Operating Systems

Spring, 2023
School of Software, CAU

Project #1
- A Thread Systems and Synchronization -

[프로젝트 보고서]

학번: 20192513

이름: 오 양 호

1. 서론

본 보고서에서는 교차로 문제의 해결 방안과 결과에 대해 기술한다.

1.1 단위 스텝 문제

단위 스텝 문제를 해결하기 위해 모든 차량이 움직였는지를 확인하기 위한 semaphore 하나와 다른 차량을 기다리느라 이동하지 못하는 차량을 세기 위한 ready_cnt, 그리고 전체 차량의 수를 저장하기 위한 total_cnt를 int형 변수로 선언하였다.

<vehicle.h> :

```
struct semaphore sema_step;
```

```
int total_cnt;
int ready_cnt;
```

1.2 교차로 동기화 문제

차량들이 교차로 안으로 진입하기 전에 자신의 경로에 한 칸씩 lock획득을 시도해보도록 했다. Lock 획득에 전부 성공한 차량은 이동하고 이동하지 못한 차량은 이미 획득한 lock은 갖고 있는 상태로 대기한다. 이러한 Hold&Wait 때문에 잠재적인 deadlock 상황이 발생할 수 있다. 따라서 circular wait를 막기 위해 semaphore로 교차로 내의 진입 가능한 차량 수를 세 대로 제한했다.

```
struct semaphore sema_crossroad;
```

2. 본론

본론에서는 문제 해결 방법에 대해 상세히 기술한다.

1.1 단위 스텝 문제

본론에서는 서론에서 설명했던 semaphore와 변수 두 개를 어떻게 사용했는지 설명한다. 각 차량이 vehicle_loop의 while문에 진입하면 전역 semaphore로 선언된 sema_step에 대해 sema_try_down을 시도한다. 만약 4대의 차량이 존재한다고 가정했을 때 init_on_mainthread에서 4로 초기화 되었을 것이다. 각 차량은 vehicle_loop에서 sema_try_down을 실행하므로 이동 전에 모든 차량이 sema_try_down에 성공하여 sema_step.value는 0이 된다. 처음 4대의 차량은 모두 이동에 성공하고 다시 while문을 반복하여 sema_try_down을 시도해 볼 것이다. 지금 상태에서 sema_step.value는 0이기 때문에 가장 먼저 두 번째 while 반복을 실행한 스레드는 sema_try_down에 실패한다. 실패를 감지했으면 crossroads_step 변수를 1 올리고, 다시 sema_init(&sema_step, total_cnt - ready_cnt)로 초기화한다. 첫 번째 반복에서 모든 차량이 이동에 성공했으므로 total_cnt - ready_cnt = 4 - 0 = 0 이므로 sema_step은 다시 4로 초기화된다. 두 번째 반복에서 4대의 차량 중 2대의 차량이 나머지 2대의 차량을 기다리느라 움직이지 못하는 상태가 되었다면 ready_cnt는 2가 된다. ready_cnt를 세는 방법은 try_move 함수 내에서 ready_cnt를 1 증가시키고 이동에 성공한 차량 (try_move의 반환값이 1인 차량)에 대해서만 다시 ready_cnt를 1 감소시키면 현재 다른 차량이 점유하고 있는 공간의 lock을 획득하기 위해 lock_try_acquire를 시도 중인, 즉, 대기 중인 차량의 수를 셀 수 있다. 대기 중인 차량은

이동 가능한 차량이 교차로를 완전히 통과할 때까지 움직일 수 없으므로 sema_step을 계속 total_cnt - ready_cnt로 초기화한다. 이렇게 하면 vehicle_loop의 while문의 두 번째 반복에서 sema_step이 2로 초기화되고 이는 이동 가능한 차량이 모두 움직였을 때만 crossroad_step을 올리는 동작을 보장한다.

이 내용을 간단하게 요약하자면 아래와 같다:

① 각 차량은 이동 전에 sema_down
② 각 차량은 이동에 성공 시 sema_up
③ Semaphore의 value가 0이 되었다면 모든 차량이 이동했다는 뜻
④ 이를 감지하면 crossroad_step++
⑤ 이동 가능한 차량 수(전체 차량 - 대기 차량)만큼 semaphore를 초기화
⑥ 반복

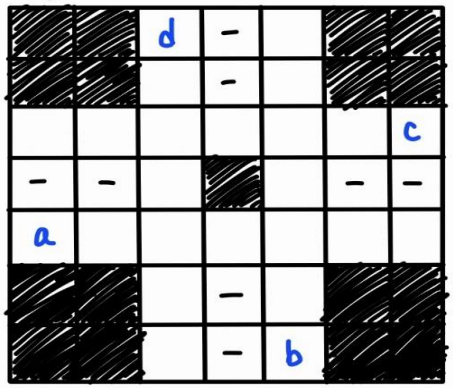
아래는 이해를 돕기 위한 소스코드의 일부분이다.

<vehicle.c 내의 vehicle_loop의 while문 내의 코드…>

```
if (sema_try_down(&sema_step)) {

    res = try_move(start, dest, step, vi);
    if (res == 1) {
        step++;
        ready_cnt--;
        unitstep_changed();
    }

    /* termination condition. */
    if (res == 0) {
        break;
    }

}
else {
    sema_init(&sema_step, (total_cnt - ready_cnt));
    crossroads_step++;
}
```

아래는 그림으로 동작 원리를 설명한 것이다:

<상황 그림>



현재 sema_step.value = 4

<loop 1>

a → sema_try_down (S=3)
b → sema_try_down (S=2)
c → sema_try_down (S=1)
d → sema_try_down (S=0)
…… 교차로 진입 직전

대기 중인 차량 : b, d ← ready_cnt = 2
전체 차량 : a, b, c, d ← total_cnt = 4

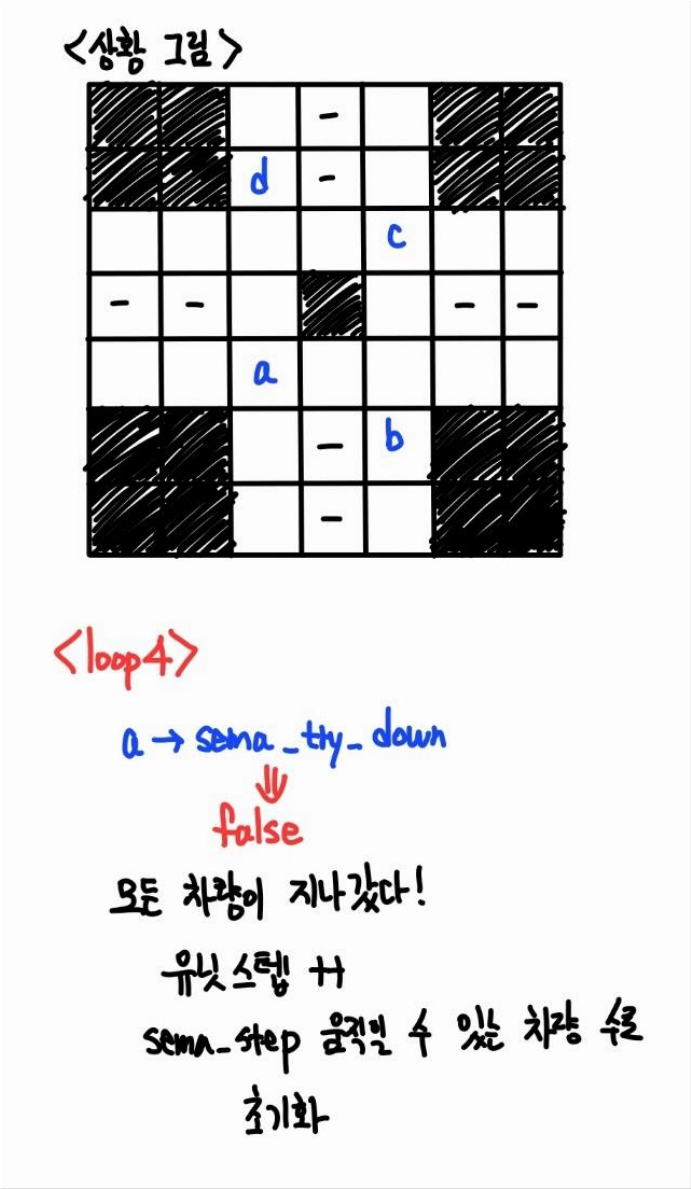
<loop 2>

a → sema_try_down
⇓
false

모든 차량이 지나갔다!
" crossroads_step++ "
&
S = total_cnt - ready_cnt
= 4 - 2 = 2
움직일 수 있는 차량 수

<loop 3>

a → sema_try_down (S=1)
c → sema_try_down (S=0)



여기서 semaphore의 value가 0이 되었음을 if문에 sema_try_down을 호출하여 실패하면 else문에서 crossraods_step을 증가시키고 semaphore를 다시 초기화하도록 설계했다. 또한 init_on_mainthread에서 sema_step을 전체 스레드 수로 초기화하면 단위 스텝을 0부터 세기 시작하는데, 이는 sema_step을 0으로 초기화하면 if문에서 sema_try_down에 실패하고 else문으로 들어가 crossroads_step을 1 올리고 시작하여 이 문제를 해결할 수 있다.

1.2 교차로 동기화 문제

먼저, 차량은 thread, 교차로 구역은 critical section이라고 할 수 있다. 차량은 같은 공간에 두 대가 존재할 수 없도록 되어있기 때문에 각 thread가 다른 thread가 이미 사용하고 있는 자원에 접근할 수 없는 [1] *Mutual Exclusion*이 적용되어 있다. 또한 다른 차량이 확보한 공간을 다른 차량이 강제로 뺏을 수 없으므로 [2] *No-Preemption*도 적용되어 있다. 본 과제에서 ***"차량이 교차로 내에 진입하면 경로가 겹치는 다른 차량의 진입이 제한된다."***는 조건이 있기 때문에 각 차량은 교차로에 진입하기 전에 자신이 진행해야 할 경로에 한 칸씩 lock_try_acquire함수로 락을 획득해보고, 필요한 모든 락을 획득하면 이동하도록 했다. 여기서 각 차량들의 시작지(start)와 목적지(dest)을 알면 교차로 내에서 필요한 락의 개수를 구할 수 있도록 count_required함수를 만들었다. 이 함수는 단순히 출발지와 목적지의 차(difference)로 필요한 락의 개수를 반환하도록 했다. count_required함수로 구한 필요한 락의 개수는 vehicle.h에 있는 vehicle_info구조체에 required라는 int형 변수를 추가하여 그곳에 저장하도록 했다. 각 차량이 try_move함수 내에서 각 step에 대해 어떤 이동 로직이 적용되어야 하는지 if문으로 작성했다.

- 차량이 각 스텝에서 어떻게 행동해야 하는지는 다음과 같다:
  ① step == 0일 때, vi->state를 VI_STATUS_READY에서 VI_STATUS_RUNNING으로 바꾸고 다음 위치에 대한 lock을 획득하고 한 칸 이동한다(vi->position을 다음 가야 할 위치로 업데이트한다).
  ② step == 1일 때, 다음 위치에 대한 lock을 획득하고 현재 위치의 lock을 해제하고 한 칸 이동한다.
  ③ step == 2 일 때, step이 2일 때는 모든 차량이 교차로 진입 직전인 상태이다. 따라서 자신의 교차로 내에서 지나가야 하는 공간에 대한 lock을 하나씩 획득해보고, 획득하는데 실패하면 획득할 때까지 대기한다. 이는 lock_try_acquire함수가 true를 반환할 때까지 while문을 반복하여 도는 것으로 구현했다. 이 과정에서 모든 필요한 lock을 획득한 차량은 이동하고 그렇지 못한 차량은 이미 획득한 lock을 확보해둔 상태로 얻지 못한 lock을 얻을 때까지 while문에 갇혀 있게 된다. 따라서 먼저 lock을 획득한 차량이 먼저 교차로를 지나가고 lock을 획득하지 못한 차량은 기다렸다가 다른 차량이 지나가고 나면 이동하게 되는 것이다.
  ④ step이 0,1,2 중 하나가 아닐 때, 다음 위치가 교차로 내에 있다면 이미 필요한 lock을 모두 획득한 상태이고, 교차로 내에서 진행할 때 다른 차량의 진입을 제한해야 하므로 이동할 때 lock을 계속

유지하고 있어야 한다(= lock을 해제하면 안 된다). 따라서 이 경우에는 그냥 차량의 이동을 위해 위치를 업데이트하는 작업만이 필요하다.

⑤ step이 0,1,2 중 하나가 아닌데 다음 위치가 교차로 내에 있지 않다면 두 가지 경우가 있다. 하나는 교차로에 빠져나가기 직전인 경우이고, 다른 하나는 교차로에서 빠져나와 목적지로 향하는 경우이다. 첫 번째 경우인 교차로에 빠져나가기 직전에 해야 할 동작은 교차로 내에서 다른 차량의 진입을 제한하기 위해 유지하던 lock을 모두 해제하고, 자신이 다음 갈 위치에 lock을 획득하고 이동하는 것이다. 이 동작이 단 한 번만 동작해야 하므로 vehicle.h의 vehicle_info 구조체에 released라는 bool형 변수를 만들어 true 초기화한 다음 교차로에서 빠져나올 때 사용했던 lock을 해제하고 나면 다시 이 동작을 중복해서 하지 않도록 released를 false로 바꿔준다. released가 false일 때는 그냥 평범하게 다음 위치의 lock을 획득->현재 위치의 lock을 해제->이동하는 순으로 동작하도록 했다.

기본적으로 교차로 내에서 자신이 진입해도 되는지 확인하는 과정에서 [3] *Hold and Wait*가 적용되어 있다. (이미 획득한 lock을 유지한 채로 다음 획득할 lock을 기다리고 있으므로) 이러한 방식으로 교차로 동기화를 구현하면 잠재적으로 deadlock이 발생할 수 있다. 현재까지 Mutual Exclusion, No-Preemption이 기본적으로 적용되어 있고 교차로 내에서 차량의 동기화를 위해 Hold&Wait도 적용되었으므로 여기에서 [4] *Circular Wait*가 발생하는 경우가 존재한다는 것을 증명하면 잠재적인 deadlock 상황이 존재한다고 할 수 있다. 그리고 Circular Wait가 발생하는 경우는 존재한다.

- 다음 그림은 Circular Wait가 발생할 수 있는 경우를 표현한 것이다:

이처럼 4가지 모든 방향에서 동시에 차량들이 접근했을 때 a는 m이 lock을 해제하기를 기다리고->m은 e를->e는 i를->i는 a를 기다리는 Circular Wait가 발생한다. 이러한 deadlock은 필요충분조건 중 하나만 제거해도 해결할 수 있다. 과제의 초반 단계해서는 required와 acquired 변수를 선언하여 차량이 교차로에 진입하기 전에 lock을 획득할 때마다 acquired++를 하여 acquired == required가 성립할 때는 이동하고 성립하지 않을 때는 획득했던 lock을 모두 해제하는 식으로 Hold and Wait를 제거하여 해결했었다. 이런 식으로 구현해도 deadlock 문제는 해결할 수 있지만, 후에 단위스텝 문제에서 ready_cnt를 세는 것이 어려워서 Circular Wait를 제거하는 방향으로 선회했다. Circular Wait를 제거하기 위해 전역 semaphore인 sema_crossroad를 선언하고 init_on_mainthread에서 3으로 초기화했다. 그리고 각 차량이 교차로에 진입하기 전 (step == 2) 일 때 sema_down을 하고 교차로에 진입한 직후(step == 3)에 sema_up을 하는 방식으로 교차로 내에 진입 가능한 차량의 수를 세대로 제한했다. 교차로 내에 진입 가능한 차량이 세대로 제한되면 4가지 방향에서 차량들이 동시에

접근하더라도 한 방향은 진행할 수 있어서 Circular Wait의 고리를 끊을 수 있다. 이러한 방식으로 차량들이 교차로를 무사히 빠져나올 수 있다.

(추가 내용: 과제의 최종 작업물과는 관련이 없으며, 만든 것이 아까워서 작성합니다…)
이외에도 Hold and Wait 조건을 제거하여 deadlock을 막을 수도 있다. 방법은 이렇다. 교차로 내에서 필요한 lock의 개수를 저장하는 required 변수와 lock을 시도해서 획득하는데 성공할 때마다 카운트를 하게 해줄 acquired 변수를 만들어 required와 acquired가 같을 때는 이동을 하고, 같지 않을 때는 지금까지 얻었던 lock들을 모두 해제해주는 것이다. 이러한 방식으로 Hold and Wait을 제거해줘서 deadlock을 막을 수 있는 방법으로 구현했었는데, 이 방법에서 여러 다른 어려움을 만나게 되어 중간에 구조를 바꿨었다.

- 아래는 그 시도에 대한 try_move함수의 일부분이다:

```
for (int i = 0; i < required; i++) {
    if(lock_try_acquire(&vi->map_locks[vehicle_pa
        acquired++;
    }
    else {
        break;
    }
}

if (acquired < required) {
    for(int i = 0; i < acquired; i++) {
        lock_release(&vi->map_locks[vehicle_path[
    }

    return -1;
}
```

3. 결론

과제를 하면서 정말 여러가지 난관을 만났었다. 예를 들어 차량이 겹치는 문제, 차량이 두 칸씩 움직이는 문제, 차량이 늦게 생성되는 문제 등 내가 생각했던 것에서 크게 벗어난 여러 이상한 결과가 나왔었다. 지금까지 했던 일반적인 코딩과는 다르게 실행의 흐름이 여러가지가 존재하는 thread적 관점에서 프로그래밍을 하려고 하니 뭔가 잘못되어도 어느 부분에서 잘못된 것인지 알기 힘들었던 것이 과제를 하면서 가장 힘들었던 점이다. 본 과제를 진행하면서 이론적으로 알고 있던 동기화 문제와 교착상태 해결법을 실제로 적용해볼 수 있는 경험이 되었고 이를 실제로 적용하는 것이 생각보다 많이 까다롭다는 것을 깨달았다. 또한 실행의 흐름이 여러가지인 상황을 고려하여 프로그램을 짤 수 있는 능력이 조금이나마 길러진 것 같아 뿌듯했다.