〈레포트〉

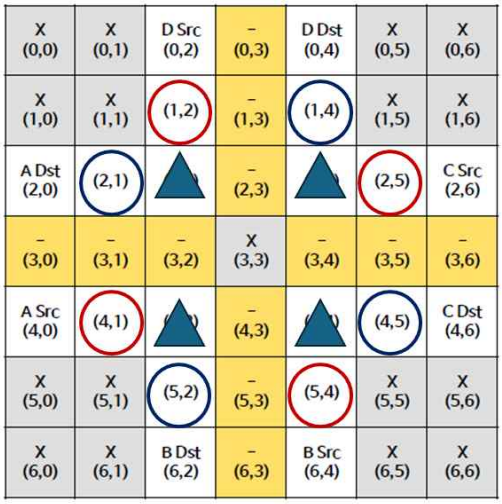
교차로 문제

강의명: 운영체제
담당교수: 박상오 교수님
작성자: 신성섭
학번: 20226XX2
학과: 소프트웨어학부

● 유의 사항

과제를 수행하기 위해 총 2번의 시도가 있었고 이 보고서는 두 번의 시도에 대해 모두 설명합니다. 두 프로그램 중 하나는 데드락 방지를 실패한 코드고, 나머지 하나의 코드는 올바르게 동작하는 코드입니다. 올바르게 동작하는 코드(success 폴더)을 가지고 Test 해주세요.

1. 데드락 방지 알고리즘 (시도1)



교착상태를 일으키는 4가지 조건은 상호배제, 점유대기, 비선점, 순환대기이다. 이 프로그램에서는 순환대기인 상황을 사전에 방지하여 교착 상태 회피를 하였다.

교착상태 회피를 하기 위해서 교차로에 진입하기 전에 우선 교차로에 진입해도 되는 지를 확인한다. 위의 그림에서 빨간색 원 위치에서 진입 가능 여부를 검사한다. 만약 진입이 불가능하다면 빨간색 원 위치에서 대기하고 다른 자동차가 교차로를 통과 할 때까지 기다린다.

빨간색 원에서 자동차는 교차로에 대한 자원을 획득 할 수 있는 지 검사하는데 교차로의 자원은 삼각형으로 표시된 4개의 자원이다. 이 4개의 자원은 오직 1개의 자동차에 의해서만 소유된다. 자동차는 자신이 지나는 경로 위에 있는 모든 교차로 자원을 요청하고, 요청한 모든 자원이 소유 가능한 경우에만 자원을 소유하게 된다.

자동차가 교차를 통과하여 파란색 원 위치를 지나게 되면 자동차가 가지고 있는 교차로에 대한 자원을 반환한다.

위와 같은 알고리즘으로 교차로를 관리할시, 삼각형으로 표시된 부분은 항상 자동차 1대만 지나 갈 수 있으므로 순환대기가 일어날 가능성이 없고, 따라서 데드락 발생 가능성을 회피할 수 있다.

2. 시도1 에 대한 코드 구현 방법

– 필요한 교차로 자원 & 진입/반환 위치

```
// move_point[i][j][k] 는 i가 시작점이고 j가 도착점인 경우 k번째 모서리를 지나가면 1 아니면 0
int move_point[4][4][4] = {
    {
        {1,1,1,1},
        {0,0,1,0},
        {0,0,1,1},
        {0,1,1,1}
    },
    {
        {1,1,0,1},
        {1,1,1,1},
        {0,0,0,1},
        {0,1,0,1}
    },
    {
        {1,1,0,0},
        {1,0,1,0},
        {1,1,1,1},
        {0,1,0,0}
    },
    {
        {1,0,0,0},
        {1,0,1,0},
        {1,0,1,1},
        {1,1,1,1}
    }
};

// 진입 결정을 내리는 위치
int decision_point[4][2] = { {1,2},{2,5},{4,1},{5,4} };

// 교차로 자원 반환을 하는 위치
int release_point[4][2] = { {1,4},{2,1},{4,5},{5,2} };
```

move_point는 총 16가지의 경우에 대해 각각의 경우에 대해 필요한 교차로 자원을 저장하는 배열이다. 1이면 교차로 자원이 필요한 것이고, 0 이면 필요 없는 것이다.
진입결정을 하는 위치와 자원을 반환하는 위치도 배열로 관리 하였다.

– 세마포어, 전역변수

```
// 세마포어 선언
struct semaphore move_decision_sema;
struct semaphore waiting_thread_sema;
struct semaphore all_threads_done;

// 전역 변수
int waiting_threads; // 대기 중인 쓰레드 개수
int position[4]; // {2,2},{2,4},{4,2},{4,4}를 지날 예정인 차가 있나?
int total_threads; // 전체 쓰레드 개수
```

멀티쓰레드 환경에서 전역변수를 올바르게 관리하기 위해서 세마포어들을 선언해주었다. move_decision_sema는 move_decision을 할 때 특정 순간에 오직 1대의 쓰레드만 교차로 자원을 변경하게 하기 위해서 했고, wating_thread_sema는 waiting_threads의 개수 관리를 위해, all_thread done은 모든 thread가 다 수행된 상태인지를 확인하기 위해 선언해주었다.

– 초기화 함수

```
void init_on_mainthread(int thread_cnt) {
    /* Called once before spawning threads */

    sema_init(&move_decision_sema, 1); // 움직일지 말지를 결정하는 세마포어 초기화
    sema_init(&waiting_thread_sema, 1); // waiting_thread 수를 관리하는 세마포어 초기화
    sema_init(&all_threads_done, 0); // 모든 스레드 완료 세마포어 초기화
    waiting_threads = 0; // 대기 중인 스레드 수 초기화
    crossroads_step = 0; // 단위 스텝 초기화
    total_threads = thread_cnt; // 전체 쓰레드 수 초기화

    for (int i = 0; i < 4; i++) { // 위치 초기화
        position[i] = 0;
    }
}
```

세마포어와 전역변수를 초기화 한다.

– 교차로 진입 결정 함수

```
// 교차로 진입 결정을 내리는 함수
void move_decision(struct vehicle_info* vi) {

    // 1이면 교차로 진입 가능 그 외는 불가능.
    int possible_move = 1;

    int start = vi->start - 'A';
    int dest = vi->dest - 'A';

    for (int i = 0; i < 4; i++) {
        if (move_point[start][dest][i] == 1) {
            if (position[i] != 0) {
                possible_move = 0;
                break;
            }
        }
    }

    // 교차로에 진입 가능 여부 업데이트
    if (possible_move != 1) {
        vi->state = VEHICLE_STATUS_STOP;
    }
    else {
        vi->state = VEHICLE_STATUS_RUNNING;
        for (int i = 0; i < 4; i++) { // 교차로 모서리를 점유
            if (move_point[start][dest][i] == 1) {
                position[i] = 1;
            }
        }
    }
}
```

교차로에 진입가능한지를 판단하는 함수이다. 모든 자원을 얻을 수 있는 경우에만 진입을 허락하고, 진입을 하는 경우 i번째 자원을 소유한 것을 의미하기 때문에 point[i]를 1로 바꿔준다. 교차로에 진입이 불가능한 경우 차의 상태를 stop으로 설정해준다.

- try_move()

```
static int try_move(int start, int dest, int step, struct vehicle_info* vi)
{
    struct position pos_cur, pos_next;

    pos_next = vehicle_path[start][dest][step];
    pos_cur = vi->position;

    if (vi->state == VEHICLE_STATUS_RUNNING) {
        /* check termination */
        if (is_position_outside(pos_next)) {
            /* actual move */
            vi->position.row = vi->position.col = -1;
            /* release previous */
            lock_release(&vi->map_locks[pos_cur.row][pos_cur.col]);
            return 0;
        }
    }

    /* lock next position */
    lock_acquire(&vi->map_locks[pos_next.row][pos_next.col]);
    if (vi->state == VEHICLE_STATUS_READY) {
        /* start this vehicle */
        vi->state = VEHICLE_STATUS_RUNNING;
    }
    else {
        /* release current position */
        lock_release(&vi->map_locks[pos_cur.row][pos_cur.col]);
    }

    if (vi->state == VEHICLE_STATUS_STOP) {
        return -1;
    }


    /* update position */
    vi->position = pos_next;

    return 1;
}
```

자동차의 상태에 따라 자동차의 위치를 결정한다.

– Vehicle_loop

1) 교차로 진입 결정

```
void vehicle_loop(void* _vi)
{
    int res;
    int start, dest, step;

    struct vehicle_info* vi = _vi;

    start = vi->start - 'A';
    dest = vi->dest - 'A';

    vi->position.row = vi->position.col = -1;
    vi->state = VEHICLE_STATUS_READY;

    step = 0;
    while (1) {
        /* vehicle main code */

        // 교차로 진입 결정을 내려야 하는 위치면 판단을 내림
        for (int i = 0; i < 4; i++) {
            if (vi->position.row == decision_point[i][0] && vi->position.col == decision_point[i][1]) {
                sema_down(&move_decision_sema);
                move_decision(vi);
                sema_up(&move_decision_sema);
            }
        }
```

교차로 진입을 결정하는 위치의 경우, 교차로 진입을 위한 판단을 내린다. 이때 자원에 대한 일관성을 보장하기 위해 세마포어를 이용했다. 교차로에 진입하기 전에 교차로의 모든 자원을 요청하기 때문에 교착 상태를 회피 할 수 있다.

2) 결과에 따라 움직이기

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

/* termination condition */
if (res == 0) {
    sema_down(&waiting_thread_sema);
    total_threads -= 1;
    sema_up(&waiting_thread_sema);
    break;
}
```

try_move 함수를 호출하고 그 반환 값에 따라 step을 증가시키거나 정지해 있거나, 종료한다. 쓰레드를 종료할때는 total_thread의 값을 1감소시키는 데, 이때도 일관성을 보장하기 위해 세마포어를 이용했다.

3) 단위 스텝 증가

```
// 단위 스텝 증가를 위한 로직
sema_down(&waiting_thread_sema);
waiting_threads++;
if (waiting_threads == total_threads) {
    waiting_threads = 0;
    sema_up(&waiting_thread_sema); // waiting_thread_sema를 다시 업
    crossroads_step = crossroads_step + 1;
    unitstep_changed();
    for (int i = 0; i < total_threads - 1; i++) {
        sema_up(&all_threads_done); // 모든 스레드 완료 세마포어 업
    }
}
else {
    sema_up(&waiting_thread_sema);
    sema_down(&all_threads_done); // 모든 스레드 완료 대기
}
```

모든 쓰레드가 작업을 수행할때까지 기다린다. 모든 쓰레드가 작업을 완료하면 단위 스텝을 증가 시킨다. 이때 step이 정확하게 반영되기 위해 2개의 세마포어를 이용하였다. 1개는 waiting thread 값을 관리하기 위한 세마포어고 나머지 한개는 모든 스레드가 완료되는지를 대기 하기 위한 세마포어이다. 세마포어의 순서가 정해져있기때문에 이부분에서는 교착상태가 일어날 가능성이 없다.

4) 교차로 자원 반환

```
        for (int i = 0; i < 4; i++) {
            if (vi->position.row == release_point[i][0] && vi->position.col == release_point[i][1]) {
                sema_down(&move_decision_sema);
                release_resource(vi);
                sema_up(&move_decision_sema);
            }
        }

    }

    /* status transition must happen before sema_up */
    vi->state = VEHICLE_STATUS_FINISHED;
}
```

교차로를 빠져 나오는 경우 교차로 자원을 반환한다. 이때 교차로 자원에 대한 일관성을 위해 세마포어를 이용했다.

3. 시도1 결과

이 프로그램은 내가 원하는 대로 잘 동작하지 않았다. 특히 step이 이상하게 변하는 현상이 일어났고 자동차의 개수를 늘렸을 때 프로그램이 데드락에 걸렸다.

이를 해결해 보려고 다양한 시도를 해 보았으나 실패했다.

디버깅을 해보고, 논리를 따라가 보았지만, 논리상의 오류는 보이지 않았다.

세마포어의 사용이 익숙하지 않아 잘못된 세마포어를 사용해서 발생한 현상이라고 추정했

고 그래서 처음부터 새로운 방식으로 코드를 처음부터 다시 작성하였다.
코드를 처음부터 다시 작성하는 김에 기존의 알고리즘을 좀 더 효율적으로 업그레이드 하였다. 그리고 검색을 해보니 pintos에는 semaphore의 사용을 쉽게 도와주는 lock과 condition이라는 것이 있어, semaphore 대신 이 구조체를 이용해 새로운 구현을 하였다.

4. 새로운 시도

X (0,0)	X (0,1)	D Src (0,2)	- (0,3)	D Dst (0,4)	X (0,5)	X (0,6)
X (1,0)	X (1,1)	(1,2)	- (1,3)	(1,4)	X (1,5)	X (1,6)
A Dst (2,0)	(2,1)	(2,2)	- (2,3)	(2,4)	(2,5)	C Src (2,6)
- (3,0)	- (3,1)	- (3,2)	X (3,3)	- (3,4)	- (3,5)	- (3,6)
A Src (4,0)	(4,1)	(4,2)	- (4,3)	(4,4)	(4,5)	C Dst (4,6)
X (5,0)	X (5,1)	(5,2)	- (5,3)	(5,4)	X (5,5)	X (5,6)
X (6,0)	X (6,1)	B Dst (6,2)	- (6,3)	B Src (6,4)	X (6,5)	X (6,6)

기존과 달리 빨간색 영역 안에 들어있는 자동차의 수를 최대 4대로 제한하여 데드락을 방지하는 코드를 작성해 보았다. 빨간색 영역에 자동차가 4대가 최대로 제한하는 경우 데드락이 회피된다.

5. 시도2에 대한 코드 설명

- Lock 과 전역변수

```
// Lock & condition
struct lock lock_step; // step의 일관성을 보장하기 위한 Lock
struct lock lock_critical; // 임계영역에 들어가는 자동차의 수의 일관성을 위한 Lock
struct condition waiting_thread; //wating 중인 thread들의 집합

// 전역 변수
int total_threads;  // 총 쓰레드(총 자동차)의 개수
int move_car_cnt; // try_move가 끝난 자동차의 총개수
int cnt_critical; // 임계영역에 들어가있는 자동차의 개수
```

데이터의 일관성을 위해 필요한 Lock들을 선언해 주었고, 전역변수로 총 자동차쓰레드의 개수, try_move가 끝난 자동차수, 임계영역에 들어가 있는 자동차의 개수를 선언해 주었다.

– 초기화 함수

```
void init_on_mainthread(int thread_cnt) {
    total_threads = thread_cnt; // 자동차 쓰레드 개수( 총 자동차 개수)
    lock_init(&lock_step);
    lock_init(&lock_critical);
    cond_init(&waiting_thread);
    move_car_cnt = 0; // 움직이는 자동차 0대
    cnt_critical = 0; // 임계영역에 들어가있는 자동차 수 0대

}
```

Lock과 전역변수를 초기화 한다.

– 움직임 판단

```
static int try_move(int start, int dest, int step, struct vehicle_info* vi)
{
    struct position pos_cur, pos_next;

    pos_next = vehicle_path[start][dest][step];
    pos_cur = vi->position;

    if (vi->state == VEHICLE_STATUS_RUNNING) {
        /* check termination */
        if (is_position_outside(pos_next)) {
            /* actual move */
            vi->position.row = vi->position.col = -1;
            /* release previous */
            lock_release(&vi->map_locks[pos_cur.row][pos_cur.col]);
            return 0;
        }
    }

    // 다음 위치의 Lock을 획득하려고 시도
    if (lock_try_acquire(&vi->map_locks[pos_next.row][pos_next.col])) {
        if (vi->state == VEHICLE_STATUS_READY) {
            vi->state = VEHICLE_STATUS_RUNNING;
        }
        else {

            // 맵 밖으로 나가지 않은 경우
            if (!is_position_outside(pos_cur)) {
                // 현재 쓰레드가 Lock을 가지고 있는 상황이면 Lock 풀기
                if (lock_held_by_current_thread(&vi->map_locks[pos_cur.row][pos_cur.col])) {
                    lock_release(&vi->map_locks[pos_cur.row][pos_cur.col]);
                }

                // 임계영역 Lock을 풀어준 후 임계영역에 들어 있는 자동차 수를 감소시킨다.
                if (vi->is_critical) {
                    lock_acquire(&lock_critical);
                    cnt_critical--;
                    vi->is_critical = false; // 임계영역 탈출
                    lock_release(&lock_critical);
                }
            }
        }
```

자동차가 Lock을 소유하고 있는 상황이라면 Lock을 풀어주고 현재 임계영역에 있는 상태라면 임계영역을 우선 탈출한거라고 생각한다.

```
    // 다음위치가 임계영역 안 인 경우
    if ((pos_next.row >= 2 && pos_next.row <= 4) && (pos_next.col >= 2 && pos_next.col <= 4)) {
        lock_acquire(&lock_critical);

        // 임계영역에 이미 4대의 자동차가 있는 경우 진입 불가능
        if (cnt_critical >= 4) {
            lock_release(&lock_critical);

            // 정지이므로 다음 위치의 map_Lock을 release 해야함.
            lock_release(&vi->map_locks[pos_next.row][pos_next.col]);

            //정지이므로 현재 위치의 Lock을 획득해야 함.
            lock_try_acquire(&vi->map_locks[pos_cur.row][pos_cur.col]);
            return -1; // 정지이므로 -1
        }

        cnt_critical++;
        vi->is_critical = true; // 자동차가 임계영역에 있음

        lock_release(&lock_critical);
    }

    // 자동차 위치 업데이트
    vi->position = pos_next;

    return 1;
}
return -1;
```

자동차의 다음 위치가 교차로 내부(빨간색 사각형 내부) 라면 임계영역에 들어가 있는 자동차의 수를 확인한다. 자동차가 최대 4대로 제한되므로 4대가 들어가 있는 상황이라면 정지하고 아니면 진입한다.

-vehicle_loop()

```
void vehicle_loop(void* _vi) {
    int res;
    int start, dest, step;

    struct vehicle_info* vi = _vi;

    start = vi->start - 'A';
    dest = vi->dest - 'A';

    vi->state = VEHICLE_STATUS_READY;
    vi->position.row = vi->position.col = -1;
    vi->is_critical = false;

    step = 0;
    while (1) {
        /* vehicle main code */

        res = try_move(start, dest, step, vi);

        if (res == 1) {
            step++;
        }

        //단위 스텝에 lock을 걸어서 step값에 일관성을 보장한다.
        lock_acquire(&lock_step);

        if (res == 0) {
            total_threads--;  // 자동차가 목적지에 도착했으므로 총쓰레드 개수(자동차의 총개수) 가 1 감소
        }
        else {
            move_car_cnt++; // 움직임이 끝난 자동차 1 증가
        }
```

단위 스텝의 일관성을 위해 우선 Lock을 건다. try_move() 함수 실행이 종료 되었으면 move_car_cnt의 값을 1 증가시킨다.

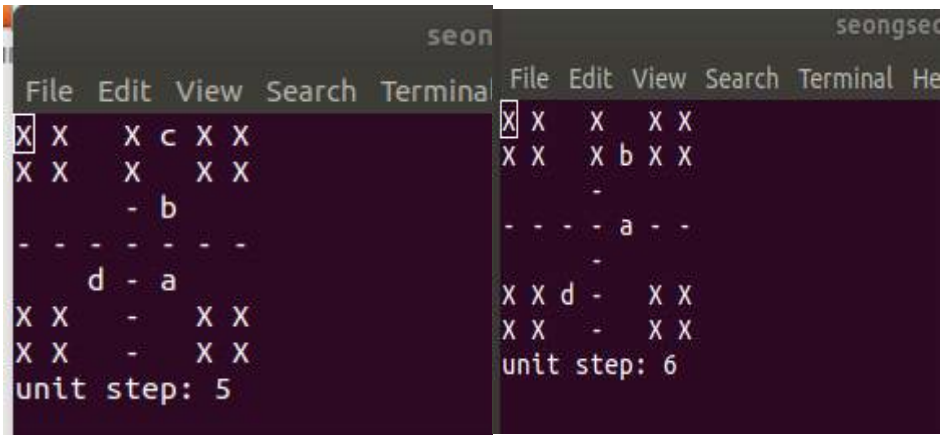
```
    if (move_car_cnt == total_threads) {
        //모든 자동차가 움직였으므로 단위 스텝 1증가
        crossroads_step++;
        unitstep_changed();
        move_car_cnt = 0;  // 다시 0으로
        cond_broadcast(&waiting_thread, &lock_step); // waiting된 thread들을 모두 활동 상태로 변경
    }
    else {
        //모든 자동차가 수행할때까지 정지
        cond_wait(&waiting_thread, &lock_step);
    }

    //단위 스텝 변경이 끝났으므로 Lock 풀기
    lock_release(&lock_step);

    if (res == 0) {
        break;
    }

}

/* status transition must happen before sema_up */
vi->state = VEHICLE_STATUS_FINISHED;
```

모든 자동차가 움직이기 전이라면 자동차 쓰레드를 wait 시킨다. 모든 자동차가 움직이 끝났으면 단위 스텝을 1증가시키고 wait된 쓰레드를 Active하게 바꿔준다.

6. 결과

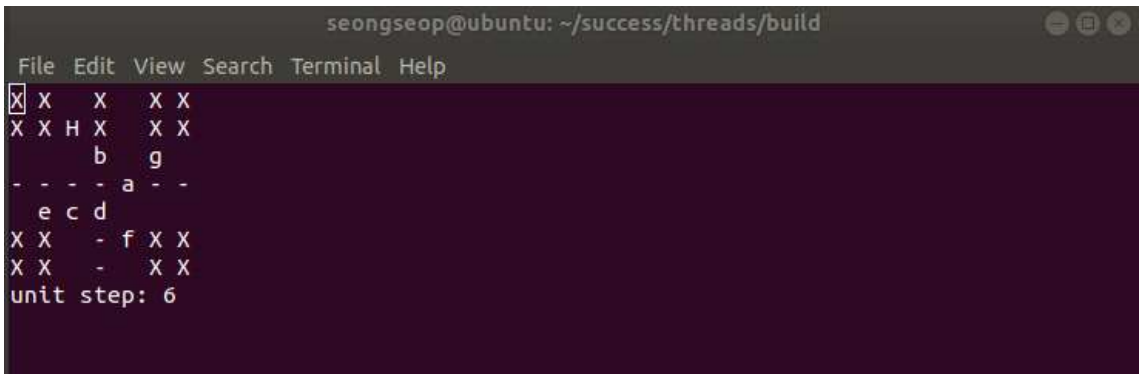
정상적으로 데드락이 안 걸리게 프로그램을 구현하는 데 성공하였다.

1. aAA:bBD:cCD:dDB 인 경우

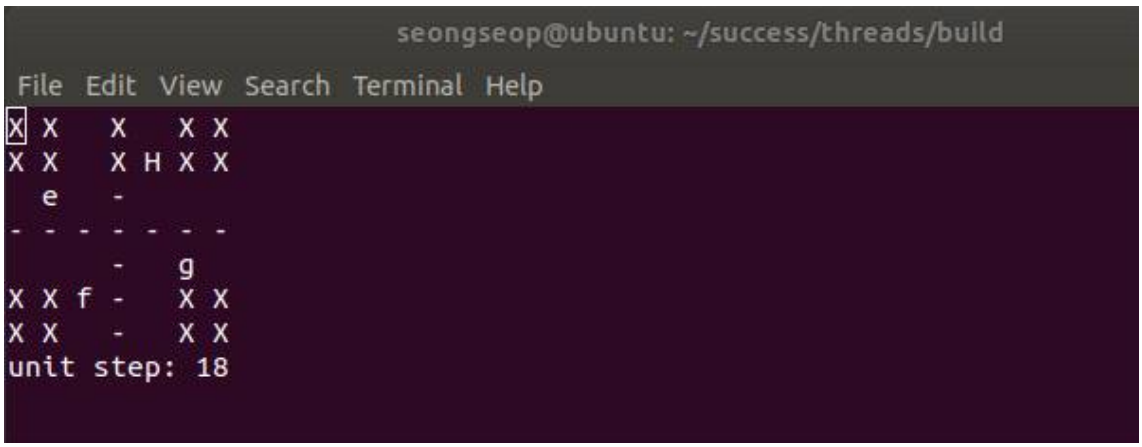


자동차들이 올바르게 움직임을 확인 할 수 있다. 또한 unit step도 올바르게 증가됨을 확인 할 수 있다.

2. aAA:bBB:cCC:dDD:eAA:fBB:gCC:HDD 인 경우



빨간색 사각형 영역은 최대 4대의 자동차만 동시에 있을 수 있으므로 e,f,g,H가 진입하지 않고 정지한 채로 기다리는 것을 확인 할 수 있다.



데드락 방지를 하지 않은 경우 이 예시 케이스는 데드락이 발생하는 경우인데, 적절한 회피 알고리즘을 사용함으로서, 데드락이 발생하지 않고 정상적으로 자동차들이 다 교차로를 통과한 모습을 보여준다.