|  |
| --- |
| **Project #1**  **-Thread Systems-** |

|  |  |  |
| --- | --- | --- |
| **학 과** | 컴퓨터공학부 | |
|  |  |  |
| **과 목** | 운영체제 | |
|  |  |  |
| **성 명** | 임기찬 | |
|  |  |  |
| **학 번** | 20145720 | |
|  |  |  |
| **제출 날짜** | 2018.04.16 | |
|  |  |  |
| **담당 교수** | 김성조 교수님 | |
|  |  |  |

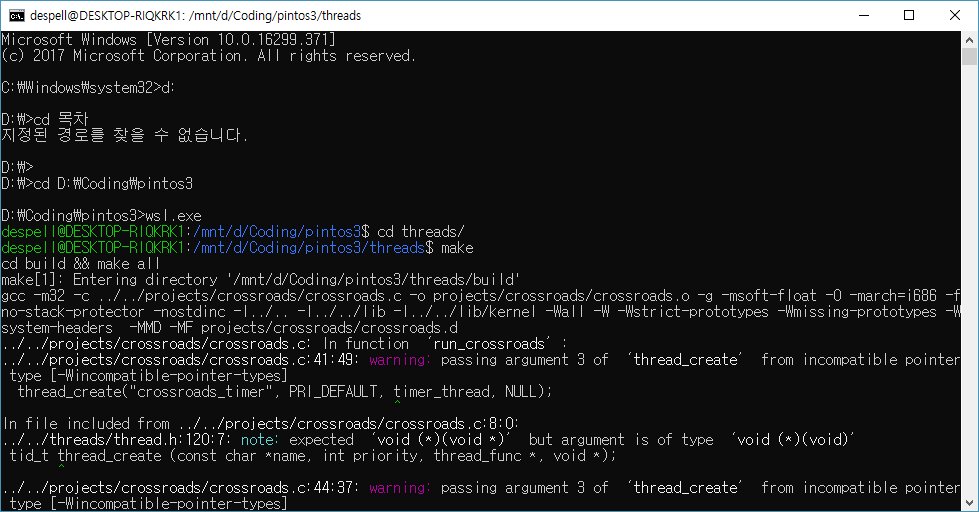


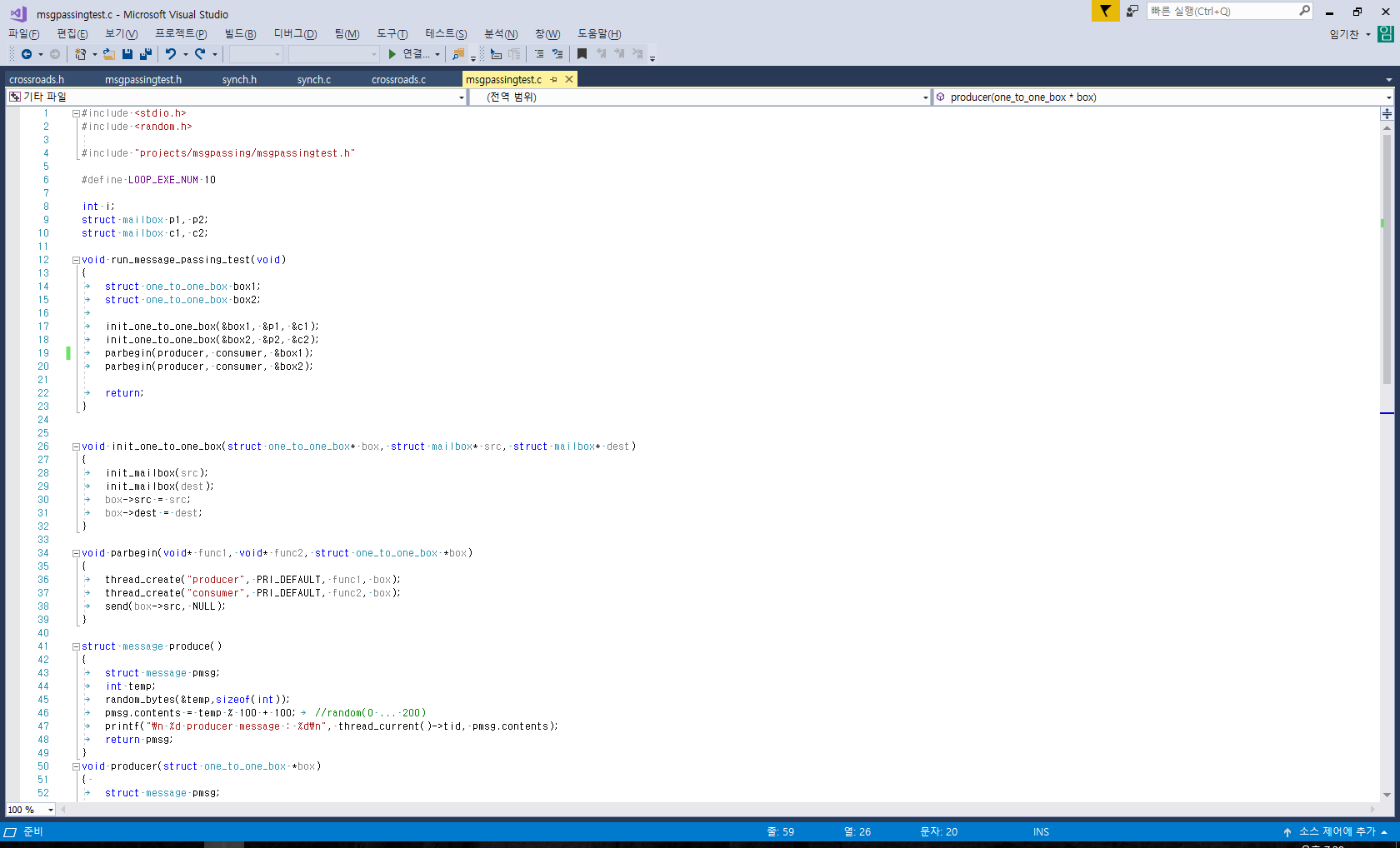
|  |
| --- |
| **목차** |

1. 프로젝트 수행 환경
2. Message Passing
   1. Send Receive 구현
   2. 테스트 코드 구현
   3. 테스트 결과
3. Crossroads
   1. 구현
   2. 테스트 결과

|  |
| --- |
| **1. 프로젝트 수행 환경** |

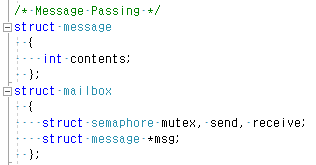
Microsoft Store에서 Ubuntu를 깔고 Windows Subsystem for Linux 기능을 사용하여 리눅스 개발 환경을 구축했습니다. 그리고 코드는 Visual Studio 2017에서 작성하여서 저장한 후 Linux 환경에서 make 한 후에 실행을 시켜서 테스트를 하였습니다.



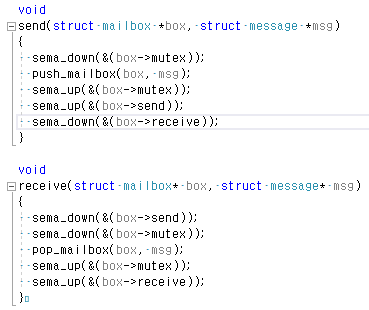


|  |
| --- |
| **2. Message Passing**  **A. Send Receive 구현** |

먼저 필요한 구조체들을 synch.h에 구현하였다.



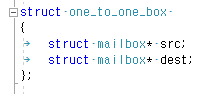
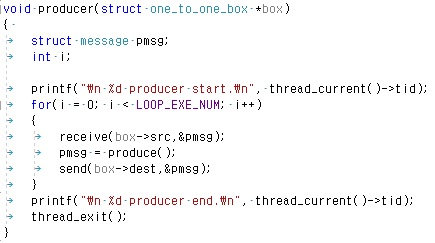
과제에서 message의 형태를 정해주지 않아서 int형으로 가정하였다. 그리고 mailbox에 message를 저장할 수 있도록 \*msg를 선언했고, 이에 대한 상호 배제를 구현하기 위해 mutex를 선언했다. 그리고 blocking send와 blocking receive를 지원하기 위해서 send와 receive semaphore를 선언해서 send와 receive 함수가 message passing이 끝날 때까지 대기하도록 구현하였다.

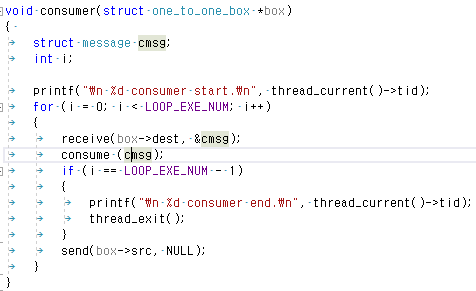


|  |
| --- |
| **B. 테스트 코드 구현** |

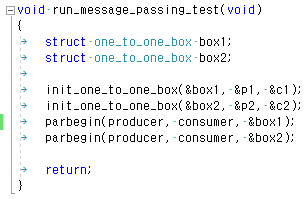
테스트 코드로는 운영체제 책에 있는 생산자/소비자 문제를 구현하였다.

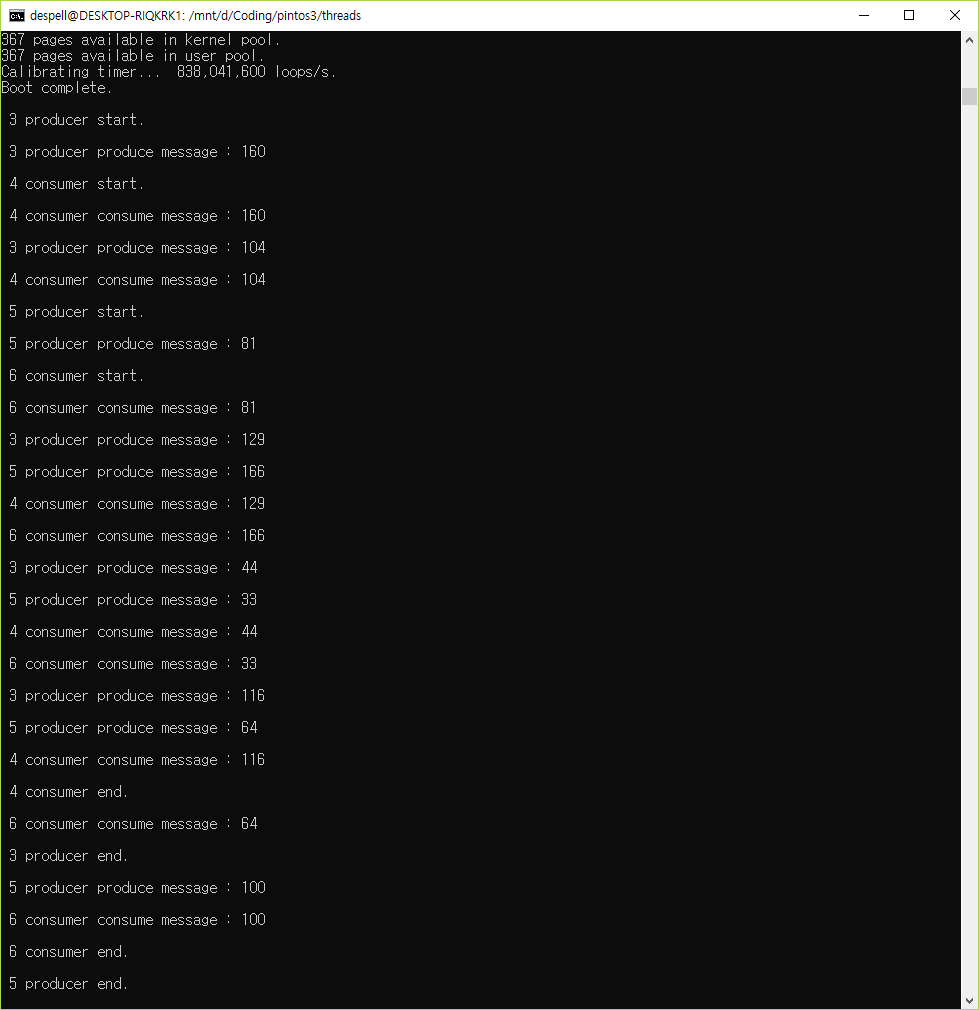
먼저 mailbox에 message를 큐가 아니라 1개만 들어가도록 정의하였으므로 producer와 consumer를 일대일로 대응 시키기 위해 one\_to\_one\_box 구조체를 선언하고 이를 묶어주었다.

producer와 consumer는 책에 있는 대로 구현하였는데, LOOP\_EXE\_NUM을 상수로 정의해서 producer와 consumer가 Produce와 consume을 LOOP\_EXE\_NUM의 수만큼 실행하도록 구현하였다. produce는 0과 200사이의 int형 자료를 무작위로 생성시키는 기능으로 구현하였고, consume은 이 자료를 출력하는 기능으로 구현하였다.



|  |
| --- |
| **C. 테스트 결과** |

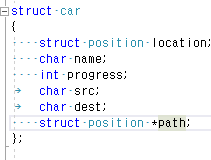


LOOP\_EXE\_NUM를 5로 정의하고, p1과 c1, p2와 c2를 producer, consumer 관계로 일대일로 대응 시키고, 이를 실행 시킨 결과는 다음과 같다.

3producer와 4consumer, 5producer와 6consumer가 각각 일대일로 대응되어 message passing을 하는 것을 볼 수 있다.

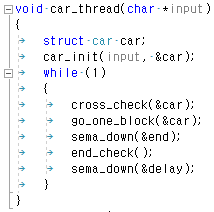
|  |
| --- |
| **3. Crossroads**  **A. 구현** |

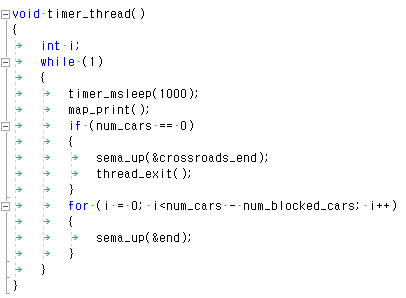
먼저 car 구조체를 정의하였다. \*path는 mapdata.c에 path 배열에 저장되어 있는 값을 가져와서 저장시키기 위한 것이고, progress는 path 배열 위에 car가 현재 있는 위치를 특정하기 위한 변수이다.



그리고나서 aAC:bBA의 형태로 입력 받은 명령어 줄을 parsing해서 car\_thread를 생성한 뒤에 입력 받은 명령어를 car이름, 출발점, 도착점으로 나눠서 car 구조체에 넣고 이를 가지고 car를 map 위에서 이동시킨다.

그리고 나서 car를 이동시키는데, 이동시키기 전에 다음에 가야할 칸이 출발점부터 삼거리 사이라면 그 칸에 갈 수 있는지 체크하기위해 A1, A2, B1, B2, C1, C2 세마포어를 선언하여 가기전에 각 세마포어에 대해 sema\_down()을 하며 기다리고 비어있다면 앞으로 간다. 그리고 그 칸을 나갈 때 sema\_up()을 하여 다른 car가 올 수 있도록 해준다. 그리고 다음에 가야할 칸이 삼거리 진입 칸 이라면 car가 삼거리에서 가야할 경로를 가려고 하는 car가 있는지 cross1, cross2, cross3, cross4를 sema\_down()을 하여 기다리고 가야할 경로가 비어있다면 진입한다. 그리고 이 칸들을 지날 때마다 sema\_up()을 하여 다른 car가 진입할 수 있도록 해준다. Car가 한칸 가고나면 end라는 세마포어를 기다리고, 모든 car가 끝난다면 이를 체크해서 delay 세마포어를 up시켜줘서 car\_thread가 다시 돌 수 있도록 세팅해준다. 그리고 car\_thread가 단위 시간(1초)마다 한 칸을 가게 하기 위해서 timer\_thread에서 단위 시간이 지날 때 마다 단위 시간 동안 한 칸 이동한 car의 수만큼 end 세마포어를 up 시켜준다.

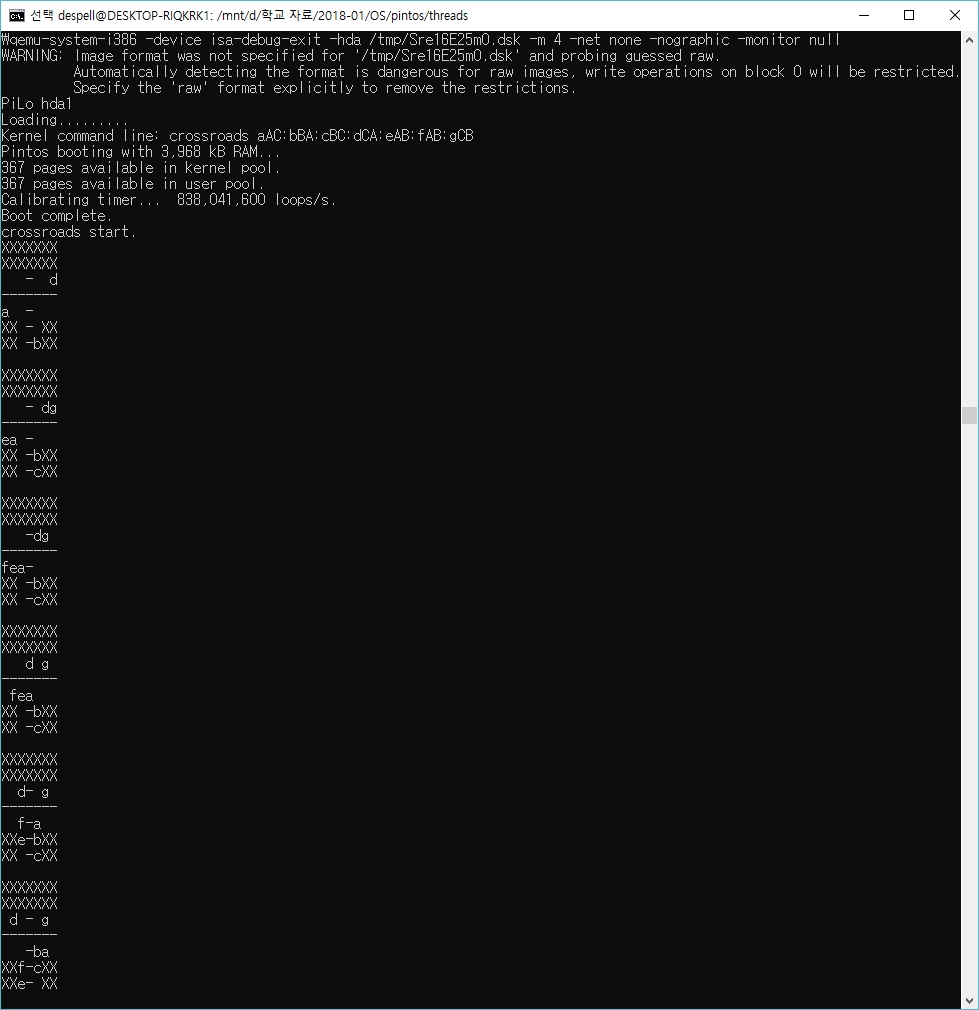
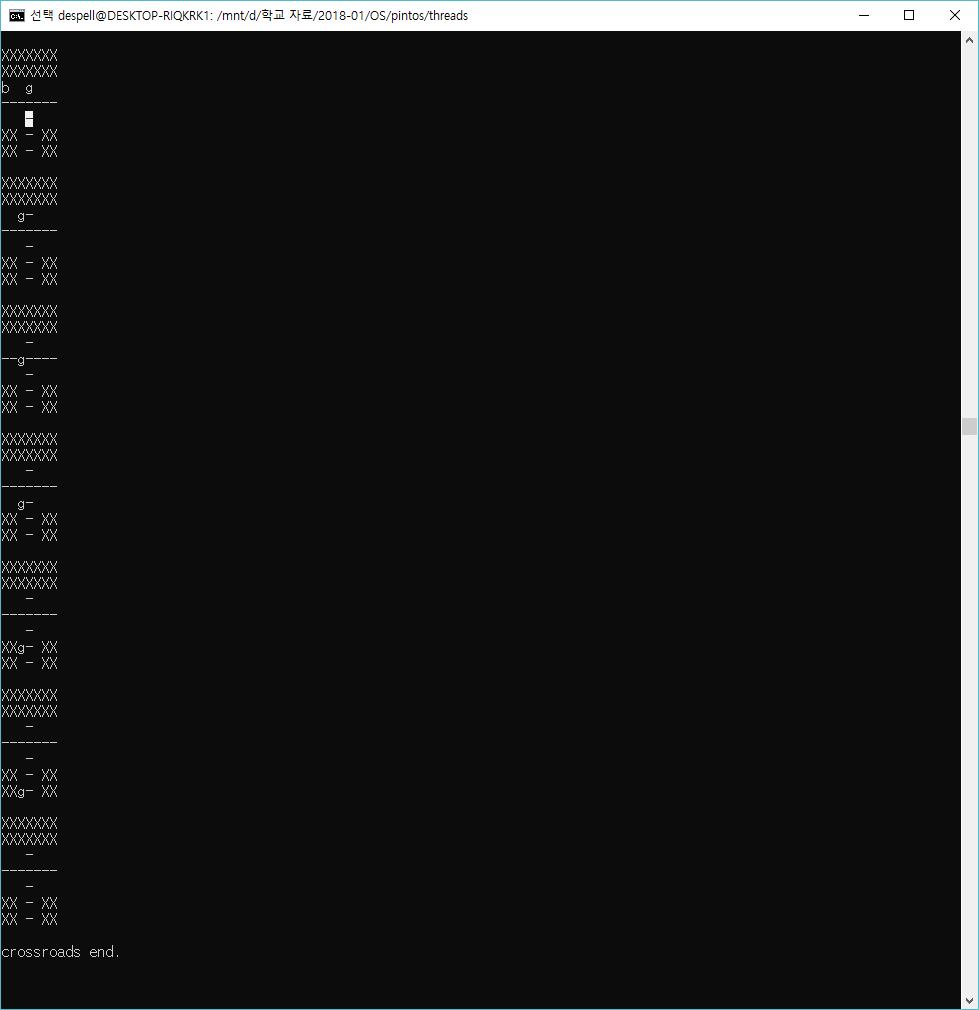
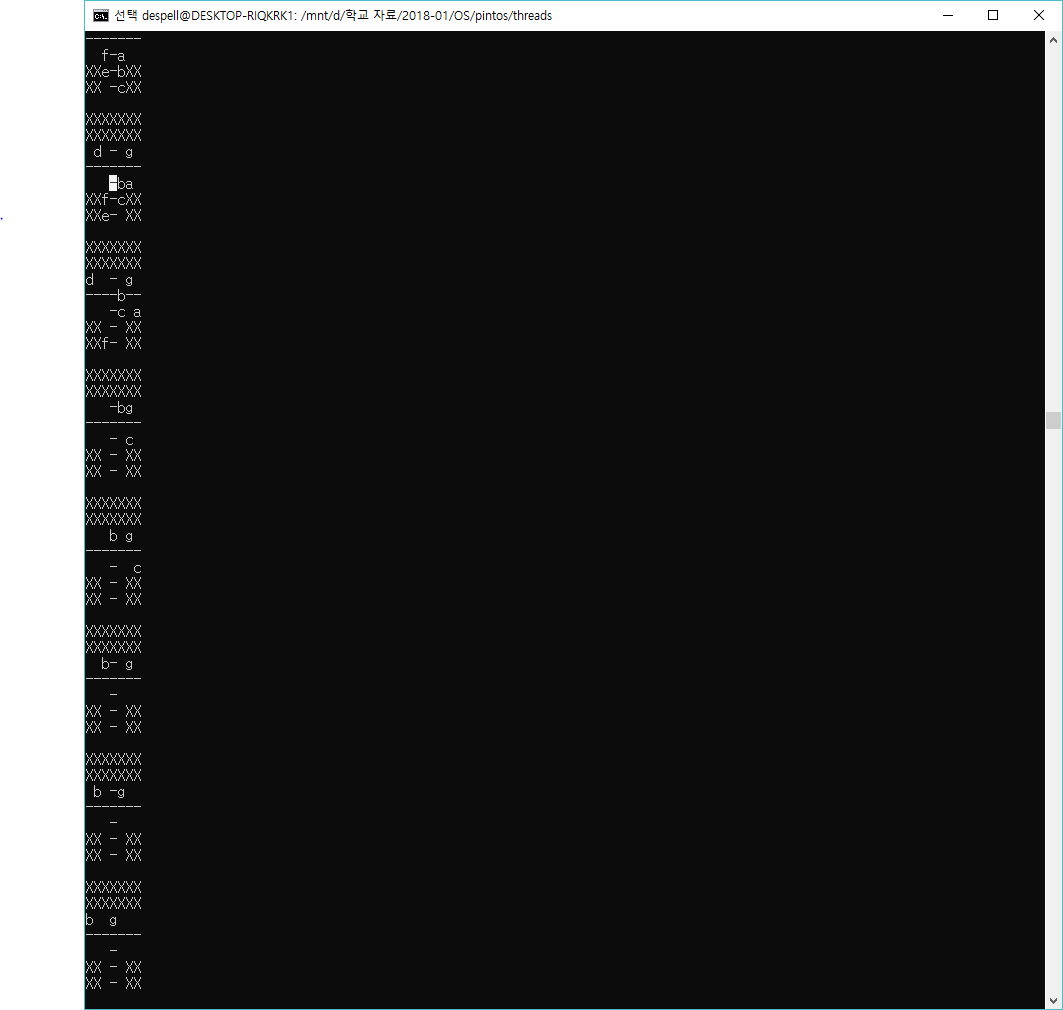




car\_thread와 timer\_thread는 위와 같이 구현되어 있다. go\_one\_block에서 car가 도착하면 num\_cars를 줄이고 thread\_exit()를 실행해 thread를 종료시킨다. 그리고 timer\_thread에서 모든 차가 도착하면 sema\_up(&crossroads\_end)을 해서 main thread가 종료 될 수 있게 하고, timer\_thread를 종료시킨다.

|  |
| --- |
| **B. 테스트 결과** |

테스트 명령어 줄은 제공된 동영상과 같은 것을 사용하였습니다.



* 🡪

위에서 아래, 왼쪽에서 오른쪽으로 실행된 순서이다. 동영상과 약간 다른데, cross 세마포어를 car가 도착하고 전부다 up하는 것이 아니라 cross 세마포어가 설정된 위치를 지나가면 그 세마포어씩 up하게해서 이런 차이가 발생했다. 현실적으로 직진신호가 있는데 우회전이 안되면 이상할 것 같아서 이렇게 구현하였다.