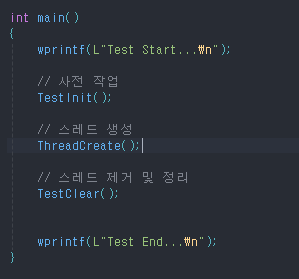
//////////////////////////////////////////////////////////////////////////////////////////////////////

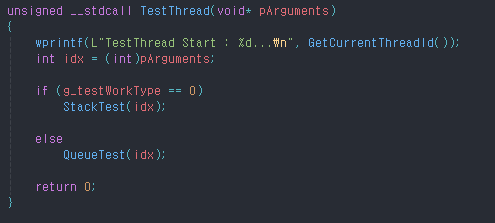
// Lock-Free vs SpinLock Performance 테스트 및 분석

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

1. 비교 테스트 코드
2. Main



1. TestThread



* 스레드 생성할 때 index 값을 인자로 주기 때문에 그걸 Arguments로 받아서 StackTest 함수에 전달함.

1. MonitorThread



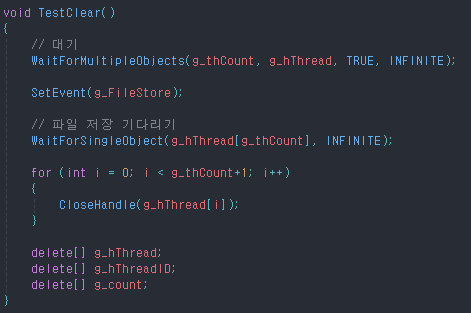
* 모니터링 스레드는 1초에 한번씩 WAIT\_TIMEOUT 로직 처리하거나 파일 저장 이벤트 객체가 시그널이 되면 파일 저장 작업을 하고 무한 루프를 탈출함.
* 모니터링 할때 각 스레드가 본인 배열의 index를 가지고 각자가 해당 배열 원소에 count값을 올리기 때문에 모니터링 스레드는 그걸 취합하는 것임. 취합하면 다시 배열 원소를 0으로 밀어서 다시 카운팅함.

1. ThreadCreate



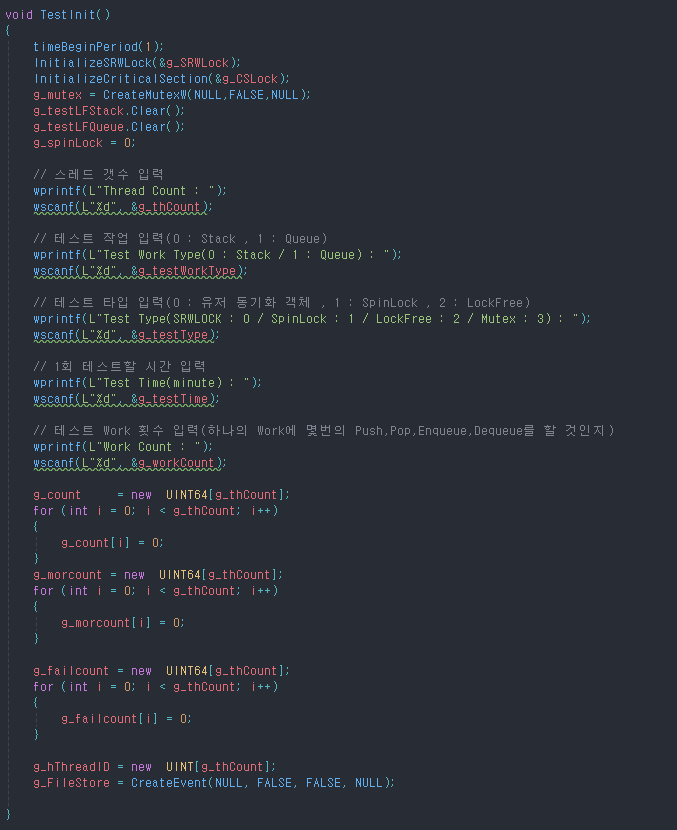
* 스레드 생성 작업임. 우선 입력 값으로 스레드 갯수를 입력받으면 그것 + 1만큼 핸들 배열을 생성함. 그리고 0번 index부터 스레드를 생성해서 나온 핸들값을 저장해줌.
* 이때 스레드 생성시 해당 배열 index값을 스레드에게 인자로 전달함.
* 스레드 생성시 상태를 suspend로 하고 스레드 생성이 끝나면 resume해서 스레드를 running 상태로 만듬.
* 마지막으로 모니터링 스레드를 만듬.

1. TestClear



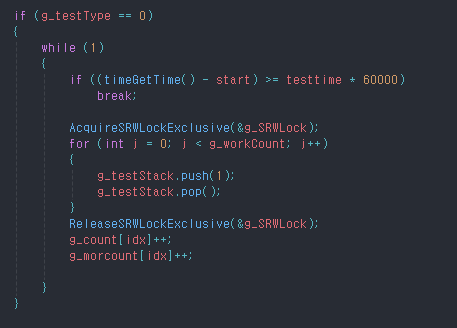
* 생성한 테스트 스레드가 끝날 때까지 WaitFor함수를 통해 기다림.
* 스레드가 테스트 끝나서 종료되면 파일 저장을 위해 이벤트 객체를 signal 시켜서 모니터링 스레드를 깨우고 파일 저장 작업을 한 뒤에 모니터 링 스레드도 종료함.

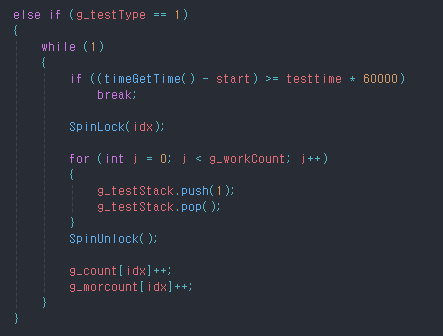
1. TestInit

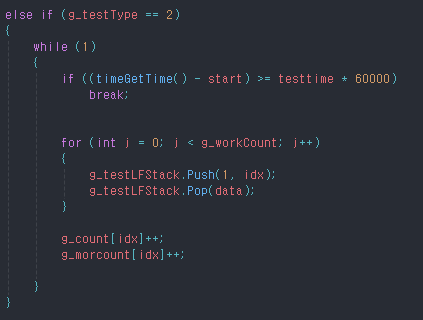


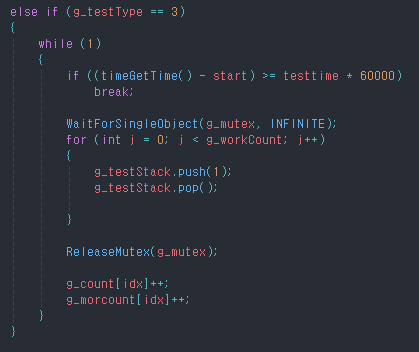
* 커널 동기화 객체인 뮤텍스, 락프리 스택, 락프리 큐, 스핀락 초기화 하고 스레드 갯수, 작업 타입, 동기화 타입, 테스트 시간, 테스트 작업 횟수(임계 영역 시간)을 입력값으로 받아서 설정함.
* 3개의 전역 카운트 배열이 있음. 첫 번째, 두번째는 각 스레드가 lock걸고 작업하고 lock 풀면 카운팅을 하는데 첫 번째는 파일 저장할 때 쓸 전역 변수고 2번째는 단순 모니터링 용도임.
* 세 번째는 스핀락에서 interlockexchange 실패시, 락프리 Push에서 CAS 실패시 카운팅하는 용도임.

1. StackTest





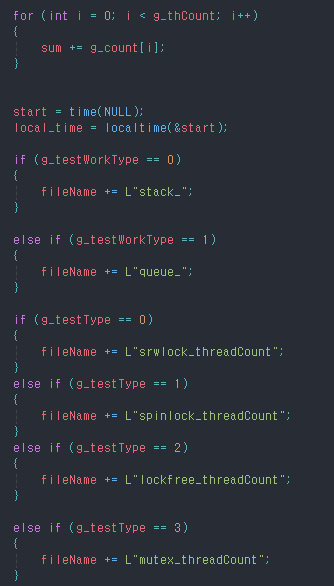




1. QueueTest

* Stack과 같음.

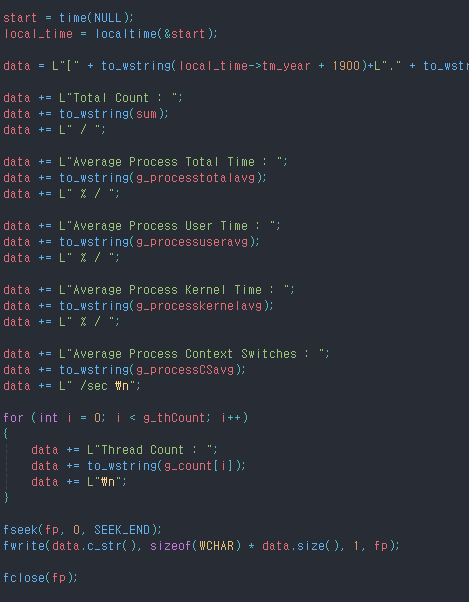
1. FileStore



* 우선 각 스레드가 각자 배열에 counting한 값을 합쳐서 sum에 저장함.
* Stack으로 테스트했는지 queue로 한건지에 따라 fileName 스트링에 파일 이름 저장함.
* TestType에 따라 srwlock인지 spinlock 인지 등에 따라 파일 이름 이어서 설정함.

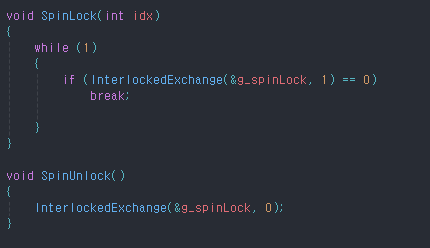


* 이름 설정하면서 스레드 갯수, 테스트 시간, 작업 갯수를 파일 이름에 붙여줌.
* 파일 스트림을 열되 파일 있으면 있던 것을 열고 바이너리로 엶.



* Data 스트링에 시간 정보 copy하고 Total Count, 프로세스 평균 cpu 사용률을 저장하고 각 스레드 마다 카운팅한 값도 저정하고 컨텍스트 스위칭 평균도 저장함.

1. SpinLock / SpinUnlock

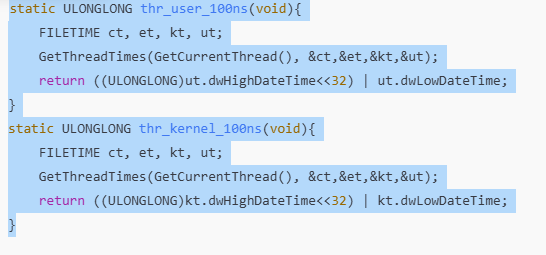


1. 결과 고찰
2. 스레드 갯수 < 논리 코어 갯수

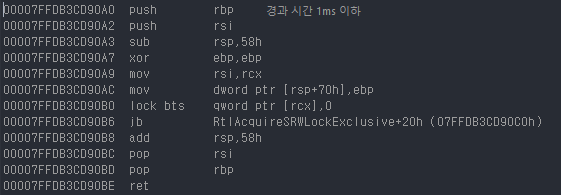
* 우선 단일 스레드에서 테스트 시간인 5분 동안 돌렸을 때 카운트 값을 보자.
* 처음에는 락프리가 제일 높았음. 그런데 이때 테스트 문제가 결국 경합이 없는 상태에서 스핀락과 srwlock은 lock 걸고 푸는 행위를 해서 std::stack에 push, pop을 하는 구조이고 락프리 스택은 그냥 락프리 스택을 가지고 push, pop을 하는 구조임.
* 동기화 방식에 따른 count 값의 차이가 테스트 목적인데 그러면 단일 스레드로 테스트하려고 하면 srwlock, spinlock에서도 락프리 스택을 사용해서 비교를 해줘야 함. 똑같은 스택이긴 하지만 하나는 락프리로 작동하고 하나는 std 형태로 작동하는 것이니 작동 형태를 동일하게 해야 해줄 필요가 있음.
* 락프리 스택의 경우 CAS 부하 + 락프리 스택 내부 구현 영향(메모리 풀 사용 영향등 여러 영향 있음)이 있고 다른 방법의 경우 interlock 부하 + std::stack 내부 push, pop 구현 영향을 같이 비교해야 함. 단순히 CAS vs interlock 부하를 비교하는것은 옳지 않음.
* 그래서 해당 테스트로 해보면 당연히 이전보다는 count값 자체가 상승했음. 그러나 락프리값까지는 오지 못함. Interlock 횟수만 봐도 락프리는 push, pop 1회 기준 2번이고 srwlock, spinlock은 lock걸고 풀고 push , pop이니 4번임. 그래서 차이가 발생함.
* 일단 단일 스레드에서 std::Stack을 사용했을 때 spinlock과 srwlock은 비슷하게 50억 count가 발생함. 이게 5분 동안의 최대 치임. 여기서 스레드 경합이 생기게 되면 50억 보다 당연히 아래로 갈것임.
* 스레드 3개 일때 total count를 보자. Srwlock(92억) > spinlock(33억) > lockfree(25억) 이 전체를 가지고 비교하는게 아니라 같은 stack을 쓴 spinlock과 srwlock을 비교하고 유저 모드에서 뺑뺑이 도는 형태의 동기화 방식인 spinlock과 락프리를 비교해야 함.
* 우선 srwlock과 spinlock을 비교해보면 srwlock이 압도적으로 성능이 높음. 경합이 없을 때는 둘다 비슷했는데 경합이 발생하는 환경에서 spinlock이 경합에 대한 부하가 컸다는 것임. 그러면 어떤 이유로 경합의 부하가 커진 것일지 생각해보자.
* Srwlock의 경우 lock 명령어로 소유권 확보 부하 + 소유권 획득 실패 시 waitonaddress 호출(srwlock 자체적으로 구현한 것)에 의한 내부 spin + 소유권 확보 실패시 커널 모드 전환 + 컨텍스트 스위칭 부하(running->block, block -> running) 이것과 spinlock의 경우 반복적인 interlockexchange 부하임.
* 만약 진짜 소유권 확보 실패해서 커널 모드 전환을 하긴 했을 수 있는데 진짜 빈번했다면 내 예상으로는 spinlock의 성능이 더 좋다고 생각함. 단일 스레드에서 Srwlock(50억)이랑 mutex(5.6억) 비교만 해도 커널 모드 전환이 상당히 부담스러운 부하인 것은 명확함. 그러니 경합시 커널 모드 전환이 빈번하지는 않았을 것임. 결국 waitonaddress 내부에서 스핀을 돌면서 소유권 누가 버린것을 확인했을 것이고 다시 lock걸어서 소유권 확보 시도를 했을 것으로 판단함.
* 그러면 srwlock의 부하에서 컨텍스트 스위칭 + 커널 모드 전환 부하는 매우 적다고 봐야 하고 그러면 lock 명령에 의한 부하 + spin 부하 vs interlock 부하임. 여기서 srwlock에서의 lock 명령 부하랑 spinlock에서의 interlock 부하를 그냥 비교하는게 아니라 단위 시간에서 저 명령어 들이 얼마나 실행되었는지를 봐야 함. 단위 시간에서 spinlock의 경우 다른 로직 없이 계속 interlock으로 확인하는 작업을 하고 있는 반면 srwlock은 lock 한번 걸어주고 소유권 확보 실패했으면 waitonaddress에서 spin돌다가 다시 소유권 확보 할거 같으면 lock걸러 갈것임. 그러면 중간에 이런 작업으로 인해 spinlock보다 오히려 단위 시간에서 interlock 호출은 적어질 것임.
* 그러면 lock 명령어 영향을 보면 cpu 내부에서 메모리 배리어를 쳐서 명령어들의 비순차적 실행을 다시 순차적으로 실행시키도록 하지만 x86 아키텍쳐의 경우 load, store 명령어 순서 재배치를 interlock으로 어떤 값을 바꾸려고 할 때 해당 캐시 메모리의 캐시라인을 잠궈서 다른 코어에서 그 캐시 라인을 대상으로 interlock으로 쓰기를 못하게 막아버림. 그러면 당연히 그 코어는 기다릴 수밖에 없음.
* 그러면 spinlock과 lockfree를 비교해보자. 이때는 spinlock에서 스택을 오히려 락프리 스택으로 바꾸게 되면 안됨. 동기화 방식에 따른 성능 차이를 봐야 하는데 spinlock을 걸어서 lock을 먹고 나서 굳이 내부 스택 push, pop 과정에서 불필요하게 cas를 하게 되니 이는 테스트가 오염될 수 있음. 우선 totalcount값으로 보면 spinlock이 더 높음.
* 다시 처음으로 돌아가자. 현재 std::stack을 사용하는 것보다는 락프리 스택에서 cas 작업이랑 뺑뺑이 도는 작업을 빼낸 stack 클래스를 만들어서 비교하는게 오히려 변수를 더 줄이기 때문에 데이터의 신뢰성이 높아질 것임.
* 그리고 단순히 push, pop에서 cas 실패 카운팅의 총합, spinlock에서 interlockexchange 실패 총합 이렇게 정량적으로 비교하는 것은 애매함.



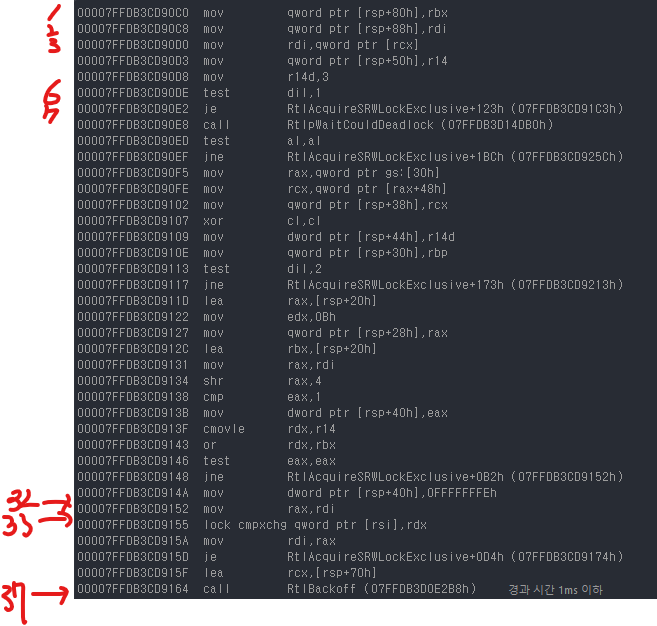
* 일단 전체적인 코드를 봤을 때 위 구조인데. 다른 코드는 다 똑같고 lock만 다르게 잡으니 여기서 생기는 차이로 인해 totalCount값이 달라진것으로 봐야 함. 그러면 어떤 것이 차이를 만들었는지 보려면 여기서 더 깊게 들어가서 어셈블리 단위까지 볼 필요가 있음.
* 그래서 이런 단위 시간당 lock 명령어 시행 횟수에 의한 결과로 판단하기 위한 정량적인 지표로 테스트 할때 lock 걸기 전에 유저 모드 시간값 측정하고 lock 걸고 나서 유저모드 시간값을 측정해서 중간에 컨텍스트 스위칭에 의한 영향을 완전 배제를 해서 이 시간 차이값을 계속 더해가다가 테스트 끝나고 spinlock, srwlock이 서로 비교를 해서 srwlock이 더 낮으면 증거의 일부로 볼 수 있음.
* 그래서 그걸 위해 GetThreadTime이라는 함수를 사용해서 100ns위로 시간 값을 측정해서 더해나갈 것임.

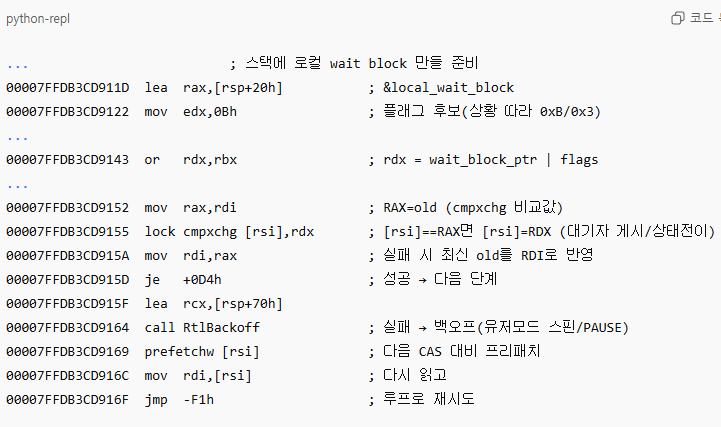


* 이런식으로 ut를 api에 전달하면 하위 4바이트, 상위 4바이트에 기준 시간(1601년 1월 1일 기준 흘러간 시간의 값을 100ns로 전달)부터 흘러간 시간값을 우리에게 알려주는데 highdatetime을 왼쪽으로 32비트 옮겨서 8바이트 메모리 기준으로 상위 32비트에 위치시키고 거기에 lowdatetime을 | 연산으로 하위 32bit값을 붙여버림.
* 이렇게 해도 되고 x86 window의 경우 리틀 앤디안이기 때문에 애초에 메모리 구조상 lowdatetime에서 아래쪽(메모리 주소 커지는 방향) 4바이트 아래에 highdatetime이 있기 때문에 그냥 lowdatetime값을 8바이트로 캐스팅해도 인식 자체는 동일할 것임.
* GetUserTime 함수를 만들어서 테스트를 해봤는데 오히려 totalCount가 srwlock과 spinlock이 비슷해짐. 오히려 100ns 누적값이 srwlock이 높았는데 단순히 이렇게 비교하는게 아니라 단위 루프 횟수 당으로 바꿔야 함. 그래야 비교가 됨. 루프 많이 돌면 당연히 높게 나옴. totalCount가 비슷해지니 GetUserTime이 테스트 결과를 왜곡시키는 것으로 볼 수 있음.
* 그래서 rdtsc로 바꿔서 할 것임. 초당 컨텍스트 스위칭이 srwlock이나 spinlock이나 비슷비슷하니(30) 어느 한쪽이 컨텍스트 스위칭이 높은게 아니라 이에 대한 영향이 totalCount에 드러났다고 하기에는 부족함. 오히려 단위 시간당 각각 비슷한 컨텍스트 스위칭이 발생했을 때 그 단위 시간당 루프 횟수는 srwlock이 더 높을 것이고 그 루프 횟수 안에서 시간 측정하는 그 사이에 컨텍스트 스위칭이 발생할 확률이 오히려 낮을 것임. 그리고 말이 30번이지 이건 평균이고 실제로 테스트 후 6~8로 낮춰짐.
* 그래서 rdtsc로 바꾸고 테스트 해보면 totalCount는 srwlock > spinlock이 되고 rdtsc sum을 totalCount로 낮춰서 루프당 시간값이 srwlock이 350.75 spinlock이 2171.40임. 즉, spinlock이 더 시간을 많이 썼다고 봐야 함.
* 그러면 이렇게 차이가 나는 이유가 나오는 원인이 단위 시간당 lock이라고 봄. 이걸 명확히 설명하기 위해 SRWLock과 SpinLock의 어셈블리 명령어를 보자.

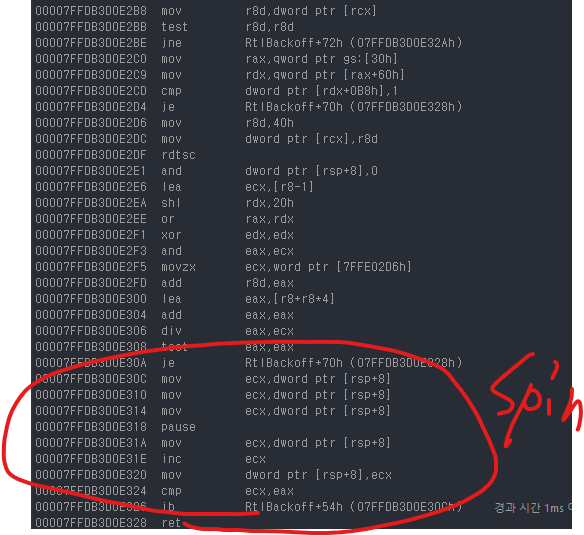


* 이게 Acquire 함수 내부 어셈블리인데 rcx는 srwlock 객체 주소값을 저장하니 그 주소값이 가리키는 srwlock 객체 값과 0과 lock bts를 통해 소유권 확보를 시도해서 만약에 실패하면 jb 명령어를 통해 어디론가 점프함. 소유권 확보했으면 함수 정리해서 리턴함.

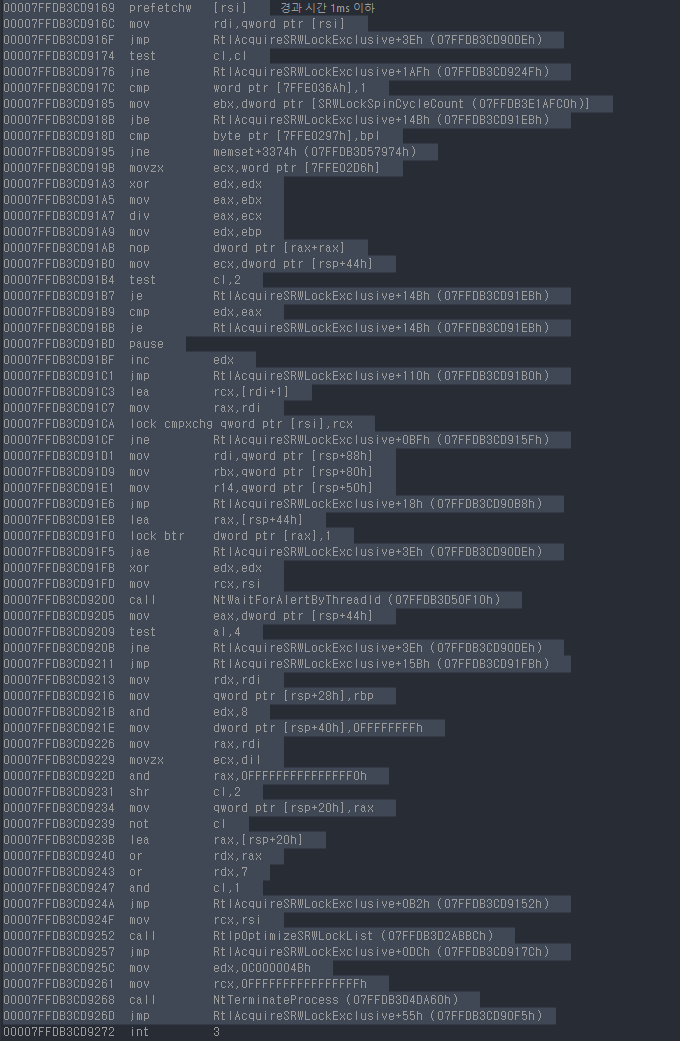


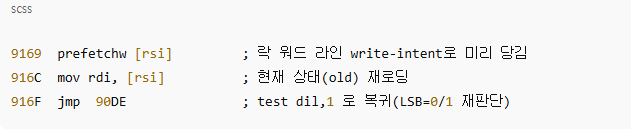


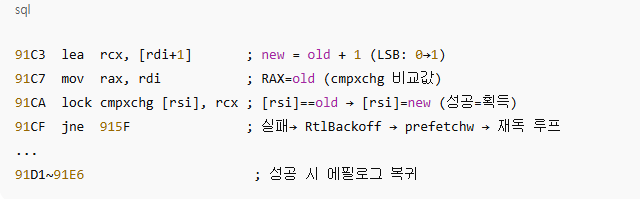
* Jb를 통해 점프한 곳이 위 시작 명령어인 mov임. 3번째 명령어에서 rcx가 가리키는 메모리의 값을 rdi로 가져오는데 그게 srwlock 객체 값임. 그걸 가져와서 rdi 레지스터에 저장하는데 6번째 명령어를 보면 test dil, 1를 통해 rdi 하위 레지스터 값과 1과 비교를 함. 즉, 간단하게 소유권이 보통 확보되어 있으면 srwlock 객체 하위 비트가 1임. 그래서 이렇게 간단 비교를 통해 소유권이 있는지 체크함. 그래서 만약 누군가 소유권 버려서 rdi에 0이 저장되면 7번째 명령어인 je를 통해 어디론가 점프를 하는데 점프하는 곳을 보면 거기서 다시 lock cmpxchg로 소유권 확보를 시도함.
* 만약에 test로 비교해도 이미 누군가 소유권을 확보한 상태라 rdi에 1이 저장된 상태면 Deadlock확인 위해 call해서 체크해서 다시 리턴되고 쭉 내려감.
* 그러다가 11D 주소 위치에서 lea 명령어를 통해 스택의 주소값을 rax로 가져오고 있음.
* 143 주소값 명령어에서 or를 통해 rdx, rbx를 하는데 위에서 122를 보면 edx에 특정 값인 b를 저장하고 rdx에 저장된 값이랑 or 연산을 통해 rdx에 저장함. 이 값을 언제 사용하냐면 155의 lock cmpxchg에서 사용함.
* 32번째 명령어에서 레지스터에 저장한 기존 srwlock 객체 값을 rax로 가져오고 lock cmpxchg를 통해 cas를 시도함. Rsi에는 srwlock 객체 주소값이 저장되어 있어서 실제 srwlock 객체랑 지금 내가 레지스터에 저장했던 값이랑 같은지를 원자적으로 체크함.
* 레지스터에 저장된 값과 원자적으로 비교한 srwlock 객체 값이 같으면 srwlock 객체 값을 rdx에 세팅한 값으로 바꿔버림. 이때 rdi 레지스터에 저장된 값은 무조건 1임. 즉, 누군가 소유권을 확보한 상태값을 레지스터에 저장한것임. 왜냐면 0이면 이미 그전에 test dil,1을 통해 빠져나갔을 것임.
* 어쨌든 다시 와서 155의 lock cmpxchg를 통해 실제 srwlock 객체값과 rdi(1)이랑 비교를 해서 만약 같으면 아직도 누군가 lock을 소유하고 있다는 의미로 봐야하니 대기하는 사람을 게시하고 상태전이를 하기 위한 특정값을 rdx에 저장했으니 rsi가 가리키는 srwlock을 이 특정 값으로 바꿔버림.
* 만약 그게 아니라면 누군가 그 사이 소유권을 버린 것임. 그러면 call BackOff까지 진입함.



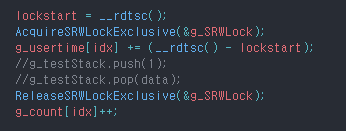
* 여기서 보면 약간의 스핀을 도는게 보면 rsp+ 8 위치에 있는 스택에 저장된 변수를 레지스터 ecx로 가져오면서 pause를 걸어서 cpu 사용을 다른 스레드에게 주고 있음. 이 작업을 eax랑 ecx랑 같아질때까지 함.
* 이 작업은 소유권 확보를 위해 뺑뺑이 도는게 아니라 그냥 약간 유저 모드에서 cpu 사용권을 다른 사람에게 주면서 대기하는 느낌으로 받아들임.
* 뺑뺑이 돌다가 나온 뒤 명령어를 보자.

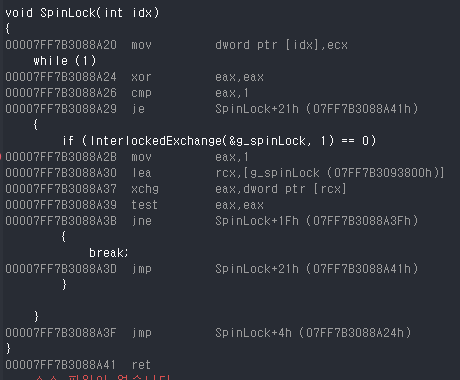






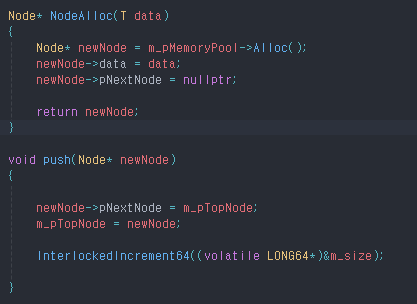
* 명령어를 보면 mov를 통해 srwlock 객체 값을 rdi로 가져오고 아까 위에서 test dil, 1을통해 비교 작업을 했는데 그걸 위해서 je로 거기 위치로 점프함.
* 거기서 테스트해서 만약에 소유권 누가 버렸으면 91c3 명령어 위치로 점프함. Rdi에 0이 저장된 상태, 즉 누가 소유권을 버린게 레지스터에서 확인이 되면 여기로 점프해서 rdi + 1을 rcx에 저장함. 즉, 1을 rcx에 저장함.
* 다시 rdi에 저장한 과거의 srwlock 객체 값을 rax에 저장함. 애초에 여기까지 왔으면 rdi에는 0이 있는 것임.
* 어쨌든 그래서 rax에 저장된 0이랑 lock cmpxchg를 통해 실제 srwlock 객체도 0인지 체크해서 만약 성공하면 rcx로 바꾸는데 srwlock 객체를 1로 바꾸는 것임. 즉 소유권을 확보한 것임.
* 만약 실패하면 915F로 점프함. 저기 위치가 BackOff 작업하다가 다시 prefetchw 점프해서 다시 lock cmpxchg로 소유권 확보 작업할 것임.
* 아래는 시간 체크 로직임.





* 그러면 이제 spinlock의 어셈블리를 보자. 1을 레지스터로 가져오고 스핀락 전역 변수의 주소값을 rcx로 가져옴.
* 그리고 xchg를 통해 rcx가 가리키는 스핀락 전역 변수 메모리 값과 eax에 있는 1과 원자적으로 교체함. Xchg자체가 앞에 lock이 생략된 것임.
* 그래서 xchg 후에 만약에 eax에 0이 있으면 소유권을 내가 먹은것임.
* 그래서 test eax, eax를 하는데 이게 뭐냐면 eax끼리 &연산을 하는 것임. 그래서 만약 그 결과가 0 이면 ZF 플래그가 1이고 0이 아니 ZF 플래그가 0임.
* Jne는 ZF 플래그가 0이면 점프함.
* 그러면 내가 소유권 먹어서 xchg후 eax가 0이면 test eax, eax 결과는 0이니 ZF 플래그가 1임. ZF 플래그가 0이면 jne에 의해서 다시 A3F로 가는데 1이기 때문에 그냥 그 아래로 와서 JMP spinlock+21h로 감. 그게 ret 명령어임. 그래서 탈출함.
* 만약 누가 lock을 먹어서 xchg 할 때 rcx가 가리키는 값이 1인 경우 eax가 1이 될 것임. 그러면 test eax, eax 결과가 1일 것이고 그러면 ZF 플래그는 0임. 그러니 jne가 가능함. ZF 가 0이니 A3F로 점프하는데 그게 JMP 07FF7B3088A24 명령어고 이 명령어는 다시 while 시작으로 가라는 것임.
* 이제 여기서 spinlock과 lockfree를 비교했을 때 비슷하긴 한데 lockfree가 살짝 더 좋음. 원래는 생각했을 때 spinlock이나 lockfree나 busy wait하게 interlockexchange를 하거나 CAS를 하는 것인데 busy wait하게 cas하는게 더 성능 안좋을 것으로 판단이 됨. 그런데 막상 락프리가 좋은데 이건 락프리 구현을 보면 Push할 때 노드를 미리 뽑아서 초기화 작업을 해주고 CAS를 함. Pop은 딱히 병렬로 처리할만한게 없음. 이 Push에서의 병렬로 노드 뽑아서 처리하는게 busy wait하게 cas하는 부하를 상쇄시켜서 락프리가 스핀락과 비슷하게 나온것이 아닌가 싶음. 어차피 스레드 별로 없어서 do while까지 여러 스레드가 들어와서 다시 사전작업하는 횟수는 많지 않을 것임.

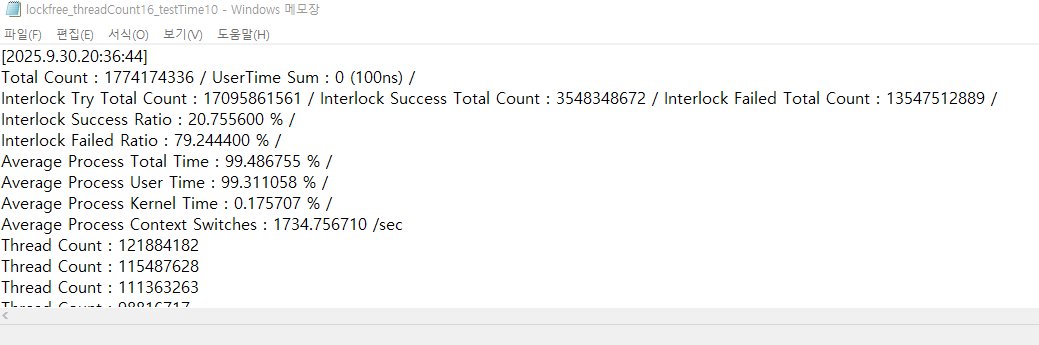


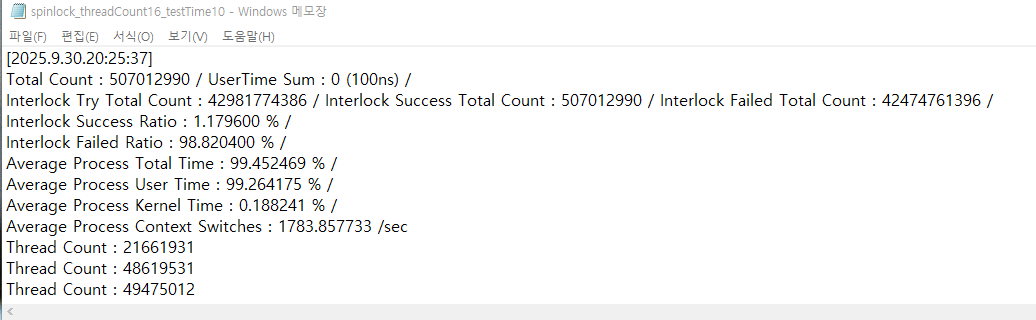


* 이를 증명하기 위한 방법으로는 저 노드를 생성하고 초기화 하는 작업을 SpinLock 걸기 전에 해버리는 것임. 어차피 저기는 여러 스레드가 병렬로 처리해도 문제가 없는 부분이게 실제 동기화가 필요한 부분은 m\_pTopNode를 newNode의 next로 설정하고 topNode를 newNode로 변경하는 부분임.
* 그래서 SpinLock에서 Lock걸기전에 NodeAlloc으로 노드 생성 작업을 병렬로 하고 약간의 작업만 동기화 걸어서 했을 때 그전 보다 SpinLock 자체는 totalCount가 늘어날 것으로 추측되고 락프리보다도 높아질것으로 보임.
* 테스트했을 때 락프리는 26.6억이고 NodeAlloc버전 뺀 기존의 SpinLock이 26.3억이고 NodeAlloc 사용해서 다시 테스트해보면 27.9억임. 기존 락프리 보다 더 높아진것임.

1. 스레드 갯수 > 논리 코어 갯수

* TotalCount 차이는 SRWLock(121.1억) > LockFree(20.2억) > SpinLock(4.6억)
* 이때 락프리 성능이 spinlock 보다 좋은 이유가 3개 중에 srwlock이 압도적으로 좋은 이유를 분석해야 함.
* 이 현상에서 srwlock이 좋은 원인이 무엇일까 보자. 이 테스트에서 중요한 측정치가 있는데 cpu사용률임. Cpu 사용률을 보면 spinlock, lockfree는 당연히 유저모드에서만 뺑뺑이 도니 유저모드에서 99%임. 그런데 srwlock도 마찬가지로 cpu 사용률이 유저모드에서 99%임.
* 이 결과가 무엇을 의미하냐면 srwlock도 스레드가 코어갯수보다 많아지면서 스레드들이 객체 얻는거 실패해서 block되고 깨어나는 행위를 했다기 보다는 이전에 어셈블리에서 분석했다시피 유저 모드에서 동기화 객체를 얻어서 push, pop을 했다는 의미임.
* 그러면 환경이 어떤지 보자. 12개의 논리 코어에서 12개의 스레드가 붙어 있고 4개는 ready중인데 12개의 스레드들이 하나의 동기화 객체에 대해서 interlock 수행할 때 동시에 걸릴 확률이 늘어날 것임.
* 이게 왜 부하냐면 하나의 코어에 붙은 스레드에서 interlock을 수행할때 그 대상의 캐시 메모리의 캐시라인을 잠궈서 내가 원자적으로 값을 바꾸게 하고 그 순간 코어에 붙은 다른 스레드들이 똑같은 캐시 라인에 대해서 interlock을 수행하려고 하면 차단해버림. 그러면 이런 스레드는 running 상태에서 lock 먹은 스레드가 캐시 라인에 값 바꾸길 기다리면서 대기타야 함.
* 그런데 스레드 갯수가 많아지면 많아 질 수록 이렇게 대기타는 스레드들도 많아질 것임. 그러면 그 대기타는 스레드는 아무것도 못하고 거기서 대기해야 하니 이게 병목이 생기는 것임.
* 그런데 srwlock에서는 이런 현상이 잘 안생기는지를 생각해보자.
* 여기서 srwlock과 spinlock, lockfree 사이에서 주된 차이를 생각해봐야함. 이전에 srwlock에 대한 어셈블리를 관찰한적 있는데 srwlock의 경우 lock 획득 실패한다고 해서 바로 다음 루프에서 lock을 시도하지 않음. 중간에 약간의 backoff과정이(의도적으로 딜레이 시간을 부여) 있음.
* Spinlock, lockfree는 lock 실패하면 다시 lock을 시도하면서 코어에 붙은 12개의 스레드들이 동시에 interlock을 수행할 확률이 높은 반면 srwlock은 lock을 시도해도 중간에 backoff 과정이 들어가면서 12개의 스레드들이 동시에 interlock을 수행할 확률 낮춰버림. 병목을 완화하게 만든 것임.
* 그래서 srwlock이 다른 동기화 방법보다 성능이 높게 나온 것으로 생각됨.
* 이제 lockfree와 spinlock이 왜 그렇게 차이가 많이 난것인지 보자. 이것도 이전 테스트처럼 노드의 생성 및 초기화를 병렬로 해서 그런것인지 NodeAlloc을 해서 테스트해보면 스핀락의 totalCnt가 높아지긴 함. 그런데 lockfree처럼 높지 않음. 그러면 어떤 다른 부하가 있다는 의미로 봐야함.
* 오히려 이런게 유저모드 뺑뺑이를 통해서 경합했을 때의 cas 성공률과 spinlock에서의 interlock 성공률을 봐보자.





* 이게 SpinLock, LockFree에서 결과 값임. Spinlock은 Interlockexchange 전에 totalCnt올리고 interlock 성공하면 successcnt 올림. 실패하면 failcnt 올림. 그래서 그걸 파일에 출력하고 비율도 계산함.
* Total값이 spinlock이 압도적으로 많은데 구조를 보면 스핀락은 실패하면 다시 interlock을 거는 반면 락프리는 cas 실패하면 다시 사전작업을 하니 단위 시간에서 interlock거는 횟수가 적을 수 밖에 없음.
* 락프리에서는 push할때 do while 시작할때 totalcnt올리고 cas 성공하면 successcnt올리고 실패하면 failcnt 올림. Pop에서도 카운팅함.
* Spinlock이 시도 자체는 엄청 많이 하는데 성공을 잘 못하고 락프리는 스핀락에 비해서 전체 시도도 높지 않지만 바로 성공(lock을 획득)을 많이 했기 때문에 최종 결과에서 lockfree가 높은 것임.
* 원래는 이런 스레드 갯수 > 코어 갯수 환경에서 락프리가 좋은 이유가 스핀락은 무조건 시작부터 1개 스레드만 가는 반면 락프리는 일단 여러 스레드가 동시에 들어가서 사전작업을 하되 cas로 1개만 통과하게 하는 것임. 그런데 스레드 갯수가 많아지면 먼저 사전작업 하던 스레드가 컨텍스트 스위칭 될 확률이 높아질 것이고 그러면 처리 흐름이 늦던 스레드가 추월해서 cas를 하게 될 것이고 컨텍스트 스위칭 되어서 out된 스레드가 다시 running와서 cas를 해서 통과하는 원리면 락프리가 당연히 좋다고 생각했음.
* 그런데 생각해보면 이 사전작업에는 노드할당도 있고 do-while문 안에서 로컬에 top 저장 및 tag 처리 하는게 있음. 그런데 do - while문에서 컨텍스트 스위칭되버리면 당연히 다른 스레드가 cas하면 내 스레드도 다시 뺑뺑이 돌아야 함.
* 좀더 근본적인 원인을 찾아보자. 스레드 갯수가 늘어날 수록 스핀락의 성능 하락이 엄청 크고 락프리는 그보다 덜함.
* 원인을 생각해보면 스핀락의 경우 스레드 갯수가 늘어나면 늘어날 수록 Spinlock 시작 위치에서 대부분의 스레드가 모이면서 병목이 생길 것임. 그러면 거기서 interlock을 수행하기 위해 한 스레드가 원자 연산하는 동안 다른 스레드는 다 러닝상태에서 대기할 것임. 반면 락프리의 경우 모든 스레드가 초반부에 모여서 병목이 일으키지 않고 일단 계속 스레드가 코드 처리하다가 cas를 하기 때문에 스핀락에 비해 cas에서의 병목이 덜 할 것임.