OS 과제 보고서

[Designing a Virtual Memory Manager]

12141579 윤찬미
010-9287-1679
cmmcme0604@gmail.com

I. 개요

• 구현의 목적

– Demand paging, Paging, TLB, LRU 에 대해 이해한다

• 요구 사항

– virtual Address에 mapping 되는 Physical Address의 Frame을 순차적으로 할당해준다.
– Page Table을 사용하여 해당 Page가 존재하는 지 확인한다.
– 모든 Page Table을 항상 확인하지 않기 위해 TLB를 사용하여 자주 사용되는 Page Table의 일부를 저장한다.
– TLB Table은 virtual address가 들어올 때 마다 LRU를 사용하여 update를 해준다.
– Frame이 모두 할당 되었을 때, LRU방식을 사용하여 victim을 선택하여 victim을 삭제하고 frame을 할당해준다.
– TLB HIT ratio와 PAGE FAULT ratio를 출력한다.

• 아이디어

– 2Byte(16bit) Address를 입력 받아 union을 사용하여 offset과 page로 나눠준다
– LRU를 stack형식으로 구현을 해야하지만 기존 stack과 다르게 head의 값을 지워야 하고, stack에 해당 page가 존재하면 삭제하고 stack의 TOP에 넣어야 하므로 연결리스트를 사용하여 stack을 구현한다.
– TLB_Table를 LRU 방식으로 update하기 위해 몇번째에 들어왔는지 알고있는 변수를 지정해 둔다.

II.자료 구조

```
typedef union
{
    unsigned short int Add;  //2바이트의 Address를 저장함
    struct  // union을 사용하며 offset과 page를 1바이트씩 가지게 한다.
    {
        unsigned char offset;    // offset = 8bit
        unsigned char page;  // page = 8bit
    };

}address;
```

- Virtual Address를 입력 받았을 때, union을 사용하여 offset과 page를 편하게 나누어 주기 위해 만든 struct 이다.
- Physical Address를 256을 곱해서 구하지 않고, offset과 page를 지정하여 2바이트의 Address를 만들 수 있다.

```
typedef struct list //stack형식
{
    unsigned char page;  // page = 8bit
    unsigned char frame; // frame = 8bit
    struct list *next; // 다음꺼 가리킨다.
}list;
```

- LRU를 Stack형식으로 구현 하기 위해 Linked List를 사용하였다.

```
typedef struct TLB_entry
{
    int lastUseTime;    // LRU형식으로 TLB를 update하기 위해
    unsigned char page;  // page = 8bit
    unsigned char frame; // frame = 8bit
    bool valid;
}TLB_entry;
```

- TLB에서 가지고 있어야 하는 값을 한 번에 관리하기 위해 Struct를 만들었다.

III.함수 설명

```
if((TLB_TABLE_IDX = find_TLB_TABLE(vPage.page, num)) != -1)   //TLB hit 발생 (TLB에 존재하면)
{
    TLB_HIT++; // hit값 증가
    pPage.page = TLB_TABLE[TLB_TABLE_IDX].frame;    //TLB_TABLE이 가지고 있는 page의 frame값을 지정함
    find_and_push(vPage.page);  //LRU를 update함
}
```

- TLB_TABLE에 해당 Page가 존재하는지 확인 한다.
- 존재하면 TLB_HIT 값을 증가 시키고, Frame num을 대입한다.
- LRU를 update 한다.

```
int find_TLB_TABLE(unsigned char page, int num) //TLB에 해당 page가 존재 하는지 확인하는 함수
{
    int i;
    for(i = 0; i < TLB_ENTRY; i++)
    {
        if(!TLB_TABLE[i].valid) break;
        if(TLB_TABLE[i].page == page)   //해당 page가 존재한다면
        {
            TLB_TABLE[i].lastUseTime = num; //언제 썼는지 update 해줌
            return i;                       //i번째에 해당 page가 존재한다.
        }
    }
    return -1;  //존재하지 않는다면 -1을 return 해줌
}
```

- TLB_TABLE에 해당 Page가 존재하는지 확인하는 함수이다.
- 해당 page가 존재하면 위치를 반환해주고, 존재하지 않으면 -1을 반환한다.

```
char find_and_push(unsigned char page)  //LRU stack list 에 해당 page의 위치를 찾고 frame값을 지정하고 LRU에 넣어줌.
{
    unsigned char cframe;   // 존재를 한다면 frame의 위치를 반환한다.
    list *prev = LRU_BEGIN;     //그 전 위치
    list *temp = LRU_BEGIN->next;   //현재 위치
    while(temp != LRU_END)  //LRU stack list에 존재 하는지 확인함
    {
        if(temp->page == page)  //해당 page를 찾으면
        {
            cframe = temp->frame;   //그 frame값을 저장하고,
            prev->next = temp->next;    //자신의 전 list가 나의 다음 것을 가리키고
            free(temp); //해당 list를 LRU에서 삭제 하고 반복문을 탈출
            break;
        }
        prev = prev->next;
        temp = temp->next;
    }                       //LRU에 존재하지 않는다면 끝에 할당, page를 찾으면 삭제하고 끝에 할당!
    push(page, cframe);
    return cframe;
}
```

- LRU에서 해당 page의 위치를 찾는다.
- 찾은 Page를 가지고 있는 temp를 LRU의 가장 끝(TOP)으로 옮긴다.

```
else    //TLB에서 찾지 못함
{
    //TLB table을 갱신 시킨다
    int idx = find_first_TLB();

    if(PAGE_TABLE[vPage.page] == -1) //frame이 할당되지 않았을 때, pagefault 발생
    {
        PAGE_Fault++;   //fault값 증가
        if(STACK_SIZE < P_FRAME)    //비어있는 프레임이 있으면
            STACK_SIZE++;   //stacksize를 증가 시킨다.

        else   //비어있는 프레임이 존재 하지 않는다면
            frame = pop();  //가장 오래전에 사용된 page를 삭제 하고 해당 frame을 할당해준다.

        push(vPage.page, frame++);  //그리고 LRU에 새로운 page를 넣어줌
    }

    else
        find_and_push(vPage.page);  //frame이 할당은 되있다. PAGE_TABLE에서 값이 존재함 = 그 해당 page를 넣어줌

    pPage.page = PAGE_TABLE[vPage.page];    //해당 virtual pagenum에 해당하는 physical pagenum을 넣음
    set_TLB_ENTRY(&TLB_TABLE[idx], num, vPage.page, pPage.page);    //TLB를 갱신 시킴!
}
```

- TLB에서 찾지 못했다면, TLB에 가장 오랫동안 있던 Page를 삭제하고 TLB를 갱신시킨다.

- PageTable에 해당 virtual page에 있는 Physical page가 할당되지 않았다면 frame을 항당하기 위해, 비어있는 프레임이 존재하지 않는다면 LRU를 사용하여 가장 오래된 Page를 POP해준다.
- 비어있는 Frame이 존재한다면 STACK_SIZE (할당된 frame의 갯수)를 증가시킨다.
- 이후 Stack에 Page를 넣어준다.
- PageTable에 해당 virtual page num의 Physical page가 할당 되어 있다면 해당 값으로 page의 frame값을 넣는다.
- TLB를 갱신 시킨다.

```
int find_first_TLB()    //TLB에서 가장 최근에 쓰이지 않는  page를 찾는다.
{
    int i = 0, ret = 0;
    int firstTime = TLB_TABLE[i].lastUseTime;
    for(i = 1; i < TLB_ENTRY; i++)
    {
        if(firstTime > TLB_TABLE[i].lastUseTime)
        {
            firstTime = TLB_TABLE[i].lastUseTime;   //가장 오래전에 쓰였다면 lastUseTime의 값이 작기때문에 그걸로 update 해준다.
            ret = i;    //해당 위치를 저장함
        }
    }
    return ret;
}
```

- TLB를 LRU 방식으로 update 하기 위해 가장 최근에 쓰이지 않은 Page를 찾는다. 이 때, 가장 최근에 쓰이지 않는 page는 그 page가 몇번째에 등장 했는지를 가지고 있는데, 이 것은 Set_TLB_ENTRY를 사용하여 정해준다.
- 가장 오래전에 쓰인 page의 TLB에서의 위치를 반환해준다.

```
void set_TLB_ENTRY(TLB_entry *tlb_entry, int time, unsigned char page, unsigned char frame)
{
    tlb_entry->lastUseTime = time;  //언제 사용 되었는지 저장
    tlb_entry->page = page; //page번호
    tlb_entry->frame = frame;   //frame 번호
    tlb_entry->valid = true;    //사용 됐다면 valid 하다고 함
}
```

- TLB_TABLE에서 TLB_ENTRY 값을 지정해주는 함수이다.
- Time은 입력을 받으면서 num번째로 입력 받았다 라는 것으로 갱신을 해준다,
- Page,Frame num을 가지고 있으며 valid하다고 지정해준다.

```
void push(unsigned char page, unsigned char frame)  //LRU에 새로운 값을 넣는다.
{
    list *NEWEND = (list*)malloc(sizeof(list)); //다시 끝(TOP)이라고 지정할 list를 만든다.
    PAGE_TABLE[page] = frame;   //pagetable의 page번째에 frame값을 저장함
    LRU_END->page = page;   //가장 최근에 들어온 것이기 때문에 stack의 가장 끝(TOP)에 들어간다.
    LRU_END->frame = frame; //frame 값을 저장한다.
    LRU_END->next = NEWEND; //새롭게 만든 빈 list를 END의 next에 단다
    LRU_END = LRU_END->next;    //END의 값을 바꿔준다
}
```

- LRU의 끝에 새로운 값을 넣어준다. (STACK의 TOP)
- LRU의 BEGIN과 END는 빈 공간으로 설정해주었다.
- END에 새로운 값을 넣어 준 후 다시 빈 공간으로 END를 바꿔주었다.

```
char pop()//가장 최근에 쓰이지 않은 (stack의 가장 바닥에 있는) list를 삭제한다.
{
    list *temp = LRU_BEGIN->next;   //Begin은 빈 list이다 그 다음에 있는 것 = 가장최근에 쓰이지 않은 것을 삭제 함
    unsigned char cframe = PAGE_TABLE[temp->page];
    PAGE_TABLE[temp->page] = -1;    //pageTable에서도 삭제 함 -1 로 바꿔줌

    LRU_BEGIN->next = temp->next;   //가장 최근에 쓰이지 않은 것을 지금 삭제 할 다음 것이라고 지정
    free(temp); //메모리 할당을 해제해줌
    return cframe;  //그리고 frame num 반환
}
```

- 가장 최근에 쓰이지 않은 Page를 삭제해준다.
- BEGIN은 빈 공간 이기 때문에 그 다음 것이 가장 최근에 쓰이지 않은 Page이기 때문에 BEGIN의 NEXT를 삭제할 temp의 NEXT로 바꿔 준 후 삭제 해 준다.

IV. 출력 결과

```
Virtual address : 16916 Physical address : 20
Virtual address : 62493 Physical address : 285
Virtual address : 30198 Physical address : 758
Virtual address : 53683 Physical address : 947
Virtual address : 40185 Physical address : 1273
Virtual address : 28781 Physical address : 1389
Virtual address : 24462 Physical address : 1678
Virtual address : 48399 Physical address : 1807
Virtual address : 64815 Physical address : 2095
Virtual address : 18295 Physical address : 2423
Virtual address : 12218 Physical address : 2746
```

V.평가 및 개선 방향

- TLB와 PAGE_TABLE, LRU 를 사용하여 DEMEND PAGING을 보다 효과적으로 구성하였다.
- Linked Listed 를 사용하여 Paging에서 사용하는 Stack을 구현하였다.
- STL을 사용하여 쉬운 구현을 가능하게 할 수 있다.
- 직관적인 변수명과 함수명을 사용하여 이해를 보다 쉽게 한다.
- TLB는 update 할 때, LRU가 아닌 보다 효과적인 방법을 사용한다.
- 자료구조를 Tree와 같은 것을 사용하면 더욱 직관적이며 성능 향상이 좋을 것으로 생각한다.

- LRU나 TLB에 저장된 page의 값을 보다 빠르게 찾을 수 있는 방법이 없을지 생각해 본다.

VI.과제 수행시 알게된 점

- TLB를 update하는 방법을 알게되었다.
- Stack 형태로 LRU를 구현할 때, 배열이 아닌 Linked List를 사용 하는 것이 효과 적이라는 것을 알았다.
- Demend Paing을 하는 순서를 보다 명확하게 이해 할 수 있었다.
- TLB HIT RATIO, PAGE FAULT RATIO에 대해 더욱 이해할기에 명확하였다.
- Page Replacement시 최적화된 알고리즘을 사용을 해야 보다 효과적으로 Demend Paging을 할 수 있다 는 것을 알게되었다.