Operation System – COSE341

실습 3 과제 레포트 – Page Replacement Algorithm – Assignment 3

고려대학교 컴퓨터학과 2017320108

고재영

개발 환경 : Oracle VM VirtualBox, Linux, Ubuntu 18.04.5, Visual Studio 2019

과제 만기일 : 2022/06/08 10:29 AM

유의 사항 : 총 3가지 버전의 LRU 알고리즘에 대한 원리와 작성한 코드 구현에 관련한 간략한 설명을 작성하였다. 또한, 각 버전별로 Reference String S = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1}에 대하여 page 참조마다 frame이나 reference bit등이 어떤 식으로 변화했는지 알 수 있는 ubuntu 환경에서의 실행 결과에 대한 스크린샷 이미지가 첨부되어 있다.

첫번째 제출 날짜 : 2022.06.05

최종 제출: 2022.06.08 새벽

재제출 사유: 2번째 version에 관한 추가 디테일 추가.

[0] – 과제 설명

3가지 버전의 LRU 알고리즘에 대해 구현 및 시뮬레이션

- 제출하는 코드는 난수 생성을 이용하여 generate_ref_arr() 함수를 통해 reference string이 생성된다.
- 제출하는 코드를 일부 변형하여 Reference String S = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1}에 대한 수행을 알기 위해 결과 이미지만 첨부한다.

[1] – Stack을 이용한 LRU Replacement 시뮬레이션

첫 번째 과제는 Stack을 이용한 LRU 구현이다. 다만, 수업에서 언급된 것처럼, 소프트웨어적으로 구현하려면 doubly linked list로 구현하는 것이 바람직하다. 따라서 필자는 `Node* frame`이란 헤더 노드를 가지는 doubly linked list를 구현했다.

해당 자료구조에 대해 간략히 소개하자면, frame이란 헤더 노드 바로 다음이 Top에서부터 Bottom으로 가는 구조이다. Stack의 기본 원리는 Last-In First-Out 의 LIFO구조를 기반으로 하므로, 구현에 있어서 push()함수 호출 시 새로운 데이터를 top으로 눌러 들어오고, pop_bot()을 통해 가장 맨 아래 bottom에 깔린 데이터에 관련한 삭제를 진행한다. 다른 함수도 물론 구현했지만, 이는 LRU 동작 원리를 설명할 다음 문단에서 설명하겠다.

필자가 구현한 Stack을 이용한 LRU는 다음과 같이 작동한다:

1- Reference string의 데이터를 하나씩 읽으면서, 해당 데이터가 frame에 존재하는지 체크한다. Stack안에 있는지 확인하기 위해, find()함수로 Top에서 Bottom방향으로 순차적으로 탐색하여 이를 구현했다.

2- 만약 해당 데이터가 frame에 존재한다면, page fault는 발생하지 않고, 현재 refer하는 page를 Top으로 갱신시켜야 한다. 따라서 search_pop()으로 해당 page를 탐색하여 pop한 후, 다시 top 위로 push시킨다.

3- 만약 해당 데이터가 frame에 존재하지 않는다면, page fault가 발생한다. 다만, frame에 존재하지 않는 경우는 다음 2가지이기 때문에 분기된다.

3.1- frame이 가득 차있진 않아서 새로 load만 해주면 되는 경우이다. 이 경우 top위치에 push만 진행시킨다.

3.2- frame, 즉 stack이 full인 상태에서 load를 해야하므로, pop_bot()함수를 이용해 bottom에 있는 것을 제거하고, top에 push한다.

```
nuketuna@nuketuna-VirtualBox:~/바탕화면$ ./lru_stack.out
+--<Ref array>--+
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
+- 7 :: Frames :: 7 - - - (page fault)
+- 0 :: Frames :: 0 7 - - (page fault)
+- 1 :: Frames :: 1 0 7 - (page fault)
+- 2 :: Frames :: 2 1 0 7 (page fault)
+- 0 :: Frames :: 0 2 1 7
+- 3 :: Frames :: 3 0 2 1 (page fault)
+- 0 :: Frames :: 0 3 2 1
+- 4 :: Frames :: 4 0 3 2 (page fault)
+- 2 :: Frames :: 2 4 0 3
+- 3 :: Frames :: 3 2 4 0
+- 0 :: Frames :: 0 3 2 4
+- 3 :: Frames :: 3 0 2 4
+- 2 :: Frames :: 2 3 0 4
+- 1 :: Frames :: 1 2 3 0 (page fault)
+- 2 :: Frames :: 2 1 3 0
+- 0 :: Frames :: 0 2 1 3
+- 1 :: Frames :: 1 0 2 3
+- 7 :: Frames :: 7 1 0 2 (page fault)
+- 0 :: Frames :: 0 7 1 2
+- 1 :: Frames :: 1 0 7 2
Total page faults :: 8
nuketuna@nuketuna-VirtualBox:~/바탕화면$
```

<주어진 Reference String S에 대해, 페이지 참조마다 Stack 내용이 어떻게 변하는지 표로 작성>

위의 결과물을 보면 알 수 있듯이, Frames :: (page) (page) (page) (page) 꼴로 표현되는데, 가장 왼쪽이 Top, 오른쪽이 Bottom을 나타낸다. Page에 대한 참조가 일어날 때마다, Top부분에 위치한 page가 갱신됨을 알 수 있다.

혹여나, 과제 내용은 표로 작성하도록 명시되어 있기 때문에 다음과 같은 표를 첨부한다.

Reference String	**7**	**0**	**1**	**2**	**0**	**3**	**0**	**4**	**2**	**3**	**0**	**3**	**2**	**1**	**2**	**0**	**1**	**7**	**0**	**1**
Stack				2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
			1	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0
		0	0	0	1	2	2	3	0	4	2	2	0	3	3	1	2	0	1	7
	7	7	7	7	7	1	1	2	3	0	4	4	4	0	0	3	3	2	2	2
Page fault																				

[2] – Clock Algorithm 시뮬레이션

두 번째 과제는 reference bits에 관해 Clock이 회전하는 양태의 원리를 기반으로 한 clock algorithm LRU의 구현이다. 필자는 reference bit의 clock의 구현을 위해 doubly linked list 자료구조를 이용하여, 이는 front와 rear란 빈 노드를 양 극단으로 가지며, 그 사이에 frame의 개수만큼의 빈 노드들을 집어넣어 구현했다. frame은 일차원 배열로 관리하며, reference bit clock과 다른 자료구조로 이용되는 만큼 동작이 atomic해야 한다. 그리고 clockpoint란 노드 포인터는 현재 clock이 가리키는 노드를 나타낸다.

필자가 구현한 Clock algorithm을 이용한 LRU는 다음과 같이 동작한다:

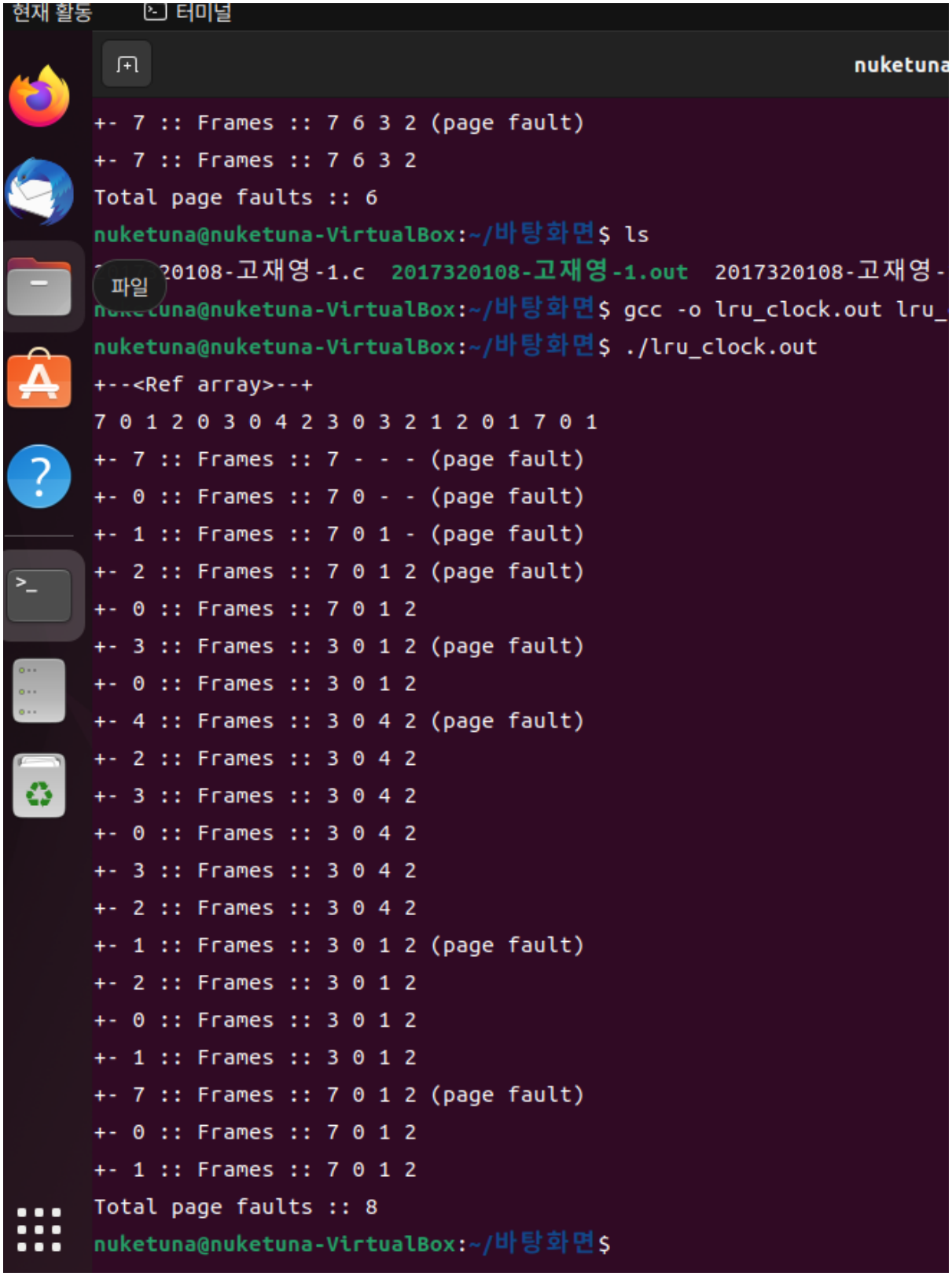
1- Reference string의 데이터를 하나씩 읽으면서, 해당 데이터가 frame에 존재하는지 일차원 배열로 관리되는 frame의 리스트를 보며 체크한다.

2- 만약 해당 데이터가 frame에 존재한다면, page fault는 발생하지 않는다. 해당 refer하는 page를 clock에서 찾아, reference bit를 1로 갱신시킨다. 중요한 건, clockpoint는 움직이지 않는다.

3- 만약 해당 데이터가 frame에 존재하지 않는다면, page fault가 발생한다. 중요한 건, clockpoint는 page fault가 발생한 지금 시점에만 회전을 시작한다는 점이다.

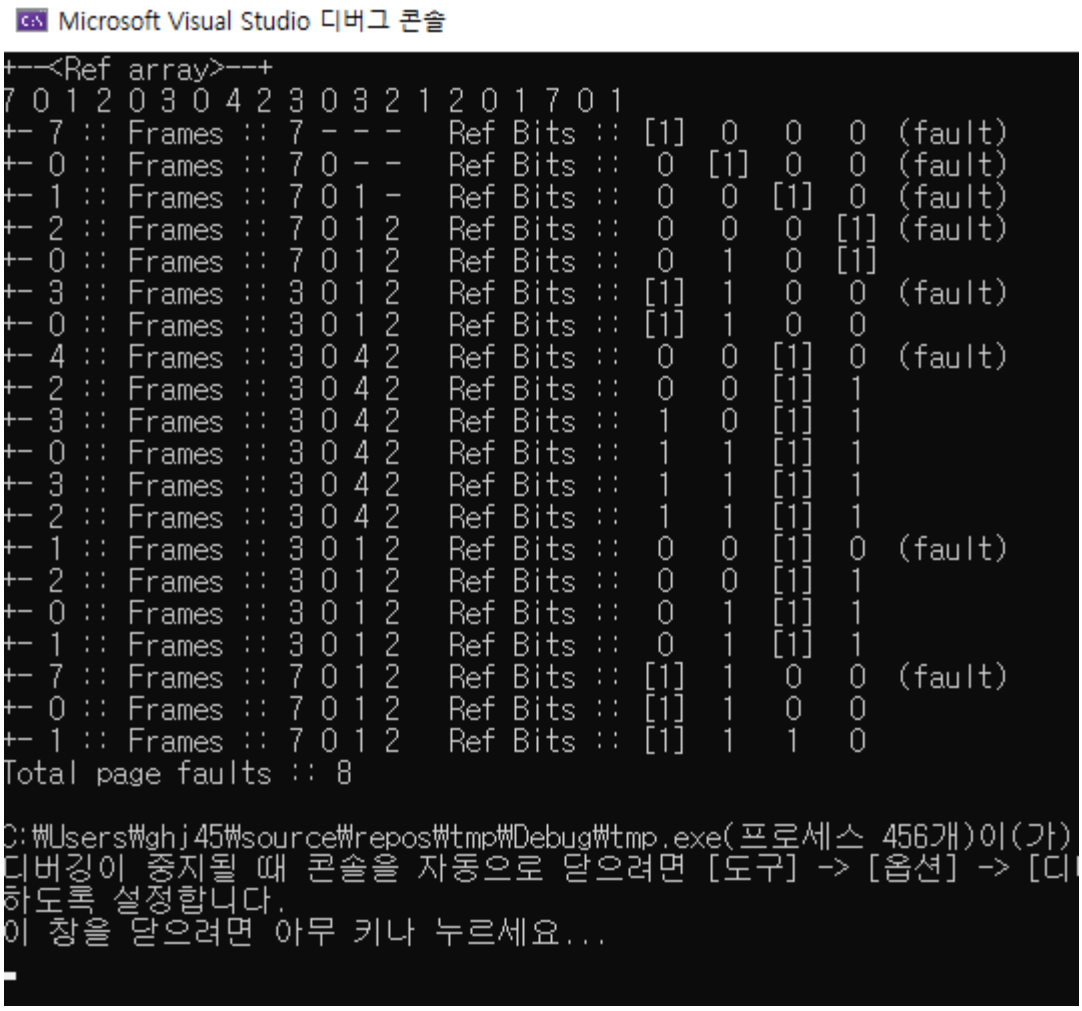
3.1- Clockpoint가 가리킨 노드의 reference bit가 1이라면, 0으로 갱신시켜준 다음 탐색을 계속한다. (코드에서 clockpoint가 rear를 가리킬 경우, front가 다음으로 가리키는 노드로 이동하기 위해 if문이 쓰였다.)

3.1- Clockpoint가 가리킨 노드의 reference bit가 0이라면, 해당 노드가 의미하는 frame을 victim으로 하여 replacement 작업을 진행한다.



위 이미지처럼, frame table에서 page fault가 발생하여 page-out되어야 하는 victim의 자리에 replace되어, page-in이 그 자리에 일어남을 알 수 있다.

아래의 이미지는, 혹시나 2번째 version에서 reference bit의 변화와 clock이 가리키고 있는 bit을 시각화하여 나타내는 것이 좋겠다고 판단하여 (재제출 사유) 이 부분을 추가적으로 구현한 결과이다. reference bits 중 대괄호 [] 로 묶여 있는 reference bits가 현재 clock이 가리키고 있는 page임을 알 수 있다.



(추신) - 추가적인 설명을 위한 이미지이기 때문에, 우분투 가상환경이 아닌 visual studio에서 바로 실행한 결과이고, 단지 시각화만 더 구현한 부분이기 때문에, 최종 제출 소스코드 및 .out 실행파일은 변경하지 않았음을 밝힌다.

[3] – Additional Reference Bits Algorithm 시뮬레이션

3번째 과제는 Additional reference bits를 이용한 LRU 구현이다. 해당하는 algorithm은, 각 page에 8-bit에 해당하는 reference bit을 각각 할당하여 reference가 일어나는 page는 앞의 bit를 1로 갱신시켜주고, 일정 시간 간격마다 (Timer interrupt) 이 reference bit가 오른쪽으로 한 칸씩 shift하며, page fault 발생가 발생한다면 frame에 있는 page 중 reference bit이 가장 작은 page를 victim으로 삼는 원리로 구현한다.

이런 8-bit reference bit 구현을 위해, 자료형 크기가 1byte인 unsigned char 자료형으로 구현했다. Frame이 일차원배열로 관리되기 때문에, page가 frame에의 존재 여부 정보를 담고 있는 ExistArr란 일차원 배열도 이용하였다.

필자가 구현한 Additional reference bit alorithm기반의 LRU는 다음과 같이 동작한다:

1- Timer interrupt의 구현을 위해, 여기선 매번 reference string을 읽어야 하는 시점이 될 때마다 timerInterrupt()함수를 통해 모든 reference bits를 오른쪽으로 한 칸씩 shift시킨다.

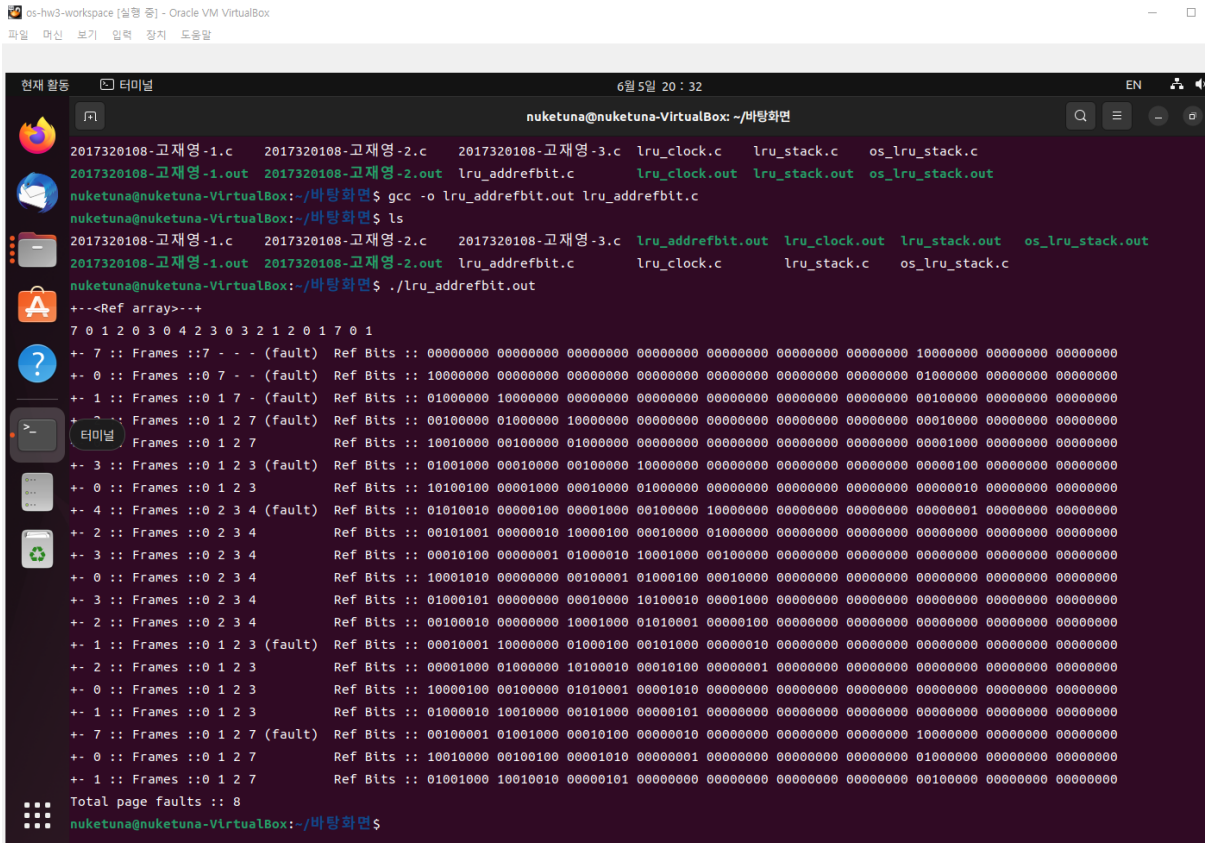
2- Reference string의 데이터를 하나씩 읽으면서, 해당 데이터가 frame에 존재하는지 ExistArr를 보며 존재 여부를 체크한다.

3- 만약 해당 데이터가 frame에 존재한다면, page fault는 발생하지 않고, 해당 page의 reference bit의 최상단 bit를 1로 갱신하기 위해, 128을 더해준다 (binary로 1000 0000)

4- 만약 해당 데이터가 frame에 존재하지 않는다면, page fault가 발생한다. 다만, 다음 2가지 경우로 분기된다.

4.1- frame이 full 상태가 아니라서 참조하는 page를 그냥 load하면 될 경우, 해당 page의 reference bit의 최상단 bit를 1로 갱신하기 위해, 128을 더해준다 (binary로 1000 0000)

4.2- frame이 full인 상태일 경우, frame에 존재하는 page들 중에서, reference bits가 가장 작은 값을 갖는 page를 victim으로 선정하여, replacement가 진행된다.



위의 이미지는 Reference String S = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1} 일 때, page reference 마다 frame과 reference bits가 어떻게 변화하는지에 대한 전체 이미지이다. 좀 더 보기 쉽게 하기 위해, 다음 이미지는 일부만 0번 reference bits 까지만 발췌한 것이다.

```
nuketuna@nuketuna-VirtualBox:~/바탕화면$ ls
2017320108-고재영-1.c    2017320108-고재영-2.c    20173
2017320108-고재영-1.out  2017320108-고재영-2.out  20173
nuketuna@nuketuna-VirtualBox:~/바탕화면$ ./lru_addrefbi
+--<Ref array>--+
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
+- 7 :: Frames ::7 - - - (fault)  Ref Bits :: 00000000
+- 0 :: Frames ::0 7 - - (fault)  Ref Bits :: 10000000
+- 1 :: Frames ::0 1 7 - (fault)  Ref Bits :: 01000000
+- 2 :: Frames ::0 1 2 7 (fault)  Ref Bits :: 00100000
+- 0 :: Frames ::0 1 2 7          Ref Bits :: 10010000
+- 3 :: Frames ::0 1 2 3 (fault)  Ref Bits :: 01001000
+- 0 :: Frames ::0 1 2 3          Ref Bits :: 10100100
+- 4 :: Frames ::0 2 3 4 (fault)  Ref Bits :: 01010010
+- 2 :: Frames ::0 2 3 4          Ref Bits :: 00101001
+- 3 :: Frames ::0 2 3 4          Ref Bits :: 00010100
+- 0 :: Frames ::0 2 3 4          Ref Bits :: 10001010
+- 3 :: Frames ::0 2 3 4          Ref Bits :: 01000101
+- 2 :: Frames ::0 2 3 4          Ref Bits :: 00100010
+- 1 :: Frames ::0 1 2 3 (fault)  Ref Bits :: 00010001
+- 2 :: Frames ::0 1 2 3          Ref Bits :: 00001000
+- 0 :: Frames ::0 1 2 3          Ref Bits :: 10000100
+- 1 :: Frames ::0 1 2 3          Ref Bits :: 01000010
+- 7 :: Frames ::0 1 2 7 (fault)  Ref Bits :: 00100001
+- 0 :: Frames ::0 1 2 7          Ref Bits :: 10010000
+- 1 :: Frames ::0 1 2 7          Ref Bits :: 01001000
Total page faults :: 8
nuketuna@nuketuna-VirtualBox:~/바탕화면$
```

<주어진 S에 대해, 페이지 참조마다 0번 페이지의 Reference Bits이 어떻게 변하는지 표로 작성>

위의 이미지를 보면 알 수 있듯이, 0번 page가 참조될 경우 0번 page의 reference bit의 최상단 bit는 1로 갱신이 된다. 그리고 매번 page reference 마다, 이 8-bit가 오른쪽으로 한 칸씩 shift되고 있음이 자명하다. (이미지로 충분히 표현 가능하여, 별도의 표는 작성하지 않았다.)