Operating Systems

Designing a Virtual Memory Manager

컴퓨터공학과

12121451

김수환

**< 목 차 >**

1. **개요**
2. **정의**
3. Demand Paging & Page Fault
4. Address Translation
5. Page Replacement
6. **구현**
7. Virtual Memory Manager
8. **개발환경**
9. **개요**

CPU가 Demand Paging을 요청할 경우 생성된 Virtual Address를 Physical Address로 변환하는 방법을 이해하여 Virtual Memory Manager를 직접 구현합니다. 그리고 Page Fault가 발생하였을 경우도 직접 구현하여 Address Translation의 과정을 익힙니다.

1. **정의**
2. Demand Paging & Page Fault

Demand Paging이란 실제로 필요한 page만 Physical Memory로 가져오는 방식입니다.

즉, 요청이 있을 때 Paging을 수행합니다. Demand Paging시에 해당 page가 Physical

Memory에 없다면 Page Fault가 발생합니다. Page Fault시 처리 과정은 다음과 같습니다.

1) Page Table의 Valid Bit가 invalid인 경우 OS에게 Trap으로 Page Fault를 알립니다.

2) Frame Table을 Look Up하여 비어있는 Frame을 확보합니다.

3) Disk에 접근하여 Page를 Physical Memory로 올립니다. ( Page-in / Page-out )

4) Physical Memory로 올린 후, Page Table을 Update합니다.

5) Page Fault를 야기한 Instruction부터 다시 수행합니다.

Demand Paging방식을 이용하면 처음에는 Multiple Page Fault가 발생하지만, 그 이후에는 “Locality of reference” 성질에 의해 Page Fault가 발생할 가능성이 감소합니다.

1. Address Translation

Address Translation은 CPU의 Virtual Address를 Physical Address로 변환하는 것을 뜻합니다. CPU의 Virtual Address를 Page Number Bit와 Offset Bit로 나누고, Page Table에 접근하여 Frame Number를 찾습니다. 그리고 찾은 Frame Number와 Offset을 합쳐서 Physical Address로 변환합니다. 이와 같은 과정을 Address Translation이라고 합니다.

1. Page Replacement

Virtual Memory에 비해 Physical Memory는 크기가 작습니다. 따라서 Page Fault가 자주 발생합니다. Page Fault가 발생하면 Disk에 접근해야 하기 때문에 시간이 오래 걸립니다. 따라서 Page Fault를 최소화 즉, 지연 시간을 최소화하기 위해 등장한 방법이 Page Replacement입니다. Page Replacement의 몇 가지 주요한 알고리즘을 살펴보겠습니다.

1) FIFO algorithm

FIFO algorithm은 가장 먼저 Page-in된 즉, Physical Memory에 오래 머문 page를

Page-out시키는 방법입니다. 프로그래밍이 단순하고 깔끔하지만, locality를 생각하면

비효율적인 알고리즘입니다.

2) LRU (Least Recently Used) algorithm

LRU algorithm은 Physical Memory에서 가장 오랫동안 사용되지 않은 Page를 Page-

out시키는 방법입니다. LRU 구현 방법 중에 제가 사용한 방법은 Stack implementation

입니다. Stack implementation은 일종의 Queue와 비슷하게 수행되지만 한가지가 다릅

니다. Queue의 중간에 있는 Page가 다시 한번 Demand된다면 최근에 사용한 것이므

로 위치를 Queue의 상단으로 옮겨줘야 합니다. 따라서 가장 오랫동안 사용되지 않은

Page는 Queue의 하단에 위치하게 됩니다. 바로 이 Page를 Page-out시키면 되므로

Search할 필요가 없습니다. 따라서 가장 많이 사용되고, 효율적인 알고리즘입니다.

3) LRU approximation algorithm

LRU approximation algorithm 중의 하나인 Second Chance Algorithm에 대하여 알아

보겠습니다. Second Chance Algorithm은 LRU와 비슷한 개념이지만, 하드웨어의 도움

을 받아 Reference Bit를 사용합니다. Page의 집합을 Circular Queue형태로 나타내고

만약 Reference가 발생하면 Bit를1로 바꿉니다. 그리고 Victim Page를 선정할 때는

Clock hand가 하나씩 탐색하여 Reference Bit를 확인합니다. 만약 Bit가 0이면 해당

Page가 Victim이 되고, 1이면 0으로 바꾼 후 다음 Page를 확인합니다. 해당 Page에

2번의 기회를 주 는 것이므로 Second Chance Algorithm이라고 명명되었습니다.

이 방식도 효율이 좋은 것으로 알려져 있습니다.

4) Counter-Based algorithm

Counter-Based algorithm은 Page가 Reference될 때마다 Counting하는 방식입니다.

* LFU : 가장 적게 Reference된 Page를 Victim으로 선정하는 방식
* MFU : 가장 많이 Reference된 Page를 Victim으로 선정하는 방식

1. **구현**
2. **Virtual Memory Manager**

**요구 조건**

1) Page Table은 256개의 entry, Frame Table은 128개의 entry를 갖는다.

2) Page와 Frame의 Size는 256Bytes이다.

3) TLB는 16개의 entry를 갖는다.

4) Page Replacement Algorithm으로 LRU와 FIFO를 사용합니다.

5) TLB Replacement Algorithm은 LRU를 사용합니다.

6) Page Fault가 발생할 경우 BACKING\_STORE.bin에서 256Bytes의 페이지를 읽고,

해당 프레임에 저장합니다.

**사용한 변수 및 함수 설명**

* **typedef enum {false, ture} bool**

C언어에서는 bool 자료형이 존재하지 않으므로 직접 정의.

* **bool inTLB**

TLB에 demand한 Page가 있으면 true, 없으면 false

* **int demandCount**

Demand paging의 횟수

* **int pageTable[256][2]**

각 Page의 Frame Number와 valid여부를 저장

* **unsigned char physicalMemory[128][256]**

Physical Memory 구조를 배열로 구현 (entry : 128, frame size : 256Bytes)

* **int emptyFrameNum**

Victim으로 선정된 Page의 Frame Number를 저장

* **struct entry {**

**int pageNum;**

**int frameNum;**

**int validation;**

**int offset;**

**struct entry\* next;**

**struct entry\* prev;**

**}**

Page와 Frame을 구조체로서 구현

* **struct table {**

**int size;**

**int hit;**

**struct entry\* head;**

**struct entry\* tail;**

**} frameTable, TLB;**

Frame Table과 TLB를 2중 연결리스트로 구현, 각각 Table마다 hit를 가짐

* **struct entry\* selectVictim(struct table\*)**

Table을 매개변수로 받고, 해당 Table의 Victim Page를 선정하여 반환하는 함수

* **void lookUpTLB(int)**

Page Number를 매개변수로 받고, TLB Look Up하고, LRU(Stack implementation)로 관하는 함수

* **struct entry\* LRU\_demandPaging(int, int)**

Page Number와 offset을 매개변수로 받고, Demand Paging시에 Page Replacement algorithm을 LRU로 총괄적인 작업을 수행하는 함수

* **struct entry\* FIFO\_demandPaging(int, int)**

바로 위의 함수와 같지만 FIFO algorithm을 이용하는 함수

* **void upToFrame(int, int)**

Page Fault시에 BACKING\_STORE.bin에서 해당 Page의 데이터를 해당 Frame에

저장하는 함수

**동작 원리**

**1) TLB : LRU algorithm, Page Replacement : LRU algorithm**

먼저 Page Replacement에 LRU를 적용하여 Address Translation하는 과정을 설명하겠습니다.

먼저 입력받은 addresses.txt 파일을 main()에서 열고, Virtual Address를 받아옵니다. 그리고 하위 16bit 중에 하위 8비트는 offset 변수에 저장하고, 상위 8비트는 pNum변수에 저장합니다. 이 후에 LRU\_demandPaging()를 호출하여 Demand Paging과정을 수행합니다.

Demand Paging이 수행되는 과정을 살펴보면 먼저 inTLB변수를 false로 초기화 합니다. 그리고 Demand Paging이 발생하였으니 demandCount변수가 1증가되고, 먼저 lookUpTLB()를 호출하여 TLB를 Look Up합니다. lookUpTLB()로 제어가 넘어가면 먼저 TLB가 비어있을 경우와 그렇지 않을 경우로 나눠집니다.

* TLB가 비어있을 경우 TLB에 entry를 동적 할당하여 추가합니다. entry의 data인 pageNum은 lookUpTLB()의 매개변수로 받은 pNum이 저장되고, TLB가 비어있다면 frame도 모두 비어있으므로 frameNum에 0을 저장합니다. 그리고 TLB를 2중 연결리스트로 구현하였기 때문에 head와 tail이 새롭게 추가된 entry로 설정됩니다.
* TLB가 비어있지 않을 경우 TLB를 탐색하여 Demand한 Page가 있는지 확인합니다. 먼저 TLB의 head, tail, 그리고 head와 tail의 사이에 Demand한 Page가 있는지 확인합니다. 만약에 Demand한 Page가 TLB에 있다면 inTLB의 값을 true로 바꿔줍니다. 그리고 TLB에서 Page(entry)를 찾은 경우 즉, inTLB값이 true인 경우에는 LRU(Stack implementation)를 적용시켰기 때문에 찾은 Page(entry)를 TLB의 tail로 옮겨줍니다. 그리고 TLB가 HIT되었으므로 TLB의 hit값을 1증가시킵니다. 반면 TLB에서 Page를 찾지 못한 경우 즉, inTLB의 값이 false인 경우에는 다시 2가지 경우로 나눠집니다. 먼저 TLB의 entry가 가득 차있을 경우 즉, entry가 16개일 경우에는 selectVictim()을 호출하여 TLB에서 Victim Page(entry)를 선정합니다. 그리고 삽입할 entry의 pageNum에 매개변수로 받은 pNum을 저장하고, Empty Frame을 아직 모르니 frameNum에 -1을 저장하고 후에 Page가 올라갈 Frame이 결정되면 다시 갱신합니다. 그리고 이 entry를 TLB의 tail로 삽입합니다. 이번에는 TLB가 가득 차지 않은 경우에는 Page가 frame에 차곡차곡 저장되므로 entry의 pageNum은 매개변수 pNum로, frameNum는 현재 TLB entry의 수로 저장됩니다. 그리고 lookUpTLB() 탈출합니다.

이제 다시 LRU\_demandPaging()로 제어가 넘어오면 TLB에서 HIT일 경우와 MISS일 경우로 나눠집니다.

* TLB에서 HIT일 경우 즉, inTLB의 값이 true이면 Demand한 Page가 Physical Memory에 있다는 뜻이므로 Frame Table만 업데이트해주면 됩니다. 역시 Frame Table도 2중 연결리스트로 구현되어있기 때문에 Demand한 Page가 head, tail, head와 tail사이에 있을 경우로 나누어서 해당 Page(entry)를 찾습니다. 그리고 역시 LRU(Stack implementation)를 적용시켰기 때문에 찾은 Page(entry)를 Frame Table의 tail로 옮겨줍니다. 그리고 찾은 Page(entry)를 반환하고 함수를 탈출합니다.
* TLB에서 MISS일 경우 즉, inTLB의 값이 false이면 Page Table에 접근하여 Frame Table에 Demand한 Page가 있는지 확인해야 합니다. 먼저 맨 처음 Demand Paging일 경우와 그 이후의 Demand Paging일 경우로 나눠집니다.

맨 처음 Demand Paging일 경우, Page Fault가 발생하므로 Frame에 Page를 올려야 합니다. 즉, Frame Table에 Demand한 Page(entry)를 삽입해야 합니다. 따라서 동적 할당한 entry의 pageNum는 매개변수 pNum, frameNum은 0, validation에 1을 저장합니다. 그리고 Frame Table의 head와 tail을 이 entry로 설정합니다. Page Fault가 발생하였으므로 upToFrame()을 호출하여 0번째 frame에 Demand한 Page를 올립니다. 그리고 Page Table배열의 Demand한 Page를 업데이트합니다. (Frame Number(0), valid(1)) 마지막으로 삽입된 entry를 반환하고 함수를 탈출합니다.

두 번째 이후의 Demand Paging일 경우, Page Table배열에서 Demand한 Page의 Valid/Invalid에 따라 경우가 나누어집니다. 먼저 Invalid일 경우 (MISS), 역시 Page Fault가 발생합니다. 따라서 미리 새로운 entry를 동적 할당합니다. 그리고 만약 Frame Table이 가득 차있다면 Victim Page(entry)를 선정하고upToFrame()을 호출하여 Victim Page의 Frame에 Demand한 Page를 올립니다. 그리고 Page Table배열을 업데이트합니다. (Frame Number, valid) 또 Page Table배열에서 Victim Page를 invalid(0)로 업데이트합니다. 그리고 앞서 동적 할당된 entry에 데이터를 저장하고, Frame Table의 tail에 삽입합니다. 만약 Frame Table이 가득 차있지 않다면 Demand한 Page가 frame에 차곡차곡 저장됩니다. 따라서 동적 할당된 entry에 데이터를 저장하고 upToFrame()을 호출하여 Frame에 Page를 차곡차곡 올립니다. 이후 Page Table를 업데이트하고, Frame Table의 tail에 entry를 삽입합니다. 이 2가지 경우에 따라 과정을 거치면 마침내 Page가 올라갈 Frame Number가 결정되므로, 앞서 TLB에서 frameNum을 -1로 초기화했던 데이터를 찾은 Frame Number로 갱신합니다. 그리고 삽입된 entry를 반환하고 함수를 탈출합니다. 이번에는 valid인 경우를 살펴보겠습니다. Valid인 경우 (HIT), hit값을 1증가 시킵니다. 그리고 Frame Table에서 Demand한 Page(entry)를 찾아 Frame Table의 tail에 삽입하고 찾은 Page(entry)를 반환하고 함수를 탈출합니다.

이제 다시 main()으로 제어가 넘어오면 LRU\_demandPaging()으로부터 반환된 entry를 struct entry\* result에 저장합니다. 그리고 과제에 필요한 파일로 데이터를 저장하고 모든 과정이 끝납니다.

**2) TLB : LRU algorithm, Page Replacement : FIFO algorithm**

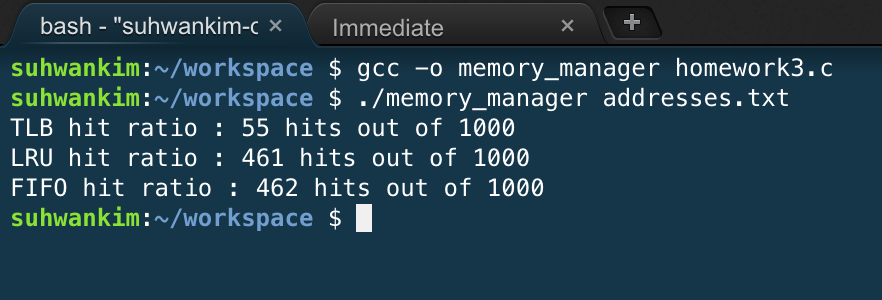
다음으로 Page Replacement에 FIFO를 적용하여 Address Translation하는 과정을 살펴보는데 1) 의 과정과 비슷하므로 간략하게 다른 점을 짚어보는 방식으로 진행하겠습니다.

1) 의 과정이 모두 끝나면 사용한 모든 변수와 Table을 초기화한 후에 새롭게 다시 FIFO\_demandPaging()을 호출하여 Address Translation과정이 진행됩니다.

먼저 lookUpTLB()를 호출하여 TLB를 Look Up합니다. Look Up을 마치고, 맨 처음 Demand Paging일 경우 1) 의 같은 경우의 과정이 진행된 후. 삽입된 entry를 반환하고 함수를 탈출합니다. 두 번째 이후의 Demand Paging일 경우 앞서 설명하였듯이 Page Table배열에서 Demand한 Page의 Valid/Invalid에 따라 경우가 나누어집니다. 먼저 Invalid인 경우 1) 의 같은 경우의 과정이 진행되지만, Valid인 경우 약간 다릅니다. Page Replacement에 FIFO를 적용시키면 Valid인 경우 LRU와 다르게 Frame Table을 관리하지 않으므로 그냥 새로운 entry를 동적 할당한 후 Page Table에 저장된 Frame Number와 offset을 참조하여 entry에 저장합니다. 그리고 entry를 반환하고 함수를 탈출합니다. 그리고 역시 FIFO\_demandPaging()으로부터 반환된 entry를 struct entry\* result에 저장합니다. 그리고 과제에 필요한 파일로 데이터를 저장하고 모든 과정이 끝납니다.

**위의 과정을 통해 요구조건 6가지를 모두 만족하도록 구현하였습니다.**

**결과**

****

**LRU / FIFO 각각 3개씩 파일 생성, 총 6개의 파일도 정상적으로 생성되는 것을 확인하였습니다.**

1. **개발 환경**

OS : Ubuntu 14.04.3

Compiler : gcc version 4.8.4

Language : C