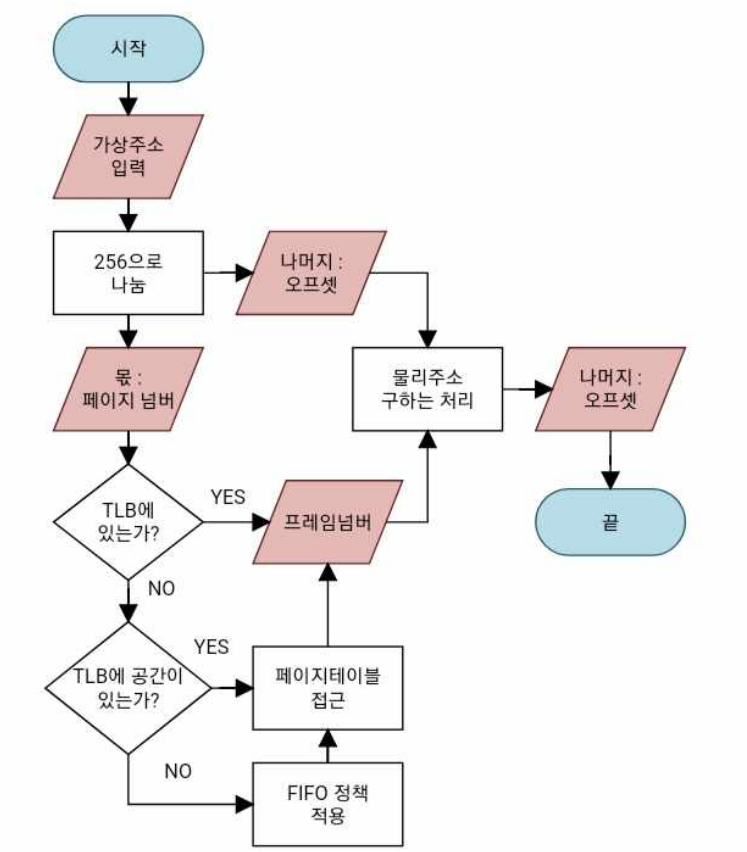
오퍼레이팅시스템 – 3 차 과제

12131819 육동현

01_프로그램 개요

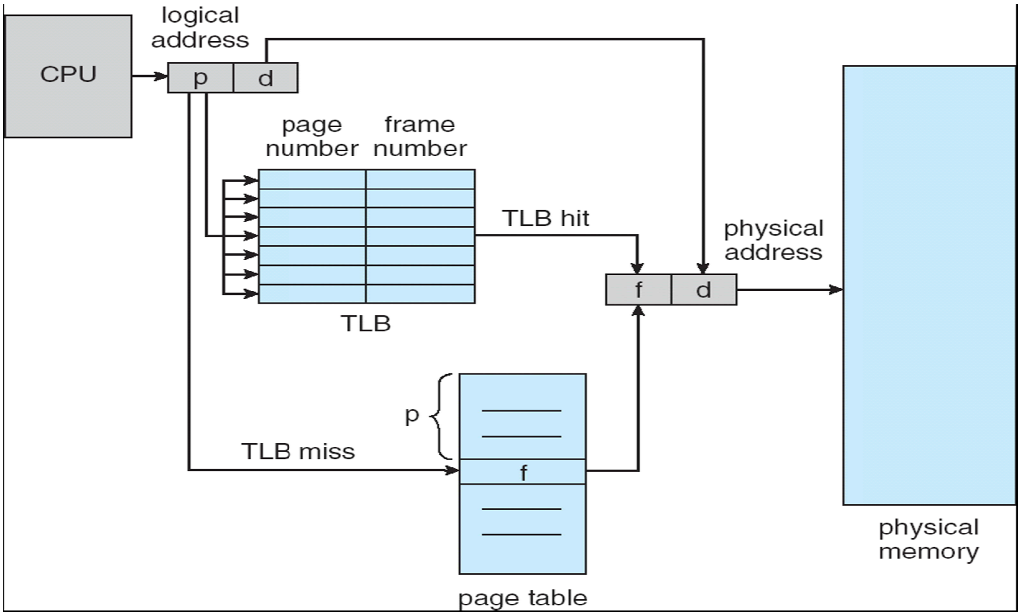
- addresses.txt 를 읽어서 physical.txt 를 생성한다.
- TLB 를 관리한다.
- 프레임 테이블을 관리한다.

02_설계 아이디어

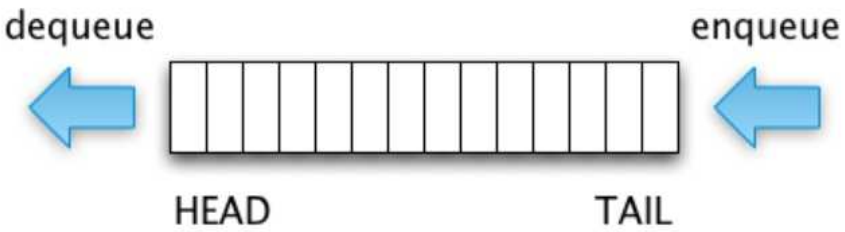


저는 이번 프로그램을 위와 같은 흐름도를 작성한 후 이에 기반하여 작성하였습니다.
프로그램이 시작되면 addresses.txt 에 저장된 논리주소를 읽어 들이고 해당 가상주소를 페이지 넘버와 오프셋으로 쪼갭니다. 이렇게 얻어 온 페이지 넘버를 TLB 내부 테이블에 존재하는지 탐색해봅니다.
만약 TLB 내부 테이블에 해당 페이지 넘버가 존재할 경우 이에 대응되는 프레임 넘버를 설정하고 이에 기반하여 물리주소를 구합니다. 만약 TLB 내부 테이블에 존재하지 않는 경우, TLB 내부에 공간이 있는 경우와 TLB 내부에 공간이 없는 경우로 나누어 처리합니다. TLB 내부에 공간이 있는 경우 즉시 페이지 테이블에 접근하여 프레임 넘버를 설정하고 이를 TLB 내부 테이블에 저장합니다. 만약 TLB 내부에 공간이 없는 경우 FIFO 정책에 의거하여 가장 먼저 들어 온 것을 victim 으로 선택하고 쫓아낸 후 페이지 테이블에 접근하여 프레임 넘버를 설정하고 이를 TLB 내부 테이블에 저장합니다. 이렇게 하고 난 후 앞의 경우와 마찬가지로 설정된 프레임 넘버에 기반하여 물리주소를 얻어오는 처리를 합니다.
(프로그램을 설계하면서 스마트폰 메모장에 적은 내용을 보고서의 06 번 부분에 참고자료로 제시합니다.)

03_자료구조 설계 아이디어



저는 자료구조를 설계 할 때 교수님의 강의교재에 있는 자료를 참고하여 만들었습니다.
페이지 테이블은 배열의 인덱스가 페이지 넘버이고 배열의 저장 값은 프레임 넘버인 1 차원 배열로 설계하였습니다. 또한 프레임 테이블은 배열의 인덱스가 프레임 넘버이고 배열의 첫 번째 열은 사용여부를 표현하는 부분, 배열의 두 번째 열은 페이지 넘버인 2 차원 배열로 설계하였습니다. TLB 내부 테이블은 첫 번째 열은 페이지 넘버, 두 번째 열은 프레임 넘버인 2 차원 배열로 설계하였습니다.



또한 위의 그림과 같이 TLB 의 저장용량을 초과한 경우 victim 을 선택하는 데 FIFO 정책을 적용하기 때문에 이를 구현할 수 있게 지원하는 FIFO 큐를 설계하였습니다. FIFO 큐의 동작원리는 가장 먼저 들어온 원소가 가장 먼저 나가는, 즉 victim 으로 설정되는 구조를 만들었습니다. 따라서 공간이 꽉 찬 경우 원소가 들어오게 된다면 가장 앞(HEAD)에 있는 원소가 나가고 그 바로 뒤에 있는 원소들을 앞으로 한 칸씩 당겨오는 처리를 한 후 가장 뒤(TAIL)에 새로 들어 온 원소를 삽입하는 처리를 합니다. (이는 FIFO 큐와 TLB 내부 테이블에 모두 적용합니다.)

04_핵심코드 설명

1) 아래는 03 에서 설명 드린 자료구조를 기반으로 코드를 구현하였습니다.

```
/* 자료구조 선언*/
int pageTable[256]; // 페이지 테이블 -> [(페이지 넘버) | 프레임 넘버]
int frameTable[256][2]; // 프레임 테이블 -> [(프레임 넘버) | 사용여부 | 페이지 넘버]
int TLB_table[32][2]; // TLB 내부에 저장된 테이블 -> [(배열 인덱스) | 페이지 넘버 | 프레임 넘버]
int FIFO_queue[32]; // victim을 결정하기 위한 자료구조 -> [(배열 인덱스) | 페이지 넘버]
```

2) 아래는 Demand Paging 을 구현하기 위해 작성한 코드입니다.
처음에는 아무 것도 올라와 있지 않으므로 -1(Invalid)로 설정합니다.

```
/* 페이지 테이블 초기화 */
for (i = 0; i < 256; i++)
    pageTable[i] = -1; // Demand Paging이므로 처음에는 비어있음

/* 프레임 테이블 초기화 */
for (i = 0; i < 256; i++)
{
    frameTable[i][0] = 0; // FREE임을 표시
    frameTable[i][1] = 0; // 비어있음을 표시
}

/* TLB 내부 테이블 초기화 */
for (i = 0; i < 32; i++)
    for (j = 0; j < 2; j++)
        TLB_table[i][j] = -1; // 비어있음을 표시

/* victim을 결정할 때 이용할 큐를 초기화 */
for (i = 0; i < 32; i++)
    FIFO_queue[i] = -1; // 비어있음을 표시
```

3) 아래의 첫 번째 코드는 논리적 주소가 저장된 파일을 읽어 들이는 처리입니다.
그 아래 세 줄은 각각 물리주소, 프레임테이블, TLB 에 대한 파일을 작성하기 위한 처리입니다.

```
/* 논리적주소 저장파일 읽기 */
FILE* flog = fopen("addresses.txt", "r"); // 논리주소를 읽어들임

/* 결과파일 생성 */
FILE* fphy = fopen("physical_address.txt", "w"); // 물리주소에 대한 파일작성
FILE* fframe = fopen("frame_table.txt", "w"); // 프레임 테이블에 대한 파일작성
FILE* ftlb = fopen("final_TLB.txt", "w"); // 최종적으로 TLB 내부에 저장된 테이블에 대한 파일작성
```

4) 아래는 주석에 작성해 놓은 것과 같이 주소변환을 위한 처리를 하는 코드입니다.
이는 02 번의 설계 아이디어에서도 말씀 드린 것과 같이 논리주소를 읽어오는 처리,
논리주소를 쪼개서 페이지넘버와 오프셋을 구하는 처리, 그리고 TLB 내부 테이블을 찾아보는 처리
그리고 마지막으로 물리주소를 최종적으로 구하는 처리를 각각 코드로 구현하였습니다.

```
/* 주소변환 */
while (!feof(flog))
{
    /* 파일로부터 논리주소 읽기 */
    int logical_addr; // 논리주소를 저장할 변수
    fscanf(flog, "%d\n", &logical_addr); // 파일로부터 논리주소를 입력받음

    // 1) 논리주소를 쪼개서 페이지 넘버와 오프셋을 구함
    int pageNum = logical_addr / 256; // 논리주소의 앞쪽 8비트 -> 256으로 나눈 몫
    int offset = logical_addr % 256; // 논리주소의 뒤쪽 8비트 -> 256으로 나눈 나머지
    paging(pageNum); // 페이징 처리

    // 2) TLB 내부 테이블을 찾아보는 함수 호출
    if (TLB_lookUp(pageNum) == 0)
        paging(pageNum);

    // 3) 물리주소를 최종적으로 구하는 처리
    physical_addr = (frameNum * 256) + offset; // 실제 저장된 물리주소를 구함
    fprintf(fphy, "%d ", physical_addr);

    totalCount++; // 전체 읽어오는 횟수 카운트
}
```

5) 아래는 각각 TLB hit ratio 를 구하는 처리, 그리고 프레임테이블과 TLB 내부 테이블을 파일에 작성하는 처리를 구현한 코드입니다.

```
// 4) TLB hit ratio 구하는 처리
printf("TLB hit ratio : %.2lf\n", (double)hit / totalCount);

// 5) 프레임 테이블을 파일에 작성하는 처리
for (i = 0; i < 256; i++)
    fprintf(fframe, "%3d %3d %3d\r\n", i, frameTable[i][0], frameTable[i][1]);

// 6) 최종 TLB 내부 테이블을 파일에 작성하는 처리
for (i = 0; i < 32; i++)
    fprintf(ftlb, "%3d %3d\r\n", TLB_table[i][0], TLB_table[i][1]);
```

TLB hit ratio 의 경우 아래의 처리에 의해 산출됩니다.

```
int totalCount = 0; // 전체 읽어들인 횟수 카운트
```

논리주소를 읽어올 때마다 카운트를 하나씩 증가시킵니다.

```
/* 주소변환 */
while (!feof(flog))
{
    /* 파일로부터 논리주소 읽기 */
    int logical_addr; // 논리주소를 저장할 변수
    fscanf(flog, "%d\n", &logical_addr); // 파일로부터 논리주소를 입력받음

    // 1) 논리주소를 쪼개서 페이지 넘버와 오프셋을 구함
    int pageNum = logical_addr / 256; // 논리주소의 앞쪽 8비트 -> 256으로 나눈 몫
    int offset = logical_addr % 256; // 논리주소의 뒤쪽 8비트 -> 256으로 나눈 나머지
    paging(pageNum); // 페이징 처리

    // 2) TLB 내부 테이블을 찾아보는 함수 호출
    if (TLB_lookUp(pageNum) == 0)
        paging(pageNum);

    // 3) 물리주소를 최종적으로 구하는 처리
    physical_addr = (frameNum * 256) + offset; // 실제 저장된 물리주소를 구함
    fprintf(fphy, "%d ", physical_addr);

    totalCount++; // 전체 읽어오는 횟수 카운트
}
```

TLB 내부 테이블에 해당 페이지넘버가 존재하는 경우 hit 카운트를 증가시킵니다.

```
// TLB 내부 테이블에서 찾아보는 처리
for (i = 0; i < 32; i++)
{
    // 페이지넘버가 TLB 내부 테이블에 있는 경우
    if (TLB_table[i][0] == pnum)
    {
        printf("TLB hit\n");
        flag = 1; // 존재한다고 플래그 설정
        frameNum = TLB_table[i][1]; // 프레임 넘버를 set
        hit++; // TLB hit count 증가
        break; // 탐색종료
    }
}
```

위와 같은 방법을 통해 얻은 hit 와 totalCount 를 통해 TLB hit ratio 를 산정하여 화면에 출력합니다.

```
// 4) TLB hit ratio 구하는 처리
printf("TLB hit ratio : %.2lf\n", (double)hit / totalCount);
```

6) 아래는 페이지테이블에 해당하는 페이지에 대응되는 프레임넘버가 없는 경우 이를 페이지테이블에 올리는 처리를 구현한 코드입니다. 프레임넘버는 0 부터 시작해서 하나씩 증가시켜가면서 대응하고 이에 따라 페이지테이블과 프레임테이블을 업데이트하는 처리를 수행합니다. 만약 페이지테이블에 해당 페이지넘버가 존재한다면 카운트를 하나 줄인 후 위와 같은 처리를 하도록 설계하였습니다.

```
void paging(int pnum)
{
    static int pagingCount = 0; // 페이징한 횟수 카운트
    if (pageTable[pnum] == -1) // 해당 페이지 넘버가 비어있는 경우
    {
        frameNum = pagingCount;
        pageTable[pnum] = frameNum; // 페이지 테이블에 프레임 넘버를 등록
        frameTable[pagingCount][0] = 1; // 프레임 테이블의 사용여부를 사용중으로 갱신
        frameTable[pagingCount][1] = pnum; // 프레임 테이블에 페이지 넘버 등록
        pagingCount++;
    }
    else
    {
        pagingCount--;
        frameNum = pagingCount;
        pageTable[pnum] = pagingCount;
        frameTable[pagingCount][0] = 1; // 프레임 테이블의 사용여부를 사용중으로 갱신
        frameTable[pagingCount][1] = pnum; // 프레임 테이블에 페이지 넘버 등록
        pagingCount++;
    }
}
```

7) TLB 내부 테이블을 찾아보는 함수는 아래와 같은 처리로 구성되어 있습니다. (아이디어는 02 번 참고)
TLB 내부 테이블에 해당 페이지넘버가 존재하는지 여부를 판단하기 위해 플래그를 만듭니다.

```
int flag = 0; // TLB 내부 테이블에 존재하는지 판단하는 플래그
```

우선 아래와 같이 TLB 내부 테이블에 해당 페이지넘버가 존재하는지 찾아봅니다.
만약에 페이지넘버가 존재하는 경우 프레임넘버를 설정하고 TLB hit 를 증가시킵니다.

```
// TLB 내부 테이블에서 찾아보는 처리
for (i = 0; i < 32; i++)
{
    // 페이지넘버가 TLB 내부 테이블에 있는 경우
    if (TLB_table[i][0] == pnum)
    {
        printf("TLB hit\n");
        flag = 1; // 존재한다고 플래그 설정
        frameNum = TLB_table[i][1]; // 프레임 넘버를 set
        hit++; // TLB hit count 증가
        break; // 탐색종료
    }
}
```

위의 처리를 수행하고도 플래그가 0 인 경우, 즉, TLB 내부 테이블에 페이지넘버가 없는 경우
아래와 같이 페이지테이블에 접근하기에 앞서 TLB 에 저장 가능한 공간이 있는지 없는지 따져봅니다.

```
if (flag == 0) // TLB 내부 테이블에 없는 경우
{
    printf("TLB miss\n");
    // 1_빈 공간 있는 경우
    if (FIFO_count < 32)
    {
        TLB_table[FIFO_count][0] = pnum; // TLB 내부 테이블에 페이지 넘버 저장
        TLB_table[FIFO_count][1] = pageTable[pnum]; // TLB 내부 테이블에 프레임 넘버 저장

        FIFO_queue[FIFO_count] = pnum; // FIFO queue의 마지막에 페이지 넘버 등록
        FIFO_count++; // enqueue가 되었으므로 카운트 증가

        frameNum = TLB_table[FIFO_count][1]; // 프레임 넘버를 set
    }
    // 2_빈 공간 없는 경우
    else
```

```
// 1_빈 공간 있는 경우
if (FIFO_count < 32)
{
    TLB_table[FIFO_count][0] = pnum; // TLB 내부 테이블에 페이지 넘버 저장
    TLB_table[FIFO_count][1] = pageTable[pnum]; // TLB 내부 테이블에 프레임 넘버 저장

    FIFO_queue[FIFO_count] = pnum; // FIFO queue의 마지막에 페이지 넘버 등록
    FIFO_count++; // enqueue가 되었으므로 카운트 증가

    frameNum = TLB_table[FIFO_count][1]; // 프레임 넘버를 set
}
```

위와 같이 빈 공간이 있는 경우 페이지테이블을 읽어 들여 TLB 내부 테이블의 가장 뒷부분(TAIL)에 (페이지넘버, 프레임넘버)의 투플을 추가해줍니다.

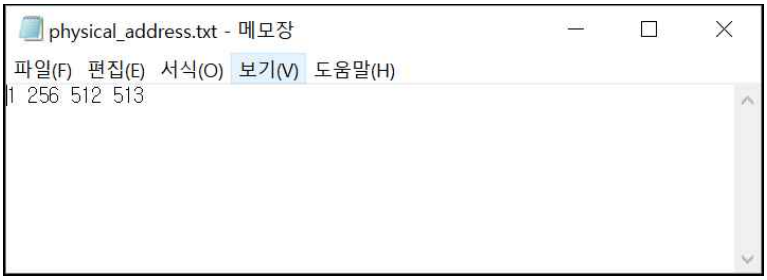
```
// 2_빈 공간 없는 경우
else
{
    // FIFO queue에서 삭제 작업을 수행
    for (i = 0; i < 31; i++)
        FIFO_queue[i] = FIFO_queue[i + 1]; // FIFO queue의 원소를 한 칸씩 앞으로 당김
    FIFO_queue[31] = -1; // 마지막 것이 비어있다 표시
    FIFO_count--; // dequeue가 되었으므로 카운트 감소

    // TLB 내부 테이블에서도 삭제 작업을 수행
    for (i = 0; i < 31; i++)
    {
        TLB_table[i][0] = TLB_table[i + 1][0]; // TLB 내부 페이지 넘버를 한 칸씩 앞으로 당김
        TLB_table[i][1] = TLB_table[i + 1][1]; // TLB 내부 프레임 넘버를 한 칸씩 앞으로 당김
    }

    /* 빈 공간에 새로 넣는 작업 수행 */
    TLB_table[31][0] = pnum; // TLB 내부 테이블에 페이지 넘버 저장
    TLB_table[31][1] = pageTable[pnum]; // TLB 내부 테이블에 프레임 넘버 저장
    FIFO_queue[31] = pnum; // FIFO queue의 마지막에 페이지 넘버 등록
    FIFO_count++; // enqueue가 되었으므로 카운트 증가

    frameNum = TLB_table[31][1]; // 프레임 넘버를 set
}
```

만약 빈 공간이 없는 경우 03 번에서 설명 드린 것과 마찬가지로 FIFO 정책을 통해 우선적으로 victim 을 선택하여 쫓아내는 처리를 한 후 (FIFO 큐와 TLB 내부 테이블 모두에서 수행), 페이지테이블을 읽어 들여 위와 마찬가지로 TLB 내부 테이블의 가장 뒷부분(TAIL)에 (페이지넘버, 프레임넘버)의 투플을 추가해줍니다.

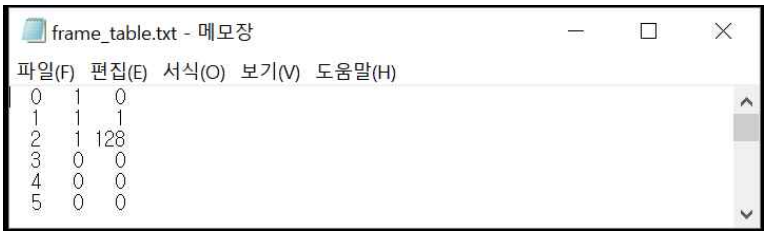
05_실행결과 설명

```
kaitou@DESKTOP-KAITOU ~
$ gcc memory_manager.c

kaitou@DESKTOP-KAITOU ~
$ ./a
TLB miss
TLB miss
TLB miss
TLB hit
TLB hit ratio : 0.25

kaitou@DESKTOP-KAITOU ~
$ ls
a.exe          final_TLB.txt    memory_manager.c  os
addresses.txt  frame_table.txt  network           physical_address.txt
```

프로그램을 컴파일하고 실행하면 위와 같이 TLB hit ratio 가 나옵니다.
또한 미리 주어진 addresses.txt 를 input 으로 하여 final_TLB.txt, frame_table.txt, 그리고 physical_addresses.txt 를 생성합니다.



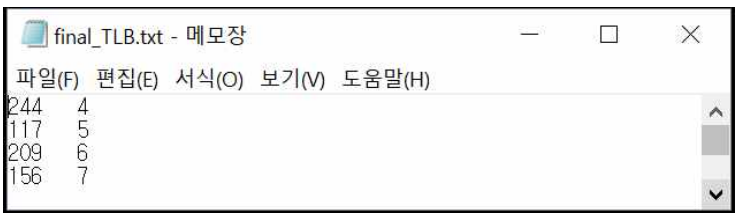
위는 addresses.txt 를 기반으로 산출된 물리주소를 파일로 작성한 것입니다.



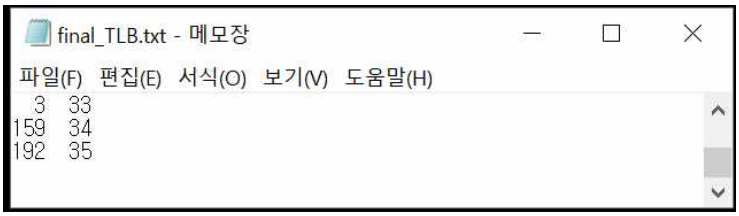
위는 프레임테이블을 파일로 작성한 것입니다. 이는 일부만 캡처한 것으로 전체를 보시려면 .txt 파일을 읽어보셔야 합니다. 첫 번째 열은 프레임넘버, 두 번째 열은 사용여부 (USED 의 경우 1, FREE 의 경우 0), 세 번째 열은 페이지넘버입니다.

위는 모든 처리를 마친 후 TLB 내부 테이블을 파일로 작성한 것입니다.
-1 의 경우 비어있다는 것을 나타냅니다. 첫 번째 열은 페이지넘버, 두 번째 열은 프레임넘버입니다.

위는 TLB 에 있어 FIFO 정책이 제대로 동작하는지 확인하기 위해 임의의 논리주소를 넣은 파일입니다.

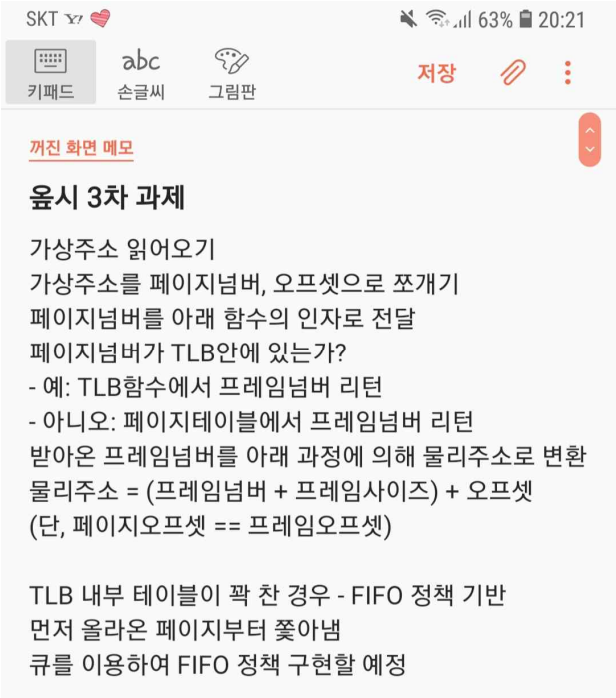


(중간에도 있으나 공간부족으로 인해 필요한 부분만 캡처)



이를 실행하면 TLB 에는 프레임넘버 0, 1, 2, 3 에 해당하는 투플이 FIFO 정책에 의해 제거됨을 알 수 있습니다. (위의 캡처는 설명을 위해 일부만 캡처한 것입니다.)

06_참고자료 (스마트폰 메모장에서 설계한 자료 캡처)




```
## 보다 세부적인 구현방안
1) 가상주소 쪼갬 -> 페이지넘버 + 오프셋
2) TLB 찾아보는 함수 호출 (TLB_lookUp)
#인자로 PageNum를 전달 TLB_lookUp(PageNum);
TLB_lookUp(int pnum)
{
~ TLB 내부 테이블을 찾아보는 처리
~ if(존재) -> 프레임넘버 반환
~ else(없음) -> 페이지테이블 접근
   -> TLB 갱신 - *빈공간 있는 경우 / 없는 경우 (분기)
      빈공간 있는 경우 : (pn, fn)의 레코드를 저장
      빈공간 없는 경우:  FIFO에 의거 쫓아냄
   -> 프레임넘버 반환
}
```

```
## 개념정리
페이지: 논리적주소공간을 동일한 크기로 나눈 것
프레임: 물리적주소공간을 동일한 크기로 나눈 것
페이지테이블: 페이지쪽수 -> 프레임쪽수
각각이 어떻게 대응되는지 관계를 나타내는 테이블
TLB hit: 프레임넘버 즉시 리턴
TLB miss: 페이지테이블 찾아가 아까 없었던 페이지넘
버와 프레임넘버를 TLB 내부에 저장
```

```
## 거시적차원 구현방안
동작흐름: 가상주소 쪼갬 -> TLB 뒤져봄
- (有: 프레임넘버 바로 반환)
- (無: 페이지테이블서 TLB 저장 후* 프레임넘버 반환)
-> 물리적주소 구함
```

```
## 사용할 자료구조 설계
여기서 ( )는 묵시적 인덱스를 의미
1) TLB_table // TLB의 저장공간
- [(#) | 페이지넘버 | 프레임넘버]
2) FIFO_queue // victim 결정에 이용
- [(FIFO 순서) | 페이지넘버]
3) pageTable // 페이지테이블
- [(페이지넘버) | 프레임넘버]
4) frameTable // 프레임테이블
- [(프레임넘버) | 사용여부 | 페이지넘버]
```