고려대학교
KOREA UNIVERSITY

KU-The Future

Page Replacement Algorithms

2022년 1학기 운영체제 수업 실습 03

조교 김명현 (freckie@korea.ac.kr)

2022.05.30

Table of Contents

Section 1

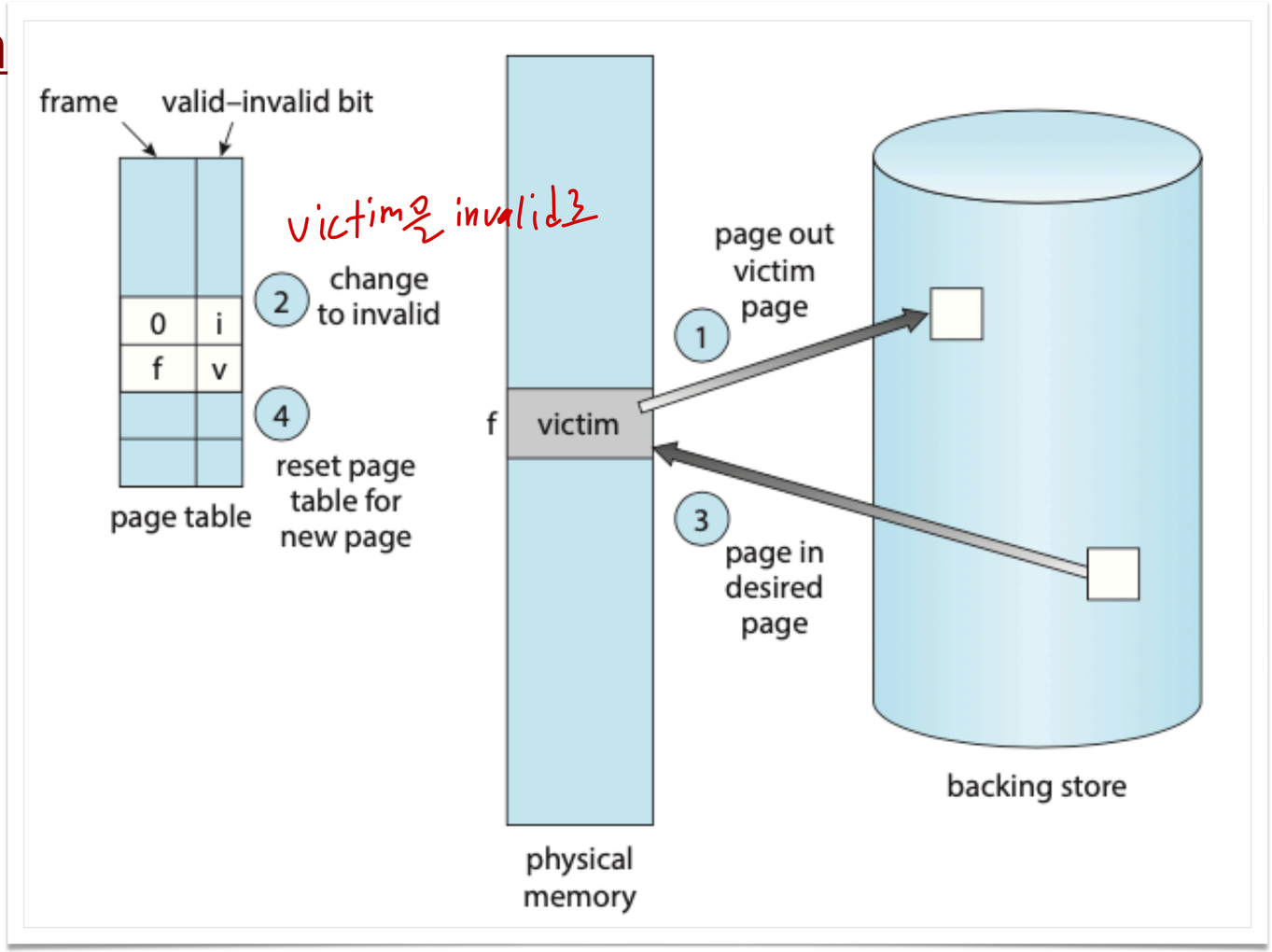
“ Basic Concepts ”

Page Replacement

- 요구되는 페이지를 모두 메모리에 유지할 수는 없으므로 페이지의 교체는 필수적임.
- 디스크 I/O의 비용이 매우 크기 때문에 적절한 페이지를 선택하여 교체하는 알고리즘이 필요함.
  교체가 최대한 적게 이루어져야함
- **Page Fault** : 요구되는 페이지가 프레임에 존재하지 않는 상황
- **Victim Page** : 빈 프레임이 없는 경우 프레임의 확보를 위해 swap-out되는 페이지

- Page Replacement 알고리즘들은 페이지 부재율(Page Fault rate)을 성능의 지표로 삼음.

Page Replacem



Page Replacement

- **Reference String** : 메모리 참조 요청들의 시간순 집합

  - Address sequence : 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

  - 페이지의 크기가 100 Bytes라고 가정.

Page Replacement

- **Reference String** : 메모리 참조 요청들의 시간순 집합

- Address sequence : 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

- 페이지의 크기가 100 Bytes라고 가정.

- Reference String : 1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

↳주어졌을 때 page fault가 얼마나 발생하는지, page의 부재율을 계산해서
이 것로 알고리즘 성능의 지표로 삼음

Section 2

“ FIFO, Optimal ”

2. FIFO, Optimal Page Replacement

FIFO (First-In, First-Out)

- Page Fault 발생 시, 프레임에 **적재된 지 가장 오래된 페이지**를 victim으로 선택

- 페이지가 적재된 순서를 FIFO Queue를 유지하여 구현

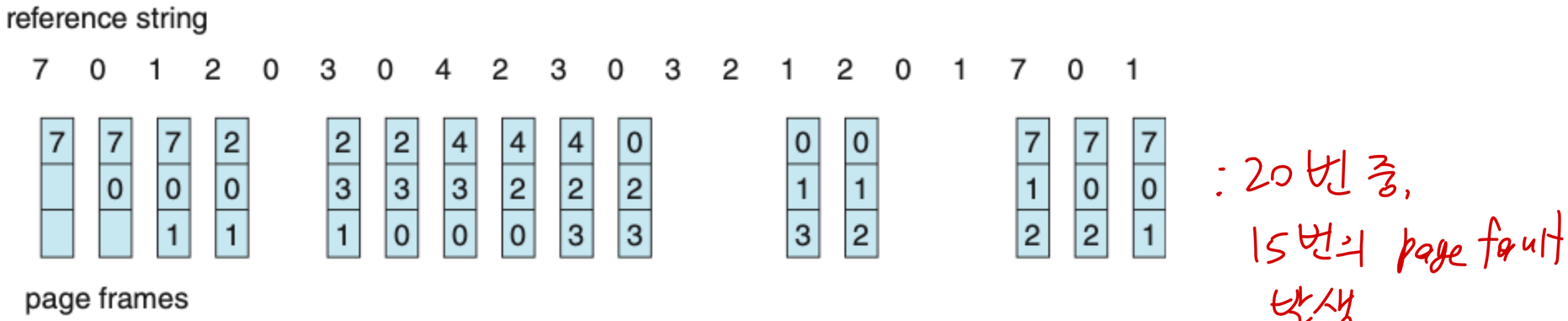


Figure 10.12 FIFO page-replacement algorithm.

2. FIFO, Optimal Page Replacement

frame 개수 : 4개로 설정

FIFO (First-In, First-Out)

reference string의 size

frame의 size

reference string

```
28  void fifo(int* ref_arr, size_t ref_arr_sz, size_t frame_sz) {
29      int i, j, is_fault;
30      int page_faults = 0, target = 0;
31
32      // Initializing frames
33      int* frames = (int*) malloc(sizeof(int) * frame_sz);
34      for (i=0; i<frame_sz; i++) frames[i] = EMPTY_FRAME;
35
36      // Iterating reference string
37      for (i=0; i<ref_arr_sz; i++) {
38          is_fault = _contains(frames, frame_sz, ref_arr[i]);
39
40          // Miss (page fault occurred)
41          if (is_fault == -1) {
42              frames[target] = ref_arr[i];
43              target = (target + 1) % frame_sz;
44              page_faults++;
45          }
46
```

frame을 size 만큼 공간 확보해서 생성

frame을 아무것도 안들어있는 -1로 설정

주어진 reference string을 iteration하면서 page fault 발생하는지 확인

frames 배열을 선형 탐색하면서 page가 존재하는지 확인

frame 순서대로 page 교체

다음 교체 대상을 target을 통해 index로 지정.

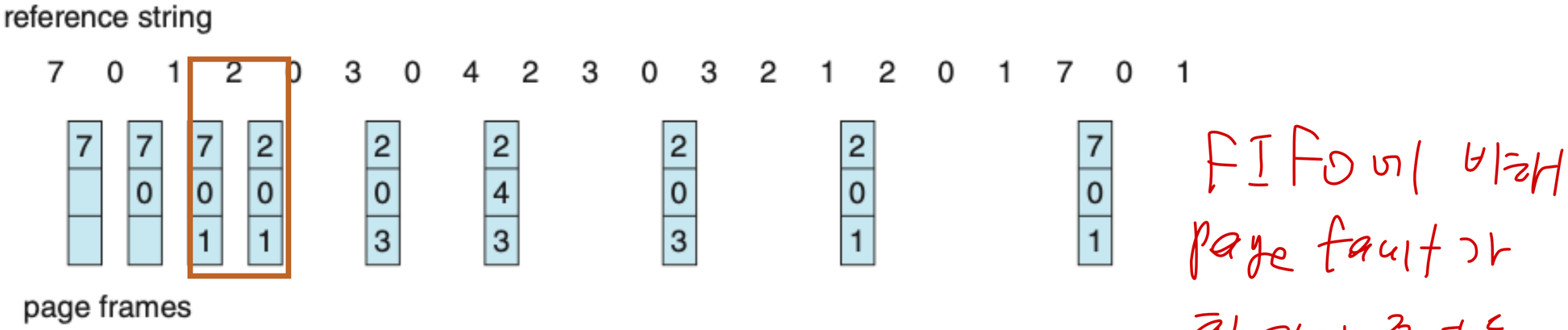
```
7 | 7 . . . (fault)
0 | 7 0 . . (fault)
1 | 7 0 1 . (fault)
2 | 7 0 1 2 (fault)
0 | 7 0 1 2
3 | 3 0 1 2 (fault)
0 | 3 0 1 2
4 | 3 4 1 2 (fault)
2 | 3 4 1 2
3 | 3 4 1 2
0 | 3 4 0 2 (fault)
3 | 3 4 0 2
2 | 3 4 0 2
1 | 3 4 0 1 (fault)
2 | 2 4 0 1 (fault)
0 | 2 4 0 1
1 | 2 4 0 1
7 | 2 7 0 1 (fault)
0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

2. FIFO, Optimal Page Replacement

Optimal

- Page Fault 발생 시, 앞으로 **가장 오랫동안 참조되지 않을 페이지**를 victim으로 선택
- 미래의 reference string까지 알 수만 있다면 가장 최적의 교체 알고리즘

실제로는 알 수 없음



FIFO의 비해 page fault가 확연히 줄어듦

Figure 10.14 Optimal page-replacement algorithm.

2. FIFO, Optimal Page Replacement

```
29  void opt(int* ref_arr, size_t ref_arr_sz, size_t frame_sz) {
30      int i, j, is_fault;
31      int page_faults = 0, target = 0;
32      int dist = -1;
33
34      // Initializing frames
35      int* frames = (int*) malloc(sizeof(int) * frame_sz);
36      for (i=0; i<frame_sz; i++) frames[i] = EMPTY_FRAME;
37
38      // Iterating reference string
39      for (i=0; i<ref_arr_sz; i++) {
40          is_fault = _contains(frames, frame_sz, ref_arr[i]);
41
42          // Miss (page fault occurred)
43          if (is_fault == -1) {
44              int empty_idx = _contains(frames, frame_sz, EMPTY_FRAME);
45
46              // Checking for empty frame slots
47              if (empty_idx != EMPTY_FRAME) {
48                  target = empty_idx;
49              }
50              else {
51                  for (j=0; j<frame_sz; j++) {
52                      int tmp_dist = _get_distance(ref_arr, ref_arr_sz, i, frames[j]);
53                      if (tmp_dist > dist) {
54                          target = j;
55                          dist = tmp_dist;
56                      }
57                  }
58              }
59
60              frames[target] = ref_arr[i];
61              page_faults++;
62              dist = -1;
63          }
```

reference string 참조해서
↗ 미래에 참조할 시점까지의 거리를 구함

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

2. FIFO, Optimal Page Replacement

Summary

FIFO (First-In First-Out)

- 페이지가 프레임에 로드된 시점을 기준으로 victim을 선택함.

OPT (Optimal)

- 페이지가 참조되는 시점을 기준으로 victim을 선택함.
- 실제로 구현이 불가능한 이상적인 알고리즘

Section 3

“ LRU ”

3. LRU Page Replacement

LRU (Least Recently Used)

- Page Fault 발생 시, 가장 **오랫동안 사용되지 않은 페이지**를 victim으로 선택
  - 가까운 미래에 대한 예측으로 가까운 과거를 사용하는 전략
  - OPT와 마찬가지로 **페이지 참조 시점**을 기준으로 사용함
    - 즉, LRU는 과거 시간에 대한 OPT 알고리즘이라 표현할 수 있음.

- Reference String S의 순서를 뒤집은 것을 S^R이라 할 때,
  - $OPT(S)$의 Page Fault rate는 $OPT(S^R)$의 Page Fault rate와 같음.
  - 마찬가지로, $LRU(S)$와 $LRU(S^R)$의 Page Fault rate는 같음.

3. LRU Page Replacement

LRU (Least Recently Used)

- Page Fault 발생 시, 가장 **오랫동안 사용되지 않은 페이지**를 victim으로 선택

- LRU는 **시간 지역성(Temporal Locality)** 또한 고려한 알고리즘
  - 최근 많이 사용된 페이지는 빠른 시간 내에 다시 사용될 확률이 높음.

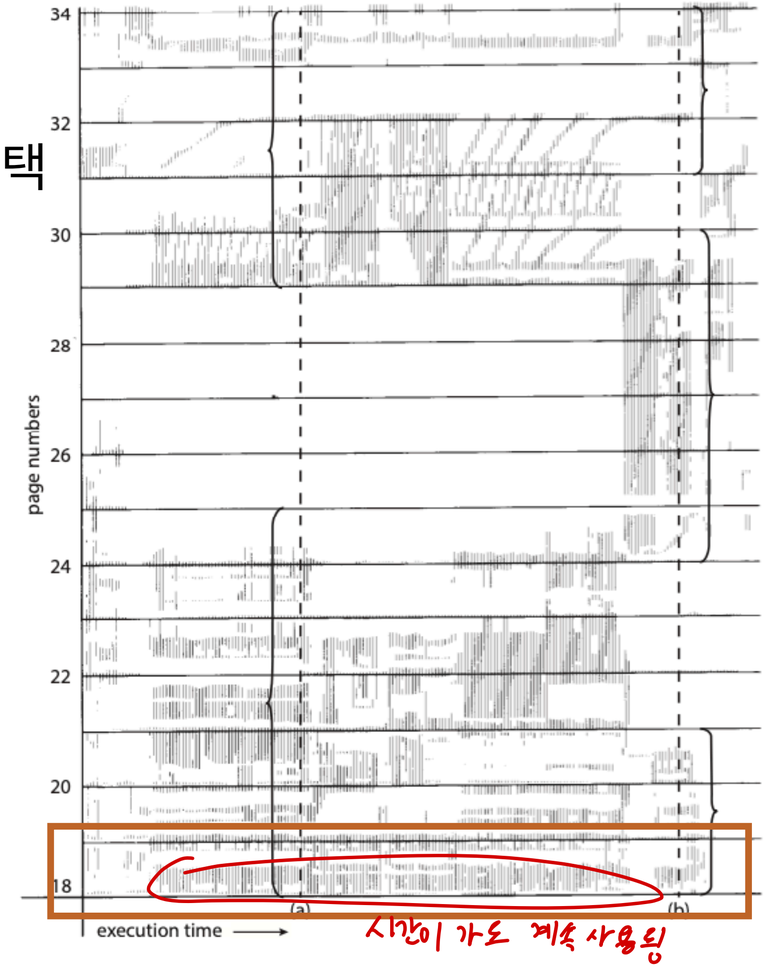


Figure 10.21 Locality in a memory-reference pattern.

3. LRU Page Replacement

LRU (Least Recently Used)

- Page Fault 발생 시, 가장 **오랫동안 사용되지 않은 페이지**를 victim으로 선택

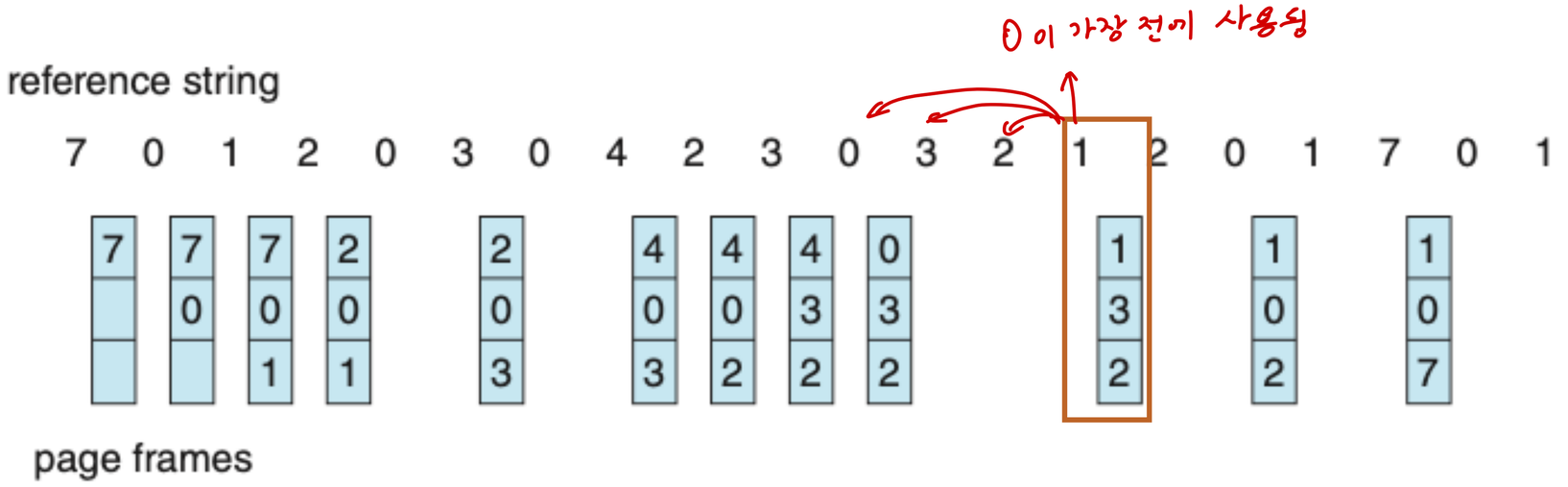


Figure 10.15 LRU page-replacement algorithm.

3. LRU Page Replacement

LRU (Least Recently Used)

Counter를 사용한 LRU 구현

counter 값 중 최솟값을 가진 것을 victim으로 사용.

- LRU 구현 중 가장 간단한 방법으로, 개별 페이지마다 **최근에 접근했던 시간을 유지**함.
- 단점:
  - 접근했던 시간이 가장 작은 페이지를 찾기 위해 매번 테이블을 탐색해야 함.
  - 페이지 테이블 수정이 일어날 때마다 시간 값을 관리해야 함.

+ 매번 interrupt 때문에 overhead 발생

  - 인터럽트를 이용해 소프트웨어로 구현 시 Page Fault마다 시간 값을 저장하고, 테이블을 탐색해야 하므로 메모리 참조 속도가 상당히 저하됨.
    - LRU를 위한 하드웨어 지원이 필요함.

3. LRU Page Replacement

```
44      // Initializing frames
45      int* frames = (int*) malloc(sizeof(int) * frame_sz);
46      for (i=0; i<frame_sz; i++) frames[i] = EMPTY_FRAME;
47
48      // Initializing accessed times table
49      int page_max = _max(ref_arr, ref_arr_sz) + 1;
50      int* atimes = (int*) malloc(sizeof(int) * page_max);
51      for (i=0; i<page_max; i++) atimes[i] = -1;
52
53      // Iterating reference string
54      for (i=0; i<ref_arr_sz; i++) {
55          is_fault = _contains(frames, frame_sz, ref_arr[i]);
56
57          // Miss (page fault occurred)
58          if (is_fault == -1) {
59              int empty_idx = _contains(frames, frame_sz, EMPTY_FRAME);
60
61              // Checking for empty frame slots
62              if (empty_idx != EMPTY_FRAME) {
63                  target = empty_idx;
64              }
65              else {
66                  int target_atime = INT_MAX;
67                  for (j=0; j<frame_sz; j++) {
68                      if (atimes[frames[j]] < target_atime && atimes[frames[j]] >= 0) {
69                          target = j;
70                          target_atime = atimes[frames[j]];
71                      }
72                  }
73              }
74
75              frames[target] = ref_arr[i];
76              page_faults++;
77          }
78
79          atimes[ref_arr[i]] = curr_time++;
```

각 page별 access time을 저장하는 table

⇓

공간적으로도 overhead가 큼

→ page fault 발생시

가장 오래된 것을 선형적으로 탐색

```
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
```

3. LRU Page Replacement

LRU (Least Recently Used)

Stack을 사용한 LRU 구현

- 페이지 참조가 일어날 때마다, 그 페이지의 번호를 스택의 top에 유지시켜두는 구현

- 스택의 **top은 항상 최근에 참조된 페이지**이며, **bottom은 오랫동안 참조되지 못한 페이지**

- 스택의 중간에 있던 번호의 페이지를 참조 시, 이를 스택의 top으로 옮겨야하는 등 중앙에서의 삭제가 빈번함. Overhead 발생
  - 따라서 소프트웨어로 구현 시 Doubly Linked List로 구현
  - 하드웨어 지원이 필요

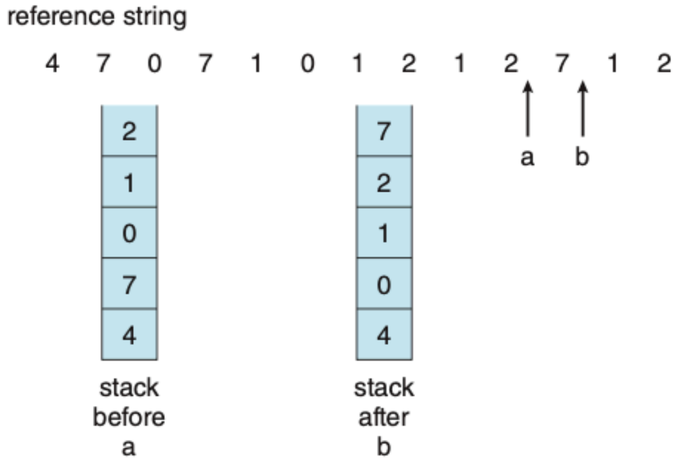


Figure 10.16 Use of a stack to record the most recent page references.

3. LRU Page Replacement

LRU Approximation

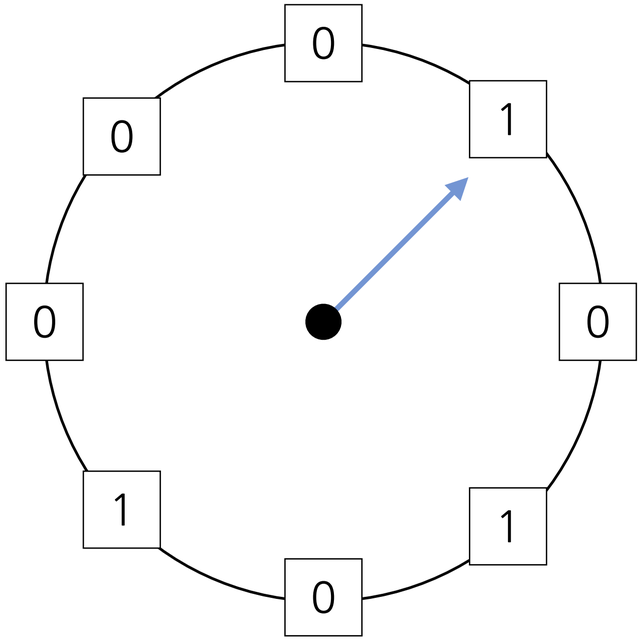
- LRU를 완벽히 지원하는 하드웨어를 가진 시스템은 거의 없으나, 대부분의 시스템이 **Reference Bit**의 형태로 제한적으로나마 지원함.
  - Reference Bit들은 0으로 초기화된 상태로 시작
  - 페이지의 참조가 일어나면 그 페이지의 비트는 하드웨어가 1로 세팅

순서는 알수 없지만 bit를 통해 사용 여부는 확실히 알 수 있음

3. LRU Page Replacement

LRU Approximation - Clock Algorithm (Second-Chance Algorithm)

- Clock 알고리즘은 Reference Bit를 이용한 LRU Approximation 알고리즘 중 하나
- 리눅스 커널 2.6부터 Clock 알고리즘 기반의 *Clock-PRO* 알고리즘을 사용함.
- Reference Bit을 **Circular Queue**로 관리함.



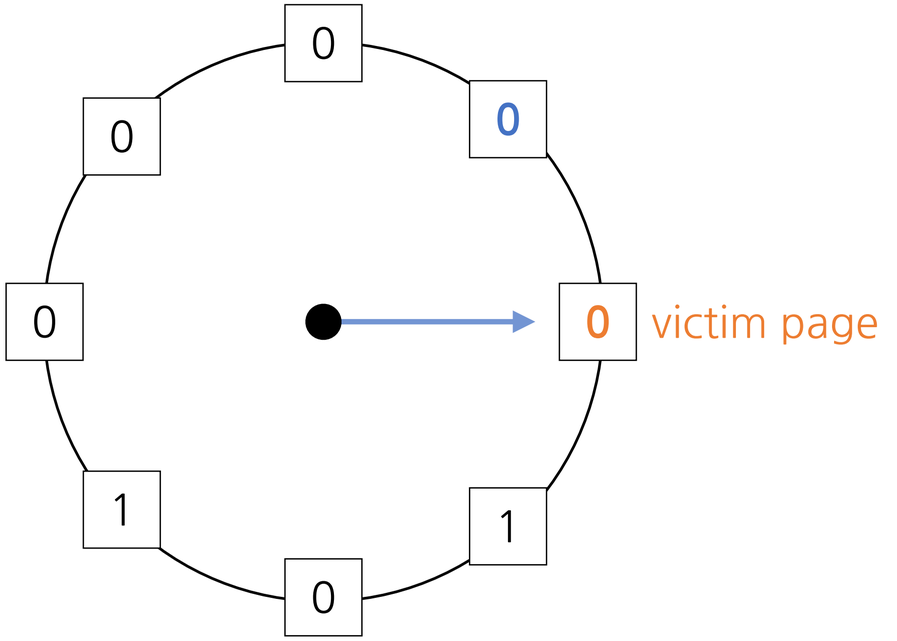
- Page Fault 발생 시, 포인터는 0을 발견할 때까지 전진함.
  - 포인터가 가리키는 Reference Bit가 1인 경우, 0으로 변경 (Second Chance)
  - Reference Bit가 0인 경우, 해당 페이지를 victim으로 선택

3. LRU Page Replacement

LRU Approximation - Clock Algorithm (Second-Chance Algorithm)

- Clock 알고리즘은 Reference Bit를 이용한 LRU Approximation 알고리즘 중 하나
- 리눅스 커널 2.6기준 Clock 알고리즘 기반의 Clock-PRO 알고리즘을 사용함.
- Reference Bit을 Circular Queue로 관리함.

- Page Fault 발생 시, 포인터는 0을 발견할 때까지 전진함.
  - 포인터가 가리키는 Reference Bit가 1인 경우, 0으로 변경 (Second Chance) 한 번 더 기회를 주는 느낌
  - Reference Bit가 0인 경우, 해당 페이지를 victim으로 선택
    ↳ swap out 함.



3. LRU Page Replacement

LRU Approximation - Additional Reference Bits Algorithm

8 bits.

- Reference Bit를 사용하면서, 각 페이지에 추가적인 비트들을 기록하여 선후 관계를 기록하는 알고리즘
  - 각 페이지에 8비트짜리 바이트만 할당함으로써 큰 오버헤드는 줄이고, 페이지들이 참조된 순서 정보를 알 수 있음.

시간적 & 공간적 overhead↓

- 페이지가 참조되면 하드웨어에 의해 Reference Bit가 1로 설정되는 것은 동일
- 각 페이지에 8비트를 할당하여 0000 0000 으로 초기화
- 일정 시간 간격마다 OS가 다음 로직 실행 (타이머 인터럽트): ex) 100ms 마다.
  - 각 페이지에 할당된 8비트를 오른쪽으로 Shift
  - Reference Bit를 8비트의 최상위 비트에 이동

(매번 수행하면 overhead 너무 커짐)

3. LRU Page Replacement

LRU Approximation - Additional Reference Bits Algorithm

(주의: 쉬운 이해를 위해, Bit Shift 수행을 타이머 인터럽트 발생 시 하지 않고 페이지 참조가 일어날 때마다 한다고 가정)

이게 제일 작으므로 victim

- Reference String: [0, 1, 2, 0, 4, 0]

Page No.	초기상태	0	1	2	0	4	0
0	0000 0000	1000 0000	0100 0000	0010 0000	1001 0000	0100 1000	1010 0100
1	0000 0000	0000 0000	1000 0000	0100 0000	0010 0000	0001 0000	0000 1000
2	0000 0000	0000 0000	0000 0000	1000 0000	0100 0000	0010 0000	0001 0000
3	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000
4	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000	1000 0000	0100 0000

- 프레임이 3개라고 할 때, 4번 페이지가 요청되었을 때 프레임에 [0, 1, 2]가 존재하며 Page Fault 발생함.
  - 이때, [0, 1, 2] 중 참조 비트의 값이 가장 작은 페이지는 1번 페이지 (= Least Recently Used)

3. LRU Page Replacement

LRU Approximation - Additional Reference Bits Algorithm

(주의: 쉬운 이해를 위해, Bit Shift 수행을 타이머 인터럽트 발생 시 하지 않고 페이지 참조가 일어날 때마다 한다고 가정)

- Reference String: [0, 1, 2, 0, 4, 0]

Page No.	초기상태	0	1	2	0	4	0
0	0000 0000	1000 0000	0100 0000	0010 0000	1001 0000	0100 1000	1010 0100
1	0000 0000	0000 0000	1000 0000	0100 0000	**0010 0000**	0001 0000	0000 1000
2	0000 0000	0000 0000	0000 0000	1000 0000	0100 0000	0010 0000	0001 0000
3	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000
4	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000	1000 0000	0100 0000

- 프레임이 3개라고 할 때, 4번 페이지가 요청되었을 때 프레임에 [0, 1, 2]가 존재하며 Page Fault 발생함.

  - 이때, [0, 1, 2] 중 참조 비트의 값이 가장 작은 페이지는 1번 페이지 (= Least Recently Used)

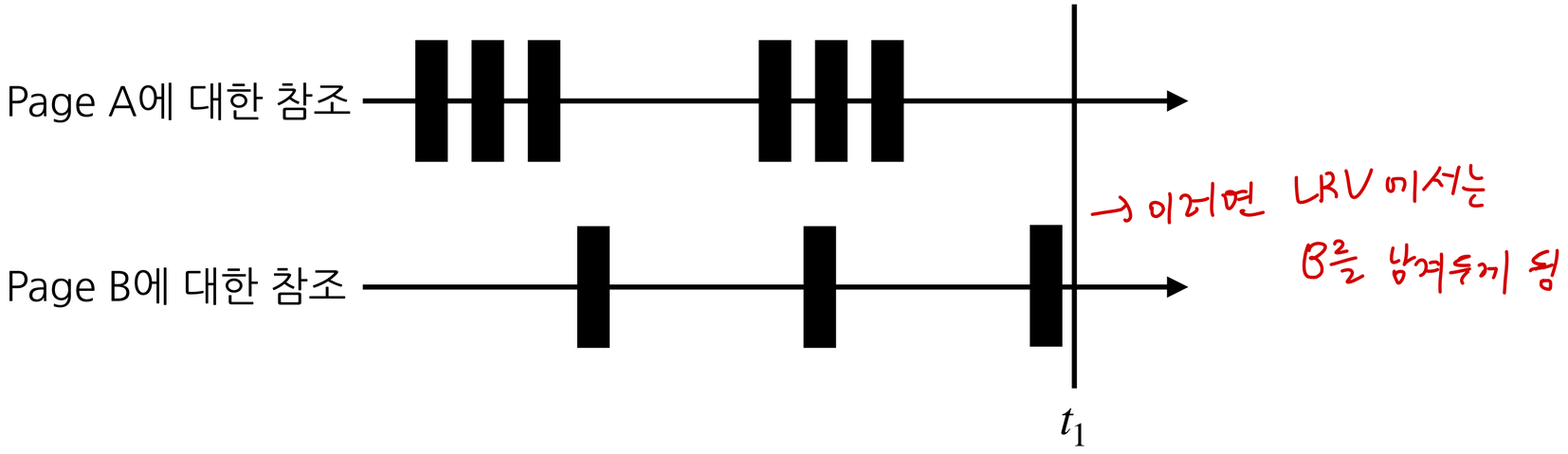
Section 4

“ LRU in Linux ”

LRU-K (Least Recently Used - Kth)

기존 LRU의 문제점

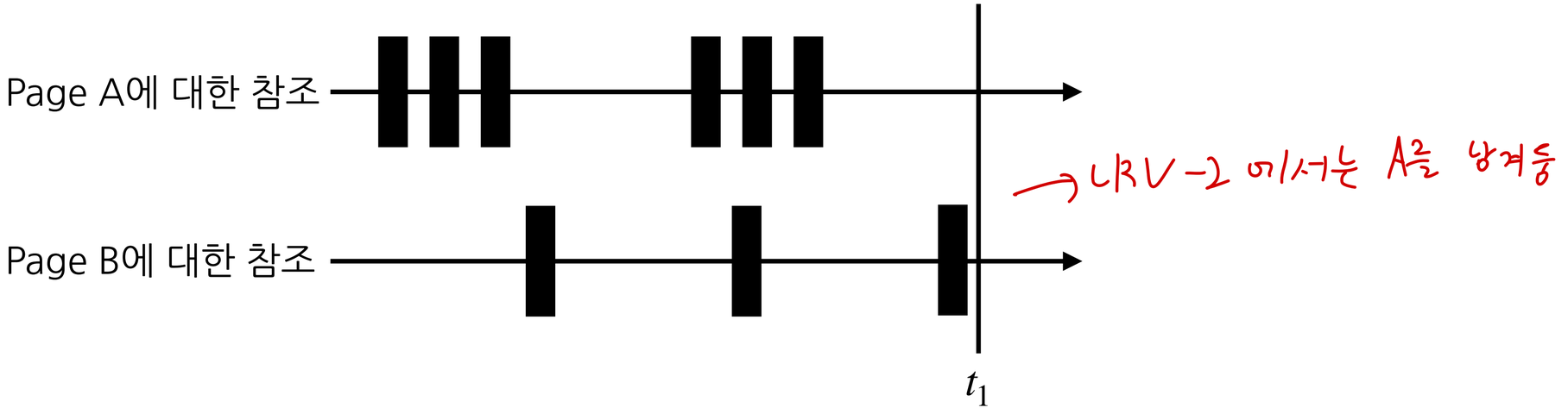
- 기존 LRU는 **최근에 참조한 페이지만을 고려**한다는 점이 있음.
  - 과거에 여러번 참조된 페이지보다, 가장 최근에 한 번만 참조된 페이지가 프레임에 남아있게 됨.



4. LRU in Linux

LRU-K (Least Recently Used - Kth)

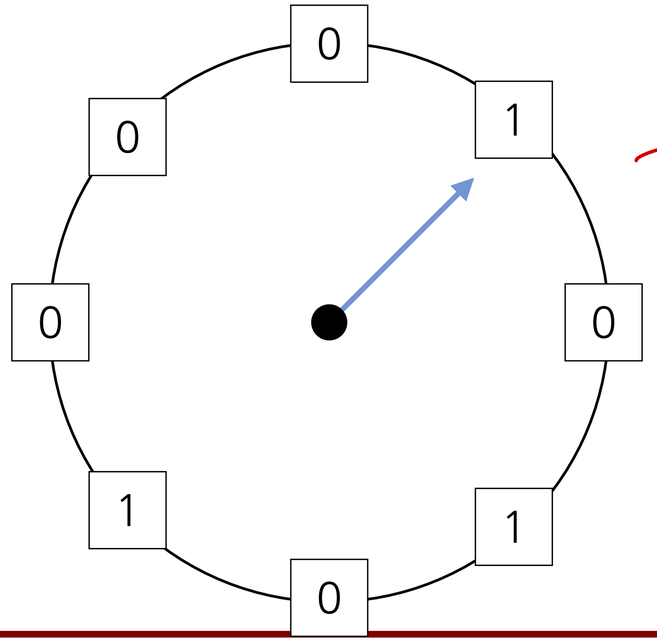
- LRU-K : K번째로 최근에 참조되었던 시간을 기준으로 victim을 결정하는 전략
  - 가장 최근에 참조된 시간을 기준으로 삼는 경우는 LRU-1이라고 할 수 있음. : 기존에 알고 있던 LRU
  - 아래 그래프에서 LRU-2를 적용할 경우 페이지 B가 victim으로 결정됨.



→ LRU-2 에서는 A를 남겨둠

LRU-K (Least Recently Used - Kth)

- LRU-K : K번째로 최근에 참조되었던 시간을 기준으로 victim을 결정하는 전략
  - 가장 최근에 참조된 시간을 기준으로 삼는 경우는 LRU-1이라고 할 수 있음.
  - 아래 그래프에서 LRU-2를 적용할 경우 페이지 B가 victim으로 결정됨.



→ LRU-2 와 같다고 볼 수 있음

LRU implementation in Linux

- Linux에서는 프레임에 존재하는 페이지들을 **active list**와 **inactive list**로 나누어서 관리함.*

  - **Active list** : 최근에 참조한 페이지들로, 다시 접근할 확률이 높은 페이지들
    - 커널에서 판단할 때 시스템이나 일부 프로세스에서 활성 사용 중

  - **Inactive list** : 프레임 할당 해제 후보 페이지들
    - 커널에서 판단할 때, 현재 사용 중이 아닌 페이지들

* 커널 버전 2.6 기준이며, 현재는 5개의 리스트를 사용함.

4. LRU in Linux

LRU implementation in Linux

- Active 페이지가 swap-out 대상으로 고려되면, 바로 메모리에서 제거되지 않고 inactive list로 이동함.

- 만약 inactive 페이지를 참조하고자 하면 **Soft Fault**가 발생하고 active list로 이동함.

LRU implementation in Linux

- 종합하자면, Linux에서 프레임에 할당된 페이지는 다음 두 가지 비트를 가지고 있음.
  - **Active Bit** : 현재 페이지가 active list에 있는지에 대한 정보
  - **Reference Bit** : 현재 페이지가 최근에 참조되었는지에 대한 정보

- 이 4가지 상태를 다루는 LRU 관련 함수들 5가지: (linux/mm/swap.c)
  - *lru_cache_add(), lru_cache_add_active(), mark_page_accessed(),*
  - *page_referenced(), refill_inactive_zone()*

4. LRU in Linux

LRU implementation in Linux

page의 상태를 부여함

- ***lru_cache_add (*page)***

  - inactive list에 페이지를 추가하는 함수 (LRU 리스트에 아직 페이지가 없는 경우 호출됨.)

    active bit = 0

- ***lru_cache_add_active (*page)***

  - active list에 페이지를 추가하는 함수 (마찬가지로 LRU 리스트에 아직 페이지가 없는 경우 호출됨.)

    active bit = 1

LRU implementation in Linux

mark_page_accessed (*page)

- 페이지 참조가 요청될 때마다 호출되는 함수
- inactive list에서 active list로 페이지를 이동시킴.

page의 상태 변환

- inactive , unreferenced -> inactive , referenced
- inactive , referenced -> active , unreferenced
- active , unreferenced -> active , referenced

LRU implementation in Linux

page의 상태를 꾸준히 관리

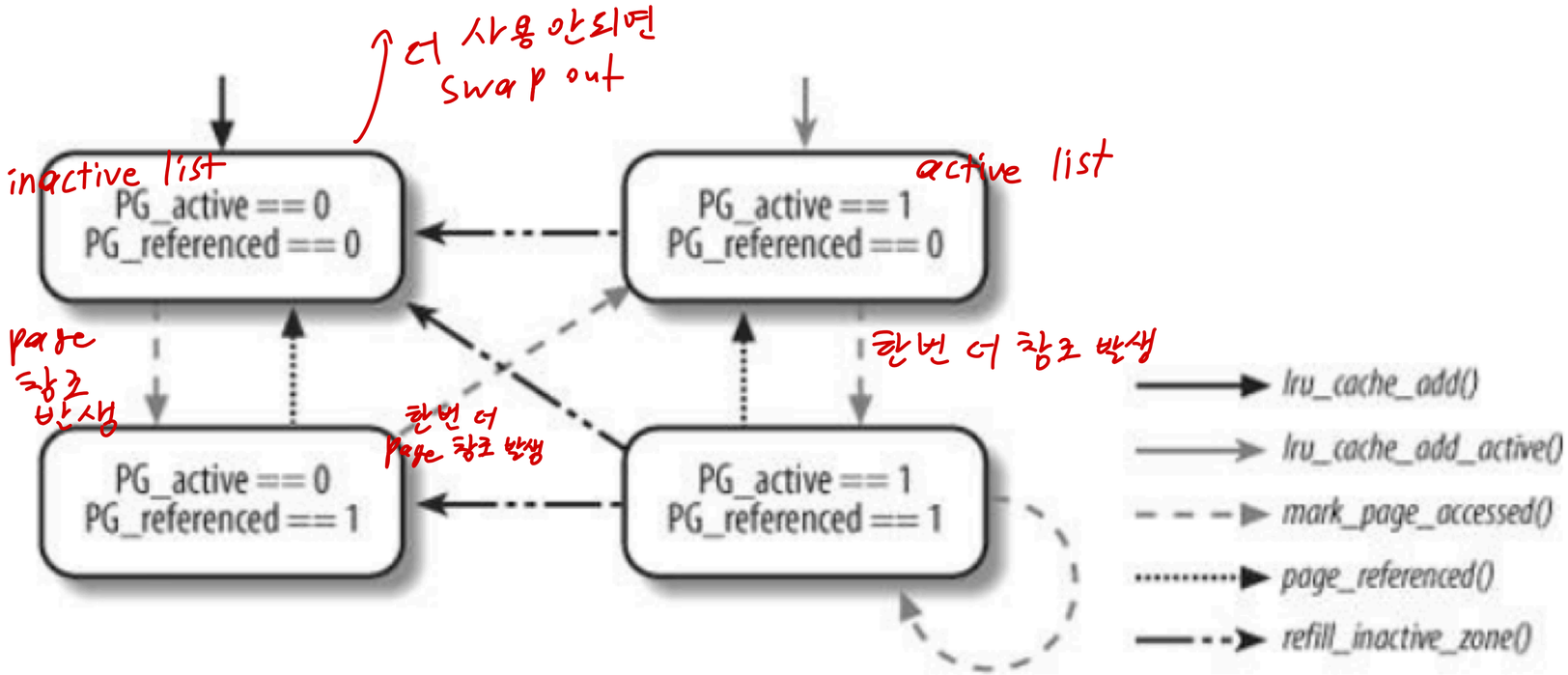
- page_referenced ()

- 주기적으로 페이지가 참조되었는지 확인하고 아니라면 Reference Bit를 0으로 설정

- refill_inactive_zone ()

- 주기적으로 페이지의 활성 상태를 판단하고, 비활성으로 판단되면 Active Bit를 0으로 설정

LRU implementation in Linux



Section 5

“ Assignment ”

5. Assignment

공통

stack, approximate 2가지

3가지 버전의 LRU 알고리즘에 대해 구현 및 시뮬레이션

- github.com/ku-cloud/22spring-os-lab03 의 assignment/skeleton.c 기반으로 구현
- INPUT: Reference String 길이(50~100), 최대 페이지 번호(5~10), 프레임 사이즈(1~6)
- OUTPUT: Page Fault 발생 횟수

실행 방법 : $ ./program {ref_arr_sz} {page_max} {frame_size}

example) ./program 50 10 5

Reference String은 주어진 ref_arr_sz와 page_max를 기반으로 **랜덤 생성**

program이 요구하는 page가 항상 예측 가능하지 않기 때문

5. Assignment

공통

3가지 버전의 LRU 알고리즘에 대해 구현 및 시뮬레이션

- **과제 결과물** (학번-이름-lab03.zip 으로 압축)
  - 각 버전의 구현 소스 파일 3개 (학번-이름-x.c)
  - 리눅스 환경에서 빌드한 각 버전 별 실행 파일 3개 (학번-이름-x.out)
  - 구현에 대해 간단히 설명한 총 2-3장 내외의 보고서 (학번-이름.pdf)
  - "블랙보드 > 과제 및 시험 > 실습3 과제" 에 제출 바랍니다.

- **기간**

  ~ 06.08 10:30am (오프라인 수업에서 답안 예시 공유 예정이므로, **기간 종료 후에는 받지 않겠습니다**)

5. Assignment

공통

3가지 버전의 LRU 알고리즘에 대해 구현 및 시뮬레이션

- 보고서에 대한 설명

- (학번-이름.pdf)
- 2-3장 내외
- 각 버전 별 구현에 대해 간단히 설명
- 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 결과를 오른쪽 그림처럼 출력하여 스크린샷 첨부

  (각 버전 별로) (프레임 개수 4로 통일)

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

5. Assignment

Assignment 3-1 (6점)

Stack을 이용한 LRU Replacement 시뮬레이션

- 파일명: 학번-이름-1.c

1. **generate_ref_arr()** 함수 구현 (랜덤 reference string 생성하여 리턴)
2. **lru()** 함수 구현 (page fault 발생 횟수 리턴)

- 보고서에 다음 내용 추가
  - 주어진 Reference String S에 대해, 페이지 참조마다 Stack 내용이 어떻게 변하는지 표로 작성

5. Assignment

Assignment 3-2 (7점)

Clock Algorithm 시뮬레이션

- 파일명: 학번-이름-2.c

1. **generate_ref_arr()** 함수 구현 (랜덤 reference string 생성하여 리턴)
2. **lru()** 함수 구현 (page fault 발생 횟수 리턴)

5. Assignment

Assignment 3-3 (7점)

Additional Reference Bits Algorithm 시뮬레이션

- 파일명: 학번-이름-3.c

  1. **generate_ref_arr()** 함수 구현 (랜덤 reference string 생성하여 리턴)
  2. **lru()** 함수 구현 (page fault 발생 횟수 리턴)

- 보고서에 다음 내용 추가

  - 주어진 Reference String S에 대해, 페이지 참조마다 0번 페이지의 Reference Bits이 어떻게 변하는지 표로 작성

5. Assignment

추가적인 정보

- skeleton.c 파일은 Git clone해서 받는 것을 권장하지만, 다음 링크에서 내용 복사해가도 무방합니다.
  - https://github.com/KU-Cloud/22Spring-OS-Lab03/blob/master/assignment/skeleton.c

- 빌드는 gcc로 진행하며, Linux, macOS 관계 없습니다.
  - 단, Windows 실행 파일(.exe) 제출은 금지합니다.
    - 빌드를 위해 Linux 환경이 필요하다면 클라우드 가상머신을 제공할 수 있으니 조교 메일로 연락바랍니다.

- **lru()** 함수는 input, output만 정확하면 되므로 함수 이름이나 파라미터 이름은 유연하게 수정해도 됩니다.

- 추가 문의 : freckie@korea.ac.kr

End of Document