경험한 이슈 중, 기억할만한 이슈를 정리한 문서입니다.

1. 락프리 자료구조에서 Hazard Pointer가 필요한 이유

- 락프리는 기본적으로 CAS를 통해 구현된다.
- CAS는 보통 CAS(Now, LocalNow, Next)와 같이, Now와 LocalNow가 같다면 Now를 Next로 이동시키는 로직이다.
- 멀티스레드 프로그램에서는 스레드 A가 CAS를 시도하는 순간, 컨텍스트 스위칭으로 인해 스레드 B가, 스레드 A가 접근하려고 하는 노드를 제거할 수 있다.
- 즉, 순식간에 LocalNow는 동적 해제된 메모리가 되는 것이다.
- 동적 해제된 메모리에 접근하게 되면, 무슨 일이 발생할 지 모른다. 보통은 힙 침범으로 프로그램이 종료된다.
- 이를 방지하기 위해서는, 해당 락프리 자료구조만 접근하는 메모리가 필요하다. 이게 Hazard Pointer다.
- 직접 제작한 LockFree-Queue / LockFree-Stack은 메모리풀을 사용했다.

2. CAS의 문제 (ABA Problem)

- Hazard Pointer를 넣은 LockFree-Queue 완성 후, 테스트를 돌려봤다.

> **테스트 상황**
> 1) main스레드에서 Queue에 UINT64를 인자로 같는 Class 4개를 넣어두고 스레드 2개 생성
> 2) 각 스레드는 [디큐 -> 디큐 -> 인큐 -> 인큐] 를 루프돌며, 중간에 클래스 내부의 UINT64값을 ++, --시키며 값이 그대로인가 확인. 다르면 Crash를 발생시킨다.

- 테스트 결과, UINT64의 값이 달라서 Crash가 발생한다.
- 확인해보니, 정말 동일한 데이터에 2개의 스레드가 접근했다. 즉, Deq가 되면 안되는 상황에 Deq가 발생한 것이다.
- 강의시간에 배운 ABA문제가 발생한 것으로 판단했다.

> **ABA 문제**
> - Head/Tail/Top 등이 내부 리스트 외부의 메모리를 가리키는 문제
> - 스레드 A가 디큐에서 CAS를 통과하려고 하는 찰나에 스레드 B와 C가 디큐/인큐를 반복한다보면 LocalNext가 메모리풀에 들어가는 상황이 발생한다.
> - 그 시점에 스레드 A가 깨어나서 아까 하려던 CAS를 통과하면 Head는 메모리풀에 있는 HeadNext를 가리킨다.

- 문제는 일반적인 CAS로는 노드의 유니크함이 보장되지 않는다는 것.
- 포인터 외의 다른 것이 필요하다고 생각되어 Unique Count 값을 추가했다.
- Unique Count라는 개념은, 직접 제작한 네트워크 모듈의 SessionID를 제작할 때도 사용하기 때문에 쉽게 떠올릴 수 있었다.

> **Unique Count**
> - 디큐/인큐를 성공할 때 마다 1씩 증가되는 값.
> - CAS 시 해당 값도 같이 확인해 head와 localhead가 정말로 동일한지 한번 더 검증한다.

- 일반 CAS는 InterlockedCompareExchnage64를 사용하는데, 이걸로는 head와 Unique Count를 한번에 비교할 수 없었다. 이 함수는 8바이트만 비교/교환이 가능하다.
- MSDN을 돌아다니다가 InterlockedCompareExchnage64 페이지에서 연관 문서로 InterlockedCompareExchnage128을 찾았다.
- 설명을 보니, 16바이트의 데이터를 CAS하는 함수다.
- 이걸로 일단 ABA 문제는 막았다.

3. Double CAS의 주의점

- Double CAS코드에서 메모리 침범이 발생했다.
- 처음엔 128이 잘못된 줄 알고 코드를 회귀시켰는데 문제는 메모리 정렬 때문이었다.
- 참고로 MSDN에 아주 자세히 적혀있다. 이걸 못 본 내 자신을 원망했다.

The parameters for this function must be aligned on a 16-byte boundary; otherwise, the function will behave unpredictably on x64 systems. See **_aligned_malloc**.

- 즉, 16바이트(128bit)로 정렬된 메모리를 사용하라는 것이다.
- 잊어버리지 않게 메모리 정렬에 대해 간단히 정리해 둔다.

> **메모리 정렬**
> - 기본적으로 CPU는 word단위(64비트 아키텍쳐는 8바이트)로 처리한다.
> - 레지스터에 데이터를 가져오거나 ALU가 처리하는 단위도 word 이다.
> - 때문에, 대부분의 변수는 8바이트(64비트 기준) 경계로 메모리가 할당된다. 그래야 메모리에 접근할 때, 한 번의 작업으로 Read/Write가 가능하다.
> - alignas() 등을 이용해 메모리 정렬을 변경할 수 있다.

- 하지만 이 문제는 단순히 메모리 정렬의 문제가 아니었다. 복합적인 문제였다. 문제 내용을 정리해본다.

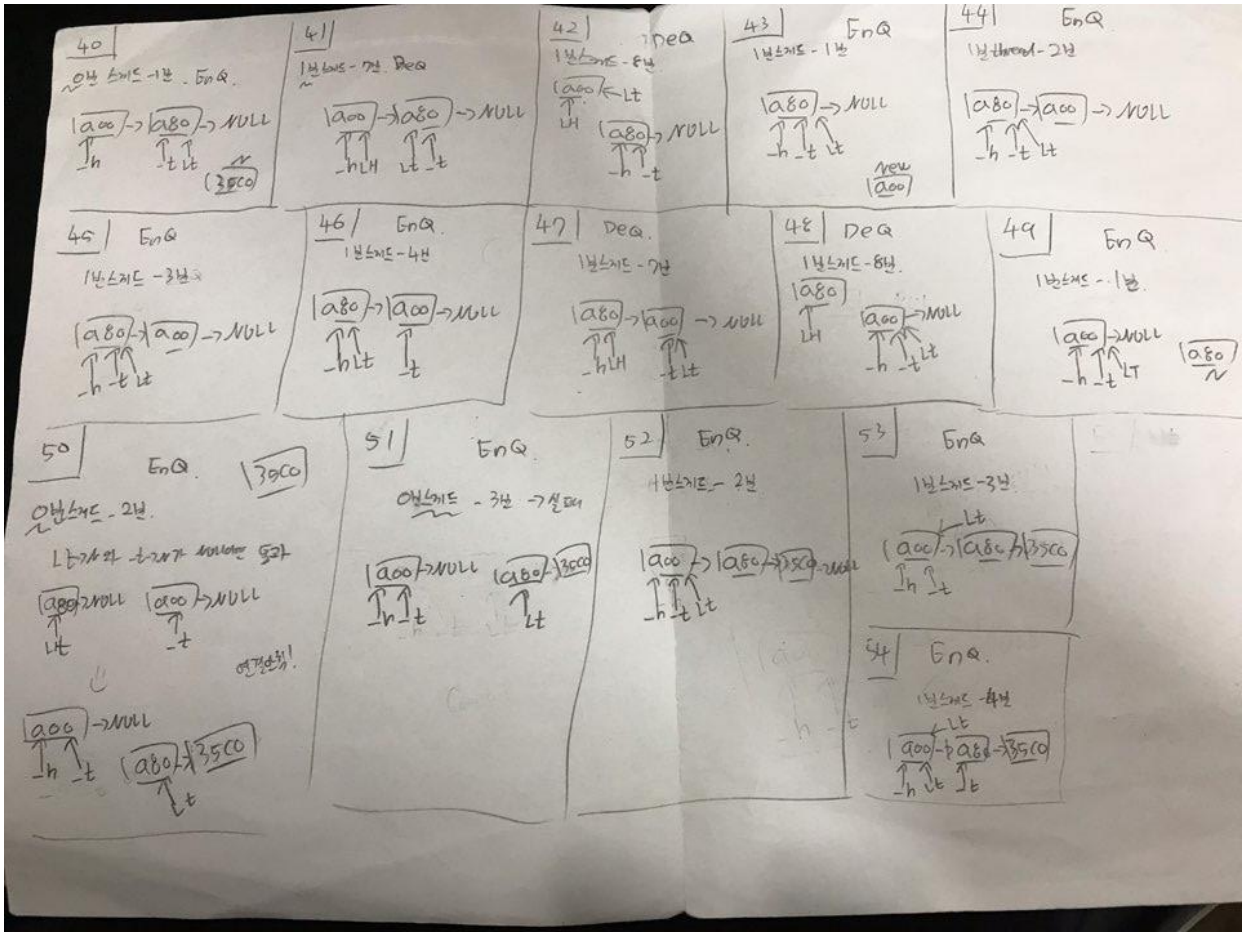
- 먼저 인터락 함수의 특징에 대해 정리한다. **매우 중요한 특징이다!**

> **인터락 함수의 특징**
> - 인자로 받은 변수가 존재하는, 실제 물리 메모리에 접근한다.
> - 예를 들어, 인터락 32함수는 인자의 시작 주소부터 4바이트의 물리 메모리 영역에 접근한다.
> - 인터락 64함수는 인자로부터 8바이트의 물리 메모리 영역에 접근한다.

- 즉, 잘못 쓰면 메모리 침범이 일어날 수 있다.
- 가상메모리 측면에서도 생각해보자.
- int Array[2]는 가상메모리 주소상에는 연속된 위치에 존재한다. (예를 들어 0x04, 0x08)
- 하지만, 이건 가상 메모리의 입장이며 실제 물리 메모리 상에는 떨어져 있을 가능성이 있다. (예를 들어 0x04, 0x40)
- InterlockedCompareExchnage128의 인자로 0x04를 넘길 경우, 0x04부터 128비트의 물리 메모리에 접근한다.
- 이 때, 8바이트는 0x04로부터 8바이트에 있지만, 나머지 8바이트가 물리메모리에 연속적으로 있다는 확신은 없다.
- 위 문제들 때문에 Double CAS에서 문제가 발생한 것이었다.
- 앞으로 MSDN을 잘 읽자...

4. Head의 Next가 null이 되는 상황

- 직접 제작한 LockFree-Queue는, Head가 가리키는 것은 Dummy이며, Head의 Next가 실제 데이터이다.
- 이는 Head와 Tail이 Null이 되는 상황을 방지하기 위함이다. (Head가 가리키는 데이터를 디큐할 경우, 데이터가 0개가 되는 순간 Head와 Tail은 null이다.)
- 때문에 Dequeue함수에서는 Head의 Next가 null인지 체크하고 null이라면 Crash가 나도록 코드를 작성했다.
- Double CAS를 하기 전에 인터락으로 내부에 노드가 있는지 확인하기 때문에 Head의 Next가 null이 되는 상황은 없다고 판단했다.
- 하지만 그 상황이 발생했다.
- 노드의 수가 적을 때 인큐/디큐를 반복하면 자주 발생했다.
- 상황을 추적하기 위해, 락프리 자료구조 내부에 리스트를 두고, 인큐/디큐하는 모든 노드의 주소를 보관하며 메모리 로깅을 시작했다.
- 얻은 덤프파일을 종이에 그려가며 추적해봤다.



- 정리하자면
  1. 0번 스레드가 Enq를 시도하는 중 컨텍스트 스위칭
  2. 1번 스레드가 디큐->인큐->디큐 성공 후, Enq를 시도하는 중 컨텍스트 스위칭
  3. 다시 0번 스레드가 Enq를 성공한 후 Tail을 이동시키지 못하면 Head의 Next는 Null이된다.
  4. 이 때 0번 스레드가 디큐를 시도하면 Head의 next가 null이기 때문에 Crash가 발생한다.
- 위 상황에서, Head의 Next가 null인 상황은 언젠가 해소된다. 인큐 중 순식간에 다른 스레드가 디큐를 시도해 발생한 상황이기 때문이다.
- 강사님과 이야기한 결과, 발생할 수 있는 상황으로 간주하고 Head의 Next가 null일 경우 continue를 하는 걸로 이슈는 종결되었다.

5. 락프리와 SRWLOCK 성능 비교

- Slim Read/Write LOCK이라는게 있다.
- 가벼운 락인데, Shared모드와 Exclusive 모드가 제공된다.
- 잊어버리기 전에 SRWLOCK에 대해 정리한다.

> **SRWLOCK**
> - 락 소유주를 체크하지 않는다. (셀프 데드락 위험)
> - 객체 크기가 8바이트 포인터로 굉장히 작다.
> - 임계영역 접근 시, bit 값을 변경하는게 끝이다. (물론, 누가 임계영역에 들어가 있으면, 다른 락과 마찬가지로 커널모드 전환 후 Event를 기다리는 형식이다)
> - 전체적으로 굉장히 가볍고 빠르다.

- 그렇다면, 정말 SRWLOCK는 락프리보다 빠를까?

Profiling.txt - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말

```
================================================================================
    ThreadID      |                  Name |    Average |        Min |        Max |       Call |
--------------------------------------------------------------------------------------------------
        8740      |            Interlock |64503397.700㎲ |64503397.700㎲ |64503397.700㎲ |          2 |
        8740      |               srwlock |10501700.200㎲ |10501700.200㎲ |10501700.200㎲ |          2 |
================================================================================
```

- SRWLOCK과 인터락 비교 결과이다.
- 스레드 50개가 1천만번 InterlockedIncrement를 한 것과 SRWLOCK을 걸고 값을 ++ 한 것이다.
- SRWLCOK이 월등하게 빠르다. 정확하게 빠른 원인이 궁금해 좀 더 고민해봤다.

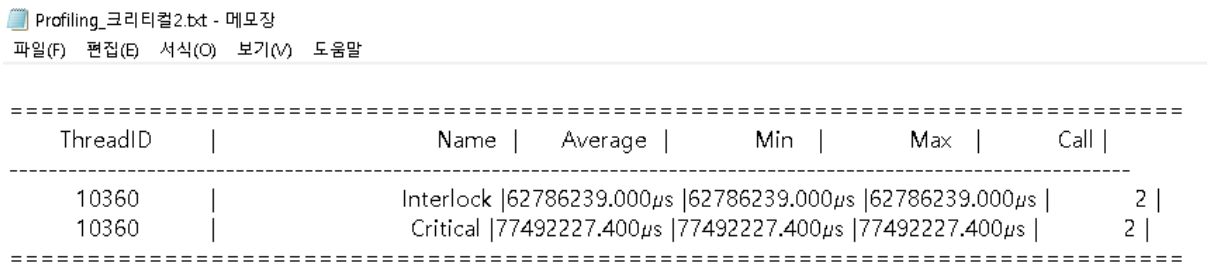
- 이것 저것 고민하다가 나온 결론은 **캐시라인** 때문이다.

> **캐시라인**
> - CPU가 메모리에서 캐시로 값을 가져오거나 메모리로 값을 보낼 때, 캐시라인 크기로 가져오고 보낸다. 이는 공간지역성을 최대한 활용하기 위함이다.
> - 1바이트만 Read/Write해도 캐시라인 크기로 읽고 쓴다.
> - 최근 Intel CPU는 L1/L2/L3 캐시라인이 64바이트이다.

- 그리고 캐시에는, 캐시무효화라는 중요한 규칙이 있는데 이것도 기억해둬야 한다.

> **캐시 무효화**
> - 캐시는 일관성이 보장되어야 한다. (cache coherency)
> - 만약, 코어A/B의 캐시가 각각 int q의 값을 가지고 있는 상태에서, 코어 A가 q의 값을 변조할 경우, 코어 B에게 해당 값이 있는 캐시라인이 틀어졌다는 것이 전달된다. (MESI 프로토콜)
> - 이후, 코어 B가 q에 접근하려고 할 때, Cache miss가 발생하며, 메인 메모리에서 새로 데이터를 캐싱한다.

- 이걸 기억해두고 락프리를 분석해보자. 일단 락프리의 CAS는 인터락이며, 인터락도 결국 값을 변조하는 것이다.
- 때문에 CAS 성공 시 캐시 무효화가 발생한다. 이 상태에서 다른 코어의 스레드가 접근하지 않는다면 메모리에서 캐시로 데이터를 캐싱하는 작업은 발생하지 않는다.
- 하지만, 락프리는 기본적으로 n개의 스레드가 경쟁해서 최소 1명이 승리하는 구조이다. 그리고 **패배한 스레드들은 즉시 다시 CAS를 시도한다.**
- 때문에, 매 번 캐시 라인을 갱신하는 오버헤드가 발생한다.
- 반면, SRWLOCK은 내가 건 락을 포기하기 전 까지는 다른 스레드가 공유자원에 접근할 수 없다.
- 락을 건 후 내부에서 값을 1억번 변경해도 캐시무효화가 발생하지 않는다. 즉, 내가 락을 풀고 다른 스레드가 그 공유자원에 접근했을 때만, 캐시 무효화가 발생한다.
- 결론적으로 SRWLOCK은 락프리보다 캐시 무효화 발생 수가 적다. 때문에 락프리보다 빨라질 수 있다. (상황에 따라 다르니, 가능성이 있다고 기억해두자)
- 여기서 의문점이 생긴다. 그럼, 모든 락이 락프리보다 빨라질 수 있는가?
- 그 예로, Critical Section도 락프리보다 빠를까?


Profiling_크리티컬2.txt - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말

ThreadID	Name	Average	Min	Max	Call
10360	Interlock	62786239.000㎲	62786239.000㎲	62786239.000㎲	2
10360	Critical	77492227.400㎲	77492227.400㎲	77492227.400㎲	2

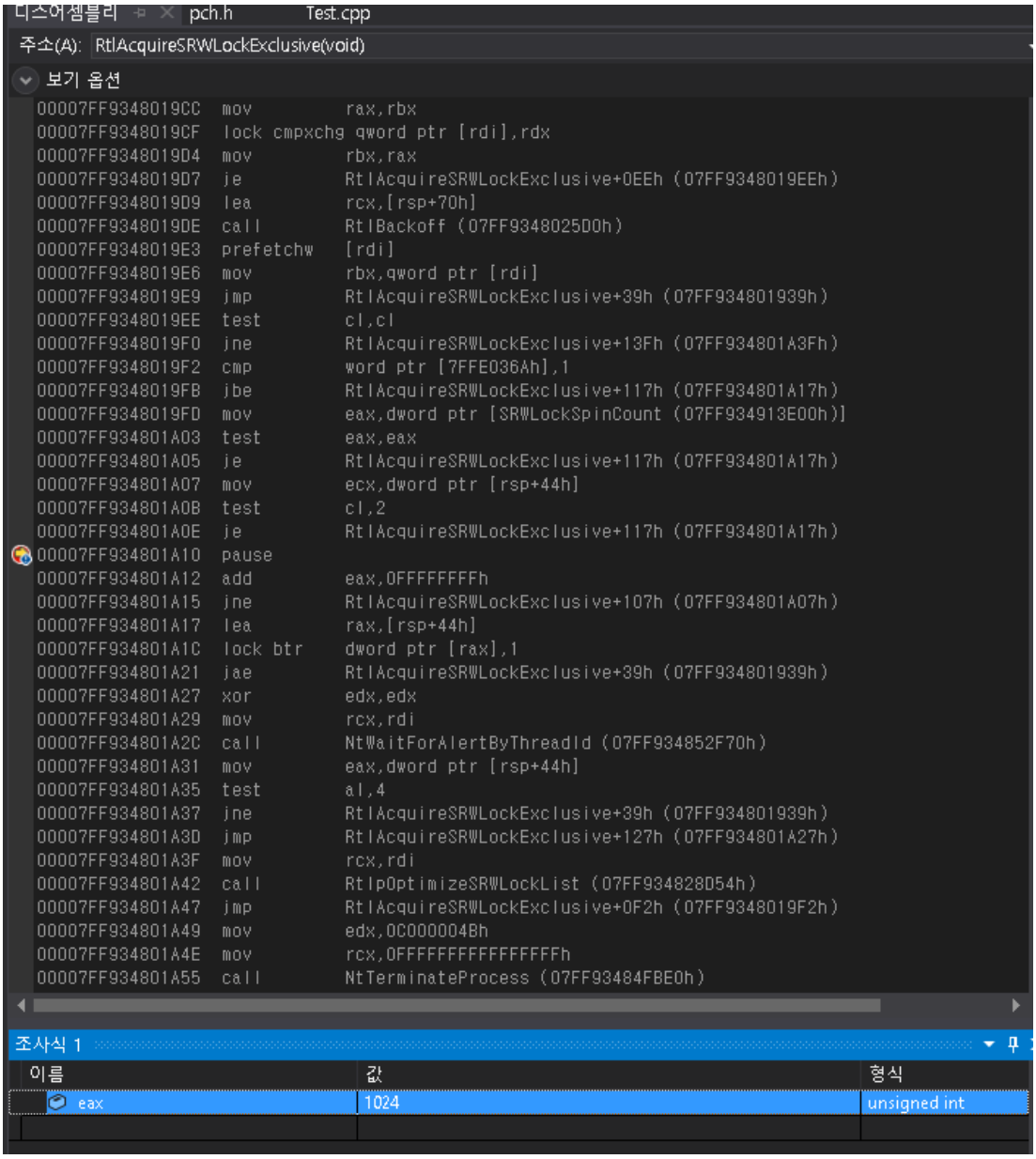
- Critical Section과 인터락 비교 결과이다. (SRW와 동일한 조건으로 테스트) 결과는 인터락이 더 빠르다.
- Critical Section은 SRWLOCK보다 하는 작업이 많고 무겁다.
- 때문에 Critical Section은, 인터락보다 느려질 수 있다.
- 주의해야 할 점은 락프리는 그냥 락이 없다는 것뿐 빠른 것은 아니라는 것이다.
- 상황에 따라 잘 사용해야 한다. 실제로 SRWLOCK이 훨씬 빠르다.

6. SRWLOCK이 빠른 이유는 Pause?

- 수업 중, 어셈 명령어 Pause에 대해 알게 됐다. SRWLOCK이 빠른 이유 중 하나라고 한다.

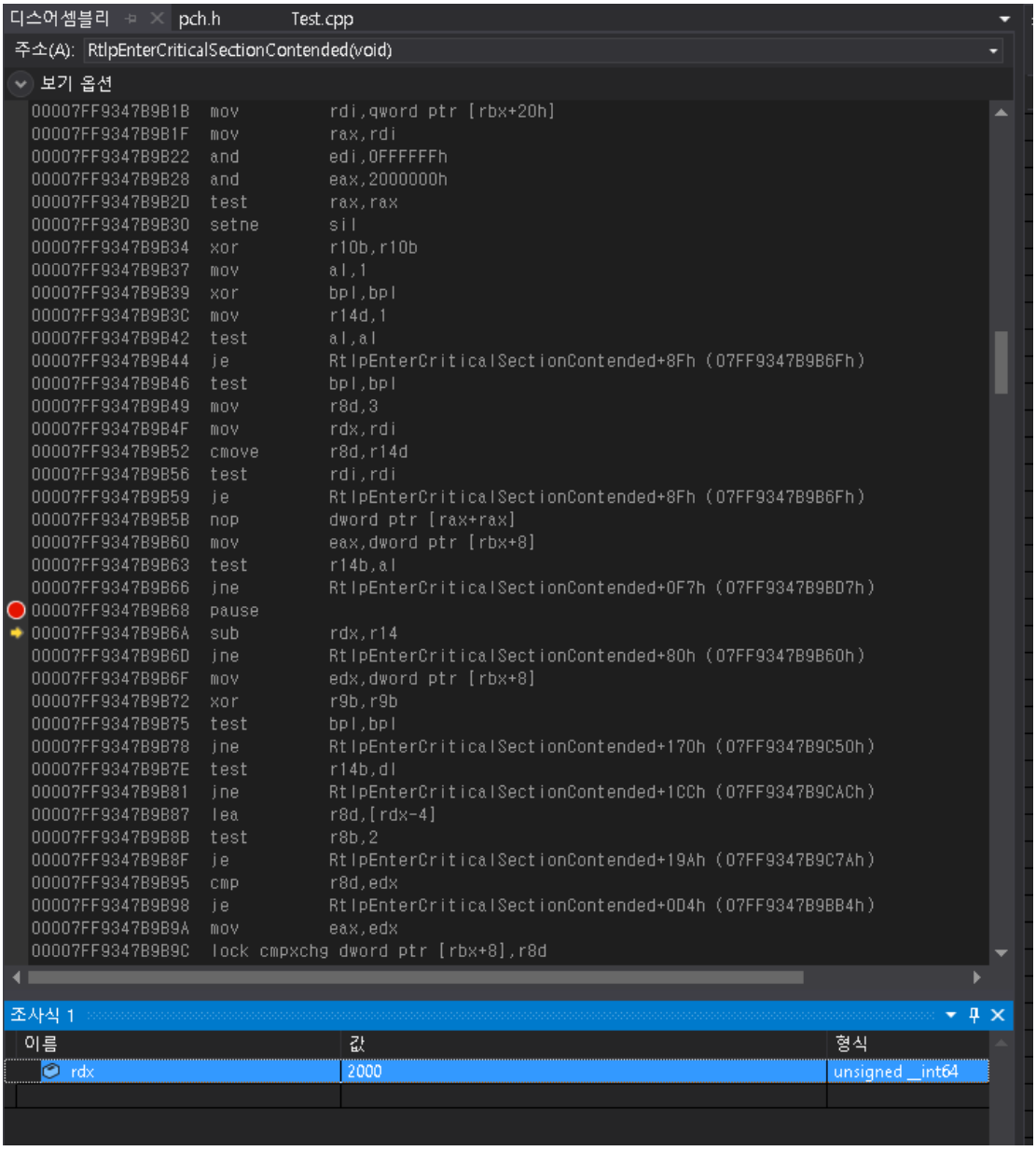
> **Pause**
> - 하드웨어 영역의 어셈 명령어. OS는 전혀 관여하지 않는 명령어이다.
> - 하이퍼 스레딩에서만 적용. 아니라면 NOP로 취급된다.
> - 잠시 스레드를 쉬도록 해, 같은 코어에 접근하는 다른 스레드가 일을 더 많이 하도록 한다.
> - 일정 횟수 루프를 돌면서 Pause를 실행한다.

- 디스어셈블리로 확인해봤다.



- 실제로 SRWLOCK에서는 Pause를 사용 중이며 1024면 루프를 돈다.
- 시나리오를 만들어보면, 같은 코어를 사용중인 스레드 A,B 중 스레드 A가 임계영역에 접근한 상태이고, 스레드 B도 같은 임계영역에 접근하길 시도할 경우, 스레드 B가 Pause를 돌면서 쉬어준다면 스레드 A는 더 빨리 작업을 처리할 수 있기 때문에, 결과적으로 스레드 B가 임계영역에 들어가는 시간도 빨라질 것이다.

- 그럼 Critical Section은 Pause를 안써서 느린것일까?



- Critical Section도 사용 중이다. 얘는 2000번 루프를 돌면서 Pause를 한다.
- 결국, Critical Section 과 SRWLOCK 의 차이점은 기존에 알고 있던 [락 객체 크기 차이, 락을 걸 시 비트 값만 변경] 이 두개인 듯 하다.

Apache/PHP/MySQL 셋팅

1. 아파치가 재시작되는 문제

- Dummy에서 아파치를 통해 PHP를 거쳐 DB에 UPDATE/SELEC를 테스트 중.
- 갑자기 Dummy가 죽는 상황이 발생했다.
- Dummy쪽 로그를 확인해보니, HTTP 통신 중, 아파치가 먼저 접속을 끊어 Dummy가 Crash를 내면서 프로그램이 종료된 것.
- 아파치 로그를 보니 정말 Dummy가 죽은 그 시간에 아파치가 재시작되었다.
- 오류 코드를 보니 c0000374(힙 침범)이었다.
- 혹시 몰라 이벤트 뷰어를 보니 역시 httpd.exe가 c0000374오류코드로 종료되었다고 남아있었다.
- 구글링 중, 스택 오버플로우에서 비슷한 문제를 확인했다.
- 답변에 링크가 있어서 들어가보니, PHP 공식 홈페이지로 연결됐다.
- 내용은 아파치 재시작 버그가 수정되었으니 PHP버전 업그레이드 하라는 내용이었다.
- PHP 버전을 7.2 스레드 세이프 버전으로 업그레이드 하니 깔끔하게 해결되었다.

2. 아파치의 메모리가 계속 증가하는 문제

- Dummy에서 아파치를 통해 PHP를 거쳐 DB에 UPDATE/SELECT 테스트 중, 더미쪽에 TPS가 점점 떨어지기 시작하는게 눈에 보였다.
- 서버의 작업관리자를 보니, 메모리가 90% 사용 중이었다.
- 메모리가 계속 늘어나, 페이지 폴트가 발생해 전체적으로 성능이 내려간 상황인 듯 하다.
- 누가 메모리를 많이 쓰나 봤더니 아파치가 거의 80%를 쓰고 있었다.
- 이번에도 스택 오버플로우의 힘을 빌렸다.
- 비슷한 질문을 찾았는데, http-mpm 설정파일의 MaxConnectionsPerChild를 건드리라는 것이었다.
- MaxConnectionsPerChild는 아파치에 연결 가능한 최대 수로, 이 이상 커넥트되면 아파치 프로세스를 kill했다가 다시 살린다.
- 답변에 더 자세히 설명은 없었지만, 커넥트 할 때 마다 쌓이는 정보가 있는데 그게 메모리를 차지했던걸로 추측된다.
- MaxConnectionsPerChild를 100만으로 수정해, 100만명마다 프로세스를 강제로 kill하는걸로 이슈는 종료됐다.

3. 포트 부족 문제

- 역시 더미테스트를 하는데, php에서 mysql connect나 fsockopen 시 에러가 발생했다.
- 에러 메시지는 다양한데, 결론은 어쨌든 실패했다는 것이다.
- 이것 저것 확인하다가 netstat –an을 보니, 사용중인 포트가 끝없이 나타났다.
- 더미가 과하게 접속을 시도함으로써 포트 고갈현상이 발생한 것으로 확인되었다.

- 일단, 동적포트를 확장시켰다.

> **동적 포트**
> - 포트에는 알려진 포트, 등록된 포트, 동적/사설포트가 존재한다.
> - 이 중 동적포트는 동적으로 할당되는 포트로, 사용자가 자유롭게 사용할 수 있는 포트이다.
> - 예를 들어 클라이언트가 connect를 하면 동적포트 중 하나가 자동으로 할당된다.
> - 동적/사설포트는 49152~65535번이며, 임의로 이 포트를 확장할 수 있다.

- **윈도우에서 동적포트 확인 :** cmd에 netsh int ipv4 show dynamicportrange tcp
- **윈도우에서 동적포트 확장 :** cmd에 netsh int ipv4 set dynamicportrange tcp start=32767 num=32768 store=persistent 숫자부분을 잘 조절하면 된다.

- 다시 테스트를 해봤는데, 그래도 여전히 동일한 문제가 발생했다.
- 다시 한번 netstat –an을 열어봤다. 여전히 포트가 쌓여있다.
- 주의깊게 보니 대부분의 포트들이 TIME_WAIT 상태였다.

> **TIME_WAIT 상태**
> - 포트의 상태 중 하나로 연결 종료된 포트에게 할당되는 상태이다.
> - 연결 종료를 요청한 쪽에 남으며, 일정 시간 후에 사라진다.
> - **존재 이유 :** 만약, 연결 종료 후 포트가 바로 재사용된다면 기존에 네트워크 혼잡 때문에 오지 못했던 패킷이 정상적이지 못한 유저에게 전송될 수 있다.

- 포트 수가 아무리 많아도 TIME_WAIT으로 포트가 남아있기 때문에 똑같이 포트 고갈 현상이 발생한 것.
- TIME_WAIT 시간을 낮춘 후 테스트하니 정상적으로 작동했다.

4. MySQL이 CPU를 100% 점유하는 상황

- Dummy에서 아파치를 통해 PHP를 거쳐 DB에 UPDATE 테스트 중.
- TPS가 너무 안나오는 상황이었다. (목표는 500TPS인데 잘나와야 100TPS였음)
- 작업 관리자를 보니, MySQL의 CPU 점유율이 거의 100%였다. MySQL에서 뭔가 과한 작업을 하고 있는 것으로 추측했다.
- 처음엔 Slave에게 보내는 Binary log가 너무 과한가 싶어 sync_binlog설정을 1에서 100으로 수정해봤다.

> **Sync_binlog**
> - Slave로 바이너리 로그를 전송하는 시점을 설정한다.
> - 예를 들어, 1이면 쿼리 1개마다 slave로에게 바로 바이너리 로그를 전송한다.
> - Replication에 사용되는 옵션이다.

- 수정 후 조금 올라가긴 했는데, 여전히 많이 부족했다
- MySQL 공식 홈페이지를 검색하다가 innodb_flush_log_at_trx_commit라는 설정을 찾았다.

> **innodb_flush_log_at_trx_commit**
> - 바이너리 로그를 디스크에 저장하는 시점을 설정. My.ini에서 설정 가능
> - 0: 초당 1회씩 트랜잭션 로그 파일(innodb_log_file)에 기록 (1초마다 디스크 기록 및 커밋)
> - 1: 트랜잭션 커밋 시 로그 파일과 데이터 파일에 기록 (새로 커밋될 때 마다 디스크에 기록 및 커밋)
> - 2: 트랜잭션 커밋 시 로그 파일에만 기록, 매초 데이터 파일에 기록 (새로 커밋되면 즉시 커밋. 1초가 되면 디스크 기록)

- 1이 기본 값이었는데, 이는 UPDATE 마다 디스크와 I/O작업이 있었던 것....
- 해당 값을 0으로 변경한 후에는 CPU 점유율도 낮아지고 TPS도 올라갔다.

- MySQL의 로그에 대해 궁금해져, 좀 더 찾아보니 로그를 정기적으로 삭제하는 기능도 있었다. 역시 my.ini에서 설정할 수 있다.

> **binlog_expire_logs_seconds**
> - 이진 로그 생존 시간. 초 단위. 이 시간이 지난 로그는 삭제.
> - 예를 들어, 이 값을 3600으로 설정하면 로그는 1시간마다 삭제된다.
>
> **max_binlog_size**
> - 로그 파일의 최대 크기. 이 크기가 되면 새로운 로그 파일을 생성한다.
> - 1G, 512M과 같이 표시한다.

- 그리고, 시간이 됐다고 바로 삭제하는게 아니라 기존의 바이너리 로그가 가득 차서 새로운 이진로그를 생성할 때 시간이 지난 파일을 체크해 삭제한다.
- 예를 들어, 파일 A가 binlog_expire_logs_seconds만큼 시간이 지나 삭제 대상이 되면, 즉시 삭제하는게 아니라 현재 쓰고있는 이진 로그 파일의 Size가 max_binlog_size가 될 경우, 새로운 로그파일을 생성하면서, 파일 A가 삭제된다.

5. PHP의 지속 연결(Persistent connect)

- Dummy에서 아파치를 통해 PHP를 거쳐 DB에 UPDATE/SELECT를 하는 중.
- 다음 단계로 넘어가는 조건은 각각 TPS 1000이었다.
- 어디가 느린지 찾기 위해, mysql connect / mysql query 의심되는 부분을 프로파일링 했다. (직접 제작한 PHP 프로파일러 사용)
- 확인해보니 mysql query는 0.001초 단위인데 connect가 평균 0.01에서 최대 0.1초까지 걸렸다.
- MySQL Connect는 PHP의 영역이니, PHP의 Mysql connect 관련 내용을 찾아보는 중

PHP 공식 홈페이지에서 지속연결이라는 것을 찾았다.

PHP의 지속연결
- 보통, TCP 통신은 Connect/Disconnect의 부하가 심하다. (3way handshake/4way handshake)
- 지속연결은 연결을 유지해 Connect와 Disconnecet가 발생하지 않게 하는 방법.
- mysqli_connect('p:'. $db_host, ...) 처럼 'p'를 붙이면 지속연결로 연결된다.

- 이걸 적용하니 일단 TPS가 850까지 올라갔다.
- 이후 OPCache라는 것을 찾아 적용했다. 그러니 TPS가 1300까지 올라갔다.

.

6. 네트워크 과부하

- 1U에 배틀,채팅,매칭 / 다른 1U에 더미,아파치 상태에서
  1U에 배틀,채팅,매칭,아파치 / 다른 1U에 더미로 변경했다.
- 아파치에 더 빠르게 접근해 배틀서버의 DB Write TPS를 올리기 위함이다.
- 근데, 이렇게 옮기고 나니 DB에 UPDATE 하다가 Lost connecting to MySQL server during query...(2013) 라는 에러가 발생했다.
- 전혀 이유를 모르는 상황이라, 일단 각 서버를 지켜봤다. 혹시 몰라 성능모니터도 켜놨다.
- 특이사항이, 성능모니터에서 1U 2개 다 네트워크 송수신 바이트가 계속 올라갔다.
- 그러다 어느 순간 Lost connecting to MySQL server during query...(2013)에러가 발생했다.
- 더미가 부하테스트도 겸하기 때문에 네트워크 송수신이 과해져 발생한 문제인 듯 하다
- 그 중에서도 채팅이 가장 과했다.
- 더미 1명당 1초에 10번, 3000명이 들어오는 중이기 때문에 약 1초에 3만TPS를 Recv하며, 방 안의 다른 유저들에게 브로드 캐스팅(한 방에 5명)하면 20만TPS를 Send하는 중이었다.
- [배틀, 매칭 아파치 / 더미, 채팅]로 변경해, 더미와 채팅을 같이 두고 이 둘은 루프백으로 통신하게 했다.
- 변경 후 정상적으로 작동했다.