

2022-Spring
운영체제

학번	2015150220
이름	이윤수

## [OS Lab 3 Page Replacement Algorithms]

### [공통]

#### - Generate\_ref\_arr() 함수

: <time.h> 헤더파일을 include 해서 srand(), time(), 그리고 rand() 함수를 통해 Reference String 값이 랜덤으로 생성되게 했습니다. 이때 rand() 값을 (page\_max+1)로 나누어 스트링 값이 [0, page\_max] 범위의 정수가 되게 했습니다. 이 함수는 3 가지 알고리즘에서 공통으로 사용됐습니다.

### Assignment 3-1 : Stack 을 이용한 LRU Replacement.

#### - Lru() 함수, \_contains() 함수

: Frames 배열을 stack 으로 구현했습니다. Reference string 에 해당하는 페이지가 frames 에 포함돼 있는지 \_contains 함수로 파악했습니다. 이때 stack 의 top 을 이용해 현재 스택이 empty 인지 여부를 확인했습니다. 페이지가 존재하면 프레임 인덱스를, 존재하지 않으면 -1 이 return 됩니다. 후자의 경우 page\_fault 가 발생한 경우 입니다. 이때 stack 의 top 을 이용해 현재 스택이 full 인지 아닌지 파악합니다.

Full 이면 스택의 모든 원소를 한칸씩 아래로 이동합니다. 이때 스택의 가장 아래 원소가 삭제되는 것으로 LRU page 를 swap out 하는 상황을 구현했습니다. Full 이 아니라면 top 을 +1 해서 빈 공간에 저장합니다.

이외에 page fault 가 발생하지 않아도 스택 내부에서 이동이 필요합니다. 프레임 내에서 가장 최근에 참조된 페이지가 항상 top 에 위치하도록 위치를 조정하는 함수를 for 문을 이용해 순차적으로 이동시키는 방법으로 구현했습니다. 주어진 S 에 대해 프로그램을 실행한 결과와 stack 의 변화는 다음과 같습니다.

```

7 | 7 . . . (fault)
0 | 7 0 . . (fault)
1 | 7 0 1 . (fault)
2 | 7 0 1 2 (fault)
0 | 7 1 2 0
3 | 1 2 0 3 (fault)
0 | 1 2 3 0
4 | 2 3 0 4 (fault)
2 | 3 0 4 2
3 | 0 4 2 3
0 | 4 2 3 0
3 | 4 2 0 3
2 | 4 0 3 2
1 | 0 3 2 1 (fault)
2 | 0 3 1 2
0 | 3 1 2 0
1 | 3 2 0 1
7 | 2 0 1 7 (fault)
0 | 2 1 7 0
1 | 2 7 0 1


```

			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

## Assignment 3-2 : Clock Algorithm.

### - Lru() 함수, \_max()함수

: \_max() 함수로 페이지 번호 최대값을 구한 다음, (page\_max+1)개의 reference bits 를 담을 수 있는 배열인 ref\_bits 를 생성하고 각각의 비트값을 0 으로 초기화했습니다. 이후 페이지 참조가 일어날때, 만약 페이지 폴트가 일어난다면, 우선 프레임에 빈 공간이 있는지를 확인합니다. 빈 공간 이 있다면 거기에 페이지를 삽입합니다. 빈 공간이 없는 경우에는 clock\_head 를 증가시키면서 프레임 내부에 존재하는 페이지들의 reference bit 을 확인합니다. While 문을 통해서 reference bit 값이 0 인 페이지가 나올 때까지 clock\_head 를 증가시키고, 이때 circular queue 방식을 구현하기 위해서 clock\_head 를 (clock\_head+1) % frame\_sz 로 업데이트 했습니다. 이를 통해 [0, frame\_sz] 범위의 정수 값을 원형(시계방향)으로 순환하게 했습니다. Reference bit 가 0 인 페이지가 나오면 해당 페이지가 속한 프레임 공간에 새롭게 참조한 페이지를 대체합니다. Replacement 가 끝난 뒤에는 참조된 페이지의 reference bit 을 1로 바꾸었습니다. 주어진 S 에 대한 실행결과와 lru 함수의 소스 코드는 아래와 같습니다.



```
7 | 7 | . | . | . (fault)
0 | 7 | 0 | . | . (fault)
1 | 7 | 0 | 1 | . (fault)
2 | 7 | 0 | 1 | 2 (fault)
0 | 7 | 0 | 1 | 2
3 | 3 | 0 | 1 | 2 (fault)
0 | 3 | 0 | 1 | 2
4 | 3 | 0 | 4 | 2 (fault)
2 | 3 | 0 | 4 | 2
3 | 3 | 0 | 4 | 2
0 | 3 | 0 | 4 | 2
3 | 3 | 0 | 4 | 2
2 | 3 | 0 | 4 | 2
1 | 3 | 0 | 1 | 2 (fault)
2 | 3 | 0 | 1 | 2
0 | 3 | 0 | 1 | 2
1 | 3 | 0 | 1 | 2
7 | 7 | 0 | 1 | 2 (fault)
0 | 7 | 0 | 1 | 2
1 | 7 | 0 | 1 | 2
```

```
int lru(int* ref_arr, size_t ref_arr_sz, size_t frame_sz) {
    int i, j;
    int page_faults = 0;
    int is_fault, target;
    // Initializing frames
    int* frames = (int*) malloc(sizeof(int) * frame_sz);
    for (i=0; i<frame_sz; i++) frames[i] = EMPTY_FRAME;

    // TODO
    int page_max = _max(ref_arr, ref_arr_sz) + 1;
    int* ref_bits = (int*) malloc(sizeof(int) * page_max);
    int clock_head = 0;
    for (i=0; i<page_max; i++) ref_bits[i] = 0;

    for (i=0; i<ref_arr_sz; i++){
        is_fault = _contains(frames, frame_sz, ref_arr[i]);

        if(is_fault == -1){
            int empty_idx = _contains(frames, frame_sz, EMPTY_FRAME);

            if(empty_idx != -1){
                frames[empty_idx] = ref_arr[i];
            }
            else{
                while(ref_bits[frames[clock_head]]!=0){
                    ref_bits[frames[clock_head]] = 0;
                    clock_head = (clock_head+1) % frame_sz;
                }
                frames[clock_head] = ref_arr[i];
            }
            page_faults++;
        }
        ref_bits[ref_arr[i]] = 1;
    }

    free(frames);
    free(ref_bits);

    return page_faults;
}
```

## Assignment 3-3: Additional Reference Bits Algorithm

### - Lru() 함수

: 3-2 알고리즘과의 차이를 중심으로 설명하겠습니다. 가장 큰 차이는 ref\_bits 를 integer 배열이 아닌 unsigned char(1 바이트, 8 비트) 배열로 구현했다는 점입니다. 또한 페이지를 참조할 때마다 ref\_bits 값이 오른쪽으로 shift 되게 했습니다. 그리고 마지막에 replacement 가 끝난 뒤 참조한 페이지의 reference bit 에 대해서 0x80 즉 1000 0000(2)과 OR 연산을 해 결과적으로 최상위 비트를 1로 만드는 작업을 수행했습니다. 페이지를 교체할 때 ref\_bits 값이 가장 작은 프레임 내부 페이지를 찾았습니다. 이를 위해 초기 Min 값은 8 비트 자료형에서 최대값인 0xFF 로 설정했습니다. 주어진 S 에 대한 실행결과와 0 번 페이지의 Reference Bit 변화는 아래와 같습니다. 실행결과와 3-2 와 동일합니다.

```

7 | 7 . . . (fault)
0 | 7 0 . . (fault)
1 | 7 0 1 . (fault)
2 | 7 0 1 2 (fault)
0 | 7 0 1 2
3 | 3 0 1 2 (fault)
0 | 3 0 1 2
4 | 3 0 4 2 (fault)
2 | 3 0 4 2
3 | 3 0 4 2
0 | 3 0 4 2
3 | 3 0 4 2
2 | 3 0 4 2
1 | 3 0 1 2 (fault)
2 | 3 0 1 2
0 | 3 0 1 2
1 | 3 0 1 2
7 | 7 0 1 2 (fault)
0 | 7 0 1 2
1 | 7 0 1 2

```

	초기상태	7	0	1	2	0	3	0	4	2
0 번	0000	0000	1000	0100	0010	1001	0100	1010	0101	0010
	0000	0000	0000	0000	0000	0000	1000	0100	0010	1001
	3	0	3	2	1	2	0	1	7	0
0 번	0001	1000	0100	0010	0001	0000	1000	0100	0010	1001
	0100	1010	0101	0010	0001	1000	0100	0010	0001	0000
	1									
0 번	0100 1000									