Operating Systems

Designing a Virtual Memory Manager

컴퓨터공학과
12121451
김수환

< 목 차 >

1. 개요

CPU 가 Demand Paging 을 요청할 경우 생성된 Virtual Address 를 Physical Address 로 변환하는 방법을 이해하여 Virtual Memory Manager 를 직접 구현합니다. 그리고 Page Fault 가 발생하였을 경우도 직접 구현하여 Address Translation 의 과정을 익힙니다.

2. 정의

(1) Demand Paging & Page Fault

Demand Paging 이란 실제로 필요한 page 만 Physical Memory 로 가져오는 방식입니다. 즉, 요청이 있을 때 Paging 을 수행합니다. Demand Paging 시에 해당 page 가 Physical Memory 에 없다면 Page Fault 가 발생합니다. Page Fault 시 처리 과정은 다음과 같습니다.

1) Page Table 의 Valid Bit 가 invalid 인 경우 OS 에게 Trap 으로 Page Fault 를 알립니다.
2) Frame Table 을 Look Up 하여 비어있는 Frame 을 확보합니다.
3) Disk 에 접근하여 Page 를 Physical Memory 로 올립니다. (Page-in / Page-out)
4) Physical Memory 로 올린 후, Page Table 을 Update 합니다.
5) Page Fault 를 야기한 Instruction 부터 다시 수행합니다.

Demand Paging 방식을 이용하면 처음에는 Multiple Page Fault 가 발생하지만, 그 이후에는 “Locality of reference” 성질에 의해 Page Fault 가 발생할 가능성이 감소합니다.

(2) Address Translation

Address Translation 은 CPU 의 Virtual Address 를 Physical Address 로 변환하는 것을 뜻합니다. CPU 의 Virtual Address 를 Page Number Bit 와 Offset Bit 로 나누고, Page Table 에 접근하여 Frame Number 를 찾습니다. 그리고 찾은 Frame Number 와 Offset 을 합쳐서 Physical Address 로 변환합니다. 이와 같은 과정을 Address Translation 이라고 합니다.

(3) Page Replacement

Virtual Memory 에 비해 Physical Memory 는 크기가 작습니다. 따라서 Page Fault 가 자주 발생합니다. Page Fault 가 발생하면 Disk 에 접근해야 하기 때문에 시간이 오래 걸립니다. 따라서 Page Fault 를 최소화 즉, 지연 시간을 최소화하기 위해 등장한 방법이 Page Replacement 입니다. Page Replacement 의 몇 가지 주요한 알고리즘을 살펴보겠습니다.

1) FIFO algorithm

FIFO algorithm 은 가장 먼저 Page-in 된 즉, Physical Memory 에 오래 머문 page 를

Page-out 시키는 방법입니다. 프로그래밍이 단순하고 깔끔하지만, locality 를 생각하면 비효율적인 알고리즘입니다.

2) LRU (Least Recently Used) algorithm

LRU algorithm 은 Physical Memory 에서 가장 오랫동안 사용되지 않은 Page 를 Page-out 시키는 방법입니다. LRU 구현 방법 중에 제가 사용한 방법은 Stack implementation 입니다. Stack implementation 은 일종의 Queue 와 비슷하게 수행되지만 한가지가 다릅니다. Queue 의 중간에 있는 Page 가 다시 한번 Demand 된다면 최근에 사용한 것이므로 위치를 Queue 의 상단으로 옮겨줘야 합니다. 따라서 가장 오랫동안 사용되지 않은 Page 는 Queue 의 하단에 위치하게 됩니다. 바로 이 Page 를 Page-out 시키면 되므로 Search 할 필요가 없습니다. 따라서 가장 많이 사용되고, 효율적인 알고리즘입니다.

3) LRU approximation algorithm

LRU approximation algorithm 중의 하나인 Second Chance Algorithm 에 대하여 알아보겠습니다. Second Chance Algorithm 은 LRU 와 비슷한 개념이지만, 하드웨어의 도움을 받아 Reference Bit 를 사용합니다. Page 의 집합을 Circular Queue 형태로 나타내고 만약 Reference 가 발생하면 Bit 를 1 로 바꿉니다. 그리고 Victim Page 를 선정할 때는 Clock hand 가 하나씩 탐색하여 Reference Bit 를 확인합니다. 만약 Bit 가 0 이면 해당 Page 가 Victim 이 되고, 1 이면 0 으로 바꾼 후 다음 Page 를 확인합니다. 해당 Page 에 2 번의 기회를 주는 것이므로 Second Chance Algorithm 이라고 명명되었습니다.
이 방식도 효율이 좋은 것으로 알려져 있습니다.

4) Counter-Based algorithm

Counter-Based algorithm 은 Page 가 Reference 될 때마다 Counting 하는 방식입니다.
- LFU : 가장 적게 Reference 된 Page 를 Victim 으로 선정하는 방식
- MFU : 가장 많이 Reference 된 Page 를 Victim 으로 선정하는 방식

3. 구현

(1) Virtual Memory Manager

① 요구 조건

1) Page Table 은 256 개의 entry, Frame Table 은 128 개의 entry 를 갖는다.

2) Page 와 Frame 의 Size 는 256Bytes 이다.

3) TLB 는 16 개의 entry 를 갖는다.

4) Page Replacement Algorithm 으로 LRU 와 FIFO 를 사용합니다.

5) TLB Replacement Algorithm 은 LRU 를 사용합니다.

6) Page Fault 가 발생할 경우 BACKING_STORE.bin 에서 256Bytes 의 페이지를 읽고, 해당 프레임에 저장합니다.

② 사용한 변수 및 함수 설명

- typedef enum {false, ture} bool

  C 언어에서는 bool 자료형이 존재하지 않으므로 직접 정의.

- bool inTLB

  TLB 에 demand 한 Page 가 있으면 true, 없으면 false

- int demandCount

  Demand paging 의 횟수

- int pageTable[256][2]

  각 Page 의 Frame Number 와 valid 여부를 저장

- unsigned char physicalMemory[128][256]

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

- int emptyFrameNum

  Victim 으로 선정된 Page 의 Frame Number 를 저장

- ```
  struct entry {
      int pageNum;
      int frameNum;
      int validation;
      int offset;
      struct entry* next;
      struct entry* prev;
  }
  ```

  Page 와 Frame 을 구조체로서 구현

- ```
  struct table {
      int size;
      int hit;
      struct entry* head;
      struct entry* tail;
  } frameTable, TLB;
  ```

  Frame Table 과 TLB 를 2 중 연결리스트로 구현, 각각 Table 마다 hit 를 가짐

- struct entry* selectVictim(struct table*)

  Table 을 매개변수로 받고, 해당 Table 의 Victim Page 를 선정하여 반환하는 함수

- void lookUpTLB(int)

  Page Number 를 매개변수로 받고, TLB Look Up 하고, LRU(Stack implementation)로 관하는 함수

- struct entry* LRU_demandPaging(int, int)

  Page Number 와 offset 을 매개변수로 받고, Demand Paging 시에 Page Replacement algorithm 을 LRU 로 총괄적인 작업을 수행하는 함수

- struct entry* FIFO_demandPaging(int, int)

  바로 위의 함수와 같지만 FIFO algorithm 을 이용하는 함수

- void upToFrame(int, int)

  Page Fault 시에 BACKING_STORE.bin 에서 해당 Page 의 데이터를 해당 Frame 에 저장하는 함수

③ 동작 원리

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

먼저 Page Replacement 에 LRU 를 적용하여 Address Translation 하는 과정을 설명하겠습니다.

먼저 입력받은 addresses.txt 파일을 main()에서 열고, Virtual Address 를 받아옵니다. 그리고 하위 16bit 중에 하위 8 비트는 offset 변수에 저장하고, 상위 8 비트는 pNum 변수에 저장합니다. 이 후에 LRU_demandPaging()를 호출하여 Demand Paging 과정을 수행합니다.

Demand Paging 이 수행되는 과정을 살펴보면 먼저 inTLB 변수를 false 로 초기화 합니다. 그리고 Demand Paging 이 발생하였으니 demandCount 변수가 1 증가되고, 먼저 lookUpTLB()를 호출하여 TLB 를 Look Up 합니다. lookUpTLB()로 제어가 넘어가면 먼저 TLB 가 비어있을 경우와 그렇지 않을 경우로 나눠집니다.

- TLB 가 비어있을 경우 TLB 에 entry 를 동적 할당하여 추가합니다. entry 의 data 인 pageNum 은 lookUpTLB()의 매개변수로 받은 pNum 이 저장되고, TLB 가 비어있다면 frame 도 모두 비어있으므로 frameNum 에 0 을 저장합니다. 그리고 TLB 를 2 중 연결리스트로 구현하였기 때문에 head 와 tail 이 새롭게 추가된 entry 로 설정됩니다.

- TLB 가 비어있지 않을 경우 TLB 를 탐색하여 Demand 한 Page 가 있는지 확인합니다. 먼저 TLB 의 head, tail, 그리고 head 와 tail 의 사이에 Demand 한 Page 가 있는지 확인합니다. 만약에 Demand 한 Page 가 TLB 에 있다면 inTLB 의 값을 true 로 바꿔줍니다. 그리고 TLB 에서 Page(entry)를 찾은 경우 즉, inTLB 값이 true 인 경우에는 LRU(Stack implementation)를 적용시켰기 때문에 찾은 Page(entry)를 TLB 의 tail 로 옮겨줍니다. 그리고 TLB 가 HIT 되었으므로 TLB 의 hit 값을 1 증가시킵니다. 반면 TLB 에서 Page 를 찾지 못한 경우 즉, inTLB 의 값이 false 인 경우에는 다시 2 가지 경우로 나눠집니다. 먼저 TLB 의 entry 가 가득 차있을 경우 즉, entry 가 16 개일 경우에는 selectVictim()을 호출하여 TLB 에서 Victim Page(entry)를 선정합니다. 그리고 삽입할 entry 의 pageNum 에 매개변수로 받은 pNum 을 저장하고, Empty Frame 을 아직 모르니 frameNum 에 -1 을 저장하고 후에 Page 가 올라갈 Frame 이 결정되면 다시 갱신합니다. 그리고 이 entry 를 TLB 의 tail 로 삽입합니다. 이번에는 TLB 가 가득 차지 않은 경우에는 Page 가 frame 에 차곡차곡 저장되므로 entry 의 pageNum 은 매개변수 pNum 로, frameNum 는 현재 TLB entry 의 수로 저장됩니다. 그리고 lookUpTLB() 탈출합니다.

이제 다시 LRU_demandPaging()로 제어가 넘어오면 TLB 에서 HIT 일 경우와 MISS 일 경우로 나눠집니다.

- TLB 에서 HIT 일 경우 즉, inTLB 의 값이 true 이면 Demand 한 Page 가 Physical Memory 에 있다는 뜻이므로 Frame Table 만 업데이트해주면 됩니다. 역시 Frame Table 도 2 중 연결리스트로 구현되어있기 때문에 Demand 한 Page 가 head, tail, head 와 tail 사이에 있을 경우로 나누어서 해당 Page(entry)를 찾습니다. 그리고 역시 LRU(Stack implementation)를 적용시켰기 때문에 찾은 Page(entry)를 Frame Table 의 tail 로 옮겨줍니다. 그리고 찾은 Page(entry)를 반환하고 함수를 탈출합니다.
- TLB 에서 MISS 일 경우 즉, inTLB 의 값이 false 이면 Page Table 에 접근하여 Frame Table 에 Demand 한 Page 가 있는지 확인해야 합니다. 먼저 맨 처음 Demand Paging 일 경우와 그 이후의 Demand Paging 일 경우로 나눠집니다.
  맨 처음 Demand Paging 일 경우, Page Fault 가 발생하므로 Frame 에 Page 를 올려야 합니다. 즉, Frame Table 에 Demand 한 Page(entry)를 삽입해야 합니다. 따라서 동적 할당한 entry 의 pageNum 는 매개변수 pNum, frameNum 은 0, validation 에 1 을 저장합니다. 그리고 Frame Table 의 head 와 tail 을 이 entry 로 설정합니다. Page Fault 가 발생하였으므로 upToFrame()을 호출하여 0 번째 frame 에 Demand 한 Page 를 올립니다. 그리고 Page Table 배열의 Demand 한 Page 를 업데이트합니다. (Frame Number(0), valid(1)) 마지막으로 삽입된 entry 를 반환하고 함수를 탈출합니다.
  두 번째 이후의 Demand Paging 일 경우, Page Table 배열에서 Demand 한 Page 의 Valid/Invalid 에 따라 경우가 나누어집니다. 먼저 Invalid 일 경우 (MISS), 역시 Page Fault 가 발생합니다. 따라서 미리 새로운 entry 를 동적 할당합니다. 그리고 만약 Frame Table 이 가득 차있다면 Victim Page(entry)를 선정하고 upToFrame()을 호출하여 Victim Page 의 Frame 에 Demand 한 Page 를 올립니다. 그리고 Page Table 배열을 업데이트합니다. (Frame Number, valid) 또 Page Table 배열에서 Victim Page 를 invalid(0)로 업데이트합니다. 그리고 앞서 동적 할당된 entry 에 데이터를 저장하고, Frame Table 의 tail 에 삽입합니다. 만약 Frame Table 이 가득 차있지 않다면 Demand 한 Page 가 frame 에 차곡차곡 저장됩니다. 따라서 동적 할당된 entry 에 데이터를 저장하고 upToFrame()을 호출하여 Frame 에 Page 를 차곡차곡 올립니다. 이후 Page Table 를 업데이트하고, Frame Table 의 tail 에 entry 를 삽입합니다. 이 2 가지 경우에 따라 과정을 거치면 마침내 Page 가 올라갈 Frame Number 가 결정되므로, 앞서 TLB 에서 frameNum 을 -1 로 초기화했던 데이터를 찾은 Frame Number 로 갱신합니다. 그리고 삽입된 entry 를 반환하고 함수를 탈출합니다. 이번에는 valid 인 경우를 살펴보겠습니다. Valid 인 경우 (HIT), hit 값을 1 증가시킵니다. 그리고 Frame Table 에서 Demand 한 Page(entry)를 찾아 Frame Table 의 tail 에 삽입하고 찾은 Page(entry)를 반환하고 함수를 탈출합니다.

이제 다시 main()으로 제어가 넘어오면 LRU_demandPaging()으로부터 반환된 entry 를 struct entry* result 에 저장합니다. 그리고 과제에 필요한 파일로 데이터를 저장하고 모든 과정이 끝납니다.

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

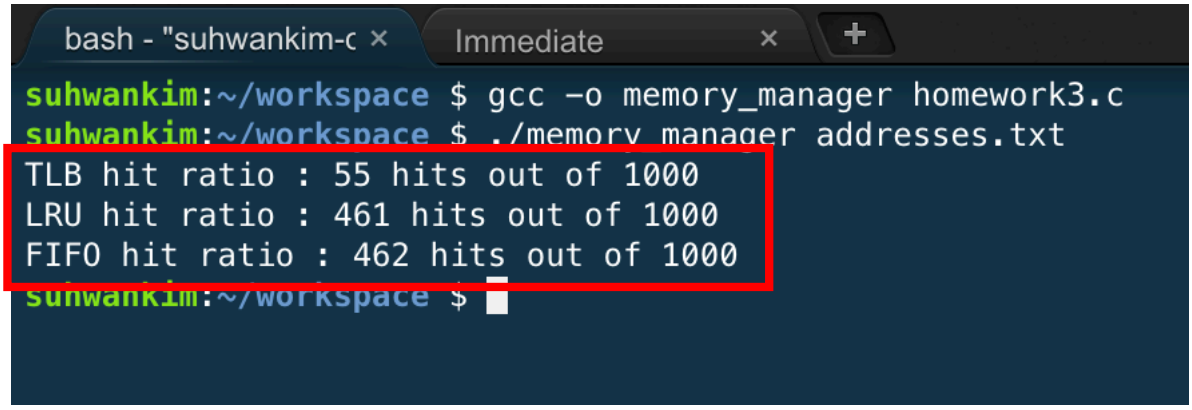
다음으로 Page Replacement 에 FIFO 를 적용하여 Address Translation 하는 과정을 살펴보는데 1) 의 과정과 비슷하므로 간략하게 다른 점을 짚어보는 방식으로 진행하겠습니다.

1) 의 과정이 모두 끝나면 사용한 모든 변수와 Table 을 초기화한 후에 새롭게 다시 FIFO_demandPaging()을 호출하여 Address Translation 과정이 진행됩니다.

먼저 lookUpTLB()를 호출하여 TLB 를 Look Up 합니다. Look Up 을 마치고, 맨 처음 Demand Paging 일 경우 1) 의 같은 경우의 과정이 진행된 후. 삽입된 entry 를 반환하고 함수를 탈출합니다. 두 번째 이후의 Demand Paging 일 경우 앞서 설명하였듯이 Page Table 배열에서 Demand 한 Page 의 Valid/Invalid 에 따라 경우가 나누어집니다. 먼저 Invalid 인 경우 1) 의 같은 경우의 과정이 진행되지만, Valid 인 경우 약간 다릅니다. Page Replacement 에 FIFO 를 적용시키면 Valid 인 경우 LRU 와 다르게 Frame Table 을 관리하지 않으므로 그냥 새로운 entry 를 동적 할당한 후 Page Table 에 저장된 Frame Number 와 offset 을 참조하여 entry 에 저장합니다. 그리고 entry 를 반환하고 함수를 탈출합니다. 그리고 역시 FIFO_demandPaging()으로부터 반환된 entry 를 struct entry* result 에 저장합니다. 그리고 과제에 필요한 파일로 데이터를 저장하고 모든 과정이 끝납니다.

위의 과정을 통해 요구조건 6 가지를 모두 만족하도록 구현하였습니다.

④ 결과



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

4. 개발 환경

OS : Ubuntu 14.04.3

Compiler : gcc version 4.8.4

Language : C