운영체제 Lab2

32203660 이현정

1) 목표

전체적으로 lock 과 생산자/소비자 역할을 하는 스레드에 대한 처리를 해보는 것입니다.
생산자와 소비자 문제에서 공유 자원에 동시에 접근하는 상황이 발생하지 않도록 lock 을 이용해서 공유 자원이 부족했을 때의 문제나 스레드가 돌아가는 순서에 따른 문제 등을 고려하는 과정이 필요합니다. 해당 과제를 수행하면서는 lock 의 유무에 따른 스레드들의 동작에서 발생하는 차이 확인하고 해당 이유에 대해서 알아보는 것, course-grained lock 과 fine-grained lock 의 구현과 해당하는 락이 어떤 측면에서 차이를 보이는지에 대해서 파악하는 것, 버킷과 큐의 자료구조를 통해서 상황에 따라서 더 편리한 쪽으로 접근하고 노드의 생성과 제거시에 해당 자료구조에서 발생하는 동작에 대해서 알아보는 것 등을 해 볼 수 있습니다.

2) 설계/구현

init_queue()에서는 각 시작노드를 만들어서 각 값에 NULL 을 적용해서 data 가 NULL 인지 여부를 기준으로 첫번째 노드인지 판별에서 들어가게 됩니다.
init_hlist_node()에서는 버킷에 대한 동적 할당으로 각 노드와 다음 노드를 가리키는 포인터가 들어갈 수 있는 구조체에 대해서 할당이 되고, NULL 로 초기화를 진행하였습니다.
value_exist() : 해당 값이 이미 노드로 존재하는지 판단을 위해서 front 부터 rear 까지 q 를 돌려서 해당 값이 있으면 1, 없으면 0 을 리턴합니다.

ComparAddSwap() : 포인터를 받아서 해당 값이 이용될 수 있는 1 상태였으면 1 을 돌려주고 자신은 0 으로 변화시키고, 0 으로 이미 이용 중인 상태면 0 을 돌려주는 방식으로 동작합니다.

<전반적인 구조>

hash_queue_insert_by_target(), hash_queue_insert_by_target_cg(), hash_queue_insert_by_target_fg()

value_exist()를 통해서 넣을 수 있는 노드라면 enqueue~()와 hash_queue_add~()가 순차적으로 실행되게 됩니다.

hash_queue_delete_by_target(), hash_queue_delete_by_target_cg(), hash_queue_delete_by_target()_fg

value_exist()를 통해서 노드로 해당 값이 존재하는지 확인 후 존재하면 hash()작업을 통해서 해당 노드가 들어있는 버킷의 최신 값에 접근하고, 그것을 기준으로 원하는 노드의 위치를 찾아서 큐와 버킷에서 지우는 작업을 합니다.

enqueue()

멀티스레드중 들어오면 ode 의 next 가 연결되지 않고, pre 로 먼저 큐에 연결이 되는 경우에는 들어있는 값이 탐색에서 오류가 발생할 수 있으므로 먼저 next 의 값이 NULL 로 들어가서 새로운 노드가 준비가 된 상태에서 pre 를 통해서 큐에 들어갈 수 있게 하였습니다. enqueue_cg()

front 와 rear 의 접근에 대해 큰 lock 을 걸기 때문에 내부 동작은 enqueue()와 같습니다.

enqueue_fg()

front 와 rear 가 접근이 되야하는 경우는 모아서 첫번째 큐 노드가 되는 경우와 그외에서 해당하는 일부분들에 대해서 sem_wait()와 sem_post()로 묶이게 하였습니다.

dequeue()

여러 스레드로 인해 target 값의 변화에 대응하기 위해 받은 target 의 값을 node 의 data 에 넣어서 노드를 전달해서 해당 노드를 찾아서 제거합니다. 마지막 노드인 경우에는 front 와 rear 이 모두 같은 노드를 가리켜서 1 번만 free()되게하고, 버킷의 자리는 free() 후 NULL 로 채웁니다. 마지막이 아닌 경우에는 우선 해당 노드의 위치를 찾고 해당 노드를 포함하는 hlist_node 도 찾습니다. 그리고 front 나 rear 인 경우에는 없어짐에 따라 다음 또는 이전 값을 가리키게 만든 후 앞과 뒤의 노드가 서로 연결되게 하고, 해당 노드를 free()한 다음에 노드를 포함하기 위해 존재했던 버킷 구조체도 free()시킵니다.

dequeue_cg()

enqueue 와 같이 cg 는 lock 이 없는 것과 차이가 없어서 같은 방식으로 돌아가게 됩니다.

enqueue_fg()

front 와 rear 가 접근이 되야하는 경우는 front 와 rear 인지 확인하는 경우와 마지막 노드라서 1 개만 리턴되게 구현하는 것으로 각 부분에 따라 작은 범위에 대해 각각 세마포어를 적용했습니다.

hash_queue_add()

새로 만든 노드가 rear 에 들어있고 해당 내용을 버킷에 넣어야 할 때, rear 를 now 에 보관해 두었다가 현재 넣을 노드(rear 에 위치)를 가장 최신에 넣어진 노드와 연결시키고, 해당 자신인 now 을 해당 버킷의 자리에 넣게 하였습니다.

hash_queue_add_cg()

앞선 것들과 마찬가지로 위와 동일하게 동작합니다.

hash_queue_add_fg()

rear 에 현재 노드가 담겨 있어서 해당 노드를 이용해야하므로 관련 코드들만 모아서 세마포어를 처리하였습니다.

3) 수행 결과 (3 가지 비교),

2-1) 락이 있을 때와 없을 때

중복된 노드가 생성이 되는 경우에는 0 부터 점점 data 값을 증가시켜서 자신이 넣을 수 있는 값을 가지게 하였습니다. 락이 있다면, 해당 스레드 1 개씩만이 독점적으로 그 시간에 해당 자원에 접근할 수 있도록 하였습니다.

2-2) coarse vs fine-grained lock 일 때

위에서 함수들의 동작을 설명하면서 언급했던 것처럼 coarse-grained lock 은 lock 이 필요한 동작들은 노드를 삽입하면 rear 값이 항상 넣을 값으로 변동되므로 lock 이 필요하고, 버킷의 값을 넣는 경우 또한 위에서 추가된 노드를 해당 버킷에 넣기 위해서 rear 에 접근해야하므로 insert 가 동작하는 전체에 lock 을 크게 걸었습니다. 또한, 노드를 삭제하는 경우에도 해쉬 구조를 통해서 큐에 비해서 상대적으로 빠르게 접근하게 되지만, 결과적으로 해당 노드를 삭제함으로써 rear 자리에 그 전의 노드가 오는 처리를 위해서 delete 동작 전체에 lock 을 크게 걸어서 총 2 개의 lock 이 걸리게 하였습니다.

fine-grained lock 은 바로 앞에서 설명한 각 부분에는 모든 코드들이 모두 공유 자원에 접근하지 않으므로 각 함수마다 내부의 공유 자원에 접근하게 되는 경우마다 잘라서 해당 부분만 한쪽으로

모아서 lock 을 걸어서 coarse-grained lock 에 비하면 범위가 적지만, 필요한 부분에서만 동작을 할 수 있게 만들어 보았습니다.

2-3) 쓰레드 개수가 달라질 때

아래는 노드의 수가 5 개 일때가 노드의 수가 2 개일때를 비교한 것인데, 랜덤으로 노드의 값을 받게 되어서 중복되는 노드를 제외하는 과정을 거치고, sleep 앞에 숫자가 해당 target 숫자로 모두 0 을 할당 받아서 처음 외의 노드들은 모두 같은 것을 삭제하려고 하므로 노드가 5 개와 2 개의 차이가 거의 없다는 사실을 확인할 수 있었습니다.

```
oslab@oslab:~/Desktop/2022_DKU_OS-main/lab2_sync$ ./lab2_sync -t=1 -c=5 -l=0
d0sleep2
d0sleep2
d0sleep2
d0sleep2
d0sleep2
end-del
=====  Multi Threads No-lock HQ  Insert experiment =====

 Experiment Info
        Test Node Count                 : 5
        Test Threads                    : 1
        Execution Time                  : 2643163919.00 seconds

=====  Multi Threads No-lock HQ  Delete experiment =====

 Experiment Info
        Test Node Count                 : 5
        Test Threads                    : 1
        Execution Time                  : 0.00 seconds

        Total Execution Time            : 0.00 seconds
```

```
oslab@oslab:~/Desktop/2022_DKU_OS-main/lab2_sync$ ./lab2_sync -t=1 -c=2 -l=0
d0sleep2
d0sleep2
end-del
=====  Multi Threads No-lock HQ  Insert experiment =====

 Experiment Info
        Test Node Count                 : 2
        Test Threads                    : 1
        Execution Time                  : 2643165020.00 seconds

=====  Multi Threads No-lock HQ  Delete experiment =====

 Experiment Info
        Test Node Count                 : 2
        Test Threads                    : 1
        Execution Time                  : 0.00 seconds

        Total Execution Time            : 0.00 seconds
```

4) 논의

lab2_sync_test.c 에서 pthread_join()을 통해 스레드가 먼저 값을 노드로 만들어서 큐에 넣은 뒤, 삭제되게 구현되어 있기 때문에 target 을 통해 노드를 만들고 버킷에 들어가는 스레드들과 target 의 노드와 버킷을 제거해나가는 스레드들을 구분되어서 구현이 되었기 때문에 서로가 공유 자원에 접근하려는 상태를 막을 수 있기에 해당 내용에 대해서는 lock 을 걸지 않았습니다.

만약 중복된 값을 미뤄뒀다가 다시 작성할 수 있는 구조로 만들게 된다면, 해당 큐에서 그 값이 없어질 때까지 기다리다가 없어진 것을 dequeue 마다 신호를 보내서 확인하면 해당 스레드가 그제서야 돌게 만들어서 큐에 노드를 넣게 할 수도 있을 것입니다.

저는 노드나 버킷이 1 개인 경우를 나머지와 구분해서 작업을 하였으나, 추가적인 segmentation 오류가 발생하는 것으로 보아서는 각 부분을 나눠서 작업을 돌려본 결과 delete 부분에서 미정확한 코드 부분이나 잘못된 접근이 있는 것까지 확인을 할 수 있었으며, enqueue 부분에서 버킷과 노드의 동적 할당에 따른 반납 부분에서 문제가 발생된 경우도 있을 수 있겠다는 생각을 하였습니다. 그에 따라서 실행 시간이 0 으로 출력이 되는 문제가 발생한다고 봅니다.