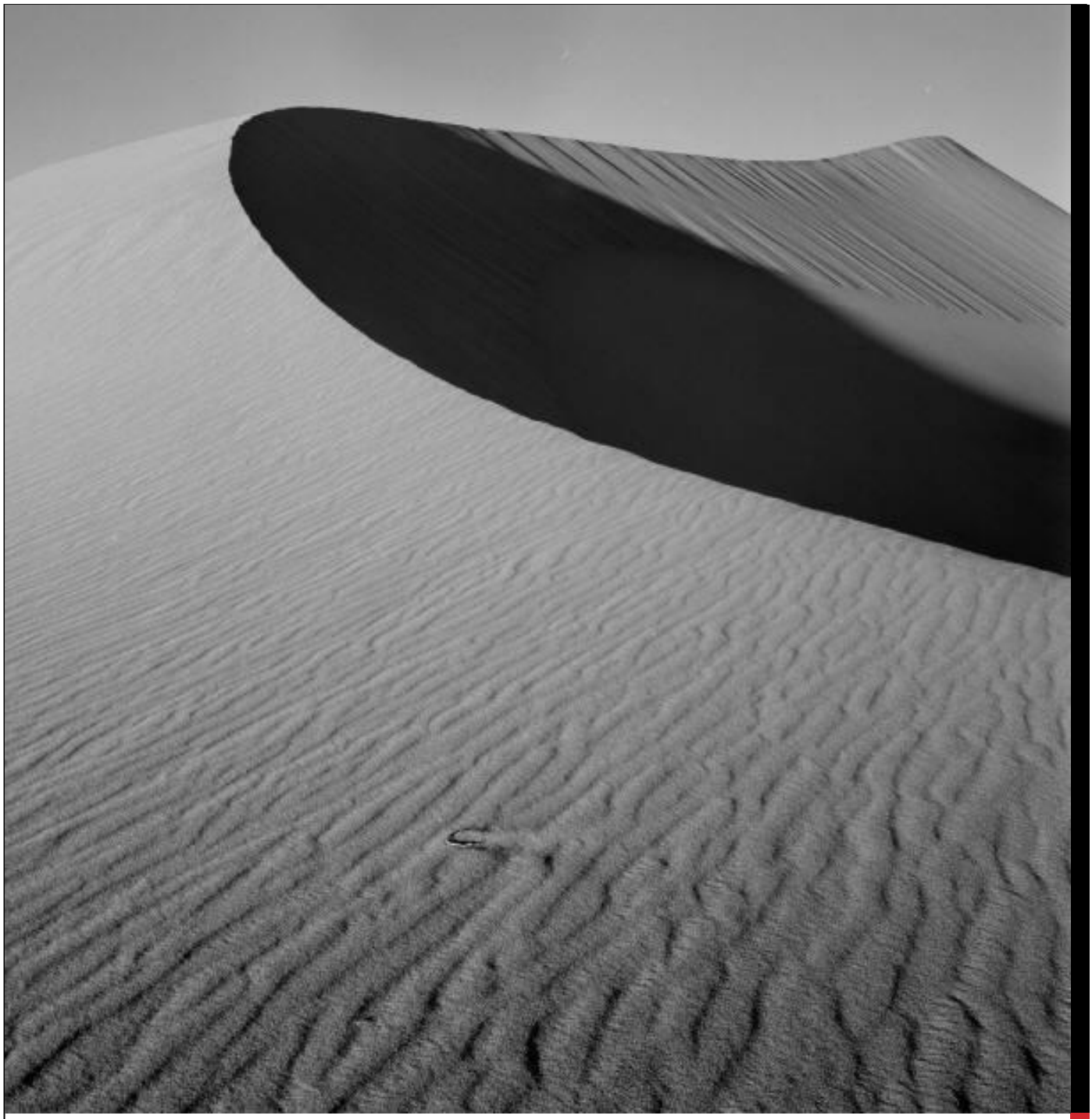


PROJECT#1

A THREAD SYSTEMS AND SYNCHRONIZATION

중앙대학교 소프트웨어학과

20173483 김휘진

목차

서론 : 문제 해석 및 해결방안 모색

Step1 문제의 경우 모든 차량 쓰레드의 step 종료에 관한 판단이 필요하고, 이를 위해서 차량 쓰레드의 block, unblock 을 관리하는 manager thread 가 필요할 것같다.

또한 각 차량 쓰레드의 lock_acquire 의 경우 step 종료에 관한 판단이 불가할 수 있다. 이를 해결해야한다.(코드변경은 crossroads 폴더의 파일으로 제한하였다.)

Manager thread 는 step 이 진행되는 도중엔 block 되어야하고, 마지막 차량 쓰레드의 trigger 으로 인해 manager thread 은 unblock 되어 다음 스텝을 관장해야 한다.

Crossroads 파일의 변경이 가능했다면 생성된 thread 의 개수, manager thread 의 생성을 run_crossroad 함수에서 진행할 수 있었지만, 쓰레드실행함수인 vehicle_loop 에서 이를 정확히 구해내는 방법이 필요하다.

Step2 문제의 경우 deadlock 상황 발생 경우를 탐색하여 그에 따른 recover 으로 진행하는 것이 좋을 것 같다. Deadlock 발생을 예상하는 경우도 존재할 듯 하다. 이는 각 차량 쓰레드가 교차로에 진입할 때 예상하는 것보다, 스텝이 종료되었을 때에 각 차량의 이동방향, 교차로 내의 차량 상태로 확인하는 것이 좋아보인다.

위에서 생성한 manager thread 에서 데드락 상황 판단 및 예상을 하고, 스텝에 따라 교차로에 입장하는 우선순위를 부여하는방법도 좋아보인다.

또한 deadlock recover 방안으로서 회전교차로에서의 차량의 움직임을 생각해서 반시계방향으로 한칸 움직이는 방법도 생각해보았다.

코드의 대략적인 흐름도

변경한 함수 : vehicle.c 의 vehicle_loop , try_move

추가한 함수 : vehicle.c 의 release_blocked_threads => manager thread 의 실행함수

Vehicle_loop{

{각 변수의 초기값 지정하는 코드}

{첫번째로 thread_cnt 에 접근하는 thread 의 경우(여러게의 thread 가 동시에 접근 가능하다) semaphore 를 초기화해준는 코드}

{여러게의 thread 가 여러 번 초기화할 경우를 대비해 sleep 함수로 모든 쓰레드가 생성되고 초기화를 마칠때까지 기다리는 sleep}

{sema_up 을 통해 처음 thread_cnt 에 접근한 차량 쓰레드일 경우 manager thread 를 생성하는 권한을 가짐}

{각 쓰레드 별로 개수와 리스트를 저장하는 코드}

{한 개의 manager thread 를 생성하고 block 되는 코드}

{차량의 움직임을 반복하는 while 문}

}

Try_move {

{차량 쓰레드의 다음 지점이 outside 인지 판단하는 if 문}

{다음 지점의 lock 을 획득하려고 시도하는 while 문}

{lock 을 획득하였을 경우 현재 위치를 다음 위치로 바꾸는 if 문}

{자신의 step 이 종료되어 block 상태로 진입하는 코드}

{block 이 풀리고 다음위치 획득여부에 따라 1 과 -1 을 반환}

}

본론 : 전역변수

1. Static Int Thread_cnt

생성된 쓰레드의 개수를 저장하는 변수이다.

만일 차량이동이 종료되면 thread_cnt 도 감소한다.

초기화 위치 : 차량 쓰레드가 생성될 때(vehicle_loop) 차량 쓰레드에서 1 을 더한다.

맨처음 쓰레드 개수를 저장하고 차량이동이 종료될 때 감소하는 것을 제외하고는 변동이 없다.

본래 static 변수를 인스턴스에서 초기화를 하면 안되지만, crossroad 파일을 수정할 수 없기 때문에 이런식의 방법을 채택했다.

여러 쓰레드에서 동시에 접근해서 동기화되지 않은 변수를 초기화하는 방법에 대해서는 코드 주석에 자세한 설명이 있다.

2. Static Semaphore Sem_TC

전역변수들을 동기화 시키기 위한 semaphore 이다.

초기화 위치 : vehicle_loop 에서 1 로 초기화 한다.

접근 가능한 쓰레드 개수를 1 개로 제한함.

3. Static Int Blocked_thread_cnt

차량 쓰레드별로 자신의 스텝을 끝낸후 블록된 상태의 쓰레드 개수를 저장한다.

Blocked_thread_cnt == thread_cnt 일 경우 한 step 이 끝난다고 판단한다.

초기화 위치 : Step 이 끝날 때마다 초기화한다.

4. Thread* Threadlist

쓰레드 포인터를 저장하는 struct thread * threadlist

대략적으로 threadlist[50]으로 초기화해 최대 쓰레드 50 개로 제한한다.

Thread_cnt 를 증가시킬 때 threadlist[thread_cnt] = current_thread 으로 현재 생성되어있는 쓰레드의 주소값을 저장한다.

차량 이동이 종료되면 threadlist 에 저장되어있는 값도 POP 한다.

초기화 위치 : 생성될때 NULL 로 초기화를 하고, thread_cnt 가 증가될 때 thread_cnt 를 index 로 사용하여 저장한다.

thread 주소값이 저장되면 차량 종료 이외에는 변경하지 않는다.

5. Static Int Destlist

차량 쓰레드의 도착지점을 저장하는 int 형 값으로 저장하는 배열 변수이다.

데드락상황을 판별하기 위해서 추가하였고, threadlist 와 같이 destlist[50] = {-1,}으로 초기화한다.

또한 threadlist 의 순서와 동일한 순서를 갖는다. Ex) threadlist[1] 의 도착지점 : destlist[1]

초기화 위치 : -1 로 초기화를 하고, threadlist 와 동일하게 저장한다.

6. Static Thread* Blocked_threadlist

Blocked_thread_cnt 와 같이 사용되는 자신의 스텝이 끝나고 움직임을 멈춘 쓰레드의 주소값을 저장하는 배열변수이다.

이는 lock 을 획득하려고 시도중인 쓰레드에게 이번 스텝에서 획득할 수 없다는 것을 알려주기 위해서 사용되었다.

초기화 위치 : 매 스텝이 끝날때마다 모두 NULL 로 초기화 되고, 차량 쓰레드가 블록상태에 들어갈 때 blocked_thread_cnt 를 인덱스로 사용하여 추가된다.

7. Static Semaphore Sem_TRI

차량 쓰레드를 관리하는 manager 쓰레드의 block , unblock 을 관장하는 trigger semaphore 이다.

초기화 위치 : Sem_TC 가 초기화 할 때 0 으로 초기화한다.

한 스텝이 시작될때 sema_down 으로 block 하고, 차량 쓰레드가 전부 움직임을 끝낸다면 차량쓰레드에서 sema_up 을 호출해 다음 스텝을 진행한다.

8. Static Bool is_deadlock_recover

이번스텝에서 데드락이 발생했고, 회복절차를 진행중인지를 나타내는 boolean 변수이다.

Manager thread 에서 스텝이 종료되면 현재 상황을 종합해 데드락 여부를 판단한다.

초기화 위치 : false 로 초기화되고, deadlock 이 발생되면 true 로 전환되어 한스텝 recover 을 진행한 후 fasel 로 되돌아온다.

Step 1 해결방안

각 차량쓰레드의 움직임이 끝났다는 것을 판단하기 위한 요소로 blocked_thread_list 를 생성하였다.

차량의 움직임에는 2 가지가 있다.

1. 이번 step 에 움직임이 가능한 경우
2. 이번 step 에 움직임이 불가능한 경우

첫번째 경우에는 목적지 lock 의 holder 가 비어있거나, 존재하지만, 다른 차량의 움직임으로 release 가 되는 경우이고,

두번째 경우에는 목적지 lock 의 holder 가 존재하지만 그 holder 의 차량의 step 이 종료되 block 된 상태인 경우이다.

따라서 스텝이 종료된 차량 쓰레드들을 blocked_thread_list 에 저장하였고,

차량 쓰레드가 lock 을 획득하려고 시도할 때(lock_try_acquire)

성공했다면 : 이전 lock 의 release 및 새로운 lock 의 holder 로 지정되고, blocked_thread_list 에 추가된다.

실패했다면 : 획득하려고 시도하는 lock 의 holder 가 존재하는 것이므로, holder thread 가 blocked_thread_list 에 존재하는지 확인한다.(lock->holder->tid == blocked_thread_list[i]->tid)

만약 존재한다면 이 차량 쓰레드는 이번 step 에 진행할 수 없다고 판단한다.

만약 존재하지 않는다면 lock 획득 시도(lock_try_acquire)를 반복한다.

이때 움직일 수 없는 차량은 bool 변수 breakflag 를 통해서 다음 목적지로 움직이지 않는고, lock release 를 수행하지 않는다.

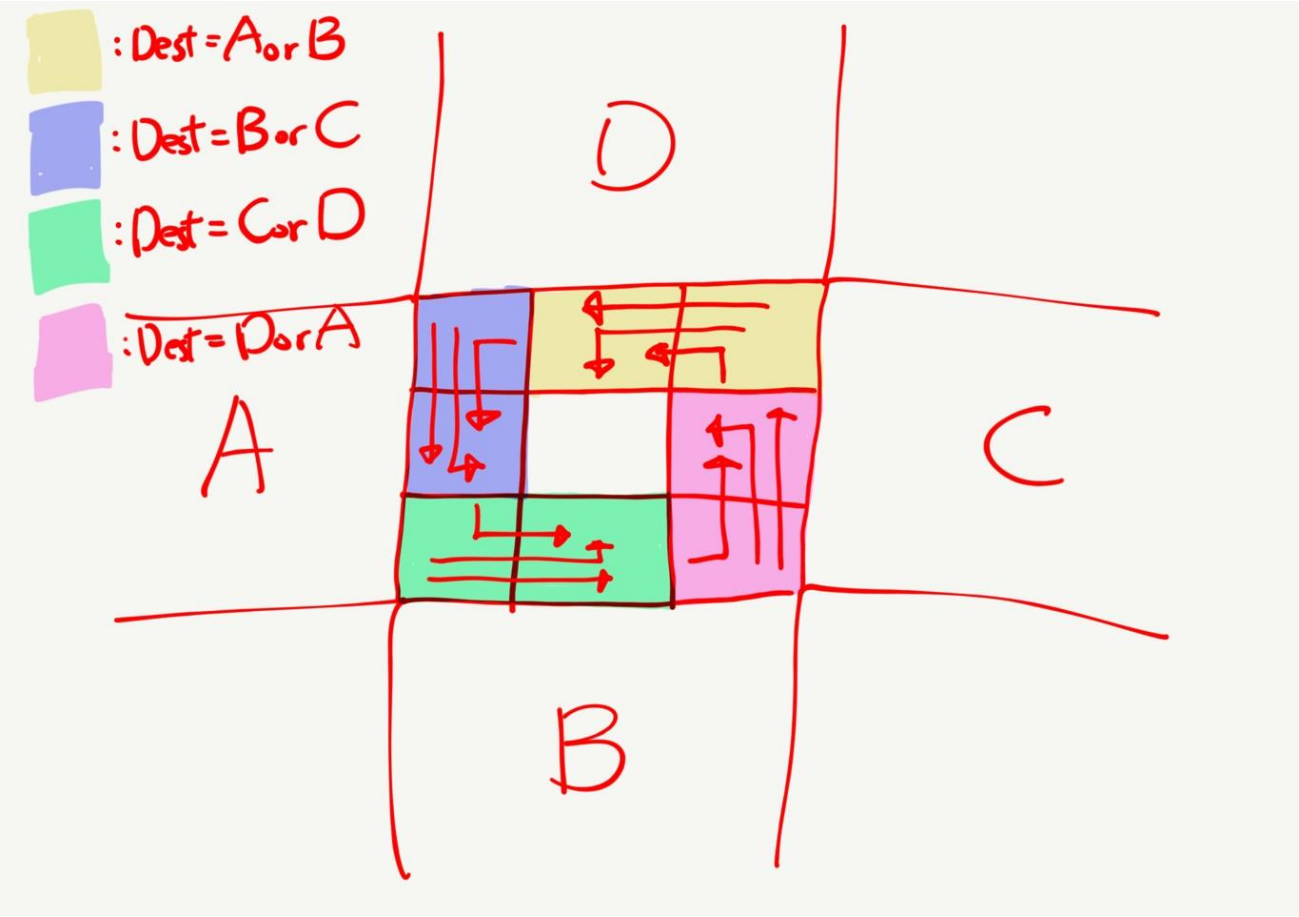
모든 차량은 자신의 step 이 종료되면 blocked_thread_list 에 추가되고, blocked_thread_cnt 를 증가시키고, thread_block 을 이용하여 block 된다.

만약 증가시킨 blocked_thread_cnt == thread_cnt 를 만족한다면 모든 차량 쓰레드의 step 이 종료되었다는 것을 의미하고 sema_up(&sem_TRI)으로 manager thread 를 깨운뒤 crossroads_step 을 하나증가시키고 다음스텝을 진행한다.

Lock_acquire 을 사용하면 쓰레드가 acquire 함수안에서 block 되기 때문에 확실하게 이 쓰레드가 움직일 수 있는 차량인지 아닌지 여부를 판단할 수가없다. 이러한 문제를 threadlist 및 blocked_thread_list 를 만듦으로서 해결하였다.

Step 2 해결방안

교차로에서의 DEADLOCK 발생 경우의 수를 고려해 보았다.



위와같이 색깔별로 칠한 부분의 차량이 특정한 방향으로 움직일 때 deadlock 일 발생한다.

이를 통해 고안한 해결방안은 step 이 종료될 때 deadlock 여부를 판단한 후에,

만일 deadlock 상태가 발생했다면 이를 회복하는 recover step 으로 진행하도록 하였다.

Deadlock 판단은 manager thread 에서 종료상태의 map_lock 을 관찰해 두가지 조건을 만족할 때 데드락임을 확신한다.

첫번째 조건 : 교차로 내의 map_lock 을 확인해 모든(8 개)의 lock 의 holder 가 존재한 경우.

두번째 조건 : 첫번째 조건을 만족한 상태에서 각 블록의 holder thread 를 찾고, 그 thread 의 dest 가 위에서 미리 지정한 방향과 모두 일치할 경우

첫번째 조건과 두번째 조건을 둘다 만족할 때 deadlock 임을 판단한다.

이를 위해서 destlist 를 통해서 각 thread 의 dest 를 저장하였다.

Deadlock 회복은 회전교차로에서 영감을 받아 교차로 내의 모든 차량을 반시계방향으로 한칸 회전시키는 것이다.
교차로 차량을 반시계방향으로 한칸 회전시키기 위해서는 다음 스텝을 deadlock recover step 으로 지정해 교차로의 lock 을 전부 release 하고 교차로 차량만 움직임을 허락하는 방식으로 고안하였다. 교차로 lock 이 전부 release 된상태라면 자연스럽게 교차로내의 차량은 반시계방향으로 한칸 회전하게 될것이다.

다음스텝을 deadlock recover step 으로 지정하기 위해서 is_deadlock_recover 변수를 생성하였다. 이는 스텝이 종료될 때마다 false 으로 초기화되고, 스텝이 종료되고 deadlock 상황이 발생하였을 때 true 로 전환된다.
또한 각 교차로 칸 lock holder 을 NULL 로 초기화하고(lock_release 를 사용하지않은이유 : lock_held_current_thread 으로 인해 오류발생함), 교차로 내의 차량 쓰레드만 unblock 한다.
왜냐하면 deadlock recover step 은 교차로 내의 차량만 움직일 뿐이고, 처음 차량 움직임이 시작된 이후에 차량이 추가되지않는한 다른 차량의 움직임은 없기 때문이다.

각 차량 쓰레드의 try_move 함수에서 deadlock recover 상황 발생시 하지 조정해야 할 부분은

1. lock_release 하는 경우
2. 모든 차량 스텝이 종료되었음을 판단하는 경우

첫번째 경우에는 is_deadlock_recover 변수를 통해 release 를 하지않으면되고,
두번째 경우에 본래 스텝이 종료됨을 판단하는 기준은 blocked_thread_cnt == thread_cnt 이지만, 현재 blocked_thread_cnt 는 최대 8 으로 성립하기 어렵다. 따라서 기존 조건을 blocked_thread_cnt == thread_cnt && !is_deadlock_recover 으로 변경하고 Is_deadlock_recover && blocked_thread_cnt == 8 일경우 deadlock recover step 이 종료되었음을 판단한다.

데드락의 recover 과정을 신호등방법, 교차로 진입시 데드락여부 확인하는 방법, 데드락 발생시 돌아가는 방법등을 생각했지만,
신호등 방법은 교차로진입이 1 스텝에 1 차량으로 효율이 떨어지고,
교차로 진입시 데드락 여부를 확인하기 위해 다음다음 스텝까지 예상을 해야되서 복잡했고,
데드락발생시 돌아가는 방법은 각 차량의 히스토리를 저장하고 있어야 가능해서
현재 고안한 회전교차로 방법으로 제일 간단히 해결되었다. 교차로 데드락의 경우 발생 상황이 정해져 있어서 고정적인 해결방법으로도 해결이 가능했다.

결론 : 개선방안

쓰레드의 개수를 결정하는 곳에서 많은 우여곡절이 있었고 결국 확실한 방법을 모색하지 못했다. 또한 처음의 block 을 해제하는 곳에서도 우여곡절의 결과가 나타났다. 만일 과제가 아니었다면 run_crossroads 의 함수에서 쓰레드개수를 가져와 확실하게 쓰레드 개수를 가져오지 않았을 까하는 개선방안이 있다.

배운점 및 어려웠던 점

개발환경 구축과 기존 제공된 함수 해석만 하루.

Sync 함수 및 thread 함수를 적응하는데에만 하루.

Semaphore 및 lock 을 적용해서 멀티 쓰레드의 프로그래밍을 적응하는데에 하루.

Step1 을 해결 및 구현하는데 이틀, Step2 을 해결 및 구현하는데 하루.

정말 마부위침이란 말이있듯이 결국 하루하루의 노력을 통해 결국 성과를 이룬 것 자체만으로도 많은 배운 점이있었다. 데드락 상황을 연출하고 자연스럽게 빠져나오는 모습을 보면서 환희에 넘쳤다.

어려운 점은 당연히 멀티 쓰레딩을 처음 맛본 코더로서 쓰레드들을 동기화 시켜준다는게 어렵기도 했다. 그런데 하다보니깐 익숙해지고 자연스럽게 semaphore 를 사용하는 내 모습을 보면서 한단계 더 성장한 느낌도 들었다. 하지만 이번 교차로 데드락 상황은 발생 상황이 예상가능하고, 회복이 가능한 상황이 었지만, 예상불가능하고 감지도 불가능한 데드락일 경우 어떻게 풀어나가야 할지 더욱더 높은 산을 만난 느낌이었다.

이번과제로 인해서 운영체제에 대한 관심이 더욱 늘어났고 자신감도 동기도 생긴 것 같다.

이상입니다. 읽어주셔서 감사합니다.