

0. generate_ref_arr() 공통 구현 설명 (3-1, 3-2, 3-3 모두 동일)

```
int* generate_ref_arr(size_t sz, size_t page_max) {
    int* ref_arr = (int*) malloc(sizeof(int) * sz);
    // TODO :: Generate Random Refernece String
    srand(time(NULL));
    for(int i = 0; i < sz; i++){
        ref_arr[i] = rand() % (page_max + 1);
    }
    return ref_arr;
}
```

- time 함수를 쓰지 않으면 seed 값을 조절할 수 없기 때문에 진짜 랜덤 생성을 할 수 없습니다. 따라서 srand(time(NULL));을 통해 seed 값을 매번 랜덤으로 조절하였습니다.
- 0부터 page_max 값 중 랜덤으로 값을 생성하여 ref_arr에 저장하였습니다.

1. Assignment 3-1

- Frames 배열을 stack으로 구현하였습니다. 사실 말이 stack이지 stack 중간의 번호를 참조하면 그 번호를 stack의 top으로 옮겨야 하기 때문에 doubly linked list로 구현하는 것이 더 효율적인 구현이지만 간단한 구현을 위해 stack 배열의 원소들을 차례대로 swap 하는 방식을 사용하였습니다. 제시된 reference string S에 대한 결과를 살펴보면 총 8번의 page fault가 일어났음을 알 수 있습니다. 각 page가 참조될 때의 stack의 값을 아래의 표에서 확인할 수 있습니다. 참조가 일어난 page frame이 항상 stack의 top에 있음을 알 수 있고 page fault가 발생했을 때 frames이 가득 찼다면 가장 밑에 있는(가장 오래 전에 참조한 페이지) 페이지를 swap out 대상으로 선택하는 것을 알 수 있습니다.

참조 page	stack bottom	stack top		
7	7	-	-	-
0	7	0	-	-
1	7	0	1	-
2	7	0	1	2
0	7	1	2	0
3	1	2	0	3
0	1	2	3	0
4	2	3	0	4
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
2	0	3	1	2
0	3	1	2	0
1	3	2	0	1
7	2	0	1	7
0	2	1	7	0
1	2	7	0	1

```
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
8
```

```
int lru(int* ref_arr, size_t ref_arr_sz, size_t frame_sz) {
    int i, j;
    int page_faults = 0, target = 0, is_fault = 0;

    // Initializing frames
    int* frames = (int*) malloc(sizeof(int) * frame_sz);
    for (i=0; i<frame_sz; i++) frames[i] = -1;

    // TODO
    // Iterating reference string
    for (i=0; i<ref_arr_sz; i++) {
        is_fault = _contains(frames, frame_sz, ref_arr[i]);

        // Miss (page fault occurred)
        if (is_fault == -1) {
            page_faults++;
            if (target != frame_sz){
                frames[target++] = ref_arr[i];
            }
            else{
                frames[0] = ref_arr[i];
                for(int j = 0; j < frame_sz - 1; j++){
                    swap(&frames[j], &frames[j + 1]);
                }
            }
        }
        else{
            for(int j = is_fault; j < frame_sz - 1; j++){
                swap(&frames[j], &frames[j + 1]);
            }
        }

        // Printing current states of frames
        printf("%d | ", ref_arr[i]);
        for(j=0; j<frame_sz; j++) {
            if (frames[j] == EMPTY_FRAME) printf(". ");
            else printf("%d ", frames[j]);
        }
        if (is_fault == -1) printf("(fault)");
        printf("\n");
    }

    // Releasing memory
    free(frames);
    return page_faults;
}
```

2. Assignment 3-2

- ➔ 핵심 알고리즘만 설명하자면 circular queue로 구현된 clock을 reference bits로 활용하는데 page가 참조될 때마다 해당 frame의 reference bit = 1로 설정합니다. Page fault가 발생하면 clock을 차례대로 돌면서 reference bit가 0인 frame을 찾고 만약 1이라면 해당 bit를 0으로 바꾸고 순회합니다. reference bit가 0인 frame을 victim으로 삼아 swap out을 진행합니다. 그 후 pointer가 다음 페이지를 가리키도록 조정합니다. 이렇게 하는 이유는 모든 reference bits가 1이면 FIFO 와 동일하게 작동이 되는데 위의 상황에서 reference bit가 0인 page를 찾아 swap out을 진행했을 때 포인터가 다음 페이지를 가리키도록 해야만 모든 reference bits가 1이면 FIFO 와 동일하게 작동한다는 조건을 맞출 수 있기 때문입니다. Reference string S에 대한 결과를 살펴보면 총 9번의 page fault가 일어났음을 알 수 있습니다. 만일 swap out이 일어난 후 pointer를 다음 페이지로 가리키지 않으면 page fault가 8번이 일어나게 되는데 second chance LRU의 정의에 따른 결과와 맞지 않게 됩니다.

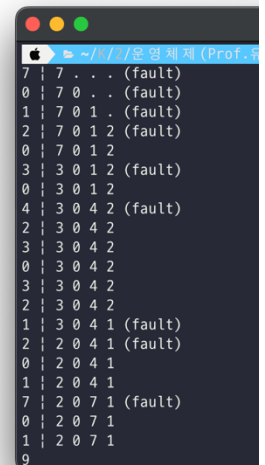
```
int lru(int* ref_arr, size_t ref_arr_sz, size_t frame_sz) {
    int i, j;
    int page_faults = 0, is_fault = 0;

    // Initializing frames
    int* frames = (int*) malloc(sizeof(int) * frame_sz);
    for (i=0; i<frame_sz; i++) frames[i] = -1;

    // TODO
    // Initializing accessed clock table
    int* clock = (int*) malloc(sizeof(int) * frame_sz);
    for (i=0; i<frame_sz; i++) clock[i] = 0;
    int clock_target = 0;

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

        // Miss (page fault occurred)
        if (is_fault == -1) {
            page_faults++;
            while(1){
                if(clock[clock_target] == 0){
                    frames[clock_target] = ref_arr[i];
                    clock[clock_target] = 1;
                    clock_target = (clock_target + 1) % frame_sz;
                    break;
                }
                else if(clock[clock_target] == 1){
                    clock[clock_target] = 0;
                    clock_target = (clock_target + 1) % frame_sz;
                }
            }
        }
        else{
            clock[is_fault] = 1;
        }
    }
}
```



```
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 4 1 (fault)
2 | 2 0 4 1 (fault)
0 | 2 0 4 1
1 | 2 0 4 1
7 | 2 0 7 1 (fault)
0 | 2 0 7 1
1 | 2 0 7 1
9
```

3. Assignment 3-3

- 구현된 lru 함수를 살펴보면 reference bits를 담는 배열인 bits를 배열의 사이즈를 page size 만큼 해서 생성한 후 처음엔 00000000으로 초기화 합니다. 이후 page 참조가 일어날 때마다 모든 reference bits에 대해 right shift 연산을 수행합니다. 만약 page fault가 발생하였는데 frame이 이미 꽉 찼다면 frame에 들어있는 pages 중에서 reference bits의 값이 가장 작은 page를 선택해 swap out을 진행하게 됩니다. 주어진 Reference string S에 대한 결과를 살펴보면 아래의 그림과 같습니다. 우선 총 8번의 page fault가 일어난 사실을 알 수 있습니다.

0번 page의 reference bit가 어떻게 변하고 있는지를 다음의 표를 통해서 살펴보겠습니다. 0이 참조될 때마다 가장 왼쪽의 bit가 1로 바뀌고 매 page 참조가 일어날 때마다 right shift 연산이 일어나는 것을 볼 수 있습니다. 이를 이용해 LRU 알고리즘을 구현할 수 있습니다.

```
int lru(int* ref_arr, size_t ref_arr_sz, size_t frame_sz, size_t page_max) {
    int i, j;
    int page_faults = 0, frame_target = 0, is_fault = 0, target = 0;

    // Initializing frames
    int* frames = (int*) malloc(sizeof(int) * frame_sz);
    for (i=0; i<frame_sz; i++) frames[i] = -1;

    // TODO
    // Initializing accessed reference bits table
    int page_size = page_max + 1;
    int* bits = (int*) malloc(sizeof(int) * page_size);
    for (i=0; i<page_size; i++) bits[i] = 0b00000000;

    // Iterating reference string
    for (i=0; i<ref_arr_sz; i++) {
        is_fault = _contains(frames, frame_sz, ref_arr[i]);
        for(int j = 0; j < page_size; j++) bits[j] = bits[j] >> 1;
        bits[ref_arr[i]] |= 0b10000000;

        // Miss (page fault occurred)
        if (is_fault == -1) {
            page_faults++;
            if (frame_target < frame_sz){
                frames[frame_target++] = ref_arr[i];
            }
            else{
                int min_target = frames[0];
                for(int j = 1; j < frame_sz; j++){
                    if(bits[min_target] > bits[frames[j]]){
                        min_target = frames[j];
                    }
                }
                target = _contains(frames, frame_sz, min_target);
                frames[target] = ref_arr[i];
            }
        }
    }
}
```

참조 page	page 0의 reference bit
7	00000000
0	10000000
1	01000000
2	00100000
0	10010000
3	01001000
0	10100100
4	01010010
2	00101001
3	00010100
0	10001010
3	01000101
2	00100010
1	00010001
2	00001000
0	10000100
1	01000010
7	00100001
0	10010000
1	01001000

```
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
0 | 3 0 4 2
3 | 3 0 4 2
0 | 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
8
```