBarrier();

**WinAPI에도 등장하는 \_ReadWriteBarrier()는 최적화 컴파일시 코드가 섞이지 않도록 하는것으로, CPU차원에서의 OutofOrder를 막는 기능이 아니다.**

**따라서 기계어로서 별도의 코드가 존재하지 않는다.**

(실제로 어셈블리 코드에서 아무런 call도 존재하지않는다.)

**MemoryBarrier라는 논의주제에 같이 등장하지만 CPU차원에서의 OutofOrder와 컴파일 차원에서의 ReadWriteBarrier()는 엄연히 다르다.**

(volatile은 해당 변수를 사용하는 메모리 지점을 최적화컴파일을 하지않는것이고, ReadWriteBarrier는 최적화 컴파일은 하되 선언된 위/아래 코드를 섞지 말라는 것.)

**Msdn에서 ReadWriteBarrier가 아닌 C++의 std::atomic\_thread\_fence()를 권장한다.**

**atomic\_thread\_fence()**

아래는 c++의 atomic\_thread\_fence()에 들어갈수있는 인자를 뜻한다.



<https://en.cppreference.com/w/cpp/atomic/memory_order>

memory\_order의 적정 방법에대해 단계를 define값으로 정의해놓은 것.

**여기서, 이 fence의 수준이 최적화컴파일의 fence인지, 메모리오더의 fence인지 잘 기억해둘것.**

(많은 단계가 정의되어있는데, 내부코드는 이 단계별로 모두 정의되어 있지 않다. 이 개념(Aquire, Release)이 x86아키텍쳐에는 존재하지 않기 때문이다.)

**우리는 mov를 사용하는데, Load/Store(읽기/쓰기)를 사용하는 CPU가 존재한다.**

**Acquire는 CPU입장에서 메모리를 읽어오는 개념(Load),**

**Release는 CPU입장에서 메모리를 저장하는 개념(Store**)

**Acquire베리어를 치게되면 Load명령어가 위에서 아래로 내려오지 못하게 된다.**

(아래에서 위로 올라갈수는 있음)

**획득하여 수행하려는 명령어(Load)에 Acquire베리어를 치는 경우 Load을 완전히 끝내고 다음 명령어를 수행하겠다는 의미이다.**

**Release메모리 디펜스는 아래있는 명령어를 위로 올리지 않는다.**

**데이터를 메모리에 반영할 것이므로, 해당 작업이 끝난다음 다음 명령어를 수행하겠다는 의미이다.**

위와 같은 Load/Store에 대한 명령어가 우리 CPU에는 존재하지않으므로, 명령어도없고 지원하지않음.

**<최근 버전>**



**<수업할 때 본 코드>**



(바뀐점 : \_Guard가 static에서 volatile로 바뀌었음)

Memory\_order\_relaex일 경우 그냥 return하여 빠져나간다.

위 코드에서는 혼동을 피하기위해 MemoryBarrier가 아닌 Compilerbarrier를 사용했다.

(Compilerbarrier는 ReadWriteBarrier를 말함)

**atomic\_thread\_fence에 어떤 인자가 들어오든 Compilerbarrier를 공통으로 호출하고,**

**Memory\_order\_seq\_cst가 인자로 들어왔을때만 어떤 행위를 한다.**

**Memory\_order\_seq\_cst**

atomic\_thread\_fence안에 Memory\_order\_seq\_cst가 들어왔을 때 로직을 살펴본다.

**1. static변수 Guard를 선언한다.**

**2. InterlockedCompareExchange()를 이용하여 Guard가 0일경우 0으로 바꾼다.**

((

최근 다시 바뀐코드

**1. static이 아닌 변수 Guard를 선언한다.**

**2. ~~InterlockedCompareExchange()~~InterlockedIncrement()를 이용하여 Guard가 0일경우 0으로 바꾼다.**

))

이 Guard변수는 이 함수에 소속된 static변수이므로 다른곳에서 건드릴수없다.

따라서 위행동은 코드자체로만 봤을때는 전혀 의미없는 코드로 보인다.

**InterLocked과 OutofOrder**

Interlocked관련된 모든함수에는 도입부에 Memorylock이 들어가고, 이 Memorylock을 기점으로는 OutOfOrdering이 발생할 수 없다.

**따라서 아무의미없는 Interlock을 호출함으로서 Interlock의 Memorylock을 MemoryBarrier기능으로 활용한 것이다.**

**mfence라는 MemoryBarrier기능 명령어가 나왔음에도 불구하고, Interlock이 더 빠르기 때문에(경합이 없다면) 대부분 이러한 방식을 사용한다.**

(InterlockIncrement, decrement 어떤것을 사용해도 상관없음)

하지만 InterLockedCompareExchange의 경우, 전역변수를 대상으로 해당Interlocked함수를 호출하고있다.

**이는 외부에서 atomic\_thread\_fence()를 여러군데서 호출했다면 잠시나마 동기화가 걸리는 모양새가 되므로 문제가 된다.**

**fastStorefence**()



새로나온 기능으로, 기존것보다 더 빠르다.

**lock or dword ptr [rsp], 0**

**rsp(스택 포인터 자체)에 있는 쓸모없는 메모리를 0으로 or연산하고 있다.**

이는 아까 static long \_Guard를 사용하여 InterlockedCompareExchange()하는것보다 훨씬 나을 것이다.(InterlockedCompareExchange()자체도 무거운데 전역변수에 시도를 하고있음)

**Memory-Barrier (Mfence vs InterLock)**

ms포럼에서, mfence를 통해 변수에 접근하는것보다 Interlocked으로 Lock을거는것이 왜 빠른지 질문이 올라온적있다.

ms에서는 이에 대한 답으로, 둘의 목적이 다르므로 성능을 비교하지 말라는 답변을 함.

**(Interlocked이 더 빠르므로 대부분이 InterLocked를 사용한다)**

위와 같은 MemoryBarrier의 기능은 사용하지않더라도 반드시 기억해둘 것.

**MemoryBarrier 적용**

임베디드나 커널쪽 같은 심오한 작업이라면 고려해볼만 하지만, 일반적인 게임서버 컨텐츠쪽에서는 유지보수가 전혀되지않기 때문에 고려대상 자체가 아니다.

**메모리 베리어 추가로 알아둬야할 것**

**Acquire semantics**

**하단부의 load, store 를 상단으로 올라오지 못하도록 한다.**

지금 획득하려는 상황이니 획득후 그 다음을 처리 해야만 한다.

**단, 상단부의 내용은 Acqure semantics 하단으로 내려갈 수 있음**

InterlockedIncrementAcquire ... 등의 함수 역할

**Release Semantics**

**상단부의 load, store 를 하단으로 내려가지 못하도록 한다.**

**지금 저장하려는 것이니 위의 모든 처리 후에 저장을 진행 한다.**

**단, 하단부의 내용은 Release Semantics 이전으로 올라갈 수 있음**

**InterlockedIncrementRelease ... 등의 함수 역할**

**이 내용은 전 플렛폼 동일한 내용이며 기능이 있음.**

**x86 아키텍쳐의 샘플**

**Load A**

**Store B**

**Load C**

**Load D**

**Store E**

**Load F**

**가능한 리오더링 순서**

**Load A**

**Load C**

**Load D**

**Store B**

**Load F**

**Store E**

**Load A**

**Load C**

**Load D**

**Load F**

**Store B**

**Store E**

**Load 뒤로 Load reordered -> x86 불가능**

**Store 뒤로 Load reordered -> x86 불가능**

**Store 뒤로 Store reordered -> x86 불가능**

**Load 뒤로 Store reordered -> x86 가능**

**store 는 load 위로 먼저 처리되지 못함.**

**load 간의 순서, store 간의 순서는 지켜짐.**

**Load 뒤의 Load 를 선처리 -> x86 불가능**

**Store 뒤의 Load 를 선처리 -> x86 불가능**

**CPU 아키텍쳐에 따라서 위의 리오더링이 모두 가능한 경우도 있음**

**x86 의 경우는 위 수준의 정도만 고려하여 작업하면 됨.**

**# 메모리 배리어를 위한 명령어로 mfence , lfence, sfence 외에 API 함수**

**void MemoryBarrier();**

**#ifdef \_AMD64\_**

**#define MemoryBarrier \_\_faststorefence**

**#endif**

**#ifdef \_IA64\_**

**#define MemoryBarrier \_\_mf**

**#endif**

**// x86**

**FORCEINLINE**

**VOID**

**MemoryBarrier (**

**VOID**

**)**

**{**

**LONG Barrier;**

**\_\_asm {**

**xchg Barrier, eax**

**}**

**}**

Lfence, mfence, sfence

mfence는 메모리 자체 fence.

Lfence는 Loadfence.

sfence는 Storefence.

Lfence + sfence = mfence다? 아니다.

셋은 완전히 다르다.

Lfence는 Load에 대한 fence.

Lfence를 했더라도 이전체가 위로 올라가버릴수있음.

Lfence는 모든로드는 코드순서를 바꾸지않기때문에 의미가없음..

그래서 컴파일러에서는 Lfence를 쓰지않는다고한다..

mfence는 해당기점으로 완전한 베리어.

**피터슨 알고리즘**

멀티스레드의 동기화를 잡는 방식

동기화객체 없이.

변수만 사용해서 동기화를 잡아보자는 개념이다.

OOO로 인해서 정상작동되지않는다.

OOO의 문제점을 표현하거나 설명할떄 항상 피터슨 알고리즘이같이나온다.

피터슨 알고리즘은 개념적으로는 완벽한데 OOO때문에 버그가 나오게된다.

따라서 메모리베리어를 치게되면 정상작동된다.

OOO때문에 CPU성능이 향상되었기 때문에.. 개념적으로 OOO는 사용해서는 안된다.

**<7.6>**

**과제 - TimeOut과 Signal을 구분하여 이용**

종료이벤트를 사용하고, CreateThread와 DeleteThread는 TimeOut을 사용한다.

이때 WaitForSingleObject로 깨어났을 때 TimeOut으로 인해 깨어났는지, EventSignal로 인해 깨어났는지 구분해서 종료를할지, 일을 수행할지 구분하면 될것이다.

WorkerThread같은 경우는 Write행위이므로 결국은 Lock이 필요하다. Write/Read가 분리되어있는 상황이므로 당연히 SRWLock을 사용하는 것이 바람직할 것이다.

(삽입 : Exclusive, SAVE,Printf는 Share로 들어간다.)

**사본을 활용하는 방식**

간단한 방법으로는 shared로 Lock걸고 읽으면서 FileWrite를 하는 것이 한가지방법.

더 나아간다면 동기화를 걸고 저장을 시도하는 것이 아니라 데이터 사본을 따로만들수도 있다.

사본만 만드는 시간이 데이터 저장보다 훨씬 빠를것이기 때문에, 동기화를 풀고 사본을 가지고 데이터를 Save한다면 성능상 이점이 있을 것이다. (단 메모리는 당연히 더 많이 쓸 것)

**동기화에서 성능이슈를 해결하기위해 사본을 활용하는 방식은 앞으로도 자주 사용하게 된다.**

클라이언트에서 일반적으로 시도할수 있는 것은 로직과 렌더를 분리시키는것이다.

**ObjList에 Lock을 건다면 로직을 돌리는동안 RenderLock을 걸고, Render가 돌 때 로직Lock을 걸기 때문에 병렬화하는 의미가 없어지므로,**

**Render에서는 사본을가지고 Render하고, 실제 로직에서는 로직을 수행한다.**

굳이 사본을 렌더에서 만들 이유는 없으므로, 로직쪽이 훨씬 가벼운 상황이므로 로직쪽에서 사본까지 만들어 주는 것이 가능하다.

렌더링에 필요한 준비를 모두 완료해놓고 실제 렌더링하는 스레드에서는 Render만을 위한 대상으로 **동기화를 걸고(?)** Render를 할수있을 것이다.

**단, 이는 사본을 만드는 시간과 동기화에 걸리는 시간을 잘 비교분석 한다음 사용해야 한다.**

**멀티스레드를 이용한 기능분리**

멀티스레드로 기능을 분리할때는 거의대부분이 역효과가 난다.

**병렬로 돈다고 해도 불필요한 작업이 들어가며(사본만들기 등), 이걸로 인한 오버헤드가 불필요하게 클수 있으므로 당연히 충분한 비교분석을 통하여 선택하는 것이 바람직하다.**

**현재 예제는 경합의 요소가 List하나뿐이기 떄문에 사본을 만드는 것이 성능상 이득이나, 이외에 스레드가 많아지고 할일도 많아지는 경우 CPU사용률이 굉장히 높아지는 상황에서는 Lock을 건상태로 파일저장을 하는 것이 나을 수도있다.**

(CPU사용률이 높다는 것은 다른로직이 돌지 못한다는 얘기가 된다. 이때 Lock으로 인해 특정 스레드가 쉬게된다면 다른쪽에서 다른일을 진행할 것이다.)

**결국 동기화를 하지않기위해 투자되는 오버헤드가 더 크지는않은지, 전체적인 CPU사용량이 높아 오버헤드로인한 추가작업이 어플리케이션입장에서 손해가될지 모두 고려해 봐야 할 것.**

이런것까지 고려한다고 한다면 상황에맞게끔 설계해야한다.

**멀티스레드에서 성능측정**

**1. 코드의 수행시간**

**2. 초당처리 횟수**

**3. 서버-클라간 응답속도**

등으로 다양하다. **모든것에서 우위를 점하는 방법이란 존재하지않는다.**

코드블럭을 지정하여 수행시간을 측정하는 프로파일러는 일부분일 뿐이다.

**만약 프로파일러로 측정한 수행시간이 점점 느려지는데, 동기화이슈로 인한 느려짐일 경우 얘가 쉬는동안 다른스레드가 돌아서 서버입장에서 처리속도는 더 빨라졌을 수도 있다.**

**채팅서버를 비동기로 설계하고 난다면 초당TPS를 높일것인지, 응답속도를 높일것인지는 상반된 모양새이므로, 적절한 선에서 처리해야 할 것.**

**Actor패턴**

MainThread가 처리하는 것이 부담되어 별도의 스레드에게 일감을 던져주는 것이 Actor패턴이다. (생산자 – 소비자 패턴이라고 하기도 한다.)

이때 일감을 던져줄 때 무엇을 던져줘야할지 애매하다.

위 예제에서는 일감을 던져주는 것이 아니라 아니라 스레드를 컨트롤 했을 뿐이다.

**(Signal을 줬지만 해당 스레드가 이미 일을하고있다면 Signal이 씹혀버린다.)**

JOP을 다른스레드에게 전달하기위해서는 JOP Queue가 나와야한다.

**WorkerThread는 Jop이있다면 일을하고, JopQueue가 비어있다면 Blocked당한 상태로 쉬고있을 것이다.**

ReActor패턴 – Workerthread가 JOP을 진행하고 마무리까지 한다.

ProActor패턴 – Workerthread가 JOP을 진행하고 결과를 알려준다.

OverlappedI/O는 그 구조자체가 Actor패턴이다.

**ProActor패턴의 동기화이슈**

길찾기를 별도의 thread로 뺐다고 가정해보자.

**이때 결과를 받아 마무리하는 ProActor패턴이라면, 메인로직스레드와 동기화이슈가 생긴다.**

**(별도의 thread로 뺀다는 것은 완전히 병렬로 돌수있기 때문에 빼놓은 것이며, 길찾기라는 이 작업 수행과 우리의 로직이 아예 경합발생요소가 없었기 때문에 분리해놓은 것이다.)**

길찾기를 실시하고 완료되면 main thread쪽으로 결과를 알리고, main thread가 마무리해야 한다. 대부분은 일을 요청한 쪽에서 마무리를 해야하는상황이 일반적이다.

**다양한 설계방법**

1. thread를 역할로 구분하고, JOP을 던지는 구조.

2. 별도의 threadPool이 존재하고, 상황에 맞게끔 종류별로 JOP을 던지면 여러 개의 Workerthread가 깨어나서 JOP을 수행한다.

3. 별도의 threadPool이 존재하고 상황에 맞게끔 종류별로 JOP을 던지면 여러 개의 WorkerThread가 깨어나서 JOP을 수행하는 방식이 있다.

**JOP Queue**

Queue는 반드시 나올수밖에 없음.

**게임서버설계**

가장 대중적으로 선호하는 방식은 main thread를 싱글로가고, 필요한 부분을 thread로 따로빼서 병렬처리하는 것이다.

(검은사막, 리니지와 같은 곳은 완전한 멀티스레드로 감)

Mainthread를 싱글로 가는 구조는, thread별로 기능이 정해져있기 때문에 코어를 마냥 늘린다고해서 좋아지는 것은 아니다. 또한 두개이상 늘리는 상황이 설계상 문제가 돌 수 있다.

**Date Base Loading**

statful서버에서 DB로딩은 해당 유저가 내 서버에 접속할때(로그인) 그 유저관련된 데이터를 DB에서 읽어와서 메모리에 올린다.

불가피하게 나오는 한두가지 경우를 제외하고는 컨텐츠 로직상에서 실시간으로 DB를 읽는행위는 최대한 없애야 한다.

**Write/read의 비동기**

**유저가 플레이하는 동안 생기는것들은 모두 쓰기이다.**

**쓰기는 비동기가 쉬우나, 읽기는 비동기가 어렵다.**

(어떤 데이터를 읽고자했는데 아직 읽혀지지않은 경우)

**읽기까지 비동기로 가는 경우는 실제 상용화된 서버에 적용된 사례도 있긴하지만 설계할 때 너무 복잡해진다.**

DB저장(Write)는 결과가 따로 필요없기 때문에 100% 될것이라고 생각하고 진행한다.

이때 만약 저장이 실패했을경우에는 서버를 종료해야 하는 상황이다. 그뒤로 저장되는 데이터가 모두 꼬일것이기 때문이다.

**JOB을 전달하는 방법**

**JOB은 해당 객체에 대한 포인터로 던질경우, 안전성을 보장받을 수 없다.**

**따라서 JOP을 던질때는 저장에 필요한 모든데이터가 들어가야한다.**

이는 우리가 만든 RingBuffer와 같은 구조로, memcpy가 수시로 일어나기 때문에 비효율적이다.

(성능이 떨어지는 것 뿐아니라, 데이터가 쌓일것이기 때문에 용량도 10MB이상이 필요하게된다.)

**이를 개선하기위해, 메시지 잡에도 [header(type, Len) – Data] 형식을 사용한다.**

지금은 이전에 만들었던 링버퍼를 그대로 사용한다.

**링버퍼 동기화이슈**

**이전에도 언급했듯, 각기 다른thread에서 Inqueue와 Dequeue를 한다면 동기화가 필요없다.**

**Inqueue – front(read) / rear(wirte)**

**Dequeue – front(write) / rear(read)**

링버퍼에 이상이 없을경우 동기화이슈는 생기지 않을 것.

**단, 같은일을 하는 여러 개의 WorkerThread가 존재한다면 어쩔수없이 동기화를 걸어야 한다.**

이후 락프리큐로 바꿀것이고, 우리는 임시로 링버퍼를 사용한다.

**링버퍼내부에 동기화객체넣기**

링버퍼내부에 동기화객체를 포함시킨다.

동기화객체는 SRWLock/CriticalSeciton 둘중 뭐를 사용하든 상관없으나, SRWLock이 좀더 가벼우므로 이를 권장한다.(Exclusive, share를 나누기위한 용도가 아니다.)

**1. 내부에서 Lock/unLock하여 사용자는 사용만 하게끔 한다.**

**2. 내부에서는 객체만 포함시키고, 사용자가 이를 사용하여 직접 Lock/unLock한다.**

**2번을 권장하는 이유**

1. 버퍼안에는 JOP에 대한 header와 메시지가 들어가야 하기 때문에, Inqueue가 두번된다.

2. TCP구조에서는 링버퍼를 사용할 때 헤더가 왔는지, 데이터길이만큼 왔는지 매번 확인했으나 지금은 데이터가왔다면 하나의 메시지가 왔다는 전제로 간다.

사실 이 방식은 너무 느리기 때문에 너무많은 고민을 하지는 말 것.

어차피 이런방식으로는 안쓴다 너무느림.

**RingBuffer**

맴버변수로 SRWLOCK 추가

맴버함수로 Lock, Unlock 추가

Enqueue,Dequeue 함수 내부에서 알아서 Lock Unlock 을 해주는 방법도 있지만

사용자에게 맡기는 편이 성능면이나 확장성이 용이함.

단 사용자가 실수해서 개판이 될 수 있음.

- 과제

1. RingBuffer - Lock, Unlock 추가.

2. RingBuffer 를 사용하여 메시지 생성 스레드, 메시지 처리 스레드 테스트 구현

- 메시지 구조

struct st\_MSG\_HEAD

{

short shType;

short shStrLen;

//문자열을 추가하는것이기 때문에 문자열Len이 들어간 것. 헤드뒤에는 문자열이 들어간다.

};

// shType 이 0 (AddStr) 인 경우는 헤더 뒤에 shStrLen 길이만큼 문자열이 들어감.

// shType 이 1,2,3 인 경우는 헤더 뒤에 문자열이 없어도 됨.

- 메시지 타입

#define dfTYPE\_ADD\_STR 0

**#define dfTYPE\_DEL\_STR 1**

#define dfTYPE\_PRINT\_LIST 2

#define dfTYPE\_QUIT 3

지금까지 우리는 스레드 종료시 flag변수를 사용하거나, Event로 signal을 사용했다.

이방법은 컨텐츠적인 입장에서는 안전하지않다. 스레드는 본인의 할일을 모두 하고 스스로 종료하는것이 제일 바람직하다. (DB의경우 100%이렇게 진행되어야 한다.)

**때문에 종료라는 메시지를 전달하면 해당스레드가 종료하게끔 한다.**

//-----------------------------------------------

// 컨텐츠 부, 문자열 리스트

//-----------------------------------------------

list<**w**string> g\_List;

string객체는 회사내규에 따른다. 사용법이 난해하기 때문에 미리 연습겸 익혀둘 것.

//-----------------------------------------------

// 스레드 메시지 큐 (사이즈 넉넉하게 크게 4~5만 바이트)

//-----------------------------------------------

CRingBuffer g\_msgQ;

**@ MainThread**

- 워커스레드 3개 이상 생성.

- 고정 문자열 하나를 둔다.(임의의 문자열을 마구 넣어볼것) ex)"PROCADEMY"

{ while()

1. 메시지 생성

st\_MSG\_HEAD.shType = 랜덤하게 메시지 생성 (0 ~ 2)

st\_MSG\_HEAD.String = 문자열을 랜덤하게 입력

"문자열" < 위 고정 문자열의 범위 내에서 랜덤하게 입력.

이런 데이터를 생성.

2. 스페이스키가 눌렸다면, 1번(메시지 생성) 무시하고 종료 메시지 (TYPE\_QUIT) 넣기.

3. 위에서 만든 메시지 큐에 넣고 워커스레드 깨우기

}

% 메인 스레드는 모든 워커스레드가 끝났다면 멈춘다.

% 메인 스레드의 루프는 50ms 대기시간 으로 개발

% 50ms 대기시간에 이상이 없으면 0ms 대기시간으로 테스트.

**@ WorkerThread (3개 이상)**

- 메인스레드 에서 메시지를 넣지 않았다면 쉬고 있어야 함.

스레드 시작로그 - ID 출력

{ while()

1. 메시지 큐에서 메시지 뽑음.

2. 헤더 타입에 맞게끔

ADD\_STR - 메시지로 온 문자열을 g\_List 에 추가.

DEL\_STR - g\_List 에서 노드 하나 삭제

PRINT\_LIST - g\_List 화면 출력

QUIT - 스레드 종료.

}

@ 실행결과 예시

List:

List:[PRO] [PROCA] [PROCADE]

List:[PRO] [PROCA] [PROCADE]

List:[PRO] [PROCA] [PROCADE]

List:[PROCAD] [PRO]

List:[PROCADEMY] [PROCAD] [PRO]

List:[PROCADEM] [PROCADEMY] [PROCAD]

List:[PROCAD] [PROC] [PROCADEM] [PROCADE]

List:[PROCADEMY] [PROCAD] [PROC] [PROCADEM]

List:[PRO] [PRO] [PROCADEMY] [PROCAD]

List:[PRO] [PRO] [PROCADEMY] [PROCAD]

List:[PROCADE] [PRO] [PRO] [PROCADEMY]

List:[PROCADEM] [PROCADE] [PRO] [PRO] [PROCADEMY]

List:[PROCADEM] [PROCADE] [PRO] [PRO] [PROCADEMY]

List:[PROCAD] [PROC] [PROC] [PROCADEM] [PROCADE] [PRO]

List:[PROCAD] [PRO] [PROCAD] [PROC] [PROC]

List:[PROC] [PROCADEM] [PROCAD] [PRO] [PROCAD] [PROC] [PROC]

List:[PROC] [PROCADEM] [PROCAD] [PRO] [PROCAD] [PROC] [PROC]

List:[PROC] [PROCADEM] [PROCAD] [PRO] [PROCAD]

List:[PROC] [PROCADEM] [PROCAD] [PRO] [PROCAD]

List:[PROCAD] [PROCADEM] [PROCA] [PROC] [PROCADEM] [PROCAD] [PRO] [PROCAD]

List:[PROCAD] [PROCADEM] [PROCA] [PROC] [PROCADEM] [PROCAD] [PRO] [PROCAD]

List:[PROCAD] [PROCADEM] [PROCA] [PROC] [PROCADEM] [PROCAD] [PRO] [PROCAD]

List:[PROC] [PROCAD] [PROCADEM] [PROCA] [PROC] [PROCADEM] [PROCAD] [PRO] [PROCAD]

List:[PROC] [PROCAD] [PROCADEM] [PROCA] [PROC] [PROCADEM] [PROCAD] [PRO] [PROCAD]

List:[PRO] [PROC] [PROCAD] [PROCADEM] [PROCA] [PROC] [PROCADEM] [PROCAD] [PRO] [PROCAD]

List:[PROCA] [PRO] [PROC] [PROCADEMY] [PRO] [PROC] [PROCAD] [PROCADEM] [PROCA] [PROC] [PROCADEM] [PROCAD] [PRO]

List:[PROCA] [PRO] [PROC] [PROCADEMY] [PRO] [PROC] [PROCAD] [PROCADEM] [PROCA] [PROC] [PROCADEM] [PROCAD] [PRO]

List:[PROCADEM] [PRO] [PRO] [PROCA] [PRO] [PROC] [PROCADEMY] [PRO] [PROC] [PROCAD] [PROCADEM] [PROCA] [PROC] [PROCADEM] [PROCAD]

List:[PROCADEM] [PRO] [PRO] [PROCA] [PRO] [PROC] [PROCADEMY] [PRO] [PROC] [PROCAD] [PROCADEM] [PROCA] [PROC] [PROCADEM]

List:[PROCADEM] [PRO] [PRO] [PROCA] [PRO] [PROC] [PROCADEMY] [PRO] [PROC] [PROCAD] [PROCADEM] [PROCA] [PROC] [PROCADEM]

List:[PROC] [PROCADEM] [PROCADEM] [PRO] [PRO] [PROCA] [PRO] [PROC] [PROCADEMY] [PRO] [PROC] [PROCAD] [PROCADEM] [PROCA] [PROC]

List:[PROC] [PROCAD] [PROC] [PROC] [PROCADEM] [PROCADEM] [PRO] [PRO] [PROCA] [PRO] [PROC] [PROCADEMY] [PRO] [PROC] [PROCAD]

# PROCADEMY 문자열 내에서 3자 ~ 9자 까지 랜덤하게 입력

# 메인 스레드는 문자열 리스트 추가 / 리스트에서 삭제 / 리스트 출력 메시지를 생성.

# 3개의 워커스레드는 메시지를 처리.

# 메인스레드는 1초마다 메시지 큐의 빈 공간 / 사용중인 공간 을 출력

List에 삭제/삽입/출력하는 스레드가 존재한다.

이때 list가 계속 늘어나면서 따라잡지 못하고있는데, 이런상황이 된다면 list가 계속해서 커지게되어 printf가 느려져서 정상적으로 잘 돌지 못하게된다.

따라서 add, del을 적절하게 빈도수를 조절해 볼 것.

만약 이 구조에서 list가 다차는경우 방법없음.

**실제로 DB thread에 저장메시지가 쌓이다가 꽉차는 경우 서버를 꺼야한다. 이는 사이즈를 늘린다고 해결될 문제가 아니므로, 계속해서 확인하는 작업이 들어가야한다.**

**따라서 mainthread쪽에서 1초마다 RingBuffer메시지큐에 빈공간or사용중인 공간을 출력한다.**

(지금 설계에서 메시지큐는 절대로 다차서는 안되기 때문이다.)

**다 차는경우 메시지 생산을 낮추던가, 서버 하드웨어 자체를 스케일업, 또는 동접을 낮춰야한다. 뾰족한 수가없음.**

MainThread에서 랜덤시간 간격으로 메시지를 넣는다.

만약 space키가 눌린다면 종료메시지를 넣음과 동시에 mainthread는 루프를 빠져나가고 스레드가 종료되길 기다린다.

**WorkerThread는 일이없으면 block당하고 메시지를 기다리고, 깨우는쪽에서 메시지를 큐잉하면서 WorkerThread를 깨우는 구조로 간다.**

WorkerThread가 끝나면 MainThread는 당연히 멈춘다.

비슷한 예시로, 아무리 서버를 잘만들었다고 해도 DB가 버티지 못한다면 방법이 없다.

DB를 어떻게든 빠르게 만들던, 동접을 낮춰야함.

//qwer

**멀티스레드 설계 소개**

무작정 정형화해서 외우는 것이 아니라, 어떠한 형태인지 감을 익힐 것.

**1.**

**AcceptThread**

Accept thread는 accept만을 한다.

listenSocket은 blockSocket으로 간다.

**I/O WorkerThread**

IO WorkerThread는 데이터에대한 수신완료 통지를 담당한다.

비동기 Send, Recv를 사용하여 데이터를 카피하는 행동들을 백그라운드 커널내부에서 일어나게하고, (ProActor 패턴) 이에대한 완료통지만 받아낸다.

**I/O Workerthread에서 데이터가 모두 받아졌으므로, 받은 데이터를 정리하여 우리쪽 컨텐츠PacketProc로 전달해주는 역할이다.**

**SendThread**

SendThread는 상황에 맞게 간다.

고전적인 IOCP설계에서는 SendDispatchThread또는 SendThread라는게 항시나왔으나, 지금은 이를 사용하지 않는 분위기가 존재하므로 둘다 익혀둔다.

**SendThread는 Send만을 담당한다.**

I/O Workerthread, Sendthread, AcceptThread는 널널한 편이다.

실질적인 모든 Proc는 UpdateThread에서 담당한다.

**UpdateThread**

메인로직이 돌아가는 곳이다.

게임특성상 멀티스레드 환경에서 메시지 처리하기가 애매하기 때문에, 실질적인 모든 로직처리를 한곳에서 하는 것이다.

따라서 모든 부담이 이쪽에서 몰리게된다.

**TimerThread**

게임서버에서는 사용자가 별다른 입력이없어도 월드가 살아있어야 한다.

TimeThread는 Queue에 들어온것들을 받아서 깨어난다.

TimerEvent Proc쪽에서 Event들에 대한 정보를 뽑아 주기적으로 도는 로직들을 담당한다.

이 구조에서는 만약 캐릭터가 이동을 한다면 지점지점을 모두 TimeEvent로 등록하는 형태이다.

**할일이 없으면 쉬고, 있으면 Timer Evenet를 받아 깨어나서 일을 하는 구조로, 언뜻보면 그럴듯해보이지만, 이는 매우느리다는 단점이 있다.**

**메시지를 받아서 던질 때 카피1, 뽑을때도 카피2, Proc에서 로직처리. 또한 Event에 쌓이는 데이터 양이 어마어마하고, TimerThread와 UpdateThread와 동기화 이슈가 발생한다.**

**이를 개선하려면 Copy를 없애고, 동기화이슈를 없애는 형태로 큐를 개량해야 하지만 이 경우 시간에 따른 모든것들을 TImer라는 개념으로 몰아넣는 설계가 문제가된다.**

**(여기서 Timer는 개념을 말하는것으로, Windows에 존재하는 Timer를 말하는게 아니다.)**

**따라서 위와 같은 Timer 설계는 구조만 복잡해지고 좋은 효과를 내지못한다.**

**2.**

**MainThread**

중심에서 모든 로직을 처리한다.

**이는 마치 SelectModel로 서버를 만들때와 같으며, 어마어마한 코드량이 내장되어있다. 모든 부하가 이곳으로 집중된다. (패킷처리포함).**

단, 네트워크 처리는 빠져있다.

**인증용Thread와 로그인전담 Thread**

**특이한 것은 인증용Thread와 , 캐릭터선택이나 로그인에 대한 작업만 전담하는 thread가 따로 존재하여, 먼저 캐릭터 로그인에 대해 정보를 처리한다음에, DB로딩을 다른쪽Thread에서 진행한다.**

**유저가 들어오는 과정에서 게임에 실제로 플레이되는 컨텐츠 로직을 제외한 다른기능들을 다른쪽 thread로 뺀 상태로, 데이터에 대한 부담이 크기때문에 순차적으로 진행하는 방식이다.**

(1)의 구조의 경우 메시지를 별도의 큐에 Inqueue하였는데, (2)의 경우 특이하게도 직접들어가서 뽑아오는 형태를 취한다. (요청과 직접처리의 차이)

이는 falling형태이나, 성능상 차라리 이게낫다.

**DB Save Thread**

DB저장용 스레드들이 따로존재해서, 큐에 DB저장할 데이터가 있다고한다면 이곳으로 쏜다.

**이 같은 구조는 Thread가 여러 개 존재하지만 메인스레드가 하나만 존재하므로 싱글스레드 로직으로 본다.**

**이와 같은 싱글스레드의 구조가 가장 일반적이지만, 듀얼CPU/멀티코어 입장에서는 가장 부담되는 일을 한쪽에 몰아넣은 것이기 때문에 비효율적인 구조이다.**

**싱글스레드 구조**

WorkerThread Pool.

이쪽으로 Recv에 대한 완료통지가 오게된다.(Send는 일단 생략한다)

**본디 IOCP구조는 완료통지가올때마다 각자가 받은 메시지를 PacketProc를 직접 처리하는 구조로, WorkerThread를 늘리고 코어가 늘어날수록 빨라지는 병렬구조이다.**

**유저간의 상호작용이 없는 컨텐츠거나, 웹서버의 경우는 이러한 구조를 따르면 되지만 게임에서는 이와 같은 병렬작업에 대해 동기화이슈가 발생한다.**

이와 같은 이유로 대부분이 메인스레드 하나를 기반으로 두고 가게된다.

**이경우 IOCP WorkerThread는 실질적인 로직을 처리하는 것이 아닌 단순히 IO Thread가 되어 IO에 대한 완료처리만 담당하게 된다.**

이는 비효율적인 구조이다.

**멀티스레드 구조**

하나의 큐에 데이터를 쌓아놓고 여러 개의 UpdateThread가 일을 처리하는 것을 생각할 수있다.

이 경우에 생기는 문제로는,

**1. UpdateThread에 대한 동기화 이슈**

**2. Session을 기준으로 한 처리순서 문제.**

**서버로직을 기준으로 가기 때문에 A유저와 B유저중 어떤유저가 먼저 처리되든 상관없다. 처리순서는 컨텐츠요소로 보는 것이 아니라, 하나의유저(세션)을 대상으로 하는 순서를 뜻한다.**

**한유저에 대한 메시지를 여러스레드가 뽑아서 처리하는 것은 존재할수 없는 구조이다.**

그렇다고해서 유저당 한 스레드를 할당하는것도 불가능하다. (웹서버에서는 가능)

**따라서 이를 해결하고자 한다면 UpdateThread마다 Queue가 나와서, 유저별로 스레드를 지정해서 넣어주는 형태가 되어야 한다.**

이는 마음에 들지 않을수도 있으나 메시지 처리를 멀티스레드 게임서버에서 언급이 되는 구조이다. 개인적으로 이구조를 반대하지만, 주장하는 사람도 존재한다.

**부정적인 쪽 입장**

메시지가 온것에 대해 WorkerThread에서 직접 처리하지않고, 굳이 이것을 별도의 스레드로 던져야하는 이유가 있는가?

직접 처리하는것도 가능하지만, 경합이 많이 발생하므로 스레드를 별도로 빼고, 스레드를 늘리자는 목적인데 이렇게 하다보니 오히려 불필요한 로직만 추가되어 역효과가 나는 것 같다.

**(?)**

**찬성쪽 주장**

WorkerThread자체는 IOThread로, 매우 중요하다.

따라서 IO의 Send, Recv에 대한 완료처리에 우선권을 주고, 해당 WorkerThread는 이 일을 전담하도록 살려두자는 것이다.

이 WorkerThread안에서 직접적으로 프로세싱, 로직을 수행했을 때 로직에 부담이 생기는경우 Send,Recv에 대한 처리에 문제가 생길수도 있다고 주장한다.

**만약 이를 대비하여 UpdateThread를 여러 개 뺐다고 가정한다고 해도, 만약 서버에 부하가 오는 경우라면 의미가 있어보임. (다시 반대쪽 주장)**

NC에 있는 프로그래머랑 열심히 논쟁했지만 결론나지않음. 이것은 직접 프로그램을 짜보고 결정해볼것.

**멀티스레드 구조 동기화**

어쩄든 멀티스레드 형식으로 뭔가의 요청이 컨텐츠쪽으로 들어오고 있다.

이때 발생할 수 있는 동기화는, 최대한 경합이 발생하지 않게끔 하는 것이 원칙이다.

어떤 객체를 쓸때마다 Lock을 건다는 것은 말이안된다.

((

프라우드넷 샘플코드에는, 모든 패킷프로시져에서 직접 처리하는구조에 전역에 크리티컬섹션 하나가 제공되고 Lock을 걸고 들어간다.

[ EnterCriticalSection – 로직 – LeaveCriticalSection. ]

이 구조는 사실상 싱글스레드와 다를바 없는 구조이다.

사람들이 엉망으로 쓰니까 일단은 돌아가는 샘플을 만든것으로, 잘모르겠다면 이렇게라도 써먹써먹으 제공해준 것.

**실제로 이렇게 모든로직을 동기화로 묶을 바에는 Select로 싱글하는게 낫다.**

))

**게임상 컨텐츠에서 물리적으로 떨어져있는 구역이라면 동기화를 분리해도된다.**

**Map별 동기화**

**맵이 따로 분리가되어있다면 병렬로 돌아도 되기떄문에, 동기화의 단위를 맵별로 가도된다.**

당연히 전역적으로 모든 플레이어, 클래스로는 관리가 되지않는다.

이런 요소요소들이 플레이어안에 첨부가 되거나 아니면 어떤 단위형태로 관리가 되어야한다.

**이 경우에는 경합이 발생하는 A맵, B맵에 Lock을걸고 나머지는 병렬로 진행되어도 된다.**

**만약 WorkerThread가 10개라 가정할때 열명의 유저가 같은맵하나에 모두 존재하는 가능성은 크지않을 것이므로, 왠만큼 병렬로 처리될 것이다.**

(지금 게임서버를 멀티서버 구조로 가야하는 상황에서의 설계를 얘기중이다.)

**경합이 발생하는 맵끼리 이동하는경우**

A맵 -> B맵

이동하는 경우, 설계마다 다르겠지만 아무장치없이 그냥 이동시킬수는없다.

**이동의 중간과정으로 전달매개체의 역할을 하는 누군가 나와서 이를통해 전달하는 형태가 가장 일반적인 형태이다.**

**물리적인 분리**

**이와 같은 물리적으로 뭔가 분리가되어있는 컨텐츠 단위로 동기화를 걸어버리는 것은 MO게임류의 방개념에서 적합하겠다.**

방단위로 동기화객체를 가는것이다.

**(맵단위로 스레드가 만들어지는 것이 아니라, 동기화를 맵단위로 걸고 들어간다는 것)**

방이 하나의 인던 느낌으로, 인던단위로 동기화를 걸고 각기 다른 인던끼리는 상호작용이 없다.

**빈번하게 EnterCriticalSection/SRWLock이 호출된다고 해도 경합이 발생하지 않는다면 InterLocked체크 후 빠지기 때문에 상관없다.**

**오픈월드 - Sector**

맵이라는 구분없이 거대한 오픈월드의 경우.

Sector단위로 가는것도 불가능한 것은 아니지만 애매한 부분이 존재한다.

**앞서 언급한 인던/맵 같은 경우는 완전히 분리되어있는 형태지만,**

**Sector는 주변 Sector에게 항상 영향을 주기 때문에 통상적인 Sector가 아닌 컨텐츠상에서 분리개념이 따로 들어가야한다.**

이경우 Sector마다 락을 9개씩 걸어야하기 때문에 Lock이 너무 빈번한 구조가 된다.

**오픈월드 – 구역단위**

**오픈월드 같은 경우 만약에 컨텐츠 상에서만 하나의 맵이고, 플레이어들이 느끼지못하게 구역이 분리되어있다면 그 단위로 가게된다.**

**이는 유저가 뛰어다닐떄는 맵의 전환이 없고 하나의 월드처럼 느껴지지만 서버상에서는 어떤 던전이나 맵을 이동한다는 개념이 들어가는 것이다.**

**이때 구역을 넘나드는 경계에서는 전투가 안되는 등의 컨텐츠 제한이 있을 수 있다.**

(또는 특정 탈것을 타고 이동해야 하는 제한이 걸려있는 등..)

**리니지 서버 / 검은사막 서버**

**검사에서 쓰는방식은 오브젝트단위 동기화를 걸었다.**

**멀티스레드로 동시에 들어가면서 캐릭터단위로 Lock을 걸기 때문에, 캐릭터마다 읽기용락 과 쓰기용 락이 존재한다.**

이런방식으로 실제로 설계가 되어있다. 이런식으로 검사는 설계되어있다.

본 원장은 이 방식에 부정적인 입장으로, 저렇게 한다고해서 성능이 나올지도 의문이고, 이렇게 까지 해야할 이유를 찾지못함.

**리니지 서버**

**리니지 서버는 오브젝트마다 메시지 큐를 활용하는 방식을 사용한다.**

뭔가를 해야할 경우 큐잉하여 병렬로 메시지를 처리하는 방식이다. 이때 경합이 발생되면 한쪽에서 포기하고 빠져나간다.

(이는 마치 tryCriticalSection과 비슷한 느낌)

매우 화려해보이지만, 이렇게 까지해야하는 이유를 아직 찾지는 못함..

**검은사막 서버**

실제 코드를 본적은없으나, 이는 실제 작업하는 개발자도 스레드의 구조는 잘 모른다.

(볼상황이 안되기 때문에 컨텐츠 규칙에 따라 컨텐츠를 짤 뿐이다.)

**[readLock 사본만들기 -> 사본컨텐츠 변경 -> WriteLock걸고 실제 반영]**

이 규칙은 모든 컨텐츠에 해당된다.

이방식은 굉장히 난해하며 일반적이지 않은 방식이다.

이러한 게임서버에서의 멀티스레드 구조는 먼훗날 차차 고민해볼 거리로 두고, 지금은 가장 일반적인 싱글스레드를 공부한다.

**<7.8>**

이전과제 확인

에서 Event는 AuotoReset으로 가야한다. 지금 모든 스레드는 같을 일을 하고 있다.

**헤테로지니어와 호모 지니어스**

**헤테로지니어스 (여러다른종류로 이루어진것)**

**호모지니어스 (동종의, 균질의.. 같은 종류)**

**호모지니어스**

스레드입장으로 마치 서버분산설계 스케일아웃으로 보면된다.

같은일을 하는 스레드를 여러 개가 돌고있는것으로, 이론적으로 스레드 개수를 늘리고 코어개수를 늘리면 늘릴수록 성능이 올라간다.

(더많은 병렬처리)

**헤테로지니어스**

스레드별로 분류하여 사용한다.

게임서버에서 주로 사용되는 방식이다.

게임서버를 만들다 보면 두가지방식이 병행되기도 한다.

우리가 만든 threadQ의 경우 호모지니어스 방식, DB저장과은 헤테로지니어스로 정해진것이다.

**DB저장**

DB저장은 비동기로 이루어지고, 데이터가 바뀐 것을 DB에 저장한다.

따라서 순서가 완벽하게 맞아야한다. 만약 저장스레드가 여러 개인 경우 순서가 꼬이기 때문에 스레드가 하나만 나와야 할것이다.

(두개이상 늘리고자한다면 아예 물리적으로 분리되어 있어야 할것)

**로그인서버**

로그인스레드를 만들어 메시지처리 워커스레드를 만드는경우.

큐잉한 Jop을 별도의 UpdateThread가 뽑아가는 형태가 아니라 수신받는 스레드자체가 여러 개가 되어 일처리를 진행하는 경우, 메시지 순서 보장만 된다면 스레드와 코어를 늘릴수록 성능이 좋아질 것이다.

**게임서버**

게임에서 대부분은 헤테로지니어스의 형태를 띄고있다. (NC나 펄어비스예외)

게임은 상호 경합이 굉장히 많기때문에 호모지니어스로 가고자한다면 굉장히 난해해진다.

용어는 크게 중요하지는않으나 알아서 나쁠것은 없음.

**(이전예제) 종료메시지**

이전예제에서의 종료메시지는 메시지자체를 꺼내지않을수도 있고, 종료메시지 자체를 여러 번을 받는 방법도 있었다.

모든 WorkerThread는 하나의 Signal을 가지고서 메시지를 종료시킨다는 것이기 때문.

이벤트 자체 종료이벤트와 일반메시지 이벤트로 분리되어있다.

또는 메시지를 바깥쪽에서 하나만 던지고 스레드가 직접 종료하면서 넣고 빠지는것도 방법이 될수있다. (내가죽는경우 다른애들역시 연타로)

이런방식도 실제로 IOCP쓸때도 사용할것이다.

**게임서버 성능**

실제 게임서버는 동접수를 최종적으로 판단하는 지표는 굉장히 종합적이다.

**1. 코드차원에서의 성능**

**2. DB저장능력**

내 코드에서는 버텨도 DB가 버티지 못한다면 어쩔수없이 동접자를 낮춰야 한다.

**3. 네트워크 트래픽**

성능이 좋지만 네트워크 트래픽이 너무 과해지는경우, 유지비용과 수지타산을 고민해볼것.

IDC센터의 왠만한 기업하나도 3~40Gb이기 때문에 7000명을 버틴다고 하더라도 서버군마다 트래픽이 4Gb씩 나온다면 회사가 망할지도 모른다.

**7.8과제**

//define

#pragma once

#ifndef \_\_DEFINE\_\_

#define \_\_DEFINE\_\_

struct st\_SESSION

{

int SessionID;

};

struct st\_PLAYER

{

int SessionID;

int Content[3];

};

//11

#define dfTHREAD\_NUM 3

#endif

// Thread\_Debug.cpp : 콘솔 응용 프로그램에 대한 진입점을 정의합니다.

//

#pragma comment(lib,"winmm")

#include "define.h"

#include <windows.h>

#include <process.h>

#include <tchar.h>

#include <list>

//#include <winsock2.h>

//#include <WS2tcpip.h>

using namespace std;

////////////////////////////////////////////////////////

// 컨텐츠 부

////////////////////////////////////////////////////////

////////////////////////////////////////////////////////

// 접속요청 목록.

//

// IOThread 에서 주기적으로 삽입,

// AcceptThread 에서 이 리스트의 값(SessionID 값)을 뽑아서 새로운 Session 을 만든다.

// 이미 존재하는 SessionID 가 나올경우 무시 해야 함

////////////////////////////////////////////////////////

CRITICAL\_SECTION g\_Accept\_cs;

list<DWORD> g\_AcceptPacketList;

//실제로 이런 define방식은 잘 사용하지않는다.

#define LockAccept() EnterCriticalSection(&g\_Accept\_cs)

#define UnlockAccept() LeaveCriticalSection(&g\_Accept\_cs)

//위와같이 define하여 사용하는것은 실제로 사용하는 방식은 아니다.

////////////////////////////////////////////////////////

// 액션 요청 목록.

//

// IOThread 에서 주기적으로 삽입(SessionID),

// UpdateThread 에서는 이 값을 뽑아서 해당 플레이어의 Content + 1 을 한다.

// 존재하지 않는 SessionID 가 나오면 무시해야 함.

////////////////////////////////////////////////////////

CRITICAL\_SECTION g\_Action\_cs;

std::list<DWORD> g\_ActionPacketList;

#define LockAction() EnterCriticalSection(&g\_Action\_cs)

#define UnlockAction() LeaveCriticalSection(&g\_Action\_cs)

////////////////////////////////////////////////////////

// 접속종료 요청 목록

//

// IOThread 에서 주기적으로 삽입,

// AcceptThread 에서 이 리스트의 값을 뽑아서 (SessionID 값) 해당 Session 을 종료 시킨다.

////////////////////////////////////////////////////////

CRITICAL\_SECTION g\_Disconnect\_cs;

list<DWORD> g\_DisconnectPacketList;

#define LockDisconnect() EnterCriticalSection(&g\_Disconnect\_cs)

#define UnlockDisconnect() LeaveCriticalSection(&g\_Disconnect\_cs)

////////////////////////////////////////////////////////

// Session 목록.

//

// 접속이 완료시 (Accept 처리 완료) st\_SESSION 를 동적 생성하여, SessionList 에 포인터를 넣는다.

// 그리고 접속이 끊어질 시 해당 세션을 삭제 한다.

////////////////////////////////////////////////////////

CRITICAL\_SECTION g\_Session\_cs;

list<st\_SESSION\*> g\_SessionList;

#define LockSession() EnterCriticalSection(&g\_Session\_cs)

#define UnlockSession() LeaveCriticalSection(&g\_Session\_cs)

////////////////////////////////////////////////////////

// Player 목록.

//

// Session 이 생성 후, 생성 될때 (Accept 처리 완료시) st\_PLAYER 객체도 함께 생성되어 여기에 등록 된다.

////////////////////////////////////////////////////////

CRITICAL\_SECTION g\_Player\_cs;

list<st\_PLAYER\*> g\_PlayerList;

#define LockPlayer() EnterCriticalSection(&g\_Player\_cs)

#define UnlockPlayer() LeaveCriticalSection(&g\_Player\_cs)

HANDLE g\_hExitThreadEvent;

HANDLE g\_hAcceptThreadEvent;

HANDLE g\_hUpdateThreadEvent;

WCHAR\* g\_szDebug;

//세션을 만드는 함수

void NewSession(DWORD dwSessionID)

{

st\_SESSION\* pSession = new st\_SESSION;

pSession->SessionID = dwSessionID;

LockSession();

g\_SessionList.push\_back(pSession);

UnlockSession();

st\_PLAYER\* pPlayer = new st\_PLAYER;

pPlayer->SessionID = dwSessionID;

//1

memset(pPlayer->Content, 0, sizeof(pPlayer->Content)/\*3\*/);

LockPlayer();

g\_PlayerList.push\_back(pPlayer);

UnlockPlayer();

}

void DeleteSession(DWORD dwSessionID)

{

LockSession();

list<st\_SESSION\*>::iterator SessionIter = g\_SessionList.begin();

for (; SessionIter != g\_SessionList.end(); SessionIter++)

{

if (dwSessionID == (\*SessionIter)->SessionID)

{

//2

//delete \*SessionIter;

//g\_SessionList.erase(SessionIter);

//break;

st\_SESSION\* dSession = (\*SessionIter);

SessionIter = g\_SessionList.erase(SessionIter);

delete dSession;

dSession = nullptr;

break;

}

}

UnlockSession();

LockPlayer();

list<st\_PLAYER\*>::iterator PlayerIter = g\_PlayerList.begin();

for (; PlayerIter != g\_PlayerList.end(); PlayerIter++)

{

if (dwSessionID == (\*PlayerIter)->SessionID)

{

//3

//delete \*PlayerIter;

//g\_PlayerList.erase(PlayerIter);

//return;

st\_PLAYER\* dPlayer = (\*PlayerIter);

PlayerIter = g\_PlayerList.erase(PlayerIter);

delete dPlayer;

dPlayer = nullptr;

break;

}

}

UnlockPlayer();

}

bool FindSessionList(DWORD dwSessionID)

{

BOOL find = false;

LockSession();

std::list<st\_SESSION\*>::iterator SessionIter = g\_SessionList.begin();

for (; SessionIter != g\_SessionList.end(); SessionIter++)

{

if (dwSessionID == (\*SessionIter)->SessionID)

{

//4

//return true;

find = true;

break;

}

}

UnlockSession();

if (find)

return true;

else

return false;

}

/////////////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////////////

// Accept Thread

/////////////////////////////////////////////////////////////////////////////////////

unsigned int WINAPI AcceptThread(LPVOID lpParam)

{

HANDLE hEvent[2] = { g\_hExitThreadEvent, g\_hAcceptThreadEvent };

//AcceptThread내부에서 종료와 Accept두가지 일을 처리하고있기 때문에,

//accceptThreadEvent에서 accept로직처리와 disconnect의 로직처리가 두번들어가는상황이다.

DWORD dwError;

DWORD dwSessionID;

bool bLoop = true;

bool bFindIt = false;

wprintf(L"Accept Thread Start\n");

srand(GetTickCount() + 1);

while (bLoop)

{

dwError = WaitForMultipleObjects(2, hEvent, FALSE, INFINITE);

//Fail이 뜰일은 없을것이나 예외처리한다.

if (dwError == WAIT\_FAILED)

{

wprintf(L"Accept Thread Event Error\n");

wprintf(L"Accept Thread Event Error\n");

wprintf(L"Accept Thread Event Error\n");

wprintf(L"Accept Thread Event Error\n");

break;

}

//Exit에 대한 신호

if (dwError == WAIT\_OBJECT\_0)

break;

//----------------------------------------------------------

//----------------------------------------------------------

// 정상 로직처리

//----------------------------------------------------------

//----------------------------------------------------------

//----------------------------------------------------------

// 접속요청 처리

//----------------------------------------------------------

while (!g\_AcceptPacketList.empty())

{

//추가

if (dwError == WAIT\_OBJECT\_0)

{

bLoop = false;

break;

}

LockAccept();

dwSessionID = \*(g\_AcceptPacketList.begin());

//LockAccept();

g\_AcceptPacketList.pop\_front();

UnlockAccept();

//----------------------------------------------------------

// SessionList 에 이미 존재하는 SessionID 인지 확인. 없는 경우만 등록.

//----------------------------------------------------------

if (!FindSessionList(dwSessionID))

{

NewSession(dwSessionID);

wprintf(L"AcceptThread - New Session[%d]\n", dwSessionID);

}

}

//----------------------------------------------------------

// 접속해제 처리

//----------------------------------------------------------

while (!g\_DisconnectPacketList.empty())

{

//추가

if (dwError == WAIT\_OBJECT\_0)

{

bLoop = false;

break;

}

LockDisconnect();

dwSessionID = \*g\_DisconnectPacketList.begin();

g\_DisconnectPacketList.pop\_front();

UnlockDisconnect();

//----------------------------------------------------------

// SessionList 에 존재하는 SessionID 인지 확인. 있는 경우만 삭제

//----------------------------------------------------------

if (FindSessionList(dwSessionID))

{

DeleteSession(dwSessionID);

wprintf(L"AcceptThread - Delete Session[%d]\n", dwSessionID);

}

}

}

wprintf(L"Accept Thread Exit\n");

return 0;

}

/////////////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////////////

// IO Thread

/////////////////////////////////////////////////////////////////////////////////////

unsigned int WINAPI IOThread(LPVOID lpParam)

{

DWORD dwError;

bool bLoop = true;

int iRand;

DWORD dwSessionID;

srand(GetTickCount() + 2);

wprintf(L"IO Thread Start\n");

while (bLoop)

{

dwError = WaitForSingleObject(g\_hExitThreadEvent, 10);

if (dwError != WAIT\_TIMEOUT)

break;

//----------------------------------------------------------

// 정상 로직처리

//----------------------------------------------------------

iRand = rand() % 3;

dwSessionID = rand() % 5000;

switch (iRand)

{

case 0: // Accept 추가

wsprintf(g\_szDebug, L"# IOThread AcceptPacket Insert [%d] \n", dwSessionID);

LockAccept();

g\_AcceptPacketList.push\_back(dwSessionID);

UnlockAccept();

SetEvent(g\_hAcceptThreadEvent);

break;

case 1: // Disconnect 추가

wsprintf(g\_szDebug, L"# IOThread DisconnetPacket Insert [%d] \n", dwSessionID);

LockDisconnect();

g\_DisconnectPacketList.push\_back(dwSessionID);

UnlockDisconnect();

SetEvent(g\_hAcceptThreadEvent);

break;

case 2: // Action 추가

wsprintf(g\_szDebug, L"# IOThread ActionPacket Insert [%d] \n", dwSessionID);

LockAction();

g\_ActionPacketList.push\_back(dwSessionID);

UnlockAction();

SetEvent(g\_hUpdateThreadEvent);

break;

}

OutputDebugString(g\_szDebug);

}

wprintf(L"IO Thread Exit\n");

return 0;

}

/////////////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////////////

// Update Thread

/////////////////////////////////////////////////////////////////////////////////////

unsigned int WINAPI UpdateThread(LPVOID lpParam)

{

HANDLE hEvent[2] = { g\_hExitThreadEvent, g\_hUpdateThreadEvent };

DWORD dwError;

DWORD dwSessionID;

st\_PLAYER\* pPlayer;

bool bLoop = true;

srand(GetTickCount() + 3);

wprintf(L"Update Thread Start\n");

while (bLoop)

{

dwError = WaitForMultipleObjects(2, hEvent, FALSE, INFINITE);

if (dwError == WAIT\_FAILED)

{

wprintf(L"Update Thread Event Error\n");

wprintf(L"Update Thread Event Error\n");

wprintf(L"Update Thread Event Error\n");

wprintf(L"Update Thread Event Error\n");

break;

}

//Exit에 대한 신호

if (dwError == WAIT\_OBJECT\_0)

bLoop = false;

//----------------------------------------------------------

// 정상 로직처리

//----------------------------------------------------------

//----------------------------------------------------------

// 플레이어 액션 처리

//----------------------------------------------------------

while (!g\_ActionPacketList.empty())

{

LockAction();

dwSessionID = \*g\_ActionPacketList.begin();

g\_ActionPacketList.pop\_front();

UnlockAction();

//----------------------------------------------------------

// PlayerList 에 이미 존재하는 SessionID 인지 확인. 있는 경우만 해당 플레이어 찾아서 + 1

//----------------------------------------------------------

LockPlayer();

list<st\_PLAYER\*>::iterator PlayerIter = g\_PlayerList.begin();

for (; PlayerIter != g\_PlayerList.end(); PlayerIter++)

{

pPlayer = \*PlayerIter;

if (dwSessionID == pPlayer->SessionID)

{

//5

// 컨텐츠 업데이트 - Content 배열마다 + 1 후 출력

for (int iCnt = 0; iCnt < 3;/\*iCnt<=3\*/ iCnt++)

{

++(pPlayer->Content[iCnt]);

}

wprintf(L"UpdateThread - Session[%d] Content[%d] \n", dwSessionID, pPlayer->Content[0]);

break;

}

}

UnlockPlayer();

//6

//UnlockAction();

}

}

wprintf(L"Update Thread Exit\n");

return 0;

}

/////////////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////////////

// Initial

/////////////////////////////////////////////////////////////////////////////////////

void Initial()

{

//7

//g\_szDebug = new WCHAR[40];

g\_szDebug = new WCHAR[48];

//------------------------------------------------

// 각각의 스레드를 깨울 이벤트

//------------------------------------------------

g\_hAcceptThreadEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

g\_hUpdateThreadEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

//------------------------------------------------

// 모든 스레드를 종료 시킬 이벤트

//------------------------------------------------

//8

g\_hExitThreadEvent = CreateEvent(NULL, /\*FALSE\*/TRUE, FALSE, NULL);

InitializeCriticalSection(&g\_Accept\_cs);

InitializeCriticalSection(&g\_Action\_cs);

InitializeCriticalSection(&g\_Disconnect\_cs);

InitializeCriticalSection(&g\_Player\_cs);

InitializeCriticalSection(&g\_Session\_cs);

}

/////////////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////////////

// Release

/////////////////////////////////////////////////////////////////////////////////////

void Release()

{

g\_AcceptPacketList.clear();

g\_ActionPacketList.clear();

g\_DisconnectPacketList.clear();

list<st\_SESSION\*>::iterator SessionIter = g\_SessionList.begin();

while (SessionIter != g\_SessionList.end())

{

//9

//delete \*SessionIter;

//g\_SessionList.erase(SessionIter);

st\_SESSION\* dSession = \*(SessionIter);

SessionIter = g\_SessionList.erase(SessionIter);

delete dSession;

dSession = nullptr;

}

list<st\_PLAYER\*>::iterator PlayerIter = g\_PlayerList.begin();

while (PlayerIter != g\_PlayerList.end())

{

//10

//delete \*PlayerIter;

//g\_PlayerList.erase(PlayerIter);

st\_PLAYER\* dPlayer = \*(PlayerIter);

PlayerIter = g\_PlayerList.erase(PlayerIter);

delete dPlayer;

dPlayer = nullptr;

}

delete[] g\_szDebug;

DeleteCriticalSection(&g\_Accept\_cs);

DeleteCriticalSection(&g\_Action\_cs);

DeleteCriticalSection(&g\_Disconnect\_cs);

DeleteCriticalSection(&g\_Player\_cs);

DeleteCriticalSection(&g\_Session\_cs);

}

/////////////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////////////

// Main

/////////////////////////////////////////////////////////////////////////////////////

int \_tmain(int argc, \_TCHAR\* argv[])

{

HANDLE hIOThread; // 접속요청, 끊기요청, 액션요청 발생. (IO 시뮬레이션)

HANDLE hAcceptThread; // 접속요청, 끊기에 대한 처리

HANDLE hUpdateThread; // 액션요청 처리.

DWORD dwThreadID;

Initial();

//------------------------------------------------

// 스레드 생성.

//------------------------------------------------

hAcceptThread = (HANDLE)\_beginthreadex(NULL, 0, AcceptThread, (LPVOID)0, 0, (unsigned int\*)&dwThreadID);

hIOThread = (HANDLE)\_beginthreadex(NULL, 0, IOThread, (LPVOID)0, 0, (unsigned int\*)&dwThreadID);

hUpdateThread = (HANDLE)\_beginthreadex(NULL, 0, UpdateThread, (LPVOID)0, 0, (unsigned int\*)&dwThreadID);

WCHAR ControlKey;

//------------------------------------------------

// 종료 컨트롤...

//------------------------------------------------

while (1)

{

ControlKey = \_getwch();

if (ControlKey == L'q' || ControlKey == L'Q')

{

//------------------------------------------------

// 종료처리

//------------------------------------------------

SetEvent(g\_hExitThreadEvent);

break;

}

}

//------------------------------------------------

// 스레드 종료 대기

//------------------------------------------------

HANDLE hThread[3] = { hAcceptThread, hIOThread, hUpdateThread };

WaitForMultipleObjects(dfTHREAD\_NUM, hThread, TRUE, INFINITE);

Release();

//------------------------------------------------

// 디버깅용 코드 스레드 정상종료 확인.

//------------------------------------------------

DWORD ExitCode;

wprintf(L"\n\n--- THREAD CHECK LOG -----------------------------\n\n");

GetExitCodeThread(hAcceptThread, &ExitCode);

if (ExitCode != 0)

wprintf(L"error - Accept Thread not exit\n");

GetExitCodeThread(hIOThread, &ExitCode);

if (ExitCode != 0)

wprintf(L"error - IO Thread not exit\n");

GetExitCodeThread(hUpdateThread, &ExitCode);

if (ExitCode != 0)

wprintf(L"error - Update Thread not exit\n");

CloseHandle(hIOThread);

CloseHandle(hAcceptThread);

CloseHandle(hUpdateThread);

return 0;

}

**과제 - 추가설명**

HANDLE hEvent[2] = {g\_hExitThreadEvent, g\_hAcceptThreadEvent};

dwError = WaitForMultipleObjects(2, hEvent, FALSE, INFINITE);

**WaitForMultipleObjects는 signal을 하나씩밖에 결과로 뽑아내지못한다.**

만약 두개이상의 Event가 Singal이 온다면 앞에 위치한것부터 받기 때문에 중요도가 높을수록 앞에 놓아야 할것.

너무 빈번하게 signal 되는 Event가 앞쪽에 배치된다면 항상 해당Event로인해 깨어날것이기 때문에 정작 뒤에있는 Event가 우선순위가 높은데 체크조차 못할수도있다.

**때문에 예제에서는 Accept가 있더라도 exit에 우선순위를 두기위해 앞쪽에 배치하였다.**

이전에 만든 과제는 여러 개의 WorkerThread가 동시에 하나의 큐를 치고들어가는 구조이기 때문에, 큐에 남아있는 공간이 있는지 확실하지않다. (분명히 신호받고 깼는데 없을수도있음)

**지금 예제는 각자 역할이 갖춰진 스레드가 전담해서 할것이기 때문에 signal받았으면 무조건 존재한다.**

이전 예제에서는 Event에 대한 Signal이 올 경우 큐에있는 모든 메시지를 뽑아서 처리했다.

하나만 처리하는 경우 Signal이라는 상태가 중복이 되지않기 때문에 큐에 Jop이 남게된다.

(이러한 스레드풀에 대한 제어와, Jop에대한 컨텐츠적인 부분들에 대한 구조가 마음에 들지않을 수도 있다.)

**스레드 풀에 대한 제어는 OS가 커널쪽에서 하고있고, Jop에 대한 메시지는 우리가 만든것이다.**

**따라서 발생하는 문제로, 이를 위해 IOCP가 워커스레드를 효율적으로 관리하기위해 등장했다.**

IOCP를 사용하면 모든 문제가 해결이 되지만, 무조건적으로 좋은 것은 아니다.(장단점 존재)

**IOCP**

IOCP를 사용하는경우 효율적이고 간단한 처리가 가능해지지만, 큐방식을 써야한다는것에 위험성이 생긴다.

IOCP의 큐는 우리쪽에서 제어할수 없기 때문에 파악할 수 있는 방법이 없다.

큐에 쌓이는 데이터만큼 Non-Paged메모리를 사용하기 때문에, 이를 항상 염두해야 한다.

**과제설명**

일단 하나의 시그널을 받아서 깨어났으면, 루프가 이중으로 들어있는것.

**예제에서 AcceptThread함수는 종료와 accept두가지 일을 처리하고 있다.(로직처리 두번)**

**이를 따로 분리한다면 구조적으로는 나아보일수 있으나, 만드는곳과 삭제하는곳이 같은곳에 배치된 것은 굉장히 좋은 구조로, 동기화이슈를 해결할 수 있는 방법이된다.**

지금예제에서는 Lock걸고 들어가기 때문에 특이점이 없으나 이후 개량해여 Lock을 없애는 구조로 바꿀때는 이러한 구조가 적합해진다.

Disconnect도 마찬가지. 이전에 Select모델할 때 네트워크 루프돌면서 종료처리를 모아서 할할 때 있었다. 이게 멀티스레딩 환경에서도 성능상 이점 때문에 많이 쓰는 방법이다.

(수시로 해제한다면 수시로 동기화가 걸리기 때문에 좋은 방법이 아니다.)

**수시로 Lock/unLock하는것보다 모아서 한번에 하는 것이 훨씬 이득이다.**

모든스레드는 종료하기직전에 AcceptThreadExit라는 이런놈을 찍어내고있기때문에 q를 눌렀을때 찍혀야 한다.

**지난시간 복습**

**IO에 대한 완료통지를 받아낼 Workerthread인 IOThread가 있고, 로직스레드인 UpdateThread가 있다.**

**1. 직접적으로 Recv를 받자마자 컨텐츠 Proc까지 돌리는 방식.**

**2. 별도의 UpdateThread를 두고 여기에 메시지를 전달해 일괄적으로 처리하는 방식.**

**3. UpdateThread 자체를 여러 개로 늘리는 방식.**

그리고 WorkerThread에서 직접적으로 컨텐츠를 치고들어갈 때 동기화의 단위를 어떤방식으로 잡을것이냐도 간략하게 설명했다.

(지역적분할의 경우 맵의 단위로 보는 등..)

**헤테로지니어스 방식**

**1.**

싱글스레드로 가서, 부담이 걸릴 법한 것들을 특정스레드로 빼서 비동기로 병렬처리하는 구조.

**부담이 걸릴만한 기능들을 별도의 스레드로 분리시킨다.**

**다 같은 스레드 로직이나, 컨텐츠 단위로 동기화를 걸면서 들어간다.**

**스레드가 직접 플레이어에게 접근하여 메시지를 처리하는 방식이다.**

**2.**

**유저들끼리 모으는 형태로 스레드를 분리하는것으로, 처음부터 컨텐츠 자체로 스레드를 나눈다.**

이는 컨텐츠 입장으로 봐야한다.

로그인/로비/필드/던전..

**이렇게 컨텐츠별로 나눠진 스레드안에 속한 유저들은 다른쪽 스레드와 전혀 상관이 없게되므로, 동기화자체가 필요없어진다.**

(어떤 방식을 택하든 플레이어를 일괄적으로 관리하는 자료구조는 존재해야한다.)

**각 스레드별로 본인한테 들어와있는 유저들을 관리하며, 하나의 유저는 두개이상의 스레드에 속해있을 수 없다.**

**(2)형태의 문제**

**어떤플레이어가 특정 스레드에서만 계속있다면 동기화가 필요없으나, 유저가 다른 스레드로 이동을 해야한다면 동기화가 필요해진다.**

A스레드 -> B스레드

이때, A스레드에서는 지우고 B스레드에서는 Push해야한다.

이 구조에서는 A스레드에 대한 삽입과 삭제가 다른스레드 에서도 일어날수 있는 상황이므로 A스레드에 Lock을 걸고 B스레드에 Push(변경)해야한다.

**따라서 동기화가 매우 빈번해진다.**

**또한 어떤 행위에서 두개의 리스트락을 건다면 데드락 위험성까지 등장하게 된다.**

**동기화이슈를 어쩔수없다고 여기고 패스하더라도, 데드락이슈를 해결하기 위해서는 내가관리하는 리스트는 나만이 사용한다는 규칙이 있어야 한다.**

이는 다른스레드에서 내 유저들을 건드릴수 없음을 뜻하며, 이 규칙이 있다면 Lock이 필요없어질 것이다.

**따라서 A스레드에서 B스레드로 이동하고싶다면, 밖에서 누군가 직접적으로 건드리는것이 아니라 A스레드에 요청해야 한다.**

이 경우 스레드간의 Player(Data)를 전달시킬 수 있는 매개체가 필요하다.

**스레드간의 Data를 전달시키는 매개체**

(A)스레드 (B)스레드

(A)리스트 - (매개체) - (B)리스트

누군가 리스트에 들어오고 싶다면, 매개체에만 넣고 빠진다.

**이 경우 내 리스트를 Lock하지만 상대 리스트를 Lock하지않는다.**

(누군가 A스레드에서 빠져야한다면, 빼버리고 B리스트에 Lock걸고 큐잉하면 끝이다.)

이방법을 본인은 가장 선호한다.

**스레드마다 본인이 처리할 대상들을 리스트로 관리하고, 삽입과 삭제를 위해 전달 매개체를 통해서 전달받는 형태이다.**

**단점1. 스레드의 한계치**

**단점은 스레드가 정해져있기 때문에 한계치가 존재한다는 것이다.**

코어가 수백개가 늘어난다고해서 성능이 좋아지는 모양은 아니다.

게임서버에서는 이정도 분산설계만 해도 충분하다.(거의 대부분의 게임서버가 이정도도 하지않음)

**단점2. 유저 분포도 불균형**

**컨텐츠별로 스레드를 분산시켰지만, 대부분의 유저들은 필드에 존재한다. 이는 하나의 컨텐츠(스레드)에 몰려있는 형태로 이러한 구조가 의미없고 성능만 떨어뜨린다는 관점이 존재한다.**

(단, 로그인 스레드는 DB로딩과 같은 작업(stateful Server는 로그인시 모든 데이터를 긁어온다)을 수행해야 되기 때문에 어쩔수없이 별도로 존재해야한다.)

**단점3. 필드스레드 순회**

**이 경우 필드스레드에서 계속해서 순회하는 모양새가 마음에 들지않을 수 있다.**

**단점3-1. 메모리 참조오류(crash)**

**List에 대해 Write를 가할때 다른곳에서 사용하고 있을수 있기 때문에 Lock을 걸었으나 중간에 누군가 들어온다고해도 문제가 100% 나는 것은 아니다.**

(List를 어떻게 구현했냐에 따라서 문제발생 유무가 달라지고, STL같은 경우는 내부코드를 정확하게 모르기 때문에 Lock을 건다. 예를들어 Lock을 걸지않아도 Push하는데는 Crash가 나지않을 수도 있다.)

**이때 해당 메모리가 언제나 그 자리에 존재하게끔 하여 메모리참조 오류를 없애고자한다면 배열을 사용하면 된다**.(하지만 이 경우 우리가 찾고자 하는 데이터인지는 확실하지않음)

동접자 MAX 수치를 정하는 이유는 많은 유저들이 접속했을때 원할하게 플레이 할수있는 수치를 정하기 위해서이다. 사람이 없을때 서버가 빠른것은 아무런 의미가 없다.

**따라서 크기가 굳이 길이가 가변적인 리스트로 갈 필요가 없이, 메인 자료구조를 배열로 가자는 것이다.**

배열로 고칠경우 아무나 치고들어가도 crash날 가능성은 없다.

**단점 3-2. 여러스레드의 접근**

crash가 나지않는다고해도, 모든 스레드들이 하나의 배열을 접근한다면 안될것.

**이 스레드 구조는 플레이어가 하나의 스레드에 배정이 된 상태여야 한다는 규칙이 존재했다.**

따라서 본인이 속한 스레드 구조영역을 순회하면서 본인스레드에 소속된 객체들만 처리한다.

**결론**

지금 설명한 방식은 염두해 두되 여러 설계중 하나로 받아들일 것.

데이터는 메모리를 효율적으로 관리하기위해 리스트를 사용한다.

((

1,2를 스레드 입장에서 비교한다면,

**스레드들이 저 플레이어한테 가서 내가처리할 애를 찾아서 처리하는 것과, 공용의 저장고 공간에 내가 처리할게 누군지 순회하면서 확인하고 처리하는것.**

**그전에꺼는 자기가 처리해야할 애들을 자기가 처리하는것.**

))

지금 방식도 동기화를 없앨 수 있다. 변수값 flag를 가지고 (인터락 X) if문으로 값을 변경하게끔.

만약 계속해서 돌고있는 서버가 아니라면 이 구조는 falling방식이므로 적합하지 않을 것이다.

IO Workerthread같은 구조는 네트워크 메시지가 올때마다 스레드가 깨어나서 일을 깨어나서 하는느낌.

물론 서버프레임을 계속해서 올릴수는 없으므로 Sleep을 통해 조건에 맞게 block당해야 한다.

UpdateThread, MsgQ InQueue, IO WorkerThread Recv가 왔을때 일을 처리한다고 해도 루프를 돌리는 스레드는 존재해야한다.

유저가 패킷을 보내지않는다고 하여 서버가 멈추면 안되기 때문에 하나이상의 스레드는 루프를 돌아야 한다.

(지금은 분산설계를 하기위해 컨텐츠별로 스레드를 나눠본것.)

이때 플레이어 배열 인덱스를 순회하는것은 성능에 영향이 거의없으므로 신경쓰지않아도 된다.

**<7.10> - 1**

**<과제>**

단순히 에러를 고치고 끝내지말고, 해당 에러상태에서는 어떤 현상이 일어나는지 정확히 파악하고 알아야 한다.

**실제 MMORPG서버코드는 매우 방대하기때문에, 단순히 확인하는 방법으로는 찾기가 불가능하다.**

**Array - Index**

이 과제에서 만약 자료구조를 List가 아닌 Array로 잡았다고 했을때 Array로 지정한 값보다 더 큰값을 Index로 사용한다면 다른 메모리를 건드리는것이기 때문에 Crash나지않고 엉뚱한값이 수정되어 코드가 꼬일것이고, 이는 굉장히 찾기힘든 오류가 된다.

(ex 또는 클래스의 가상함수 포인터 메모리공간을 건드릴수도 있다)

**동적할당 에러**

**동적할당한 메모리를 잘못사용했을 경우, 해당메모리를 해제(free)할때 예외가 발생한다.**

**이 예외는 malloc 이나 free가 하는것이 아니라 heap이 발생시켜준다.(호출스택에도 나타남)**

**배열 사용**

우리가 실질적으로 관리하는 데이터만 가지고 플레이어를 대상으로 Lock을 없애는것이 목표.

playerlist에서는 List자체(노드)의 오류와 Player(데이터)오류 두가지의 문제가 존재한다.(crash)

이 문제는 배열을 사용한다면 해결된다.

세션은 그대로두고, 플레이어를 array로 바꿔본다.

**list<st\_PLAYER\*> g\_PlayerList;**

**->**

**st\_PLAYER g\_PlayerArray[dfMAX\_PLAYER];**

**배열로 바꿨기 때문에 메모리 참조 오류는 나지않지만, 여러 스레드가 접근하여 Write하기때문에 원하는 데이터가 아닐 수 있다는 문제가 존재한다.**

**동기화 제거**

Update는 실질적인 main스레드이고, Accept함수에서 삽입/삭제를 같이한다.

따라서 Update에서 [n]을 사용중일때 AcceptThread에서의 접근을 막아야 한다.

(삭제만 막으면 해결될 것)

한줄만 내려가도 값이 보장되지않기 때문에, flag값을 통해 동기화하는것은 불가능하다.

**내가 쓰는동안 절대 삭제가 되서는 안되는데 동기화 개념을 넣지 않으려면 결국 삭제의 주체를 바꿔 한곳에서만 사용해야 한다.**

**내가 처리하고있는 대상은 [내 스레드]라는 상태의 변수를 가지고있고, 이 상태인 경우 나만이 Write(변경/삭제)가 가능하다.**

**Player State**

플레이어에 대해서는 Update에서만 권한을 가진다.

이 플레이어에다가 상태를 넣는다.

ON\_STATE\_NONE Empty(Accept 삽입가능)

ON\_STATE\_CONNECT 접속 후 플레이 대기상태 (Accept에서 사용중)

ON\_STATE\_LOGIN 플레이중인 상태 (Update로 넘어온상태)

무조건 단방향으로 설계되어야 한다.

NONE > CONNECT Accept

CONNECT > LOGIN Update

LOGOUT > NONE Update

반복문을 돌면서 Player의 상태가 LOGIN이라면 내가 처리할 대상.

(내 스레드에서 건드릴수 있다고 판단)

**State 누락**

이때 [NONE > CONNECT > LOGIN]과같이 순차적으로 넘어가지않고 AcceptThread에서 곧바로 CONNECT를 거치지않고 LOGIN상태로 바뀐 경우에는 어떤 문제가 있을 것인가?

**이는 예상치 못하는 상황으로, 동기화의 문제는 없을 것이나 설계상 문제이다.**

**미리 준비해야할 작업들(ex 플레이어를 맵에 push, Sector에 알리기, ItemLoading 등)을 하지못하기 때문에 여기서 문제가 발생한다.**

**WaitLogOut변수**

결국 종료 후 플레이어를 삭제하는것은 UpdateThread가 해야한다.

요청은 누구든지 할수 있는 상황이어야 하고, 이는 요청하는쪽에서 전달할 수 있는 방법이 필요하게 된다.

**따라서 WaitLogOut이라는 변수를 두고 [끊어지거나 삭제가 되어야 하는 객체]라고 체크해두는 것이다.**

**최종적으로 NONE상태로 바꾸는 것은 Update가 해줄것이지만, WaitLogOut라는 변수를 건드리는것은 어디에서든지 가능하다.**

(이러한 구조는 일반적인 구조는아님)

**결론**

결국 UpdateThread에서 Lock이 없어졌고, 접속상태를 UpdateThread에서 처리하지만 UpdateThread하단에서 일괄적으로 처리되도록 구현하는것이다.

꼭 게임서버 라이브러리가 아니더라도, 이런느낌의 처리방식이 필요한 때가 있을것이다.

**장점**

**Lock의과정, Lock에대한 오버헤드, 컨텐츠에 대한 오버헤드, 동기화실수 를 모두 제거하고자 하는 의도로 설계했다.**

**(동기화 객체 자체가 나쁘다거나, Acquire, Release, Enter, Leave가 많다고해서 문제가 되는것은 아니다. 중요한것은 경합이 많이 발생되는지에 대한 유무이다.)**

**단점**

사실 최종적인 구조를 보면 루프가 꽤많이 등장하게 된다.

**상단부에서는 내 스레드를 끌어오는 작업의 루프, 중단부에서는 로직에 대한 반복문, 하단부에서는 내 상태에서 나가는애들에 대한 루프로 스레드당 세개의 루프가 돌게된다.**

(진입 – 처리 – 나가기) 와같은 설계.

**기타 다른로직**

**NewSession**

**여기서 다 뒤지면서 Player에 대한 상황을 넣기위해 비어있는 State를 찾아 이에 대한 셋팅을 한다. 그리고 OnStateConnect를 변경하고 언제든지 변경을 할것이다.**

**이 변경은 항시 최하단부에서 이루어지고, 끝까지 반복문 로직을 돌게되므로 이를 개선하려면 비어있는 공간을 따로 관리하면 될것이다.**

플레이어가 나갈때마다 그자리에 Index를 별도 공간에 보관하고, 그냥 뽑아서 쓰면 되는것.

(Stack과 같은 자료구조에 보관)

**Stack vs Queue**

자료구조 차원에서 스택이 더 가벼우나, 잘못 만들어졌을때 버그가 많이 나게된다.

큐는 이전에 썼던것을 재사용하지않기 때문에 버그가 거의 나지않는다.

**성능상은 자료구조 차원에서도, 캐시로봐도 스택이 더 좋다.**

**DeleteSession**

DeleteSession에서 지워야 하는데 이곳에서 함부러 지울수없다.

**accept스레드에서 확인후에 호출이 되는데, 여기서는 지울수있는 권한이 없기 때문에 flag만 체크한다. 여기서 지워버리면 Update에서 쓰고있을지도 모르기 때문에 위험하기 때문이다.**

이 스레드에서 삭제하는 로직까지 포함시킨다면

**1. 상태끌어오기 2. 원래 하던일(Send/Recv등 서버IO처리) 3. 삭제하는 로직**

으로, 일이 매우 많아지게 된다.

**따라서 삭제하는 스레드를 별도로 생성하여 가는것도 나쁘지않음.**

별로 할게없다면 빼지않는게 낫다.

**별도의 DeleteSession마련**

**별도의 DeleteSession을 만들어 사용할 것이라면,**

**체크된 객체를 마냥 삭제하는 것이 아니라 별도의 상태를 마련해야 할것이다.**

현재는 UpdateThread가 Player를 단독으로 쓰기 때문에 직접 삭제해도 상관없지만,

플레그는 셋팅이 됐는데 따로 스레드가 나왔다면 다른쪽에서 사용중일 수 있다.

**Release라는 Thread가 나왔다면 Release라는 상태가 나와야하며, flagLogOut을 확인해서 전이시켜 주면 해당스레드에서 알아서 처리하도록 한다.**

(단, Release라는 작업의 부하가 크다면 그런 절차를 넣어줄만 하다. 걔 자체도 별도의 스레드로 빼버리는것.)

**UpdateThread**

이전과 같은 형태. 모든상황 체크할때마다 상태체크, 그리고 걔가 맞는지 체크한다.

실질적으로 삭제하는파트이다.

Connect상태면 UpdateThread로 보내고 이를 UpdateThread가 받아들인다.

그래서 필요한 초기화 작업을 완료한 후 DB로딩이 아니라 로딩된 데이터를 셋팅하는 것.

컨텐츠 작업들을 UpdateThread에서 진행한다.

**추가언급**

이러한 설계는 동기화에 비해서 불필요한 부분이 많이 들어가긴하지만, 사실 If문 for문이 몇천번 들어가는 것은 성능입장에서 아예 느껴지지않는다.

만약 서버로직에서 UpdateThread 프레임자체를 제한없이 4~5천번 돌린다고하면 CPU를 약간 차지할 것이나, 실제 프레임은 보통 빨라야 100, FPS같은 경우도 4~50이면 충분하기 때문에 이를 문제삼을 필요는 없다.

===============================================