운영체제

쓰레드 동기화의 교착 상태 문제

학번 : 20165968
이름 : 맹채정

2020년 5월 9일

1. 프로그램 코드 및 설명

Explanation	Code

먼저 struct 구조체로 차로 내 4구간인 NW, NE, SW, SE, INTER에 대한 세마포어를 선언한다. message_function() 함수를 호출하는 횟수를 의미하는 messafunc_callnum 변수를 선언하여 0으로 초기화한다. inititems() 함수에서는 생성한 구조체 NW, NE, SW, SE, INTER에 대한 세마포어를 sempah_create()를 이용해 추가로 생성하며 파라미터인 세마포어 이름과 initial count 값을 지정한다. NW, NE, SW, SE의 initial count는 1로 지정하고 INTER의 initial count는 3으로 지정한다.	if (done_semaph == NULL) {
	done_semaph=semaph_create("donesem", 0);
	if (done_semaph == NULL) {
	panic("synchtest: sem_create failed\n");
	}
	}
	// (NW, NE, SW, SE, INTER 세마포어 생성)
	if (NW == NULL) {
	NW = semaph_create("NW", 1);
	if (NW == NULL) {
	panic("synchtest: sem_create failed\n");
	}
	}
	if (NE == NULL) {
	NE = semaph_create("NE", 1);
	if (NE == NULL) {
	panic("synchtest: sem_create failed\n");
	}
	}
	if (SW == NULL) {
	SW = semaph_create("SW", 1);
	if (SW == NULL) {
	panic("synchtest: sem_create failed\n");
	}
	}
	if (SE == NULL) {
	SE = semaph_create("SE", 1);
	if (SE == NULL) {
	panic("synchtest: sem_create failed\n");
	}
	}
	if (INTER == NULL) {
	INTER = semaph_create("INTER", 3);
	if (INTER == NULL) {
	panic("synchtest: sem_create failed\n");
	}
	}
	}

static struct semaphore* NW; // NW 방향 추가
static struct semaphore* NE; // NE 방향 추가
static struct semaphore* SW; // SW 방향 추가
static struct semaphore* SE; // SE 방향 추가
static struct semaphore* INTER; // 교차로에 들어올 수 있는 차의 개수
int messafunc_callnum = 0;//message_function호출 횟수
 	
static void inititems(void)
{
if (test_semaph == NULL) {
test_semaph=semaph_create("testsem", 2);
if (test_semaph == NULL) {
panic("synchtest:sem_create failed\n");
}
}
 	
if (testlock == NULL) {
testlock = lock_create("testlock");
if (testlock == NULL) {
panic("synchtest:lock_create failed\n");
}
}
 	
if (testcv == NULL) {
testcv = cv_create("testlock");
if (testcv == NULL) {
panic("synchtest: cv_create failed\n");
}
}	

Explanation	Code

s_message_print() 함수는 쓰레드가 생성되었을 때 메시지를 출력하며 쓰레드로 차량이 생성되어 차량번호, 시작 위치, 움직이는 차량 번호(car_number)와 시작 위치(start), 이동 방향(str)를 파라미터로 받는다. P(test_semaph)의 세마포어는 kprint 함수가 한번에 하나씩 호출되도록 하고 사용한 자원은 semtest()에서 V(test_semaph)으로 회수된다. 이때 semtest()에서 P(done_semaph)으로 사용된 자원은 s_message_print의 V(done_semaph)으로 회수된다. 이렇게 쓰레드(차량)가 생성되었을 때 메시지 출력이 하나씩 되도록 세마포어를 통해 관리된다.
차량이 움직이고자 하는 방향을 출력하는 방식은 다음과 같다. if else 문으로 회전 횟수(turn_num)에 따라 strcpy로 str에 문자열을 복사한다. turn_num이 0일 때는 직진, 1일 때는 우회전, 2일 때는 좌회전에 해당하는 문자열을 복사하여 kprintf()에서 차량의 이동 방향을 출력한다. 이처럼 메시지를 출력하는 kprintf() 함수도 세마포어로 감싸 메시지들이 동시에 출력되어 문자가 섞이는 것을 방지한다.
	static void s_message_print(int car_number, char start, int turn_num) {
	
	//쓰레드가 생성되었을 때 메시지 출력
	char str[15];
	
	if (turn_num==0) strcpy(str, "go straight");
	else if (turn_num==1) strcpy(str, "turn right");
	else if (turn_num==2) strcpy(str, "turn left");
	else strcpy(str, "error");
	
	P(test_semaph);
	kprintf("car: %d, waiting in %c to %s\n", car_number, start, str);
	V(done_semaph);
	}

message_function() 함수는 차량의 교차로 진입, 교차로 내 움직임, 교차로에서 빠져나와 목적지 도달, 이 3가지 움직임 경우에 대한 메시지를 출력한다. strcmp로 시작위치(start)와 이전 위치(prev)를 비교하여 같다면 차량이 방금 교차로에 진입했음을 의미하므로 이전위치(prev)에서 진입 후 현재위치(current)로 이동했음을 의미하는 메시지를 “car: 차량번호, enter 이전위치 -> 현재위치”와 같은 형식으로 출력한다.
차량의 현재위치(current)와 목적지(destination)를 비교하여 같은 경우는 교차로 내 이동을 마치고 목적지에 도달하여 빠져나올 때이므로 차량의 목적지(destination)와 시작위치(start)를 알려주는 메시지를 "car: 차량번호, arrive 목적지, start: 시작위치” 형식으로 출력한다.
앞서 두 가지 경우에 모두 해당이 안 된다면 차량이 진입하거나 목적지에 도달한 경우가 아니므로 교차로 내에서 이동하는 경우일 것이다. 이 경우 차량이 어디에서(prev) 어디로(current) 움직였는가와 시작위치(start), 다음위치(next), 목적지(destination)을 출력하며 "car: 차량번호, moved 이전위치 -> 현재위치, start: 시작위치, after: 다음위치, dest: 목적지\n" 형식으로 출력한다.
	static void message_function(int car_number, const char* start, const char* prev, const char* current, const char* next, const char* destination) {
	
	//메시지 출력
	if (strcmp(start, prev) == 0) {
	//교차로에 방금 진입
	P(test_semaph);
	kprintf("car: %d, enter %s -> %s\n", car_number, prev, current);
	V(done_semaph);
	}
	
	else if (strcmp(current, destination) == 0) {
	//목적지에 도착, 교차로에서 나옴
	P(test_semaph);
	kprintf("car: %d, arrive %s, start: %s\n", car_number, destination, start);
	V(done_semaph);
	}

이 세가지 경우 각각에서 차량(쓰레드)이 자원을 하나씩 사용하여 이동할 때마다 P(test_semaph)로 세마포어 값을 1 감소시키며 자원을 하나 사용한다. 사용한 자원은 semtest()에서 V(test_semaph)으로 회수된다. 이때 semtest()에서 P(done_semaph)으로 사용된 자원은 message_function()의 V(done_semaph)으로 회수되며 세마포어 값을 1 증가시켜 자원을 release 한다.

여기서 P()는 proberen이며 semWait() 역할을 하고 세마포어를 획득하여 요청된 것을 실행한다. 세마포어가 0일 경우 자원이 release되어 세마포어가 다시 1 이상이 될 때까지 쓰레드는 대기 상태에 있는다. V()는 verhogen이고 semSignal() 역할로 실행이 종료된 후 세마포어를 release하여 자원을 다시 사용할 수 있게 해준다. 이처럼 세마포어가 release되어 다시 1 이상이 되면 대기상태였던 쓰레드가 실행되며 세마포어는 1 감소된다. 이와 같이 사용한 자원을 사용 후 바로 semtest()에서 release시켜 reusable하게 하는 것은 세마포어를 이용해 자원 사용에서의 Mutual exclusion을 지키기 위함이다. 이는 자원을 하나의 프로세스(쓰레드)만이 사용할 수 있도록 하여 각 차량이 자원을 사용해 이동하는 동안 다른 차량들이 동시에 이동해 같은 자리에 2개의 차량이 들어와 사고가 나는 것을 방지하는 기능을 한다. 이와 같은 Mutual exclusion(상호배제)는 교착상태(데드락)를 예방하는데 도움이 된다. 또한 inititems()에서 세마포어 test_semaph의 initial count를 2로 선언하였으므로 사용할 수 있는 자원은 최대 2개이며 하나는 kprint()로 출력하는데, 하나는 차량이 이동하는데 사용되어 한번에 하나의 차량만 이동 가능하도록 함으로써 교착상태(데드락)가 발생하지 않도록 한다. 이는 철학자 문제에서 철학자들에게 하나의 포크로 먹는 방법과 비슷한 원리임을 알 수 있다.

```
        else {
        //교차로 내 이동
                P(test_semaph);
                kprintf("car: %d, moved %s -> %s, start: %s, after: %s, dest: %s\n", car_number, prev, current, start, next, destination);
                V(done_semaph);
        }
}
```

Explanation	Code

semtest()는 자동차인 쓰레드를 생성하고 차량 이동을 수행하도록 semtestthread()를 호출하는 함수이다.
먼저 첫 번째 for문에서 thread_fork()를 이용해 32개의 쓰레드(차량)를 생성하여 각 쓰레드들이 semtestthread를 실행하도록 한다. 여기서 NTHREADS는 쓰레드 개수를 의미하며 차량 수가 32개이므로 쓰레드 개수도 32개가 된다.
 	
프로세스가 실행된 후 자원은 다시 사용하기 위해 회수되어야 한다. 따라서 두 번째 for문은 message_create()에서 쓰레드 개수(NTHREADS)만큼 V((test_semaph)로 test_semaph 세마포어 값을 증가시켜 사용된 자원을 release한다. 그 다음 P(done_semaph)로 done_semaph 세마포어는 감소시킨다.
또한 세 번재 for문을 이용해 message_function에서 사용된 test_semaph 세마포어도 이에 대한 자원을 release하고 V(test_semaph)로 세마포어 값을 증가시키고 P(done_semaph)로 done_semaph 세마포어는 감소시킨다.
이러한 과정을 통해 32개의 쓰레드가 생성되고 각각의 프로세스를 수행하여 결과가 출력되는데 문제가 없는 이유는 다음과 같다. 이는 32개의 쓰레드가 동시에 실행되어도 프로세스가 Critical Section에 들어가는 쓰레드들을 P()와 V(), 즉 semWait()와 semSignal() 연산을 통해 세마포어를 제어하여 가능한 것부터 순차적으로 진행되도록 하기 때문이다.
	int semtest(int nargs, char** args) {
	
	int i, result;
	(void)nargs;
	(void)args;
	
	inititems();
	kprintf("Starting semaphore test...\n");
	kprintf("If this hangs, it's broken: ");
	
	P(test_semaph);
	P(test_semaph);
	kprintf("ok\n");
	
	for (i = 0; i < NTHREADS; i++) {
	result = thread_fork("semtest", NULL, semtestthread, NULL, i);
	if (result) {
	panic("semtest: thread_fork failed: %s\n",
	
	strerror(result));
	}
	}
	for (i = 0; i < NTHREADS; i++) {
	//message_create에서의 testsem, donesem 회수
	V(test_semaph);
	P(done_semaph);
	}
	
	for (i = 0; i < messafunc_callnum; i++){
	//message_function에서의 testsem, donesem 회수
	V(test_semaph);
	P(done_semaph);
	}
	
	/* so we can run it again */
	V(test_semaph);
	V(test_semaph);
	
	kprintf("Semaphore test done.\n");
	kprintf("ver 4.1\n");
	return 0;
	}

Explanation	Code

semtest()에서 생성된 32개의 쓰레드들은 각각 semtestthread()는 실행하여 차량의 직진, 우회전, 좌회전을 수행하도록 한다.
먼저 차량의 시작 위치를 정하기 위해 선언한 변수 start_num은 4가지 시작위치 각각의 경우를 나누어 처리하기 위해 random()으로 생긴 숫자를 4로 나눈 나머지 값을 받아 저장한다. 차량의 이동 방향을 정하기 위한 변수인 turn_num은 3가지 이동 방향 각각에 대한 경우를 나누어 주기 위해 random() 숫자를 3으로 나눈 나머지를 저장한다.
 	
차량 쓰레드의 시작 위치는 if else문으로 start_num의 값에 따라 정해지는데 N, E, S, W 4개 방향에서 가능하므로 0일 때는 북쪽(N)이며 1일 때는 동쪽(E), 2일 때는 남쪽(S), 3일 때는 서쪽(W)된다. 각각의 방향은 start에 저장된다.
s_message_print() 함수를 호출하여 쓰레드를 생성하고 교차로 내 차량 대수(num)와 시작지점(start), 회전 횟수(turn_num)를 인자로 한다.
 	
차량의 이동 방향은 회전 횟수(turn_num)에 따라 정해지며 직진, 우회전, 좌회전 3가지 이동이 가능하며 0일 때는 직진, 1일 때는 우회전, 2일 때는 좌회전을 하게 된다. 직진일 경우 message_function 호출 횟수인 messafunc_callnum는 3이 되고 go_straight() 함수를 호출하여 차량번호(num)와 시작위치(start)를 인자로 실행한다. 같은 방식으로 우회전일 경우 messafunc_callnum는 2가 되어 message_function을 2회 호출하며 right_turn() 함수를 실행한다. 좌회전일 경우 messafunc_callnum는 4가되어 message_function을 4회 호출하며 left_turn() 함수를 실행한다.
	static void semtestthread(void* junk, unsigned long num) {
	
	(void)junk;
	
	//Only one of these should print at a time.
	
	int start_num = random() % 4;
	// 시작위치 생성
	int turn_num = random() % 3;
	// 직진, 우회전, 좌회전
	char start;
	// 시작위치
	
	//시작 위치 N, E, S, W 지정
	if (start_num == 0) start = 'N';
	else if (start_num == 1) start = 'E';
	else if (start_num == 2) start = 'S';
	else start = 'W';
	
	//쓰레드 생성
	s_message_print(num, start, turn_num);
	
	//직진, 우회전, 좌회전
	if (turn_num == 0) {
	messafunc_callnum += 3;
	// 직진 - message_function 호출 횟수 = 3
	go_straight(num, start);
	}
	else if (turn_num == 1) {
	messafunc_callnum += 2;
	// 우회전 - message_function 호출 횟수 = 2
	right_turn(num, start);
	}
	else {
	messafunc_callnum += 4;
	// 좌회전 - message_function 호출 횟수 = 4
	left_turn(num, start);
	}
	}

Explanation	Code

< 직진 NW-SW >
gostraight_process()는 직진 프로세스를 나타내는 함수이며 차량번호(car_number), 시작위치(start), 처음 교차로 진입 시 교차로 내 위치(step_1), 진입 후 그 다음 이동한 교차로 내 위치(step_2), 목적지(dest)를 인자로 받는다. 직진은 3번 이동으로 목적지에 도달할 수 있으므로 message_function()을 3번 호출한다.
먼저 첫 번째 이동인 교차로 진입은 P(INTER)로 INTER 세마포어를 사용하여 실행하고 세마포어 값을 1 감소시킨다. message_function을 호출하여 차량 번호(car_number), 시작위치(start), 이전위치(prev), 현재위치(current), 다음위치(next), 목적지(destination)를 인자로 실행한다. 진입 시에는 시작위치와 이전위치가 모두 start이고 현재위치는 step_1이 되며 다음 위치는 step_2가 된다. step_1과 step_2는 sem_name에 저장되고 이 세마포어 이름으로 차량 위치를 message_function에서 출력되어 확인할 수 있도록 한다.
두 번째 이동은 P(step_1)로 step_1 세마포어를 사용하고 1 감소시킨다. message_function()을 호출 시 차량번호, 시작위치, 목적지는 같은 차량이므로 동일한 값이 되고 이전 위치는 진입 후 이동한 위치이므로 step_1이고 현재위치는 step_2, 다음 위치는 목적지인 dest가 된 것을 출력한다. 그런 다음, step_1에 대한 세마포어는 프로세스의 수행이 끝났으므로 V(step_1)로 자원을 release하고 세마포어 수를 증가시킨다.
마지막 목적지로 가는 세 번째 이동은 P(step_2)를 이용해 step_2 세마포어를 사용하고 세마포어 값을 1 감소시킨다. 앞과 같은 방식으로 message_function을 호출하여 이전위치는 step_2이고 현재 위치는 목적지인 dest, 다음 위치도 dest가 된 것을 출력한다. V(step_2)로 step_2에 대한 자원을 release하고 세마포어 값을 1 증가시키고 앞서 수행한 세마포어 INTER에 대해서도 자원을 release하고 세마포어 값을 1 증가시켜 자원을 reusable하게 해준다.
go_straight()는 차량(쓰레드)의 시작 위치에 따라 교차로에서 직진을 할 때 이동하는 위치들에 대한 값들을 gostraight_process()에 할당하여 호출하는 함수이다. 차량은 동서남북(E, W, S, N) 모든 방향에서 출발할 수 있으므로 case로 각각의 경우에 대해 이동 위치를 지정해주면 된다. 예를 들어 북쪽(N)에서 출발하는 경우 직진을 할 때 교차로 내 다음 이동 위치는 NW가 되고 그 다음은 SW, 목적지는 S가 된다. 동쪽(E)에서 출발하면 NE, NW, W 순으로 이동하고 남쪽(S)은 SE, NE, N 순으로, 서쪽(W)은 SW, SE, E 순으로 이동하면 된다.
	static void gostraight_process(int car_number, const char* start, struct semaphore* step_1, struct semaphore* step_2, const char* dest) {
	
	//직진 과정
	// step1은 처음 교차로에 진입 시 교차로 위치
	// step2는 그 다음 이동한 교차로 내 위치
	P(INTER); //교차로 진입
	P(step_1);
	message_function(car_number, start, start, step_1->sem_name, step_2->sem_name, dest);
	P(step_2);
	message_function(car_number, start, step_1->sem_name, step_2->sem_name, dest, dest);
	V(step_1);
	message_function(car_number, start, step_2->sem_name, dest, dest, dest);
	V(step_2);
	V(INTER); // 교차로 진입
	
	}
	
	static void go_straight(int car_number, char start) {
	
	//차량이 교차로에서 직진
	switch (start) {
	case 'N': gostraight_process(car_number, "N", NW, SW, "S");
	break;
	case 'E': gostraight_process(car_number, "E", NE, NW, "W");
	break;
	case 'S': gostraight_process(car_number, "S", SE, NE, "N");
	break;
	case 'W': gostraight_process(car_number, "W", SW, SE, "E");
	break;
	default: P(test_semaph);
	kprintf("Wrong input..\n");
	V(done_semaph);
	break;
	}
	}

Explanation	Code

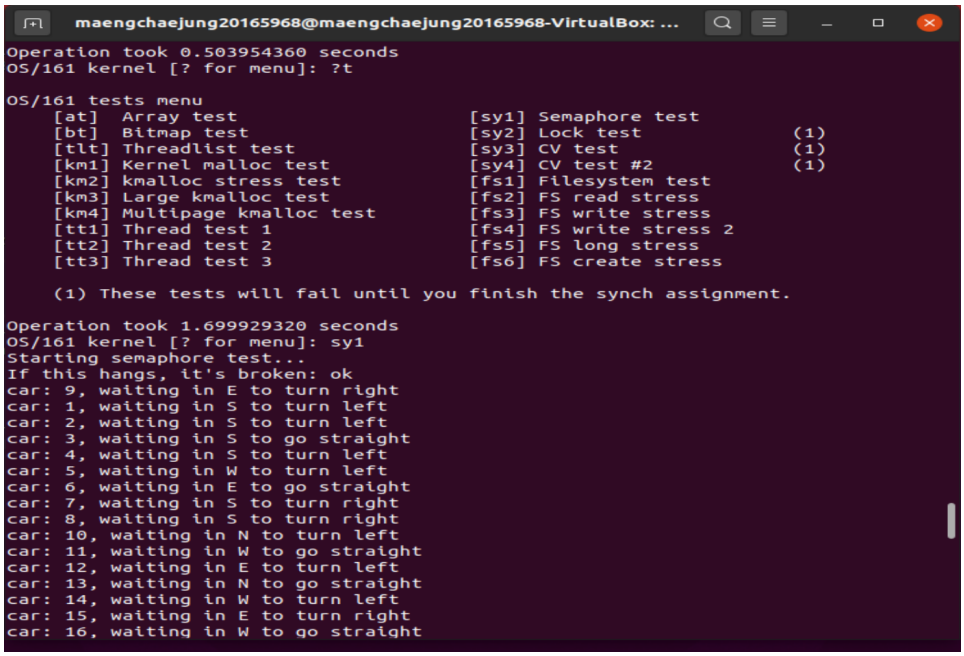
< 좌회전 NW-SW-SE >
 	
leftturn_process()는 좌회전 프로세스를 나타내는 함수이며 차량번호(car_number), 시작위치(start), 처음 교차로 진입 시 교차로 내 위치(step_1), 두 번째 위치(step_2), 세 번째 위치(step_3), 목적지(dest)를 인자로 받는다. 좌회전은 교차로 내에서 4번 이동으로 목적지에 도달할 수 있으므로 message_function()을 4번 호출한다.
먼저 첫 번째 이동인 교차로 진입은 P(INTER)로 INTER 세마포어를 사용하여 실행하고 세마포어 값을 1 감소시킨다. message_function() 호출 시 교차로 진입은 시작위치와 이전위치가 모두 start이고 현재위치는 step_1이 되며 다음 위치는 step_2가 된다.
두 번째 이동은 P(step_1)로 step_1 세마포어를 사용하고 1 감소시킨다. message_function()을 호출 시 이전 위치는 step_1, 현재위치는 step_2가 되고 그 다음 위치는 step_3가 되어 출력한다. 그런 다음, step_1에 대한 세마포어는 프로세스의 수행이 끝났으므로 V(step_1)로 세마포어 수를 증가시킨다.
세 번째 이동은 P(step_2)를 이용해 step_2 세마포어를 사용하고 세마포어 값을 1 감소시킨다. message_function을 호출하여 이전위치는 step_2, 현재 위치는 step_3, 다음 위치는 목적지인 dest가 된 것을 출력한다. V(step_2)로 step_2 세마포어 값을 1 증가시킨다.
네 번째 이동은 P(step_3)를 이용해 step_3 세마포어를 사용하고 세마포어 값을 1 감소시킨다. message_function을 호출하여 이전위치는 step_3, 현재위치는 목적지인 dest, 다음 위치도 dest가 된 것을 출력한다. V(step_3)로 step_3 세마포어 값을 1 증가시키고 앞서 수행한 세마포어 INTER도 값을 1 증가시킨다.
 	
left_turn()는 차량(쓰레드)의 시작 위치에 따라 교차로에서 좌회전 시 이동 위치들에 대한 값들을 leftturn_process()에 할당하여 호출하는 함수이다. 차량 이동은 차량의 출발 위치(E, W, S, N)에 따라 각각의 경우에 대해 지정해주면 된다.
예를 들어 북쪽(N)에서 출발하는 경우 좌회전 시 교차로 내 진입 위치는 NW가 되고 두 번째는 SW, 세 번째는 SE, 목적지는 E가 된다. 동쪽(E)에서 출발하면 NE, NW, SW, S 순으로 이동하고 남쪽(S)은 SE, NE, NW, W 순으로, 서쪽(W)은 SW, SE, NE, N 순으로 이동하면 된다.
	static void leftturn_process(int car_number, const char* start, struct semaphore* step1, struct semaphore* step2, struct semaphore* step3, const char* dest) {
	
	//좌회전 과정
	P(INTER); //교차로 진입
	P(step1);
	message_function(car_number, start, start, step1->sem_name, step2->sem_name, dest);
	P(step2);
	message_function(car_number, start, step1->sem_name, step2->sem_name, step3->sem_name, dest);
	V(step1);
	P(step3);
	message_function(car_number, start, step2->sem_name, step3->sem_name, dest, dest);
	V(step2);
	message_function(car_number, start, step3->sem_name, dest, dest, dest);
	V(step3);
	V(INTER); // 교차로 진입
	}
	
	static void left_turn(int car_number, char start) {
	//차량이 교차로에서 좌회전함을 의미하는 함수
	switch (start) {
	case 'N': leftturn_process(car_number, "N", NW, SW, SE, "E");
	break;
	case 'E': leftturn_process(car_number, "E", NE, NW, SW, "S");
	break;
	case 'S': leftturn_process(car_number, "S", SE, NE, NW, "W");
	break;
	case 'W': leftturn_process(car_number, "W", SW, SE, NE, "N");
	break;
	
	default: P(test_semaph);
	kprintf("잘못된 입력입니다.\n");
	V(done_semaph);
	break;
	}
	}

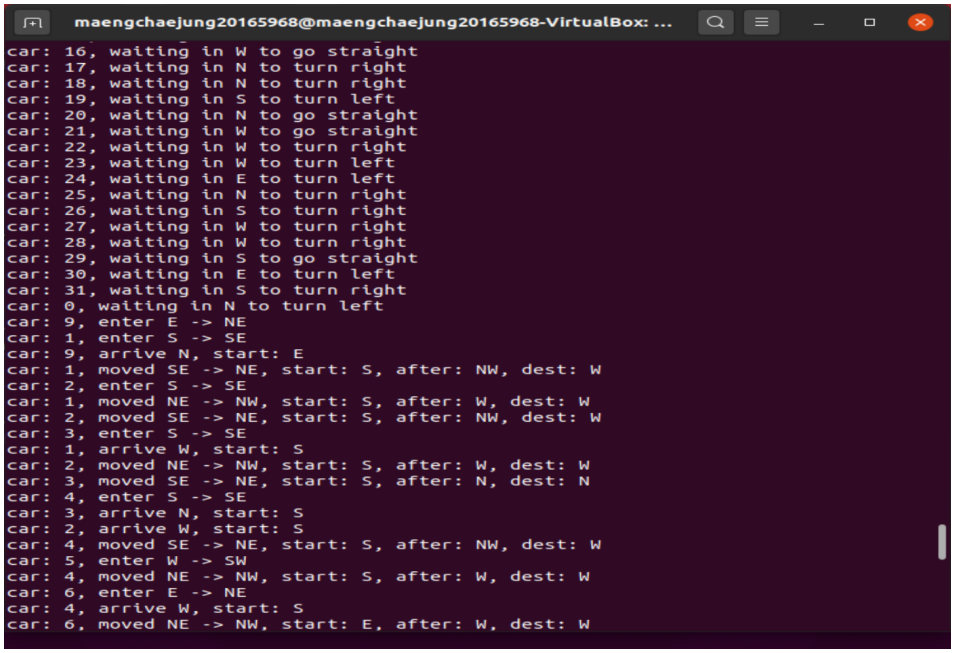
Explanation	Code

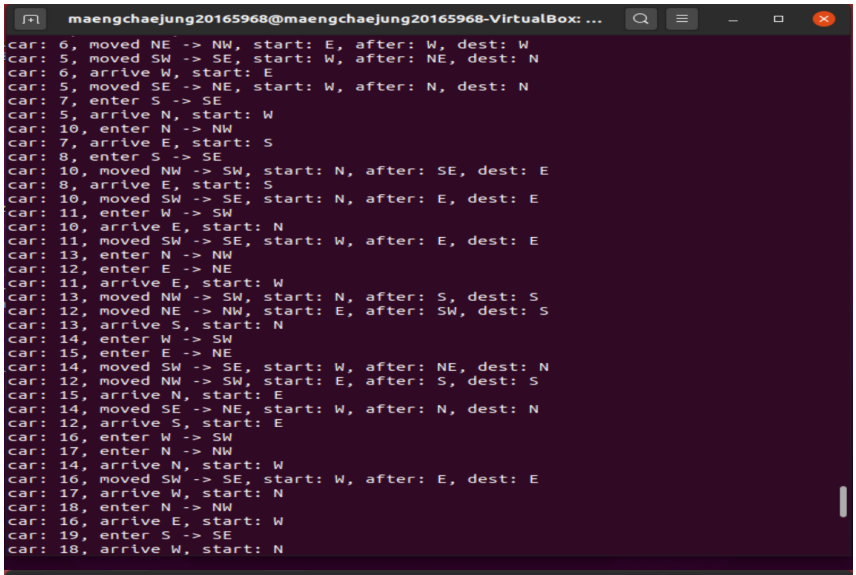
< 우회전 : NW >
 	
rightturn_process()는 우회전 프로세스를 나타내는 함수이며 차량번호(car_number), 시작위치(start), 처음 교차로 진입 시 교차로 내 위치(step)와 목적지(dest)를 인자로 받는다. 우회전은 교차로 내에서 2번 이동으로 목적지에 도달할 수 있으므로 message_function()을 2번 호출한다.
먼저 첫 번째 이동인 교차로 진입은 P(INTER)로 INTER 세마포어를 사용하여 실행하고 세마포어 값을 1 감소시킨다. message_function() 호출 시 교차로 진입은 시작위치와 이전위치가 모두 start이고 현재위치는 step이 되며 다음 위치는 목적지인 dest가 된다.
두 번째 이동은 P(step)로 step 세마포어를 사용하고 1 감소시킨다. message_function()을 호출 시 이전 위치는 step, 현재위치는 목적지 dest가 되고 그 다음 위치도 dest가 되어 출력한다. 그런 다음, step에 대한 세마포어는 프로세스의 수행이 끝났으므로 V(step)로 세마포어 수를 증가시키고 앞서 수행한 세마포어 INTER도 값을 1 증가시킨다.
 	
right_turn()는 차량(쓰레드)의 시작 위치에 따라 교차로에서 우회전 시 이동 위치들에 대한 값들을 rightturn_process()에 할당하여 호출하는 함수이다. 차량 이동은 차량의 출발 위치(E, W, S, N)에 따라 각각의 경우에 대해 지정해주면 된다.
예를 들어 북쪽(N)에서 출발하는 경우 우회전 시 교차로 내 진입 위치는 NW가 되고 목적지는 W가 된다. 동쪽(E)에서 출발하면 NE, N 순으로 이동하고 남쪽(S)은 SE, E 순으로, 서쪽(W)은 SW, S 순으로 이동하면 된다.
만약 입력이 잘못 되면 default에서 잘못되었다는 메시지 출력하며 이 실행은 P(test_semaph)와 V(done_semaph)로 자원 할당과 회수를 통해 수행된다.
	static void rightturn_process(int car_number, const char* start, struct semaphore* step, const char* dest) {
	
	//우회전 과정
	P(INTER); //교차로 진입
	P(step);
	message_function(car_number, start, start, step->sem_name, dest, dest);
	message_function(car_number, start, step->sem_name, dest, dest, dest);
	V(step);
	V(INTER); // 교차로 진입
	}
	
	static void right_turn(int car_number, char start) {
	
	/* 차량이 교차로에서 우회전함을 의미하는 함수 */
	switch (start) {
	case 'N': rightturn_process(car_number, "N", NW, "W");
	break;
	case 'E': rightturn_process(car_number, "E", NE, "N");
	break;
	case 'S': rightturn_process(car_number, "S", SE, "E");
	break;
	case 'W': rightturn_process(car_number, "W", SW, "S");
	break;
	default: P(test_semaph);
	kprintf("잘못된 입력입니다.\n");
	V(done_semaph);
	break;
	}
	}

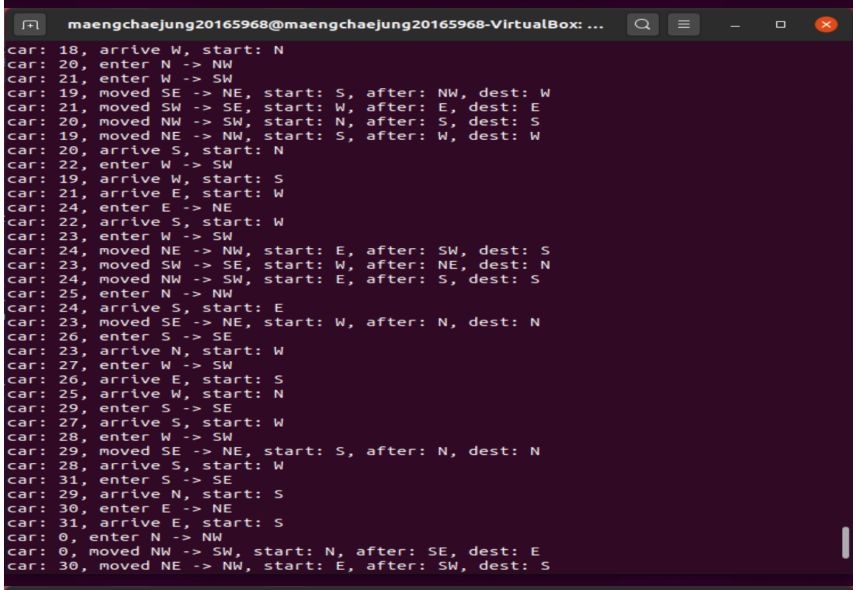
2. 출력 결과

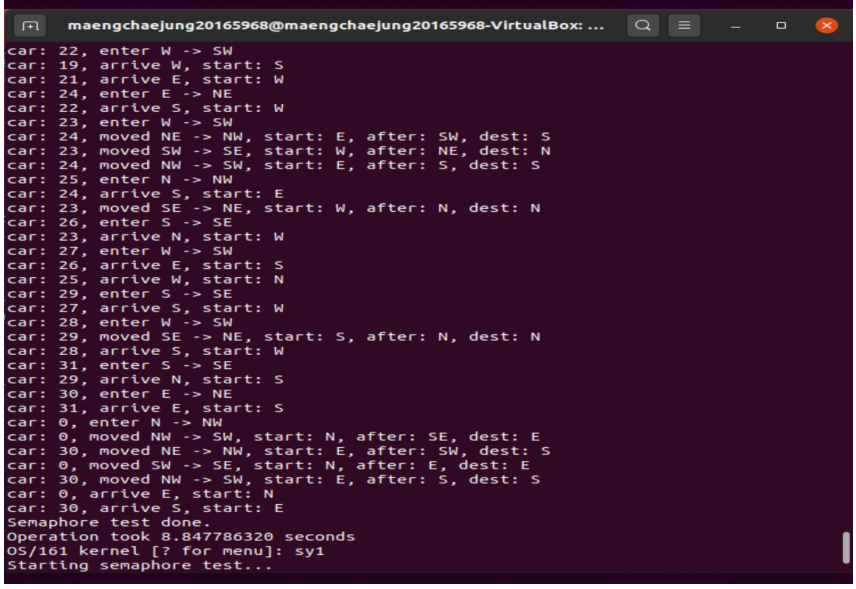
32개의 자동차(쓰레드)가 N, S, W, E 4 방향으로 접근하여 직진, 좌회전, 우회전 이동 방향을 정한 후 이동한 것을 출력한 것이다. 차량의 시작 위치와 이동 방향, 이동 위치, 목적지를 움직이는 순으로 출력하였다.






maengchaejung20165968@maengchaejung20165968-VirtualBox: ...
car: 6, moved NE -> NW, start: E, after: W, dest: W
car: 5, moved SW -> SE, start: W, after: NE, dest: N
car: 6, arrive W, start: E
car: 5, moved SE -> NE, start: W, after: N, dest: N
car: 7, enter S -> SE
car: 5, arrive N, start: W
car: 10, enter N -> NW
car: 7, arrive E, start: S
car: 8, enter S -> SE
car: 10, moved NW -> SW, start: N, after: SE, dest: E
car: 8, arrive E, start: S
car: 10, moved SW -> SE, start: N, after: E, dest: E
car: 11, enter W -> SW
car: 10, arrive E, start: N
car: 11, moved SW -> SE, start: W, after: E, dest: E
car: 13, enter N -> NW
car: 12, enter E -> NE
car: 11, arrive E, start: W
car: 13, moved NW -> SW, start: N, after: S, dest: S
car: 12, moved NE -> NW, start: E, after: SW, dest: S
car: 13, arrive S, start: N
car: 14, enter W -> SW
car: 15, enter E -> NE
car: 14, moved SW -> SE, start: W, after: NE, dest: N
car: 12, moved NW -> SW, start: E, after: S, dest: S
car: 15, arrive N, start: E
car: 14, moved SE -> NE, start: W, after: N, dest: N
car: 12, arrive S, start: E
car: 16, enter W -> SW
car: 17, enter N -> NW
car: 14, arrive N, start: W
car: 16, moved SW -> SE, start: W, after: E, dest: E
car: 17, arrive W, start: N
car: 18, enter N -> NW
car: 16, arrive E, start: W
car: 19, enter S -> SE
car: 18, arrive W, start: N


maengchaejung20165968@maengchaejung20165968-VirtualBox: ...
car: 18, arrive W, start: N
car: 20, enter N -> NW
car: 21, enter W -> SW
car: 19, moved SE -> NE, start: S, after: NW, dest: W
car: 21, moved SW -> SE, start: W, after: E, dest: E
car: 20, moved NW -> SW, start: N, after: S, dest: S
car: 19, moved NE -> NW, start: S, after: W, dest: W
car: 20, arrive S, start: N
car: 22, enter W -> SW
car: 19, arrive W, start: S
car: 21, arrive E, start: W
car: 24, enter E -> NE
car: 22, arrive S, start: W
car: 23, enter W -> SW
car: 24, moved NE -> NW, start: E, after: SW, dest: S
car: 23, moved SW -> SE, start: W, after: NE, dest: N
car: 24, moved NW -> SW, start: E, after: S, dest: S
car: 25, enter N -> NW
car: 24, arrive S, start: E
car: 23, moved SE -> NE, start: W, after: N, dest: N
car: 26, enter S -> SE
car: 23, arrive N, start: W
car: 27, enter W -> SW
car: 26, arrive E, start: S
car: 25, arrive W, start: N
car: 29, enter S -> SE
car: 27, arrive S, start: W
car: 28, enter W -> SW
car: 29, moved SE -> NE, start: S, after: N, dest: N
car: 28, arrive S, start: W
car: 31, enter S -> SE
car: 29, arrive N, start: S
car: 30, enter E -> NE
car: 31, arrive E, start: S
car: 0, enter N -> NW
car: 0, moved NW -> SW, start: N, after: SE, dest: E
car: 30, moved NE -> NW, start: E, after: SW, dest: S


maengchaejung20165968@maengchaejung20165968-VirtualBox: ...
car: 22, enter W -> SW
car: 19, arrive W, start: S
car: 21, arrive E, start: W
car: 24, enter E -> NE
car: 22, arrive S, start: W
car: 23, enter W -> SW
car: 24, moved NE -> NW, start: E, after: SW, dest: S
car: 23, moved SW -> SE, start: W, after: NE, dest: N
car: 24, moved NW -> SW, start: E, after: S, dest: S
car: 25, enter N -> NW
car: 24, arrive S, start: E
car: 23, moved SE -> NE, start: W, after: N, dest: N
car: 26, enter S -> SE
car: 23, arrive N, start: W
car: 27, enter W -> SW
car: 26, arrive E, start: S
car: 25, arrive W, start: N
car: 29, enter S -> SE
car: 27, arrive S, start: W
car: 28, enter W -> SW
car: 29, moved SE -> NE, start: S, after: N, dest: N
car: 28, arrive S, start: W
car: 31, enter S -> SE
car: 29, arrive N, start: S
car: 30, enter E -> NE
car: 31, arrive E, start: S
car: 0, enter N -> NW
car: 0, moved NW -> SW, start: N, after: SE, dest: E
car: 30, moved NE -> NW, start: E, after: SW, dest: S
car: 0, moved SW -> SE, start: N, after: E, dest: E
car: 30, moved NW -> SW, start: E, after: S, dest: S
car: 0, arrive E, start: N
car: 30, arrive S, start: E
Semaphore test done.
Operation took 8.847786320 seconds
OS/161 kernel [? for menu]: sy1
Starting semaphore test...