운영체제 프로젝트 1 보고서

20225679 서규민

스레드 관리:
자동화 물류 센터 시뮬레이터 개발

Operating System Project#1

목차

b. 'S' 위치에 하나의 로봇만 접근하도록 제어

c. 로봇 상태에 따른 동적 목표 지점 변경 (적재 후 하역)

4. 결과

a. Test Case 1: 5 2A:4C:2B:2C:3A

b. Test Case 2: 4 3B:3B:3B:3B

c. Test Case 3: 3 1B:2B:3B

개발 과정

1. Pintos 개발 환경 설정

a. Pintos 는 x86 아키텍처를 기반으로 설계되어 있으므로 Ubuntu 환경에서 Pintos 를 빌드하고 실행할 수 있는 개발 환경을 구성하였습니다. 개발 환경은 Windows 기반 시스템에 VMware Workstation Player 를 활용하여 Ubuntu 18.04 LTS 를 설치하고, 그 위에서 Pintos 프로젝트를 수행하는 방식으로 구성하였습니다.

2. Pintos 이해

a. 부트로더에서 커널 진입점까지

i. Pintos 의 커널이 실행되기 위해서는 먼저 메모리로 로드되어야 하며, 이 역할을부트로더가 수행합니다. 부트 로더는 컴퓨터가 부팅될 때 가장 먼저 실행되는 프로그램으로, loader.S 파일에 구현되어 있으며, loader.S 는 커널 이미지를 메모리에 적재한 후, 커널의 엔트리 포인트로 점프시키 진행합니다. 하지만 이 엔트리 포인트에서 곧바로 init.c 에 정의된 main() 함수로 연결되지 않고, 그 이유는, main() 함수는 gcc 에 의 컴파일되면 그 과정에서 함수의 실제 진입 주소를 명확히 알기 어려우며, 디어셈블링 및 주소 계산이 필요하기 때문입니다. 그래서 이를 보완하기 위해, 중간 연결 지점으로 어셈블리 코드인 start.S 가 사용되고 start.s 에서 main()함수를 호출하며, main 함수는 실제 커널 초기화의 수행을 맡게 됩니다.

ii. main() 함수에서는 커널의 핵심 컴포넌트를 초기화하는 함수들이 순차적으로 thread_init() : 스레드 시스템 초기화, console_init() : 콘솔 입출력

설정, palloc_init() : 페이지 프레임 할당자 초기화, malloc_init() : 동적 메모리 할당자 초기화, paging_init() : 가상 메모리 페이징 초기화, intr_init() : 인터럽트 설정, timer_init() : 하드웨어 타이머 초기화, syscall_init() : 시스템 콜 초기화 등 많은 기능들이 초기화됩니다.

iii. 그 후, main() 함수 내부에서는 read_command_line() 함수를 통해 사용자 입력을 분석하여 명령줄 인자를 argv 에 저장하고, 이 정보를 바탕으로 run_actions() 함수가 호출되어 매핑된 함수로 진입하여 본격적인 실행이 시작됩니다. 이번 프로젝트에서는 run_automated_warehouse 라는 함수로 진입하며 시작합니다.

3. 구현 과정

a. 입력 인자 파싱 및 초기화

b. Robot 생성 & 스레드 생성

c. Block() & unblock() 함수 구현

d. Message Box 구현

e. determine_next_move() 및 bfs() 구현

f. 스레드 실행 함수 구현

구현 방법

1. 입력 인자 파싱 및 초기화

```c
// automated_warehouse.c
// run_automated_warehouse
n = atoi(argv[1]); // 로봇 수 파싱
    char *config_string = NULL;
    if (argv[2] != NULL) {
        config_string = malloc(strlen(argv[2]) + 1);
        strlcpy(config_string, argv[2], strlen(argv[2]) + 1);
    }

    // 파싱 (예: "2A:1B:3C" → ["2A", "1B", "3C"])
    char *robot_configs[n + 1];
    char *saveptr;
    char *config_token = config_string ? strtok_r(config_string, ":", &saveptr) : NULL;
    int config_idx = 0;
    while (config_token != NULL && config_idx < n) {
        robot_configs[config_idx++] = config_token;
        config_token = strtok_r(NULL, ":", &saveptr);
    }

    initialize_obstacle_map(); // 이동에 참고할 maps 초기화
    message_boxes(n+1); // 모든 스레드가 참고할 message_box 초기화
```

첫 번째 인자인 로봇 수를 문자열에서 정수로 변환하여 n 에 저장합니다. 두 번째 인자로는 각 로봇의 작업 스펙이 문자열 형태로 입력되며, 이 값을 안전하게 다루기 위해 동적 메모리 할당을 통해 복사하고 strlcpy 를 사용해 내용을 저장합니다. 이후 strtok_r 함수를 활용해 : 구분자를 기준으로 스펙 문자열을 분할하고, "2A", "1B", "3C"와 같은 형태의 작업 단위를 robot_configs[] 배열에 저장합니다. 이를 통해 각 로봇이 어떤 작업을 수행할지 사전에 정의할 수 있습니다. 또한, 이동에 참고할 maps 과 모든 스레드가 참고할 message_box 를 초기화하는 과정을 진행했습니다.

```c
// automated_warehouse.c
// initialize_obstacle_map 함수
void initialize_obstacle_map() {
    for (int i = 0; i < ROW; i++){
        for (int j = 0; j < COL; j++){
            if (map_draw_default[i][j] == 'X' || (map_draw_default[i][j] >= '0' && map_draw_default[i][j] <= '9')) {
                maps[i][j] = 1;
            } else {
                maps[i][j] = 0;
            }
        }
    }
};
```

Initialize_obstacle_map 함수에서는 map_draw_default(맵들)를 참고하여 'X'(벽)와 숫자(적재장소)가 있는 곳에는 1, 나머지 위치는 0 인 maps 를 만드는 함수입니다. 이를 통해 각 로봇들이 벽에 충돌하는 상황이나 다른 적재장소에 가지 않도록 제어할 수 있게 합니다.

```
// aw_message.c
// message_box 초기화 함수
void message_boxes(int n) {
    boxes_from_central_control_node = malloc(sizeof(struct message_box) * (n+1));
    boxes_from_robots = malloc(sizeof(struct message_box) * (n+1));

    for (int i = 0; i < n; i++) {
        boxes_from_central_control_node[i].msg.row = ROW_W;
        boxes_from_central_control_node[i].msg.col = COL_W;
        boxes_from_central_control_node[i].dirtyBit = 1;

        boxes_from_robots[i].msg.row = ROW_W;
        boxes_from_robots[i].msg.col = COL_W;
        boxes_from_robots[i].dirtyBit = 0;
    }
}
```

Message_boxes 함수는 로봇들과 중앙 제어 노드 간의 메시지 송수신을 위한 박스들을 초기화 하는 역할을 합니다. 각각 중앙 제어 노드 → 로봇, 로봇 → 중앙 제어 노드 방향의 메시지를 저장하고, 중앙 제어 노드 → 로봇 방향은 초기값을 1 로 설정하여 메시지가 준비되어 있음을 표시, 반대로 로봇 → 제어 노드 방향은 0 으로 설정하여 아직 메시지가 없음을 표시합니다.

2. Robot 생성 & 스레드 생성

```c
// automated_warehouse.c
// run_automated_warehouse
// 로봇 생성
    step = 0;
    robots = malloc(sizeof(struct robot) * (n+1));
    for (int i = 0; i < n; i++) {
        char name[10];
        snprintf(name, sizeof(name), "R%d", i+1);
        NumCharPair robot_config = split_string(robot_configs[i]); // Ex) "2A" => {2, 'A'}

        // Ex) 2와 'A'에 해당하는 좌표 각각 구하기
        Position loading_pos = findPosition(robot_config.num + '0'); // '0' 넣은 이유 : 숫자를 문자로 바꾸기 위해
        Position destination_pos = findPosition(robot_config.letter);

        char* robot_name =  create_string_copy(name);
        setRobot(&robots[i], robot_name, ROW_W, COL_W, 1, 0, loading_pos.row, loading_pos.col,
destination_pos.row, destination_pos.col, 0);
    }
```

전체 진행 단계를 나타내는 step 변수를 0 으로 초기화하고, n+1 크기의 로봇 배열을 동적 할당하여 로봇 정보를 저장할 준비를 합니다. 그 후 반복문을 통해 각 로봇에 대해 고유한 이름을 "R1", "R2"와 같은 형태로 생성하며, 주어진 작업 스펙 문자열(예: "2A")을 숫자와 문자로 분리하는 split_string 함수를 통해 적재 위치와 하역 위치 정보를 얻습니다. 그리고 숫자에 '0'을 더해 문자로 변환한 후 findPosition 을 통해 실제 적재 좌표를 계산하고, 문자 역시 findPosition 을 통해 하역 좌표로 변환합니다. 이렇게 파악된 적재 위치와 하역 위치는 해당 로봇이 수행할 경로를 결정하는 핵심 정보로 사용됩니다. 이후 생성한 이름을 힙 메모리에 안전하게 복사한 뒤, setRobot 함수를 통해 로봇의 이름, 초기 위치(ROW_W, COL_W), 적재 및 하역 위치 좌표, 상태 정보 등을 설정하여 로봇 구조체를 초기화하며, 이를 통해 모든 로봇이 시뮬레이션에서 사용할 준비를 마치게 됩니다.

```c
// automated_warehouse.c
// run_automated_warehouse
// 스레드 생성
tid_t* threads = malloc(sizeof(tid_t) * (n+1));
int* idx = malloc(sizeof(int) * (n+1));
for (int i = 0; i < n; i++) {
    idx[i] = i;
    threads[i] = thread_create(robots[i].name, 0, &robots_thread, &idx[i]);
}

// 중앙 제어 노드 스레드 생성
thread_create("CNT", 0, &cnt_thread, NULL);
```

각 로봇과 중앙 제어 노드(Central Node)에 대한 스레드를 생성하는 과정입니다. 먼저 n+1 크기의 스레드 ID 배열과 인덱스 배열을 동적으로 할당한 뒤, 각 로봇에 대해 인덱스를 전달하며 robots_thread 함수를 실행하는 스레드를 생성합니다. 이후 중앙 제어 노드 스레드 "CNT"를 별도로 생성하여 전체 제어 흐름을 관리할 수 있도록 합니다.

3. Block() & unblock() 함수 구현

```c
// aw_thread.c
struct list blocked_threads;

int check_blocked_thread_full(int n) {
    int size = list_size(&blocked_threads);
    return size == n;
}

void aw_thread_init(void) {
    list_init(&blocked_threads);
}

void block_thread(void){
    enum intr_level old_level;
    old_level = intr_disable ();

    list_push_back(&blocked_threads, &thread_current()->elem);
    thread_block ();

    intr_set_level (old_level);
}

void unblock_threads(void){
    enum intr_level old_level;
    old_level = intr_disable ();

    /* 리스트가 빌 때까지 순회하며 언블록 */
    while (!list_empty(&blocked_threads)){
        struct thread *t = list_entry(list_pop_front(&blocked_threads), struct thread, elem);
        thread_unblock(t);
    }

    intr_set_level(old_level);
}
```

aw_thread_init 함수는 블로킹된 스레드를 저장할 리스트 blocked_threads 를 초기화합니다. 이후 block_thread 는 로봇 스레드에서 호출되며, 현재 스레드를 리스트에 추가하고 실행을 중단시키고, 반대로 unblock_threads 는 중앙 제어 스레드에서 호출되어, 리스트에 있는 모든 스레드를 꺼내어 다시 실행 가능하게 만듭니다. check_blocked_thread_full 함수는 블로킹된 스레드의 수가 특정 개수와 일치하는지 확인하여 모든 로봇의 동기화 여부를 판단하고, 이를 통해 로봇들이 동시에 대기하며, 중앙 제어 스레드가 이를 제어할 수 있도록 합니다.

4. Message Box 구현

```c
// aw_message.c
void message_from_robots(int idx, struct message_box* msg_box, int robot_row, int robot_col) {
    msg_box->msg.row = robot_row;
    msg_box->msg.col = robot_col;
    msg_box->dirtyBit = 1;
}

void message_from_center(int idx, struct message_box* msg_box, int next_row, int next_col) {
    msg_box->msg.row = next_row;
    msg_box->msg.col = next_col;
    msg_box->dirtyBit = 1;
}
```

message_from_robots 함수는 로봇이 자신의 현재 위치(robot_row, robot_col)를 메시지 박스(msg_box)에 기록하고, dirtyBit 을 1 로 설정하여 새 메시지가 도착했음을 표시합니다. message_from_center 함수는 중앙 제어 노드가 로봇에게 보낼 명령 위치(next_row, next_col)를 메시지 박스에 기록하고, 마찬가지로 dirtyBit 을 1 로 설정해 메시지가 갱신되었음을 알립니다.그래서 이 방식으로 메시지 박스를 공유 메모리처럼 사용하여 양방향 통신을 구현했습니다.

5. determine_next_move() 및 bfs() 구현

```c
// automated_warehouse.c
int is_within_bounds(int row, int col) {
    return row >= 0 && row < ROW && col >= 0 && col < COL;
}

int determine_next_move(int robot_idx, int cur_row, int cur_col, int target_row, int taget_col, int reserved[ROW][COL]) {
    int min_distance = 1000;
    int next_pos = 4;

    // 일반 경로 탐색
    for (int i = 0; i < 4; i++) {
        int nr = cur_row + dr[i];
        int nc = cur_col + dc[i];

        // 만약 다음 지점이 목표 지점이라면, 그 방향으로 이동
        if (nr == target_row && nc == taget_col) {
            return i;
        }

        if (is_within_bounds(nr, nc) && maps[nr][nc] == 0 && !reserved[nr][nc]) {
            // 최단 거리 계산
            int distance = bfs(robot_idx, nr, nc, target_row, taget_col);
            if (distance < min_distance) {
                min_distance = distance;
                next_pos = i;
            }

        }
    }

    return next_pos;
}
```

is_within_bounds 는 주어진 좌표가 유효한 맵 범위 내에 있는지를 검사하고, determine_next_move 함수는 현재 로봇의 위치(cur_row, cur_col)에서 목표 위치(target_row, taget_col)로 이동하기 위한 최적의 방향을 계산합니다. 우선 4 방향(상하좌우)을 확인하며, 인접한 칸이 곧바로 목표 지점이면 해당 방향을 즉시 반환합니다. 그렇지 않은 경우에는 각 인접 칸이 이동 가능(maps 가 0 이고 예약되지 않음)한지 확인하고, 그 위치에서 목표까지의 거리를 bfs 알고리즘을 활용하여 bfs()함수로 계산하고, 가장 짧은 거리로 가는 방향을 선택합니다. 최종적으로 선택된 방향 인덱스(0~3)를 반환하도록 합니다.

```c
// automated_warehouse.c
int bfs(int robot_idx, int cur_row, int cur_col, int target_row, int target_col) {
        int visited[ROW][COL] = {0}; // 방문 배열 초기화
        int distance[ROW][COL] = {0}; // 거리 저장 배열

        // 큐 초기화
        Point queue[ROW * COL];
        int front = 0, rear = 0;

        // 시작점을 큐에 삽입 후, 방문 표시
        queue[rear++] = (Point){cur_row, cur_col};
        visited[cur_row][cur_col] = 1;

        while (front < rear) {
                // 큐의 맨 앞에 있는 요소 꺼내기
                Point cur = queue[front++];

                // 만약 목표 지점에 도달했다면, 거리 반환
                if (cur.row == target_row && cur.col == target_col) {
                        return distance[target_row][target_col];
                }

                for (int i = 0; i < 4; i++) {
                        int nr = cur.row + dr[i];
                        int nc = cur.col + dc[i];

                        // 만약 다음 지점이 목표 지점이라면, 거리 반환 (이게 없으면, 진입 불가)
                        if (nr == target_row && nc == target_col) {
                            return distance[cur.row][cur.col] + 1;
                        }
```

```c
                        if (is_within_bounds(nr, nc) && maps[nr][nc] != 1 && !visited[nr][nc]) {
                            visited[nr][nc] = 1;
                            distance[nr][nc] = distance[cur.row][cur.col] + 1;
                            queue[rear++] = (Point){nr, nc};
                        }
                }
        }
        return 1000;
}
```

이 함수는 로봇이 현재 위치에서 목표 위치까지 도달하는 최단 거리를 계산하기 위한 BFS 알고리즘을 구현한 함수입니다. bfs 함수는 로봇 인덱스(robot_idx)와 현재 위치(cur_row, cur_col), 목표 위치(target_row, target_col)를 받아, 2 차원 맵 상에서 가장 짧은 경로의 거리 값을 정수로 반환합니다. 방문 여부를 나타내는 배열 visited 와 거리 누적을 위한 distance 배열을 초기화한 후, 큐를 사용해 BFS 탐색을 수행합니다. 시작 위치를 큐에 넣고 방문 표시를 하며 시작하고, 이후 큐에서 하나씩 꺼내면서 상하좌우로 이동 가능한 인접 칸을

확인하며, 목표 위치에 도달하면 누적된 거리를 반환합니다. 인접 칸이 유효한 위치이고 장애물이 없으며 아직 방문하지 않았다면, 거리를 1 증가시키고 큐에 넣습니다. 도달하지 못하면 큰 수(예: 1000)를 반환하여 이동 불가능함을 나타내고, 이렇게 여러 방향 중 목표 지점과 가장 가까워지는 방향으로 이동할 수입니다.

6. 스레드 실행 함수 구현

```c
// automated_warehouse.c
/* 중앙 제어 노드 스레드 */
void cnt_thread() {
    while (1) {
        int is_full = check_blocked_thread_full(n);  // 모든 로봇이 block 되었는지 확인
        int final_cnt = 0;
        // 모든 로봇이 도착했는지 확인
        for (int i = 0; i < n; i++) {
            if (robots[i].is_finish == 1) {
                final_cnt++;
            }
        }
        // 모든 로봇이 도착했다면 종료
        if (final_cnt == n) {
            printf("\n--------- Mission Complete ---------\n");
            break;
        }
        if (is_full) {
            print_map(robots, n);  // 현재 상태 시각화
            // 이동 충돌 방지용 예약 배열
            int reserved[ROW][COL] = {0};

            // 중요: 모든 로봇의 현재 위치를 먼저 reserved에 표시
            for (int i = 0; i < n; i++) {
                if (!robots[i].is_finish) {  // 완료되지 않은 로봇만 고려
                    int cur_row = boxes_from_robots[i].msg.row;
                    int cur_col = boxes_from_robots[i].msg.col;
                    reserved[cur_row][cur_col] = 1;  // 현재 위치 점유 표시
                }
            }

            // S 위치가 현재 점유되어 있다면 예약 배열에도 표시
            if (s_location_occupied) {
                reserved[ROW_S][COL_S] = 1;
            }
```

```
// S 위치가 현재 점유되어 있다면 예약 배열에도 표시
if (s_location_occupied) {
    reserved[ROW_S][COL_S] = 1;
}

for (int i = 0; i < n; i++) {
    if (boxes_from_robots[i].dirtyBit == 1) {
        int cur_row = boxes_from_robots[i].msg.row;
        int cur_col = boxes_from_robots[i].msg.col;
        if (robots[i].is_finish == 1) {
            message_from_center(i, &boxes_from_central_control_node[i], cur_row, cur_col);
            boxes_from_robots[i].dirtyBit = 0;
            continue;
        }

        // 목적지 설정 (적재 or 하역)
        int target_row, target_col;
        if (robots[i].current_payload == 1) {
            target_row = robots[i].f_row;  // 하역 장소
            target_col = robots[i].f_col;
        } else {
            target_row = robots[i].l_row;  // 적재 장소
            target_col = robots[i].l_col;
        }

        if (cur_row == robots[i].l_row && cur_col == robots[i].l_col && robots[i].current_payload == 0) {
            robots[i].current_payload = 1;
        }

        // If reached final point and is loaded, mark as finished
        if (cur_row == robots[i].f_row && cur_col == robots[i].f_col && robots[i].current_payload == 1) {
            robots[i].is_finish = 1;
        }
```

```
int direction = 4;  // 기본값: 제자리 유지
int next_row = cur_row;
int next_col = cur_col;

// 완료되지 않은 로봇만 이동 경로 계산
if (!robots[i].is_finish) {
    // 현재 위치는 비워질 예정이므로 reserved에서 제거
    reserved[cur_row][cur_col] = 0;

    direction = determine_next_move(i, cur_row, cur_col, target_row, target_col, reserved);

    if (direction < 4) {
        next_row = cur_row + dr[direction];
        next_col = cur_col + dc[direction];

        bool is_moving_to_S = (next_row == ROW_S && next_col == COL_S);
        bool is_leaving_S = (cur_row == ROW_S && cur_col == COL_S && !is_moving_to_S);
        bool is_moving_to_special = false;

        // A, B, C 위치를 제외한 나머지 위치에서 충돌 확인
        is_moving_to_special = (next_row == robots[i].f_row && next_col == robots[i].f_col);

        // 이동 충돌 방지 로직
        if (reserved[next_row][next_col] && !is_moving_to_special) {
            // 이미 다른 로봇이 예약한 칸이면 대기
            direction = 4;
            next_row = cur_row;
            next_col = cur_col;

            // 현재 위치 다시 점유 표시
            reserved[cur_row][cur_col] = 1;

            // S 위치에서 나가려고 했으나 대기하게 된 경우, S 위치는 계속 점유 상태 유지
            if (is_leaving_S) {
                s_location_occupied = 1;
                reserved[ROW_S][COL_S] = 1;
```

```
                        // S 위치에서 나가려고 했으나 대기하게 된 경우, S 위치는 계속 점유 상태 유지
                        if (is_leaving_S) {
                            s_location_occupied = 1;
                            reserved[ROW_S][COL_S] = 1;
                        }
                    } else {
                        // 다음 위치 예약
                        reserved[next_row][next_col] = 1;

                        // S 위치 특별 처리
                        if (is_moving_to_S) {
                            s_location_occupied = 1;
                        } else if (is_leaving_S) {
                            // S 위치에서 나가는 경우 성공적으로 이동
                            s_location_occupied = 0;
                            reserved[ROW_S][COL_S] = 0;
                        }
                    }
                } else {
                    // 제자리 유지의 경우 현재 위치 다시 점유 표시
                    reserved[cur_row][cur_col] = 1;
                }
            }

            // 명령 전달 (다음 좌표 및 방향 포함)
            message_from_center(i, &boxes_from_central_control_node[i], next_row, next_col);
            // 메시지 처리 완료
            boxes_from_robots[i].dirtyBit = 0;
        }
    }
    increase_step();     // 시각화용 step 증가
    unblock_threads();   // 모든 로봇 스레드 깨우기
  }
 }
}
```

cnt_thread() 함수는 중앙 제어 노드 스레드로서, 전체 로봇의 상태를 관리하고 명령을 전송하는 핵심 역할을 수행합니다. 이 스레드는 무한 루프 안에서 반복적으로 로봇들의 상태를 점검하며, 모든 로봇이 작업을 완료했는지 확인합니다. 만약 모든 로봇의 is_finish 플래그가 1 이라면, "Mission Complete" 메시지를 출력하고 루프를 종료합니다. 로봇이 모두 멈춰 대기(block) 상태인지 확인하는 함수 check_blocked_thread_full(n)의 반환값이 참일 경우, 현재 맵 상태를 시각화한 후 이동 충돌을 방지하기 위한 reserved 배열을 초기화합니다. 이 배열은 각 셀의 예약 여부를 나타내며, 우선 로봇들의 현재 위치를 표시하고, S 구역도 점유되어 있다면 예약 처리합니다. 그 후 각 로봇의 메시지 박스(boxes_from_robots)를 검사하여 dirtyBit 가 1 인 경우, 즉 새 위치 정보가 전달된 경우에만 처리합니다. 로봇이 목적지에 도달했는지 확인하고, 적재 장소에 도착하면 current_payload 를 1 로 설정하고,

하역 장소에 도달했을 때는 is_finish 를 1 로 설정합니다. 이동 방향은 기본적으로 제자리를 유지하는 값(4)으로 설정되며, 로봇이 완료되지 않았을 경우 determine_next_move() 함수를 통해 다음 위치를 결정합니다. 이때 충돌 방지를 위해 이동하려는 칸이 이미 reserved 된 경우 해당 로봇은 제자리에 정지하게 됩니다. 단, 목적지 셀(A, B, C 등)은 예외로 처리되며, S 지점에서 이동하는 경우 특별히 s_location_occupied 플래그도 관리합니다. 이후 결정된 다음 위치와 방향을 각 로봇에게 message_from_center() 함수를 통해 전달하고, 메시지 박스의 dirtyBit 를 0 으로 초기화합니다. 모든 로봇 명령 전송이 끝나면 시각화 단계 수를 증가시키고, unblock_threads()를 호출하여 대기 중인 로봇 스레드들을 깨웁니다. 이러한 과정을 통해 중앙 제어 노드는 각 로봇의 경로를 실시간으로 조정하며, 충돌 없이 물류 작업을 진행하도록 합니다.

```c
// automated_warehouse.c
/* 물류 로봇 스레드 */
void robots_thread(void* aux) {
    int idx = *((int *)aux);

    while (1) {
        if (boxes_from_central_control_node[idx].dirtyBit == 1) {
            int next_row = boxes_from_central_control_node[idx].msg.row;
            int next_col = boxes_from_central_control_node[idx].msg.col;

        robots[idx].row = next_row;
            robots[idx].col = next_col;

            // 센터에 메시지 보내기
            message_from_robots(idx, &boxes_from_robots[idx], robots[idx].row, robots[idx].col);
            boxes_from_central_control_node[idx].dirtyBit = 0;
        }

        block_thread();  // 스스로 블록 상태로
    }
}
```

robots_thread() 함수는 로봇의 인덱스를 전달받아, 무한 루프 내에서 중앙 제어 노드로부터 새로운 위치 명령이 왔는지 확인합니다. dirtyBit 가 1 이면 명령이 도착한 것이므로, 메시지에서 새로운 위치(row, col)를 받아 로봇의 현재 위치를 갱신하고, 이를 다시 중앙 제어 노드에 전달합니다. 메시지 처리가 끝나면 dirtyBit 를 0 으로 초기화합니다. 이후

block_thread()를 호출해 로봇 스레드는 스스로 대기 상태에 들어가며, 중앙 제어 노드가 unblock_threads()를 호출할 때까지 멈춰 있습니다.

트러블 슈팅 과정

a. 로봇 간 충돌 방지를 위한 이동 예약 배열 reserved[ROW][COL] 도입

i. 문제 상황:

초기에 모든 로봇이 동시에 경로를 탐색하고 이동하면서, 동일한 위치로 동시에 진입하려는 현상이 발생했고, 이로 인해 충돌이 발생하거나 무한 루프에 빠질 수 있는 상황이 생겼습니다.

ii. 해결 방법:

각 로봇의 위치 정보를 기반으로 한 reserved 배열을 도입해 이동을 예약하고, 경로 탐색 시 해당 위치가 예약되어 있으면 다른 경로를 선택하게 하였습니다. 특히, 로봇의 현재 위치를 reserved 에 미리 표시하고, determine_next_move() 함수에서는 reserved 를 고려하여 경로를 판단함으로써 충돌을 방지할 수 있었습니다.

b. 'S' 위치에 하나의 로봇만 접근하도록 제어

iii. 문제 상황:

여러 로봇이 동시에 S 지점에 접근하려고 할 때 충돌이 발생하거나 S 위치를 공유하는 문제가 생겼습니다.

iv. 해결 방법:

s_location_occupied 플래그를 도입해, 해당 위치에 로봇이 이미 있거나 접근 중일 경우에는 다른 로봇이 접근하지 못하도록 했습니다. 또한, S 위치에 진입하려는 경우에는 reserved[ROW_S][COL_S] = 1; 처리를 통해 다음 이동 예약에도 반영했습니다.

c. 로봇 상태에 따른 동적 목표 지점 변경 (적재 후 하역)

v. 문제 상황:

로봇이 적재 지점에 도달해도 상태가 변경되지 않아 계속 적재 지점으로 향하거나, 하역 지점에 도달해도 종료되지 않는 문제가 있었습니다.

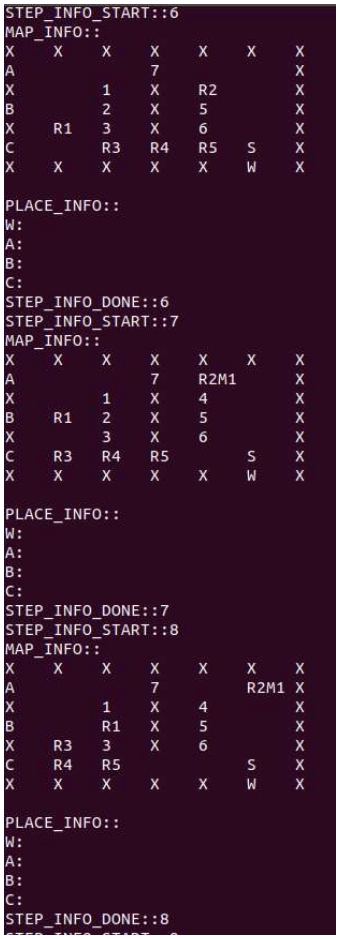
vi. 해결 방법:

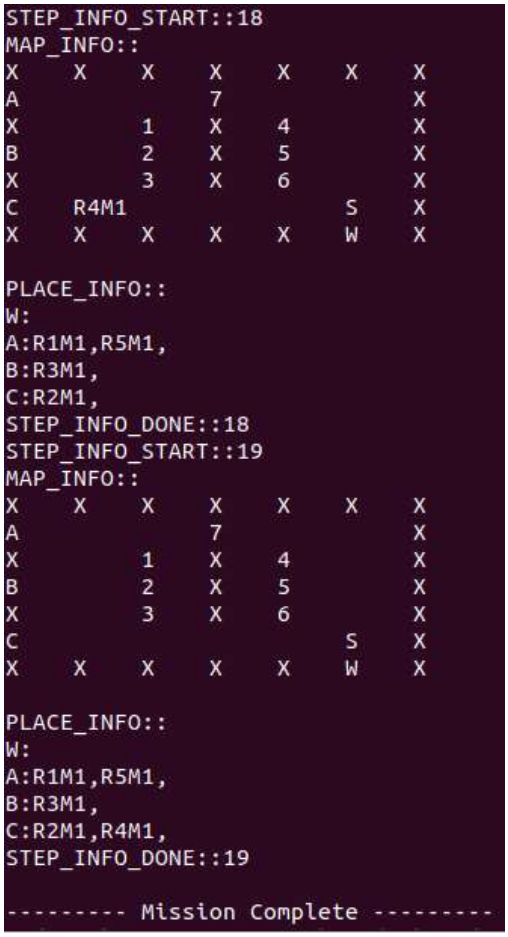
로봇의 현재 위치가 적재 지점이면 current_payload = 1 로 설정하고, 하역 지점에 도달하면 is_finish = 1 로 설정하여 상태 전이를 명확히 했습니다. 이를 통해 상태 기반의 목표 전환을 실현하고, 경로 탐색의 목적지도 동적으로 설정할 수 있게 됐습니다.

결과

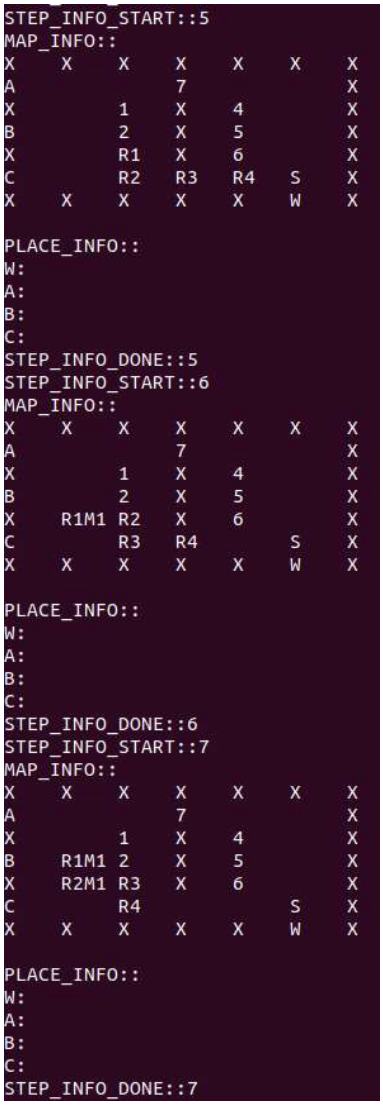
각 로봇들을 동시에 제어하면서 적재한 뒤, 하역 장소까지 완벽하게 도착합니다. 실행 과정과 결과 화면을 첨부합니다.

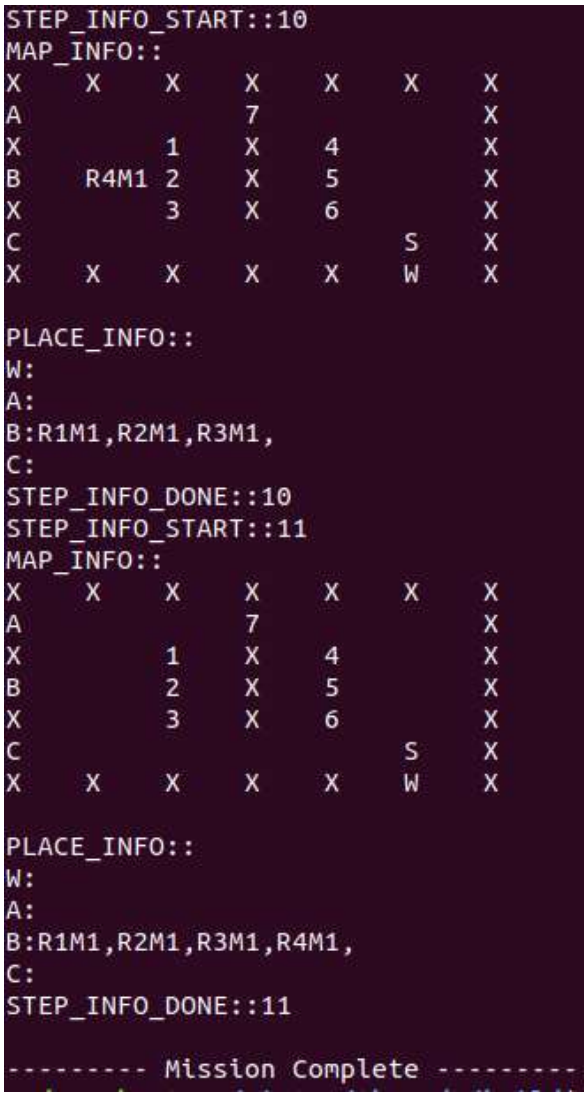
1. **Test Case 1: 5 2A:4C:2B:2C:3A**





2. Test Case 2: 4 3B:3B:3B:3B





3. Test Case 3: 3 1B:2B:3B

```
STEP_INFO_START::6
MAP_INFO::
X    X    X    X    X    X    X
A              7              X
X         1    X    4         X
B         2    X    5         X
X    R1   3    X    6         X
C    R2   R3                S    X
X    X    X    X    X    W    X

PLACE_INFO::
W:
A:
B:
C:
STEP_INFO_DONE::6
STEP_INFO_START::7
MAP_INFO::
X    X    X    X    X    X    X
A              7              X
X         1    X    4         X
B    R1   2    X    5         X
X    R2   R3   X    6         X
C                        S    X
X    X    X    X    X    W    X

PLACE_INFO::
W:
A:
B:
C:
STEP_INFO_DONE::7
STEP_INFO_START::8
MAP_INFO::
X    X    X    X    X    X    X
A              7              X
X    R1   1    X    4         X
B    R2   2    X    5         X
X    R3M1 3    X    6         X
C                        S    X
X    X    X    X    X    W    X

PLACE_INFO::
W:
A:
B:
C:
STEP_INFO_DONE::8
```

```
STEP_INFO_START::13
MAP_INFO::
X    X    X    X    X    X    X
A              7              X
X         1    X    4         X
B    R1M1 2    X    5         X
X         3    X    6         X
C                        S    X
X    X    X    X    X    W    X

PLACE_INFO::
W:
A:
B:R2M1,R3M1,
C:
STEP_INFO_DONE::13
STEP_INFO_START::14
MAP_INFO::
X    X    X    X    X    X    X
A              7              X
X         1    X    4         X
B         2    X    5         X
X         3    X    6         X
C                        S    X
X    X    X    X    X    W    X

PLACE_INFO::
W:
A:
B:R1M1,R2M1,R3M1,
C:
STEP_INFO_DONE::14

--------- Mission Complete ---------
```