Operating Systems

Spring, 2022

School of Software, CAU

Project #1

- A Thread Systems and Synchronization -

[프로젝트 보고서]

Template

학번: 20175371

이름: 강 명 석

1. 서론

1.1 단위 스텝 문제

교차로 문제는 여러 개의 차량 스레드가 동시에 이동을 진행할 때 발생가능한 병행성 문제를 다루는 문제이다. 하나의 턴에는 N개의 차량이 1)자신의 다음 경로로 한칸 이동을 하거나 2)자신의 다음 경로에 이미 차량이 존재하여 대기를 하는 행위를 수행하게 된다. 이때, 단위 스텝은 하나의 턴에서 모든 차량이 움직임을 수행했음을 표시해주는 역할을 한다.

1.2 교차로 동기화 문제

차량들은 교차로(intersection)에 진입을 하면, 잠재적인 교착상태(Deadlock)의 위험이 발생하게 된다. 본론에서는 이를 해결하기 위한 방법을 제시하여, 교차로에 진입한 차량들의 병행성 처리에 대한 방법을 기술한다.

2. 본론

1.1 단위 스텝 문제

1.1.1 배경

단위 스텝은 모든 전진 가능한 차량 스레드가 한 칸 전진 하거나, 전진해야 하는 경로에 차량이 있거나 교차로 문제에 의해서 전진하지 않고 대기를 하는 경우가 있다. 그래서 단위 스텝은 현재 턴에서 block 되지 않은 쓰레드 개수를 저장해 놓고, 각각의 쓰레드들이 이동을 하거나, 대기를 하는 로직을 완료하면 개수를 하나씩 차감 하여 마지막 쓰레드가 실행되어 저장된 변수가 0이 되면 단위스탭을 1 증가시키는 방법으로 구현할 수 있다.

단위 스텝의 기능을 구현하기 위해 사용된 변수 및 구조체는 다음과 같다.

[표 1 - 변수 및 구조체 목록]

Int remain_count	현재 턴에서 아직 이동을 하지 않은 쓰레드 수
Int lock_wait	이전 턴에서 block되었던 쓰레드가 현재 턴에서 이동된 다른 쓰레드에 의해서 block이 풀린뒤 이동을 하게 되면 전체 remain_count 수에 대한 보정이 필요. 이를 위한 변수
Semaphore *x	동시에 여러 쓰레드가 remain_count에 접근하지 못하도록 세마포어 이용
Condition2 *cond	한칸 이동이 끝난 쓰레드들은 현재 턴의 모든 쓰레드들의 움직임이 끝날 때 까지 block 되어야 함 그 기능을 구현하기 위해서 sync.c 파일에 있는 Condition 구조체를 변형하여 사용

1.1.2 프로그램 흐름

단위 스탭 문제에서는 크게 2가지 Case를 고려하여 구현하였다. 한 가지는 정상적으로 이동을 한 쓰레드이고, 다른 한 가지는 다음 이동 칸이 lock이 되어 있어서 block이 된 경우이다.

1) 한칸 이동을 한 쓰레드

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

```
sema_down(x);    // remain_count 보호
remain_count -= 1; // 이동을 완료해서 1 차감
if(remain_count > 0){
          sema_up(x);
        cond2_wait(cond);
}else{/*모든 쓰레드의 이동이 끝난 경우*/
          crossroads_step += 1;
          unitstep_changed();

/*cond2_wait에 들어있는 쓰레드 깨움*/
          cond2_broadcast(cond);

/*현재 턴에 lock에서 block이 풀린 쓰레드들 다음 턴에서
이동되도록 lock_wait로 수 보정*/
          avilable_move_count += lock_wait;
          remain_count = avilable_move_count;
          lock_wait = 0;
          sema_up(x);
}
```

다음 칸으로 이동 시 블록이 없다면 try_move에서 리턴을 한다. 리턴이 온다면 semaphore(x)를 이용해서 remain_count를 보호하며 1을 차단하고 remain_count의 수치를 보고 판단을 한다.

현재 남은 remain_count > 0 이면 cond2_wait()함수 를 통해서 해당 쓰레드는 block이 된다.
남은 remain_count가 0이라면 unitstep을 1 증가 시킨 뒤 unitstep_change()함수를 호출한다.
cond2_broadcast() 함수를 호출하여 지금까지 block된 모든 쓰레드들을 깨우고 다음 턴을 위한 변수를 설정한 뒤, 단일 스탭이 종료된다.

2) block이 된 경우의 동기화

```
void before_lock_wait(){
          sema_down(x);
          remain_count -= 1;
          avilable_move_count -= 1;
          sema_up(x);
}

if(lock_try_acquire(&vi-
>map_locks[pos_next.row][pos_next.col])){
}else{
          before_lock_wait();
          lock_acquire(&vi-
>map_locks[pos_next.row][pos_next.col]);
          lock_wait++;
          sema_down(x);
          sema_up(x);
}
```

기존 코드는 lock_acquire()함수를 통해서 다음 칸이 lock 되어 있다면, 바로 쓰레드가 block이 되는 구조였다. 하지만 이렇게 되면 전체 쓰레드 남은 개수를 저장하고 있는 remain_count 변수를 차감하는 로직을 추가할 수가 없기 때문에,
lock_try_acquire()함수를 대신하여 사용한다.

lock_try_acquire() 함수는 lock_acquire()함수와 다르게 만약 인자로 준 lock 구조체가 잠겨 있지 않다면, 바로 block을 하는 것이 아닌, false를 return 해 준다. 따라서 이를 이용하여 만약 인자로 주어진 lock이 잠겨 있다면 before_lock_wait()함수를 호출하여, 해당 쓰레드가 block 되기 전에 remain_count변수를 차감하는 로직을 수행하고 그 뒤에 lock_acquire()함수를 수행하여 해당 쓰레드가 block되도록 중간 과정을 추가해 줄 수 있다.

1.1.3 struct condition2

유닛 스탭을 구현하는 과정에서 주요 고려사항은 2가지 이다. 첫째는 이전까지 설명한 몇 개의 쓰레드들이 현재 턴에서 수행을 하고 있는지를 동기화 하는 과정이고 이는 remain_count를 통해서 관리하여 해결하였다.

두 번째는, 자신의 턴이 끝나고 대기를 하고 있는 쓰레드들의 처리에 대한 문제이다. 위에서 설명한 바와 같이 하나의 쓰레드는 하나의 턴에서 이동을 완료 하거나, 다음 lock이 잠겨있어서 block이 되는 경우로 나눌 수 있다. 이때 이동을 완료한 쓰레드들도 해당 턴의 다른 쓰레드가 완료될 때까지 block을 해주는 기능이 필요하고, 또한 그렇게 block된 여러 개의 쓰레드들을 마지막엔 모두 wake up 해주는 기능이 필요하다.

이 기능은 pintos 에서 제공해주는 자료구조 중 struct condition 구조체를 변형하여 구현하였다. 다음은 sync.h 파일에 존재하는 struct condition의 구조와 그것과 관련된 함수들이다.

```
/* Condition variable. */
struct condition
  {
    struct list waiters;/* List of waiting threads. */
  };

void cond_init (struct condition *);
void cond_wait (struct condition *);
void cond_signal (struct condition *);
```

```
void cond_broadcast (struct condition *);
```

세부 기능은 생략하고 주요하게 변형하여 사용한 함수는 cond_wait() 함수와 cond_broadcast()함수이다.

Cond_wait(cond)함수는 인자로 주어진 cond 변수에 있는 waiters 리스트에 현재 쓰레드를 넣어서 block을 시키는 기능을 하고, cond_broadcast(cond) 함수는 cond 변수에서 대기를 하고 있는 모든 waiters 들을 wake up 하는 기능이 있다.

이 구조체를 이용하여, 하나의 턴에서 이동이 완료된 쓰레드들은 cond_wait()함수를 호출하여 block 시키고, 마지막 쓰레드에서는 cond_broadcast()함수를 호출하여 모든 쓰레드들을 wakeup 하여 단일 스탭의 기능을 구현했다.

1.2 교차로 동기화 문제

1.1.1 배경

차량 스레드들이 교차로에 동시에 진입할 때 발생하는 동기화 문제로는 Deadlock 문제가 있다. 본 프로젝트에서는 해당 문제를 해결하기 위해서 하나의 차량이 교차로에 진입할 때, 해당 차량의 교차로 상의 경로를 모두 lock을 시킨 뒤, 차량이 교차로를 탈출 할 때, 교차로 상의 모든 lock을 해제 시켜서 Deadlock을 방지하였다.

교차로 동기화 문제를 해결하기 위해 사용한 변수 및 구조체들의 구조는 다음과 같다.

[표 1 - 변수 및 구조체 목록]

Bool intersection_locked[8]	교차로 8칸에 대해서 해당 칸의 잠김 유무를 확인하기 위한 배열
const struct position intersection_path[4][4][10]	Start,Dest 경로에 따른 교차로에서의 경로를 저장한 배열 Ex)start = A, dest = B일 때,instruction_path[start][dest][0] = {4,2}
Semaphore *y	교차로에 접근 가능 여부를 판단하는 코드 영역을 보호하기 위한 세마포어.
Const int num_intersection_hold[start][dest]	교차로에 진입할 때, 몇 칸의 교차로를 잠궈야 하는지 저장하고 있는 변수
int is_position_intersection(struct position pos)	주어진 좌표가 intersection에 위치하는 좌표인지 확인하는 함수
bool can_enter_intersection(int start, int dest)	Start, dest 로 가는 쓰레드가 현재 교차로에 진입을 해도 되는지 판단하는 함수

1.1.2 프로그램 흐름

교차로(intersection)를 추가하면 크게 3가지 case를 구분하여 try_move 함수를 나누어서 구현한다. 첫 번째 경우는 교차로 내부에서의 이동이고, 두 번째 경우는 교차로 밖에 있지만 다음 위치가 교차로인 경우, 세 번째 경우는 교차로 밖에 있으면서 다음 경로 또한 교차로 밖인 경우이다. 3가지 case에 따른 이동을 나눔으로써 기존 프로그램(교차로 밖에서의 이동)에서 try_move 함수를 이용하여 교차로를 추가하는 로직을 독립적으로 추가하여 구현할 수 있다.

```
if(is_position_intersection(pos_cur)){
        // 1.현재 위치가 intersection인 경우
}else{
        if(is_position_intersection(pos_next){
        //2 다음 위치가 intersection 인 경우
        }else{
        //3. 다음 위치가 intersection이 아닌 경우
        }
}
```

1) 현재 위치가 교차로 내부인 경우

해당 쓰레드가 자신이 교차로 상에서 지나갈 경로를 교차로를 진입할 때 모두 lock을 한 상태로 들어왔기 때문에 고려 사항 없이 한 칸 전진한다. 전진 시 다음 경로가 교차로를 벗어난다면, 자신이 lock 했던 교차로 내에서의 경로를 모두 release 한다.

```
//1. 교차로 내에서의 이동 상세 코드
vi->position = pos_next; // 한칸 이동
if(!is_position_intersection(vi->position)){
        // 다음 위치가 교차로를 벗어났을 때 처리
        lock_acquire(&vi-
>map_locks[pos_next.row][pos_next.col]);
        sema_down(y);
        int num = num_intersection_hold[start][dest];
        int i;
        // lock 된 부분을 모두 release 하는 코드
        for(i=0; i<num; i++){
                struct position pos =
intersection_path[start][dest][i];
                int pos_idx = get_intersection_idx(pos);
                intersection_locked[pos_idx] = false;
                lock_release(&vi-
>map_locks[pos.row][pos.col]);
        }
        sema_up(y);
}
return 1;
```

2) 다음 위치가 intersection 인 경우

현재 쓰레드가 교차로를 건널 때 지나야 하는 경로에 대해서 다른 쓰레드가 지나고 있는지 확인을 한다. 만약 다른 쓰레드가 현재 쓰레드가 지나야 하는 경로를 점유하고 있다면, 대기를 하고 해당 경로가 비어 있다면 모두 lock을 한 뒤, 교차로에 진입한다.

```
bool can_enter_intersection(int start, int dest){
// 교차로 진입 가능여부를 판단하는 코드
        sema_down(y);
        int num = num_intersection_hold[start][dest];
        int i=0;
        bool result = true;
        for(i=0; i<num; i++){
                struct position pos =
intersection_path[start][dest][i];
                int pos_idx = get_intersection_idx(pos);
                if(intersection_locked[pos_idx] == 1){
                        result = false;
                        break;
                }
        }
        return result;
}
```

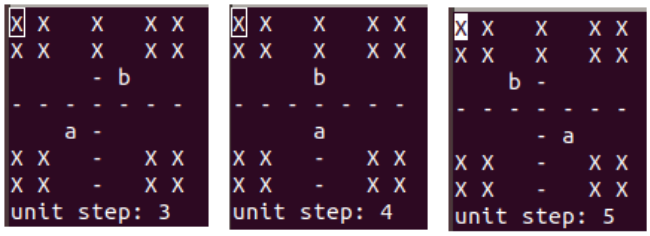
```
if(can_enter_intersection(start,dest)){
        int num = num_intersection_hold[start][dest];
        int i;
        // 해당 쓰레드가 교차로를 건널때 지나야 하는 경로
lock
        for(i=0; i<num; i++){
                struct position pos =
intersection_path[start][dest][i];
                int pos_idx = get_intersection_idx(pos);
                intersection_locked[pos_idx] = true;
                lock_acquire(&vi-
>map_locks[pos.row][pos.col]);

        }
        sema_up(y);
        vi->position = pos_next;
        lock_release(&vi-
>map_locks[pos_cur.row][pos_cur.col]);
        return 1;
}else{
        sema_up(y);
        return 2;
}
```

3) 다음 위치가 교차로가 아닌 경우

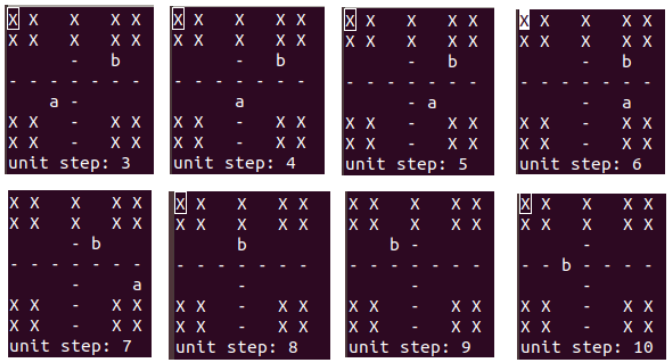
현재 위치가 교차로 내부가 아니고, 다음 위치도 교차로 내부가 아닌 경우이다. 이 경우는 기존에 기본으로 생성되었던 부분의 코드를 그대로 이용하였다.

3. 실험

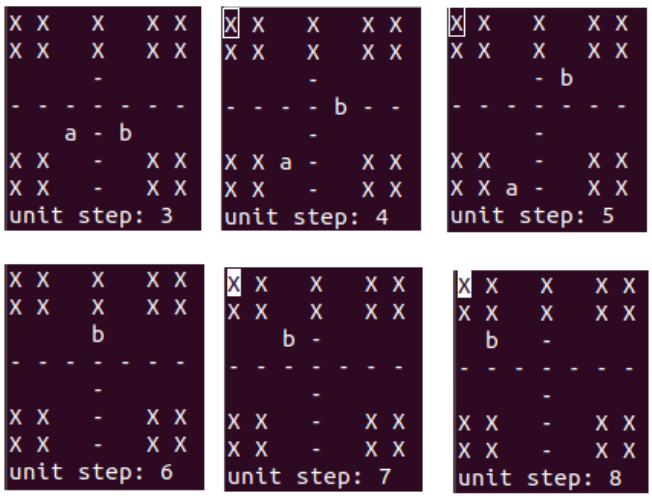
1) aAC:bCA



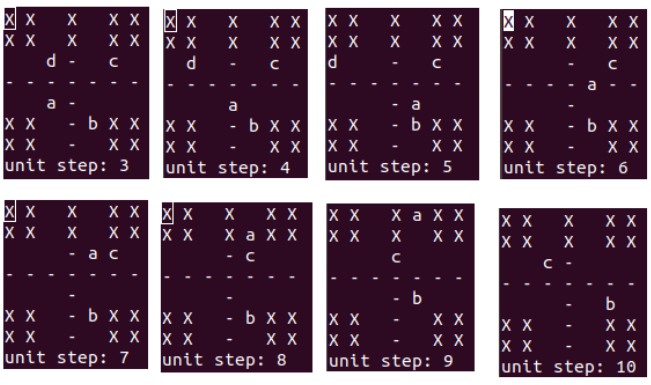
2) aAC:bCB



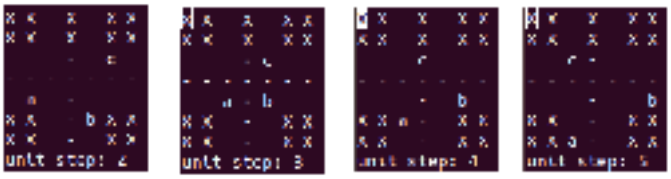
3) aAB:bBA



4) cCA:bBC:aAD:dDA



5) aAB:bBC:cCA



4. 결론
본 프로젝트를 하면서 이론으로 배운 병행성의 특징을 실제로 구현을 하면서 그 어려움을 이해할 수 있었다. 특히 여러 쓰레드들 간의 동기화의 필요성과, 세마포어 및 lock의 필요성을 확실히 이해할 수 있었다.

Unit Step을 구현하면서, Semaphore의 기능을 확실히 이해할 수 있었다. 여러 Thread에서 참조하는 remain_count 변수에 semaphore를 통해서 동시에 여러 Thread가 접근하지 못하고, 하나의 쓰레드씩 접근하도록 구현하여 쓰레드간의 데이터 동기화를 이해하였다. Unit Step을 구현하는 과정 중 새로 이해한 내용은 Semaphore는 단순히 semaphore 변수의 수치만을 가지고 접근 가능여부를 판단하기 때문에, 한계가 있음을 알았다.

예를 들어 7개의 쓰레드를 unit step이 적용되도록 구현하기 위해서 이를 세마포어로 구현한다면, 하나의 쓰레드가 반복문을 돌면서 세마포어 자원 7개를 모두 사용하는 경우가 발생하였다. 이를 해결하기 위해서 프로젝트에서 condition variable을 공부하고, 상황에 맞게 수정하면서 모니터의 개념에 대해서도 이해할 수 있게 되었다.

Intersection 문제를 구현하면서, 예측되는 경로를 미리 하나의 thread가 점유하도록 하면서 deadlock을 방지하고 있다. 이는 deadlock prevention 방식을 이용하여 deadlock을 처리한 방식이고, prevention 중에서 hold & wait 을 미리 차단한 방식이다. 즉 자신의 자원을 소유한 상태로 추가자원을 요청하는게 아니라, 자신이 미리 사용할 자원을 한번에 요청을 하고난 뒤, 자신의 작업이 끝났을 때, 한번에 모든 자원을 release하여 hold와 wait을 분리시킨 방식이다. 그리고 배운대로 이 방식이, 몇몇 케이스에서는 확실히 비효율적이라는 것도 볼 수 있었다.