**[ automated\_warehouse.c ]**

void test\_cnt(void){

bool \*received = malloc(sizeof(bool) \* num\_robots);

for (int i = 0; i < num\_robots; i++) {

received[i] = false;

}

while(1){

로봇 정보를 출력

print(robots, num\_robots);

100ms 동안 대기

thread\_sleep(100);

bool received\_all = false;

모든 로봇으로부터 메시지를 받을 때까지 대기

while (!received\_all) {

received\_all = true;

for (int i = 0; i < num\_robots; i++) {

if (received[i]) {

continue;

}

received\_all = false;

Message msg;

모든 로봇에게 메시지 수신

int res = receive(i, &msg, CENTRAL\_CONTROL);

if (res == 0) {

Command cmd = find\_path(&msg);

set\_command(&msg, cmd);

send(i, &msg, CENTRAL\_CONTROL);

received[i] = true;

}

}

}

for (int i = 0; i < num\_robots; i++) {

received[i] = false;

}

카운터를 증가

increase();

스레드를 재개

unblock\_threads();

}

}

void test\_thread(void\* aux){

int idx = \*((int \*)aux);

int test = 0;

int res = 0;

초기 메시지를 보내 로봇의 위치를 설정

Message msg;

msg.cmd = CMD\_NOP;

move\_robot(robots, idx, num\_robots, &msg);

send(idx, &msg, ROBOT);

다른 로봇으로부터 메시지를 받아 처리

for (int i = 0; i < idx \* 5; i++) {

if (i != idx) {

res = receive(idx, &msg, ROBOT);

if(res == 0){

msg.cmd = CMD\_NOP;

move\_robot(robots, idx, num\_robots, &msg);

send(idx, &msg, ROBOT);

}

}

}

무한 루프를 실행하며 메시지 송수신

while(1){

memset(&msg, 0, sizeof(Message));

printf("thread %d : %d\n", idx, test++);

res = receive(idx, &msg, ROBOT);

if(res == 0){

move\_robot(robots, idx, num\_robots, &msg);

send(idx, &msg, ROBOT);

}

block\_thread();

}

}

int parse\_args(char \*payload\_list, Payload \*payloads) {

로봇 정보를 파싱

char \*saveptr;

콜론을 기준으로 토크나이징

char \*token = strtok\_r(payload\_list, ":", &saveptr);

int count = 0;

while (token != NULL) {

char number = token[0];

char letter = token[1];

로봇 수 확인

if (count >= num\_robots) {

printf("Too many robots\n");

return -1;

}

로봇 정보 숫자 확인

if (number < '0' || number > '9') {

printf("Invalid number format: %s\n", token);

return -2;

}

로봇 정보 알파벳 대문자 확인

if (letter < 'A' || letter > 'Z') {

printf("Invalid letter format: %s\n", token);

return -3;

}

if (token[2]) {

printf("Invalid format: %s\n", token);

return -4;

}

배열에 저장

payloads[count].current = number;

payloads[count].required = letter;

count++;

token = strtok\_r(NULL, ":", &saveptr);

}

for (int i = 0; i < count; i++) {

printf("%c%c\n", payloads[i].current, payloads[i].required);

}

성공적으로 완료되면 0을 반환

return 0;

}

int initialize\_robots(Payload \*payloads) {

로봇 배열 동적 할당

robots = malloc(sizeof(Robot) \* num\_robots);

스레드 배열을 동적 할당

tid\_t\* threads = malloc(sizeof(tid\_t) \* num\_robots);

로봇 이름을 저장할 문자열 배열을 동적 할당

char \* rnames = malloc(sizeof(char) \* num\_robots \* 3);

로봇 인덱스를 저장할 배열을 동적 할당

int \* robot\_idxs = malloc(sizeof(int) \* num\_robots);

for (int i = 0; i < num\_robots; i++) {

robot\_idxs[i] = i;

snprintf(rnames, 3, "R%d", i + 1);

로봇 이름, 위치, 목표 설정

set\_robot(&robots[i], rnames, ROW\_W, COL\_W, payloads[i].required, payloads[i].current);

로봇 스레드를 생성

threads[i] = thread\_create(rnames, 0, &test\_thread, &robot\_idxs[i]);

if (threads[i] == TID\_ERROR) {

printf("Thread creation failed\n");

return -1;

}

rnames += 3;

}

성공적으로 완료되면 0을 반환

return 0;

}

int initialize\_node() {

중앙 제어 노드 스레드를 생성

tid\_t\* cnt\_thread = thread\_create("CNT", 0, &test\_cnt, NULL);

if (cnt\_thread == TID\_ERROR) {

printf("Thread creation failed\n");

return -1;

}

성공적으로 완료되면 0을 반환

return 0;

}

void run\_automated\_warehouse(char \*\*argv)

{

프로그램 초기화

init\_automated\_warehouse(argv);

로봇 수와 로봇 정보를 매개변수로부터 입력

num\_robots = atoi(argv[1]);

char \*payload\_list = argv[2];

Payload\* payloads = malloc(sizeof(Payload) \* num\_robots);

if (payloads == NULL) {

printf("Memory allocation failed\n");

return -1;

}

로봇 정보를 파싱하여 배열에 저장

int res = parse\_args(payload\_list, payloads);

if (res < 0) {

printf("Parsing failed\n");

return;

}

메시지 박스 초기화

res = initialize\_message\_boxes(num\_robots);

if (res < 0) {

printf("Message box initialization failed\n");

return;

}

중앙 제어 노드 초기화

res = initialize\_node();

if (res < 0) {

printf("Central control node initialization failed\n");

return;

}

로봇 초기화

res = initialize\_robots(payloads);

if (res < 0) {

printf("Robot initialization failed\n");

return;

}

}

**[ aw\_messeges.c ]**

int initialize\_message\_boxes(int num\_robots) {

로봇 수에 맞게 메시지 박스를 동적으로 할당

NUM\_ROBOTS = num\_robots;

boxes\_from\_node = malloc(sizeof(MesssageBox) \* num\_robots);

if (boxes\_from\_node == NULL) {

printf("Memory allocation failed\n");

return -1;

}

각 메시지 박스의dirtyBit를 초기화

for (int i = 0; i < num\_robots; i++) {

boxes\_from\_node[i].dirtyBit = 1;

}

노드 수에 맞게 메시지 박스를 동적으로 할당

boxes\_from\_robots = malloc(sizeof(MesssageBox) \* num\_robots);

if (boxes\_from\_robots == NULL) {

printf("Memory allocation failed\n");

return -1;

}

각 메시지 박스의dirtyBit를 초기화

for (int i = 0; i < num\_robots; i++) {

boxes\_from\_robots[i].dirtyBit = 0;

}

성공적으로 완료되면 0을 반환

return 0;

}

int send(int robotIdx, Message \* msg, MessageEndpoint endpoint) {

올바른 로봇 인덱스인지 확인

if (robotIdx < 0 || robotIdx >= NUM\_ROBOTS) {

printf("Invalid robot index: %d\n", robotIdx);

return -1;

}

로봇 인덱스와 메시지를 받아 해당 메시지 송신

if (endpoint == ROBOT) {

return send\_to\_node(robotIdx, msg);

} else if (endpoint == CENTRAL\_CONTROL) {

return send\_to\_robot(robotIdx, msg);

} else {

printf("Invalid message endpoint\n");

return -1;

}

}

int receive(int robotIdx, Message \* msg, MessageEndpoint endpoint) {

로봇 인덱스와 메시지를 받아 해당 메시지 수신

if (endpoint == ROBOT) {

return receive\_from\_node(robotIdx, msg);

} else if (endpoint == CENTRAL\_CONTROL) {

return receive\_from\_robot(robotIdx, msg);

} else {

printf("Invalid message endpoint\n");

return -1;

}

}

int send\_to\_node(int robotIdx, Message\* msg) {

메시지 박스의 dirtyBit를 확인

이미 다른 스레드가 메시지를 쓴 경우

if (boxes\_from\_robots[robotIdx].dirtyBit == 1) {

printf("Message box is already written by others\n");

return -1;

}

로봇에서 중앙 제어 노드로 메시지 송신

boxes\_from\_robots[robotIdx].msg.row = msg->row;

boxes\_from\_robots[robotIdx].msg.col = msg->col;

boxes\_from\_robots[robotIdx].msg.current = msg->current;

boxes\_from\_robots[robotIdx].msg.required = msg->required;

boxes\_from\_robots[robotIdx].dirtyBit = 1;

INFO("msg", "send robot2cnt row: %d, col: %d, current: %c, required: %c", boxes\_from\_robots[robotIdx].msg.row, boxes\_from\_robots[robotIdx].msg.row, boxes\_from\_robots[robotIdx].msg.current, boxes\_from\_robots[robotIdx].msg.required);

return 0;

}

int send\_to\_robot(int robotIdx, Message\* msg) {

메시지 박스의 dirtyBit를 확인

이미 다른 스레드가 메시지를 쓴 경우

if (boxes\_from\_node[robotIdx].dirtyBit == 1) {

printf("Message box is already written by others\n");

return -1;

}

노드에서 로봇으로 메시지 송신

boxes\_from\_node[robotIdx].msg.cmd = msg->cmd;

boxes\_from\_node[robotIdx].dirtyBit = 1;

INFO("msg", "send cnt2robot cmd: %d", boxes\_from\_node[robotIdx].msg.cmd);

return 0;

}

int receive\_from\_node(int robotIdx, Message\* msg) {

메시지 박스의 dirtyBit를 확인

메시지를 받기 전에 다른 스레드가 메시지를 쓴 경우

if (boxes\_from\_node[robotIdx].dirtyBit == 0) {

printf("Message box is not written yet\n");

return -1;

}

중앙 제어 노드에서 로봇으로부터 메시지 수신

msg->cmd = boxes\_from\_node[robotIdx].msg.cmd;

boxes\_from\_node[robotIdx].dirtyBit = 0;

INFO("msg", "recv cnt2robot cmd: %d", msg->cmd);

return 0;

}

int receive\_from\_robot(int robotIdx, Message\* msg) {

메시지 박스의 dirtyBit를 확인

메시지를 받기 전에 다른 스레드가 메시지를 쓴 경우

if (boxes\_from\_robots[robotIdx].dirtyBit == 0) {

printf("Message box is not written yet\n");

return -1;

}

로봇에서 중앙 제어 노드로부터 메시지 수신

msg->row = boxes\_from\_robots[robotIdx].msg.row;

msg->col = boxes\_from\_robots[robotIdx].msg.col;

msg->current = boxes\_from\_robots[robotIdx].msg.current;

msg->required = boxes\_from\_robots[robotIdx].msg.required;

boxes\_from\_robots[robotIdx].dirtyBit = 0;

INFO("msg", "recv robot2cnt row: %d, col: %d, current: %c, required: %c", msg->row, msg->col, msg->current, msg->required);

return 0;

}

void set\_command(Message\* msg, Command cmd) {

메시지의 cmd필드를 주어진 명령으로 설정

msg->cmd = cmd;

}

**[ robot.c ]**

bool robot\_is\_unloaded(Robot\* robot) {

로봇의 위치가 알파벳 대문자인지 확인

bool is\_pos\_alpha = map[robot->row][robot->col] >= 'A' && map[robot->row][robot->col] <= 'Z';

로봇의 현재 상태가 해당 위치의 알파벳과 일치하는지 확인

bool is\_pos\_current = robot->current == map[robot->row][robot->col];

return is\_pos\_alpha && is\_pos\_current;

}

void update\_msg(Message\* msg, Robot\* robot) {

로봇의 정보를 메시지에 복사

msg->row = robot->row;

msg->col = robot->col;

msg->current = robot->current;

msg->required = robot->required;

INFO("robot", "update msg row: %d, col: %d, current: %c, required: %c", msg->row, msg->col, msg->current, msg->required);

}

bool isOutOfBound(int row, int col) {

주어진 행과 열이 범위를 벗어났는지 확인

return row < ROW\_MIN || row > ROW\_MAX || col < COL\_MIN || col > COL\_MAX;

}

bool isWall(int row, int col) {

주어진 위치가 벽인지 확인

return map[row][col] == 'X';

}

bool isAnotherCurrentPayload(int row, int col, int current) {

주어진 위치에 다른 로봇의 현재 상태가 있는지 확인

return map[row][col] >= '0' && map[row][col] <= '9' &&

map[row][col] != current;

}

bool isAnotherRequiredPayload(int row, int col, int current) {

주어진 위치에 다른 로봇의 요구되는 목표가 있는지 확인

return map[row][col] >= 'A' && map[row][col] <= 'Z' &&

map[row][col] != 'W' && map[row][col] != 'S' &&

map[row][col] != current;

}

bool isAnotherRobot(Robot\* robots, int number\_of\_robots, int idx, int row, int col) {

주어진 위치에 다른 로봇이 있는지 확인

for (int robotIdx = 0; robotIdx < number\_of\_robots; robotIdx++){

if (robotIdx == idx) {

continue;

}

Robot\* robot = &robots[robotIdx];

if (robot->col == col && robot->row == row){

return true;

}

}

return false;

}

Command find\_movable\_direction(Robot\* robots, int idx, int number\_of\_robots) {

Robot\* robot = &robots[idx];

int row = robot->row;

int col = robot->col;

char current = robot->current;

char required = robot->required;

int dr[] = {-1, 1, 0, 0};

int dc[] = {0, 0, -1, 1};

로봇의 위치에서 상하좌우로 이동 가능한지 확인

for (int i = 0; i < 4; i++) {

int new\_row = row + dr[i];

int new\_col = col + dc[i];

if (isOutOfBound(new\_row, new\_col) || isWall(new\_row, new\_col) || isAnotherRobot(robots, number\_of\_robots, idx, new\_row, new\_col)) {

continue;

}

if (isAnotherCurrentPayload(new\_row, new\_col, current) || isAnotherRequiredPayload(new\_row, new\_col, current)) {

continue;

}

가능한 경우 해당 방향을 반환

switch (i) {

case 0:

INFO("robot", "robot %s can move up", robot->name);

return CMD\_UP;

case 1:

INFO("robot", "robot %s can move down", robot->name);

return CMD\_DOWN;

case 2:

INFO("robot", "robot %s can move left", robot->name);

return CMD\_LEFT;

case 3:

INFO("robot", "robot %s can move right", robot->name);

return CMD\_RIGHT;

}

}

return CMD\_NOP;

}

int move\_robot(Robot\* robots, int idx, int number\_of\_robots, Message\* msg) {

Robot\* robot = &robots[idx];

로봇의 현재 위치와 요구되는 목표를 비교

도달한 경우 정상 종료

if (robot->current > robot->required && map[robot->row][robot->col] == robot->current) {

return 0;

}

받은 명령을 확인

Command cmd = msg->cmd;

일정 횟수 이상의NOP명령이면 움직일 방향을 탐색

if (robot->nop > 5) {

robot->nop = 0;

cmd = find\_movable\_direction(robots, idx, number\_of\_robots);

}

새로운 위치를 계산

int new\_row = robot->row;

int new\_col = robot->col;

switch (cmd) {

case CMD\_UP:

new\_row--;

break;

case CMD\_DOWN:

new\_row++;

break;

case CMD\_LEFT:

new\_col--;

break;

case CMD\_RIGHT:

new\_col++;

break;

case CMD\_NOP:

robot->nop++;

break;

default:

ERROR("robot", "robot %s received invalid command %d", robot->name, cmd);

update\_msg(msg, robot);

return -1; // Invalid command

}

char new\_pos = map[new\_row][new\_col];

벽이나 다른 로봇과 충돌하는지 확인

if (isOutOfBound(new\_row, new\_col)) {

...

}

새로운 위치에 따라 로봇의 상태를 업데이트

robot->row = new\_row;

robot->col = new\_col;

INFO("robot", "new position: (%d, %d), current: %c", new\_row, new\_col, robot->current);

로봇의 상태를 메시지에 업데이트하고 결과를 반환

if (new\_pos == robot->current) {

if (robot\_is\_unloaded(robot)) {

INFO("robot", "robot %s unloaded %c at (%d, %d)", robot->name, robot->current, new\_row, new\_col);

} else {

INFO("robot", "robot %s loaded %c at (%d, %d)", robot->name, robot->current, new\_row, new\_col);

int tmp = robot->current;

robot->current = robot->required;

robot->required = tmp;

}

}

update\_msg(msg, robot);

INFO("robot", "move %s to (%d, %d)", robot->name, new\_row, new\_col);

return 0;

}

**[ node.c ]**

#define MAX\_QUEUE\_SIZE ((ROW\_MAX + 1) \* (COL\_MAX + 1))

행과 열을 저장하는 데 사용

typedef struct Point {

int row;

int col;

} Point;

큐를 구현하는 데 사용

typedef struct Queue {

Point data[MAX\_QUEUE\_SIZE];

int front;

int rear;

} Queue;

void initQueue(Queue \*queue) {

큐를 초기화

queue->front = 0;

queue->rear = 0;

}

bool isQueueEmpty(Queue \*queue) {

큐가 비어있는지 확인

return queue->front == queue->rear;

}

bool isQueueFull(Queue \*queue) {

큐가 가득 찼는지 확인

return (queue->rear + 1) % MAX\_QUEUE\_SIZE == queue->front;

}

void enqueue(Queue \*queue, Point point) {

큐가 가득 차있으면 종료

if (isQueueFull(queue)) {

return;

}

큐가 가득 차있지 않은 경우

rear를 증가시키고 데이터를 추가

queue->rear = (queue->rear + 1) % MAX\_QUEUE\_SIZE;

queue->data[queue->rear] = point;

}

Point dequeue(Queue \*queue) {

큐가 가득 차있으면 종료

if (isQueueEmpty(queue)) {

Point emptyPoint = {-1, -1};

return emptyPoint;

}

큐가 비어있지 않은 경우

front를 증가시키고 해당 위치의 데이터를 반환

queue->front = (queue->front + 1) % MAX\_QUEUE\_SIZE;

Point point = queue->data[queue->front];

return point;

}

bool msg\_is\_loaded(Message \*msg) {

메시지의 현재 상태가 요구되는 목표보다 크면 언로드 상태로 판단

return msg->current > msg->required;

}

int find\_path(Message \*msg)

{

INFO("cnt", "find\_path row: %d, col: %d, current: %c, required: %c", msg->row, msg->col, msg->current, msg->required);

int row = msg->row;

int col = msg->col;

로봇의 현재 위치와 요구되는 목표를 비교

목표에 도달한 경우 NOP 반환

if (msg\_is\_loaded(msg) && map[row][col] == msg->current) {

return CMD\_NOP;

}

int dr[] = {-1, 1, 0, 0};

int dc[] = {0, 0, -1, 1};

큐 생성 및 초기화

Queue queue;

initQueue(&queue);

방문 확인 및 이전 방문 위치 저장 배열

bool visited[ROW\_MAX + 1][COL\_MAX + 1] = {false};

Point prev[ROW\_MAX + 1][COL\_MAX + 1] = {0};

Point start = {row, col};

enqueue(&queue, start);

visited[start.row][start.col] = true;

BFS를 사용하여 로봇이 움직일 수 있는 경로 탐색

while (!isQueueEmpty(&queue)) {

Point current = dequeue(&queue);

for (int i = 0; i < 4; i++) {

int nr = current.row + dr[i];

int nc = current.col + dc[i];

if (isOutOfBound(nr, nc)) {

...

}

Point next = {nr, nc};

enqueue(&queue, next);

INFO("cnt", "enqueue row: %d, col: %d", nr, nc);

visited[nr][nc] = true;

prev[nr][nc] = current;

}

}

경로를 따라 로봇이 움직일 방향을 결정

Point target;

bool found = false;

for (int i = 0; i < ROW\_MAX + 1; i++) {

for (int j = 0; j < COL\_MAX + 1; j++) {

if (visited[i][j] && map[i][j] == msg->current) {

target.row = i;

target.col = j;

found = true;

break;

}

}

if (found) {

break;

}

}

if (!found) {

return CMD\_NOP;

}

while (prev[target.row][target.col].row != row ||

prev[target.row][target.col].col != col) {

target = prev[target.row][target.col];

}

if (target.row == row - 1) {

return CMD\_UP;

}

else if (target.row == row + 1) {

return CMD\_DOWN;

}

else if (target.col == col - 1) {

return CMD\_LEFT;

}

else if (target.col == col + 1) {

return CMD\_RIGHT;

}

return CMD\_NOP;

}

**[ aw\_thread.c ]**

void block\_thread() {

struct thread \*cur;

cur = thread\_current();

현재 스레드를 blocked\_threads리스트에 추가

list\_push\_back(&blocked\_threads, &cur->elem);

enum intr\_level old\_level;

인터럽트를 비활성화

old\_level = intr\_disable ();

스레드를 블록

thread\_block ();

intr\_set\_level (old\_level);

}

void unblock\_threads() {

enum intr\_level old\_level;

인터럽트를 비활성화

old\_level = intr\_disable ();

blocked\_threads리스트에서 스레드를 하나씩 꺼내서 해제

while (!list\_empty (&blocked\_threads)) {

thread\_unblock (list\_entry (list\_pop\_front (&blocked\_threads), struct thread, elem));

}

intr\_set\_level (old\_level);

}