REPORT

Operating System

* MessagePassing & Crossroad Problem

20133950

공찬형

Index

1. MessagePassing
   1. 구현
   2. 테스트
   3. 참조
2. Crossroads Problem
   1. 구현
   2. 동기화 방법
   3. 결과
   4. 참조
3. 추가 코드 일람표
4. MessagePassing
   1. 구현

쓰레드 혹은 프로세스 간의 동기화 또는 통신을 위한 방법으로써 고안된 MessagePassing기능 중에서, Blocking Receive와 Blocking Send를 구현하기 위하여 Semaphore의 개념을 이용하였습니다. Message를 이용한 상호 배제, 즉 Semaphore 와 같은 기능을 구현하는 것과, Semaphore를 이용하여 MessagePassing을 구현하는 것 모두 동일한 기능을 함으로, 이미 pintOS 상에서 구현되어있는 Semaphore를 이용하여 MessagePassing 기능을 구현하였습니다.

-Mailbox[[1]](#footnote-1) : 여러 개의 서로 소통해야 할 프로세스 혹은 쓰레드 간의 간접 주소지정 방식의 패싱을 이용하기 위한 구조체입니다. 이 Mailbox 구조체는, 그 안에 멤버로 메시지 큐의 Inqueue를 위한 Head, Dequeue를 위한 Tail의 포인터, 같은 Mailbox를 공유하는 쓰레드간의 상호 배제를 위한 Semaphore가 정의되어 있습니다.

-Message\_queue : 메시지의 내용을 멤버로 하는 LinkedList 구조체입니다. LinkedList 구조를 위한 next, prev 그리고 실제 메시지를 멤버로 가집니다.

-Message : 실제 메시지 구조체입니다. Windows나 다른 운영체제에 구현되어 있는 메시지의 구조에서, 실제 메시지 패싱 테스트를 위하여 그 메시지를 보내는 쓰레드의 이름을 멤버로 구현하였습니다.

-Semaphore : 같은 메일박스를 공유하는 쓰레드 간의 상호배제를 구현하기 위하여, 최초 쓰레드를 생성한 메인 쓰레드에서 “Idle” 이라는 내용을 담은 기본 메시지를 메일박스에 보냅니다. 생성된 테스트 쓰레드들은, 경쟁 상태에서[[2]](#footnote-2) 메일박스에 접근하여 메시지를 받음과 동시에, semaphore를 감소시켜 메일박스에 접근하여 메시지를 받으려는 다른 쓰레드를 블락시킵니다. 이후 메시지를 성공적으로 수신하고 임계 영역에서의 코드 수행을 마치면, Mailbox에 자신의 쓰레드 이름을 입력하여 메시지를 송신합니다. 송신과 동시에 semaphore에 signal을 보내 다른 쓰레드가 자신이 보낸 메시지를 확인할수 있도록 합니다. 임계 영역에는, Mailbox에서 메시지를 수신한 쓰레드만이 들어갈 수 있기 때문에, 구현된 메시지 패싱은 상호 배제와 긴밀한 동기화를 보장합니다.

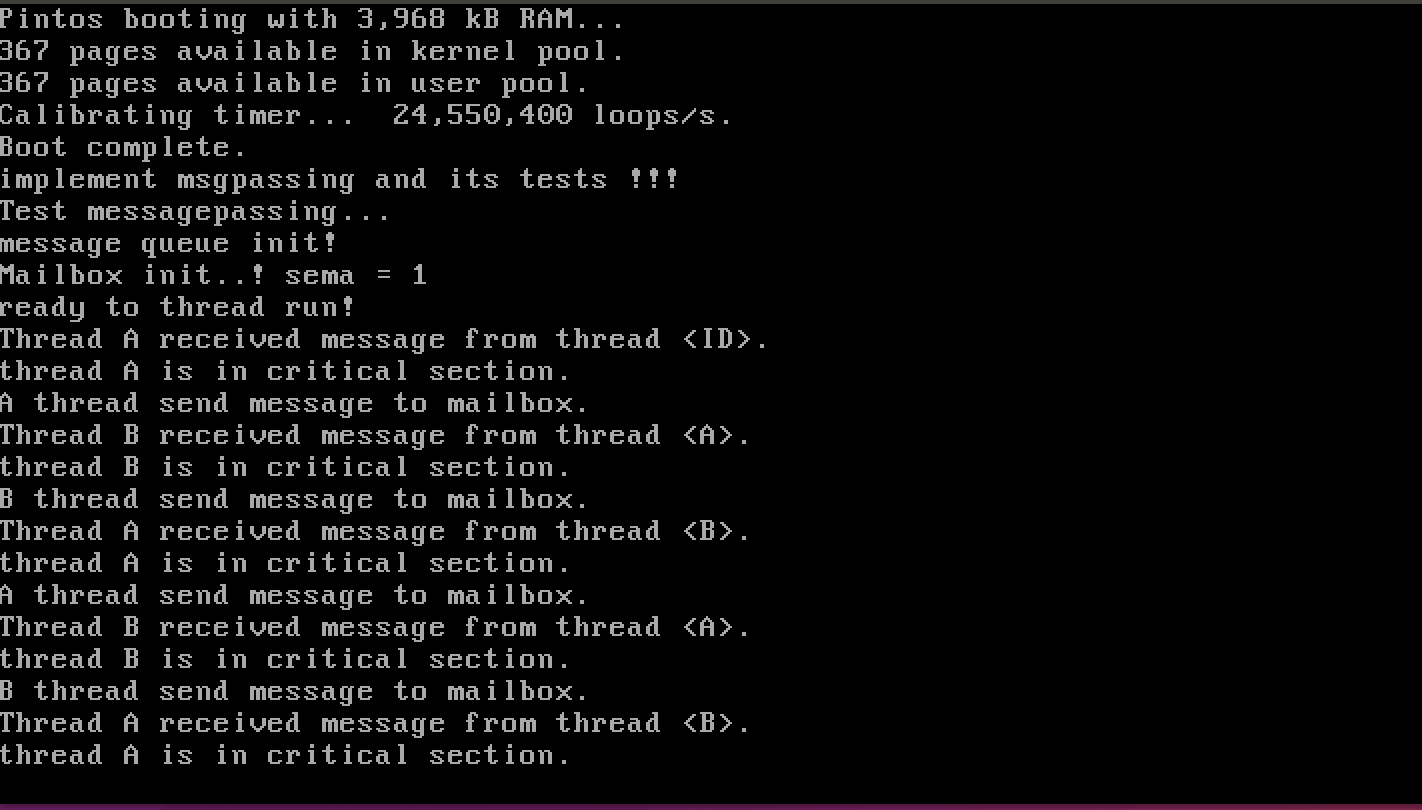
<그림 1>

* 1. 테스트

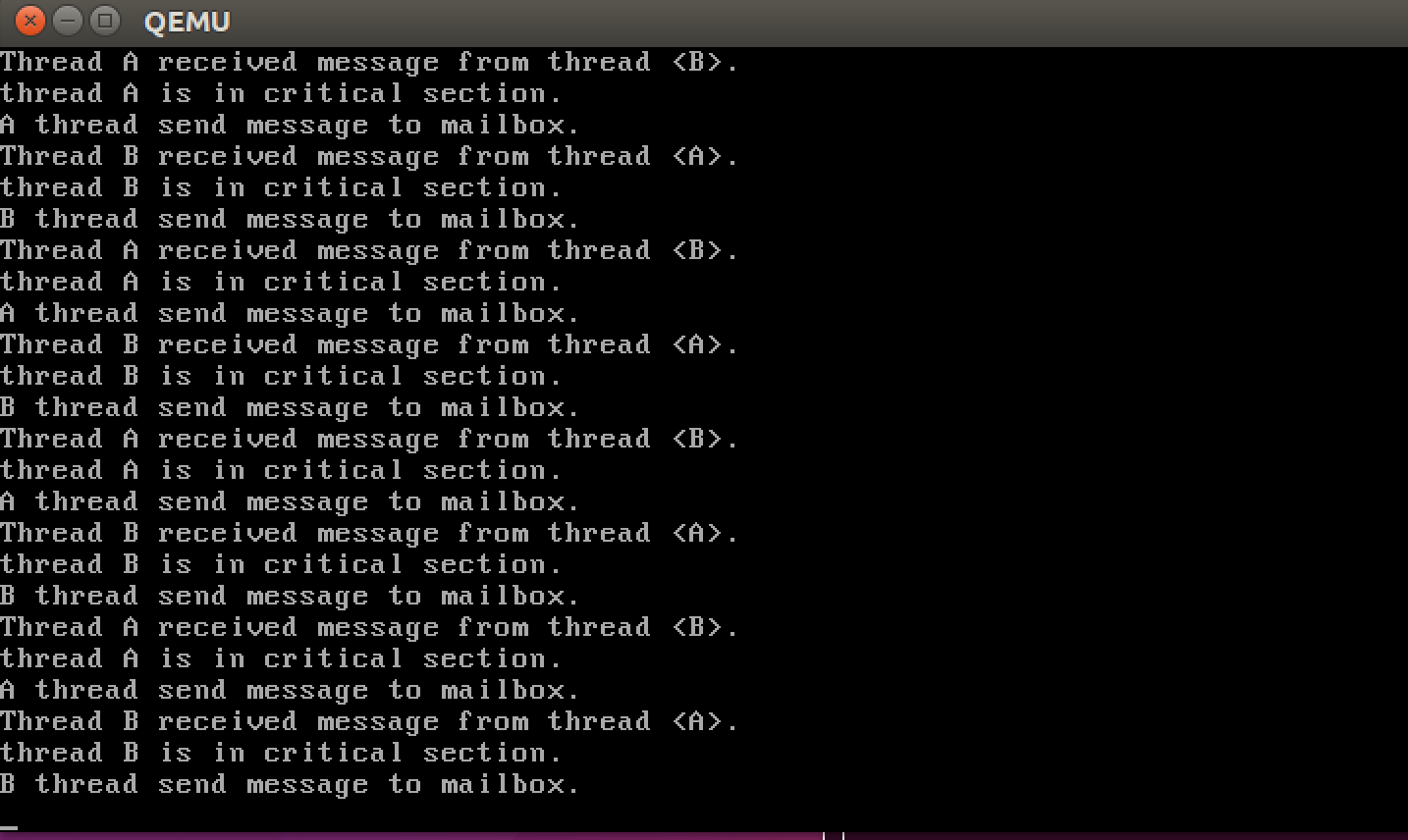
구현된 Blocking Receive와 Blocking Send를 테스트 하기 위하여, 서로의 쓰레드 이름을 송수신 하는 가상의 쓰레드를 생성하였습니다. 이 두 쓰레드(A,B)는 매개변수로 Mailbox를 가지며, 같은 Mailbox를 공유함으로써 소통합니다. 쓰레드 진행 순서는 다음과 같습니다.

1. Thread A, B 생성. 경쟁상태에서 Mailbox에서 “Idle\_message”를 수신하려고 함.
2. Idle\_message를 수신한 쓰레드는 메시지의 송신자를 출력하고 timer\_sleep을 호출하여 임계영역 내부에서 cpu 양보.
3. 이어서 Cpu를 점유한 쓰레드는 mailbox의 semaphore가 양수가 아니므로, 대기열 큐로 진입.
4. 임계영역의 code 수행이 끝나며, 자신의 이름을 입력값으로 한 메시지를 mailbox에 송신. 동시에 semaphore에 signal을 보내어 다른 쓰레드가 실행될수 있도록 함.
5. 다른 쓰레드에서 signal을 받은 쓰레드는 mailbox에서 이전 쓰레드가 보낸 메시지를 수신하고, 그 송신자의 이름을 출력. 2부터 반복.

-테스트 결과



-최초 ID 쓰레드에서 -> A -> B -> A -> B 순으로 실행되는 화면.



테스트를 위하여 임의 설정한 횟수만큼 반복후 종료되는 화면.

* 1. 참조

메시지 패싱 기능 구현을 위하여, synch.c와 synch.h에 다음과 같은 함수를 추가하였습니다. pintOS 실행시 실행되는 messagepassing.c 파일은 synch.c의 message\_test 함수를 호출하며 시작됩니다. 자세한 함수 설명은 문서 하단 표를 참고하시기 바랍니다.

void init\_message\_queue(struct mailbox \*);

void mailbox\_init (struct mailbox \*, unsigned value);

void inqueue (struct message\_queue \*, struct message\_queue \*);

struct message\_queue \* dequeue (struct message\_queue \*, struct message\_queue \*);

void blocking\_send (struct mailbox \*, struct message\_queue \*);

struct message\_queue \* blocking\_receive (struct mailbox \*, struct message\_queue \*);

void message\_test (void);

1. Crossroads Problem
   1. 구현

입력값에 따른 쓰레드 생성을 위하여, Crossroads 함수 실행과 입력되는 argv[1]의 값을 토큰을 이용하여 개별적으로 분리하였습니다. 이 분리된 string은 각각 자동차의 이름, 출 발지, 목적지를 나타냅니다. 자동차의 이동을 위한 쓰레드는 이 세가지 정보를 매개변수 로 생성되어, 개별적으로 진행됩니다. 개별적인 자동차들은 다음과 같은 조건을 만족해야 만 진행될수 있습니다.

* + 1. 가야할 곳을 다른 자동차가 선점하고 있지 않을 것.
    2. 교차로 진행시 다른 자동차의 충돌을 야기하지 않을 것.

다음과 같은 조건을 만족한 자동차는 process\_status의 변수를 증가시킴으로써 전진합니다. Process\_status는 최초 0으로 초기화되며, 해당 자동차가 다음에 가야할 경로상의 진행정도를 나타냅니다. 만약 위의 조건 1을 만족하지 못할 때에는, process\_status가 증가하지 않아, 쓰레드는 실제 동작하였으나, 자동차의 위치는 변화하지 않습니다. 또한 조건 2를 만족하지 못할 경우에는 semaphore에 의해 쓰레드의 진행 자체가 block됨으로, process\_status가 증가하지 않습니다. Process\_status를 이용한 쓰레드의 진행은 다음과 같습니다.

* + - 1. 출발지와 목적지에 따른 조건문 탐색.
      2. 교차로 진입시, 거쳐가야 할 교차로 내부의 외곽부분에 대한 semaphore 획득. Semaphore가 선점되어 있을시, 해당 쓰레드는 semaphore 획득 대기.
      3. 교차로를 진입하였거나, 교차로가 아닌 곳에서의 진행이라면, set\_move 함수를 호출하여 지도상에서 자신의 위치 갱신. 위치 갱신이 성공적으로 이루어졌다면, process\_status를 증가. 이미 선점되어 진행하지 못하였다면, process\_status는 유지. 종착지에 다다랐다면, process\_status를 최대값으로 증가.
      4. 진행할 수 있는 모든 쓰레드가 동작했다면, 커널에 갱신된 지도를 출력.
      5. Process\_status가 최대값(종착지)이라면, 루프 탈출 이후 쓰레드 종료. 아직 도착하지 못했다면, 1부터 다시 실행.
  1. 동기화 방법

쓰레드, 즉 자동차들의 순차적인 진행과 한 쓰레드의 cpu독점 방지를 위하여, 각각 생성된 쓰레드는 같은 우선순위를 지니며, 1번의 루프, 즉 한번의 진행이 끝나면 timer\_sleep함수를 호출하여 cpu를 다음 쓰레드에게 양보합니다.

쓰레드가 교차로에 들어갔을 때, 충돌 위험이 있는 쓰레드를 블락시키거나, 혹은 이미 진행중인 쓰레드를 인식하고, 자신을 블락시키기 위하여 교차로의 각 4개의 외곽부분을 semaphore로 정의하였습니다.[[3]](#footnote-3)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Control[0] |  | Control[1] |  |
| A |  |  |  | C |
|  | Control[2] |  | Control[3] |  |
|  |  | B |  |  |

<그림 2>

예를 들어, A에서 B로 진행하는 쓰레드는 control[0] 만을, B에서 A로 진행하는 쓰레드는 control[0], control[1], control[3]의 semaphore 획득을 필요로 합니다. B에서 A로 진행하는 쓰레드는, 현재 process\_status의 값이 2(교차로 진입)일 때, semaphore의 값을 확인하고, 교차로 진행중일 때, process\_status의 값이 7(교차로 탈출)일 때, 가지고 있던 semaphore들에게 signal을 보내 다른 쓰레드가 진행할 수 있도록 합니다.

이때, 만약 B에서 A로 가는 쓰레드가 control[0], control[3]을 소유하고 control[1]을 대기하고, C에서 A로 가는 쓰레드가 control[1]을 소유하고 control[0]을 대기하는 상황에서 발생하는 deadlock 문제를 해결하기 위하여, semaphore는 작은 숫자를 우선하여 획득을 시도한다는 규칙을 적용하였습니다.[[4]](#footnote-4)

실제 현실의 교차로와는 달리, A에서 B로 진행하는 두개의 쓰레드가, 교차로에 연이어서 진입하여 진입할 수 는 없습니다. 이는 과제 문서 하단에 첨부된 실행 예제에서도 동일하므로, 이를 구현하지는 않았습니다.

마지막으로 화면상에 지도를 출력하기 위하여, 자동차를 진행시키는 쓰레드와 별개의 쓰레드를 생성하였습니다. 이 쓰레드는, 모든 쓰레드중 가장 후순위로 생성되어, 지도상 자동차의 위치를 커널상에 출력해줍니다. 이 쓰레드와 다른 쓰레드간의 시간 동기화와, 테스트의 편의성을 위하여, 다른 쓰레드와 같은 우선순위로, 다음과 같은 semaphore와 timer함수를 사용하였습니다.

|  |  |
| --- | --- |
| void print\_thread() | void car\_thread() |
| while : number of car != 0  sema\_wait(print)  //Print map data  timer\_mdelay(1000)  sema\_signal(print)  timer\_sleep(1) | while : it’s procedure is done  sema\_wait(print)  sema\_signal(print)  //Do car procedure  timer\_sleep(1) |

Print\_mapdata 함수는 지도를 화면에 출력한 후, print에 대한 semaphore를 획득하여, 다른 car 쓰레드들이 진행하지 못하도록 막습니다. 이후 일정시간(1000ms = 1 sec)이 지난뒤, print semaphore에 signal을 보내어 다른 쓰레드가 진행되도록 한후, timer\_sleep 함수를 호출하여 다른 쓰레드에게 cpu를 양보합니다.

Car\_thread 함수는 print\_thread 쓰레드에게 print semaphore를 획득한 후, 다음 쓰레드에게 timer\_sleep를 호출함으로써 cpu를 양보합니다.

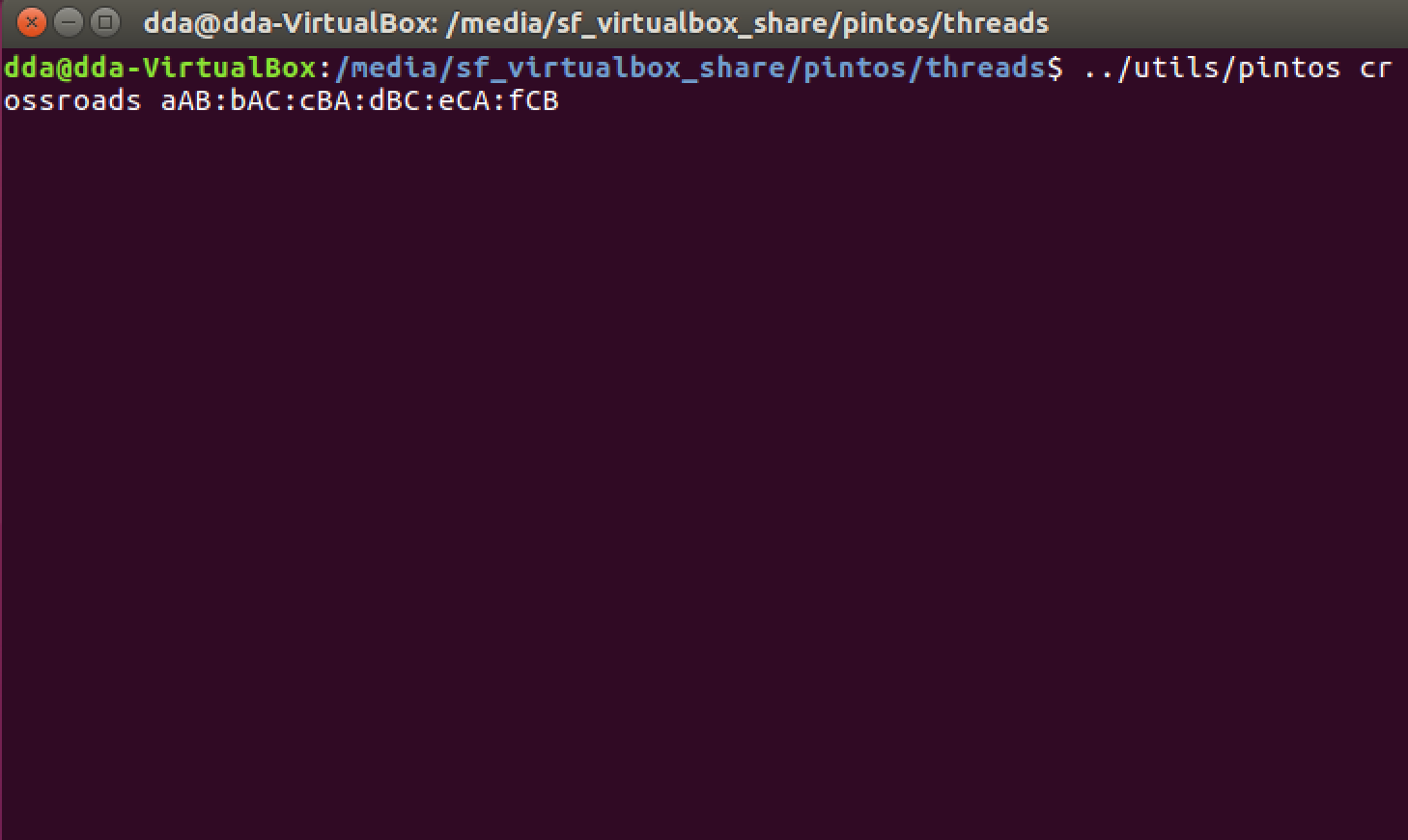
움직일수 있는 모든 자동차가 움직이는 것을 확인한 후, 맵 상에 지도를 출력하기 위하여, print semaphore를 이용하였습니다. Print 쓰레드는, 먼저 생성된 자동차 쓰레드 후순위로 배정되었기 때문에, 최초 실행시에는 모든 자동차의 이동 이후에 실행됩니다.[[5]](#footnote-5) 쓰레드는 실행 중에, 다른 쓰레드의 진입을 방지하기 위하여 print semaphore를 감소시키고, 일정 시간동안 busy-wating을 하게 되는데, 이 semaphore의 waiter 관리 규칙 또한, FIFO 정책을 따르므로, semaphore의 대기자를 언블락 시키는 순서를 이용하여, print 쓰레드 이후에 자동차 이동을 동기화 하였습니다. 실행 예제는 다음과 같습니다.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Tick | 1 | 2 | 3 | 4 | | 5 | 6 | 7 | 1000ms  +1 | 1000ms  +2 |
| Thread | A | B | C | Print  wait | Call  delay | A  wait | B  wait | C  wait | Print  signal | A |
| Tick | 1000ms  +3 | 1000ms  +4 | 1000ms  +5 | | 1000ms  +6 | 1000ms  +7 | 1000ms  +8 | 2000ms+1 | 2000ms  +2 | 2000ms  +3 |
| Thread | B | C | Print  wait | Call  delay | A  wait | B  wait | C  wait | Print  signal | A | B |

위의 표에서 볼 수 있듯, 최초 A,B,C 쓰레드 실행 이후, print 쓰레드는 print semaphore 를 획득하여 1000ms 만큼 대기합니다. Print 쓰레드 대기 도중, cpu를 점유한 A,B,C 쓰레 드는 print semaphore를 획득하지 못함으로, semaphore의 대기자 리스트에 들어갑니다. 이후 대기가 끝나면, semaphore에 signal을 보냄으로써, 대기자 리스트의 가장 선두로 나올 쓰레드가 언블락 되도록 하기 때문에, 쓰레드 간의 동기화가 보장된다는 것을 알수 있습니다.

* 1. 테스트

다음과 같은 쓰레드를 생성하였다.



실행단위(1초) 에 대응하는 결과이다.

a(A -> B), b(A -> C), c(B -> A), d(B -> C), e(C -> A), f(C -> B)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 교차로 진입 |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  | | |

이 외에 결과는 다음과 같다.

* + 1. aAA 와 같은 쓰레드는 생성과 동시에 종료.
    2. aAB:bAB:cAB 와 같은 같은 곳을 향하는 쓰레드는 같은 방향 진입임에도 교차로에 의해 상호배제. (B. 구현 참조)
  1. 참조

교차로 문제를 해결하기 위하여, crossroads.c 와 crossroads.h에 다음과 같은 함수를 추가하였습니다. 자세한 기능 구현은 문서 하단 일람표를 확인해 주시기 바랍니다.

Void print\_mapdata(void)

Int set\_move(int, int, int, char)

Bool is\_occupied(int, int)

Static Void print\_thread(void)

Static Void car\_thread(void)

1. 추가 코드 일람표

<synch.c, synch.h>

|  |  |
| --- | --- |
| Void  Init\_message\_queue | Initiate message\_queue in struct mailbox. |
| Void  Mailbox\_init | Initiate mailbox and semaphore of mailbox.  This calls init\_message\_queue. |
| Void  inqueue | Inqueue message\_queue in queue of mailbox. |
| Struct message\_queue \*  Dequeue | Dequeue message\_queue in queue of mailbox and return dequeued message\_queue. |
| Void  Blocking\_send | Inqueue message\_queue to mailbox and signal to semaphore of mailbox. |
| Struct message\_queue \*  Blocking\_receive | Dequeue message\_queue from mailbox and wait for signal of semaphore from mailbox. Return message\_queue to use in threads that called this function. |
| Void  Message\_test | Main test function. It calls mailbox\_init and send “Idle message” to mailbox. And it makes threads(message\_test\_helper) |
| Static void  Message\_test\_helper | Used for main function. This function allocates memory for message, and calls blocking\_receive, timer\_sleep for test and blocking\_send sequentially. |

<crossroads.c, crossroads.h>

|  |  |
| --- | --- |
| Void  Run\_crossroads | Main function. It Initiates all static variable and semaphore used in program, tokenizes and save each data into car\_data. And makes threads based on each car\_data and also makes thread for print map on kernel. |
| Void  Print\_mapdata | Function for thread print\_thread. It print fixable map called map\_draw. |
| Int  Set\_move | Renewal car\_location that is integer 7x7 array insisted of 0,1 exisitence of car in map and map\_draw visualized of map and car name. It checks condition of car\_location so that increases process\_status or not. If procedure is done, return maximum value of process\_status to break outer threads, otherwise return own value. |
| Int  Is\_occupied | Return Boolean which means position to go is already occupied or not. |
| Static void  Print\_thread | Function called by thread for printf .After call print\_mapdata, it down semaphore and busy\_wait for test, and fixed time has gone, it calls timer\_sleep to yield cpu other thread wait for semaphore(print). |
| Static void  Car\_thread | Based on its thread name, consisted of car name, source, and destination, it checks condition of source and destination to check its related semaphore. If next step of process\_status is in critical section, which means crossroads, wait for all semaphore on the route. End of critical section, it signal to semaphore its thread had. Otherwise, or end of critical section, it calls set\_move function to renewal its location, and checks whether finished or not considering value of process\_status. Finally it sleeps for sleeptime to yield cpu to another thread. |

1. 그림 1 참조. [↑](#footnote-ref-1)
2. 구현된 테스트 쓰레드 내에서는 기본적으로 먼저 생성된 쓰레드 순으로 진행되기 때문에, 완전한 경쟁 상태를 가정할 수 없음. [↑](#footnote-ref-2)
3. 그림 2 참조 [↑](#footnote-ref-3)
4. 쓰레드 C->A는 control[0]을 획득하기 전까지 control[1]을 소유할수 없음. [↑](#footnote-ref-4)
5. pintOS scheduling is based on FIFO round robin. [↑](#footnote-ref-5)