RePORT

Designing a Virtual Memory Manager

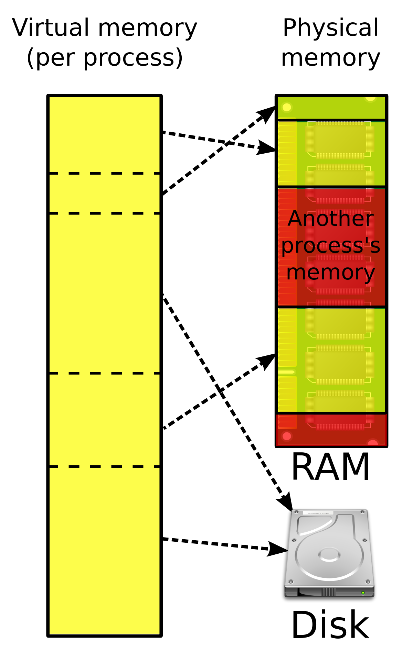


|  |  |  |
| --- | --- | --- |
| 과 목 명 | : | 오퍼레이팅시스템 |
| 학 과 | : | 컴퓨터정보공학과 |
| 학 번 | : | 12111453 |
| 담당교수 | : | 송민석 교수님 |
| 제 출 일 | : | 17년 06월 06일 |
| 성 명 | : | 김 우 진 |

1. 개요

- 목표

Logical address와 physical address의 변환을 simulation하는 code를 작성함으로써 virtual memory를 좀 더 깊이 있게 이해하기

- Virtual memory

Virtual memory란 memory 관리 기법의 일종으로 program에 physical memory address가 아닌 logical memory address를 넘겨주는 방식으로 사용된다. 이 기법을 통하여 physical address space 보다 더 큰 공간을 사용할 수 있게 되었다. Logical address를 physical address로 빠르게 변환하기 위해서는 MMU (Memory Management Unit)의 도움이 필요한데, 이것의 종류로는 TLB (Table, Lookaside Buffer), PTBR (Page Table Base Register), PTLR(Page Table Length Register) 등이 있다. Program은 page라는 block 단위로 나뉘어 지고, 필요 시에만 memory로 swap in을 하는 on demand 방식으로 사용이 된다.

2. 요구되는 개념

- Page

Paging은 memory의 사용을 위해 disk에 data를 저장 및 검색하는 memory management technic으로 virtual memory를 같은 크기의 block으로 나눠서 사용한다. Contiguous allocation과는 다르게 process를 memory 공간에 연속적으로 할당할 필요가 없다. Page는 logical address와 physical address간의 변환을 위하여 page table을 이용하는데, 이것은 process마다 존재하기 때문에 PCB에 저장하여 관리한다. Page의 경우에 size가 정해져 있기 때문에 internal fragmentation이 발생하는데, 이것은 할당한 memory 공간이 요청한 memory 공간보다 커서 생기는 낭비이다. 이것을 감소 시키기 위해 page size를 줄이는 방법이 있지만 page의 size가 작으면 page table entry가 많기 때문에 table 관리에 많은 overhead가 발생한다. Page size가 증가하면 internal fragmentation으로 인한 낭비가 심해지지만 page table의 크기가 작아지고, locality가 좋아진다는 장점이 있다.

- Frame

Physical memory를 고정된 block size로 나눈 것을 의미하며, 일반적으로 block size는 page의 block size와 동일하다. Frame table은 총 frame의 수와 각 frame의 할당 여부가 저장되어 있다.

- 변환방법

Logical address와 physical address의 변환 방법은 두가지가 있다. 첫번째는 (page size) X (page number) + a <-> (frame size) X (frame number) + a를 이용하여 number와 offset으로 접근하는 방식이다. 두번째는 Logical address space 2m, page size 2n일 때, page number = m – n, offset = n으로 하여 page number를 page table의 index로 사용하고 page number에 frame number를 대체하여 physical address의 시작주소를 찾아내는 방식이다. 이 시작주소에 offset을 더해서 access하고자 하는 위치를 찾을 수 있다.

- page fault

Page fault는 memory에 해당 page가 존재하지 않는 상태로, demand paging 방식에서는 처음 page를 가져올 때 항상 fault가 발생한다. 그러나 locality로 인하여 page fault가 많이 발생하지는 않는다. Page fault가 발생하였는데 만약 invalid reference라면 abort가 되고, page가 존재하지만 memory에만 없는 상태라면 swap in이 진행 된다. Swap in을 하기 위하여 frame을 선택할 때 비어 있는 frame이 존재한다면 해당 frame을 선택하고, 그렇지 않으면 replacement algorithm을 통하여 victim frame을 선택하게 된다. 해당 frame이 비어 있지 않았다면, frame에 있던 page를 backing store로 swap out해야한다. 이때 modify bit가 존재한다면 수정이 안된 page에 한해서 swap out 없이 덮어씌울 수 있다. 이 경우 disk 연산이 줄기 때문에 상대적으로 더 빠르다. Swap in이 진행된 후에 page table과 frame table을 갱신하고 요청한 page의 validation bit을 valid하다고 setting 해야 한다. 그 후 page fault가 발생한 instruction을 재실행 한다. Page fault 시에 disk로 접근해야 하기 때문에 overhead가 발생한다. 따라서 swap을 최소로 해야 한다. Swap을 최소로 하는 방법은 page size를 크게 하는 방법, indirect allocation을 최소로 하는 방법, 마지막으로 fault rate을 최소화하는 방법이 있다.

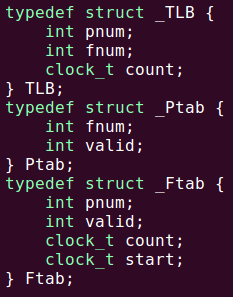
- MMU(Memory Management Unit)

Virtual address와 physical address의 변환을 돕는 Hardware적인 장치이다. MMU는 memory management를 위한 특별한 register로 구성이 되는데, 이것은 범용적으로 사용되는 register와는 차이가 있다. Paging 기법에서 사용되는 MMU로는 PTBR, PTLR, TLB가 있다. PTBR (Page Table Base Register)은 page table이 memory에 보관되어 있기 때문에 빠르게 접근하기 위해 존재하는 register로, page table의 시작주소를 가리킨다. PTLR (Page Table Length Register)는 page table의 size를 가지며, 주로 protection에 이용된다. 이번 구현에 중요한 부분을 담당하는 TLB (Table Lookaside Buffer)는 page table의 빠른 lookup이 가능하기 위한 hardware cache로 page table의 일부를 저장해 놓고 모든 entry를 동시에 비교면서 page number를 찾는다. TLB에는 ASIDs (Address space identifiers)가 존재하는데 이것은 각 TLB entry에 존재하며, address space protection을 위해 각 process를 식별한다. Process를 구분할 수 있기 때문에, 수행되지 않는 process여도 자주 사용된다면 TLB에 table을 유지해 둔다. TLB는 table의 일부만 저장하였지만 locality로 인하여 hit ratio가 매우 높다.

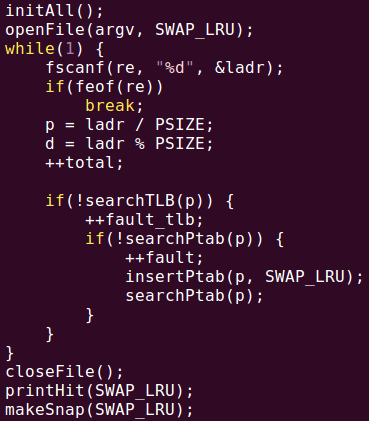
- page replacement algorithm

Page replacement algorithm은 page fault가 발생했을 때 교체할 victim을 결정하기 위한 전략으로, page fault를 최소화하는 형태의 교체를 하는 것이 목표이다. FIFO (First In First Out)방식은 memory를 먼저 차지한 page를 victim으로 선택하는 전략으로 belady’s anomaly가 발생할 수 있다. Belady’s anomaly란 memory의 크기를 늘릴수록 page fault가 증가 할 수 있다는 것으로, 일반적인 예측에 모순된다. LRU (Least Recently Used)는 가장 오랫동안 사용되지 않은 page를 victim으로 선택하는 전략으로 시간의 개념이 들어가 있기 때문에 구현이 복잡하다. 구현 방식은 두가지가 있는데, 첫번째는 counter를 이용한 구현이다. 이 방식은 각 page entry에 counter를 갖도록 하고 page를 reference할 때마다 clock을 기록해 두고, victim 선택 시 counter값이 가장 작은 것을 선택하는 방식으로, 모든 entry를 검사해야 하기 때문에 시간이 오래 걸린다는 단점이 있다. 또 다른 구현 방식은 stack을 이용한 것으로 double link현태로 page number를 stack에 보관하는 방식이다. 매 reference 시마다 해당 page를 top으로 올려 순서를 보존하기 때문에 searching 필요 없어서 빠르다.

3. 코드설명 및 실행결과

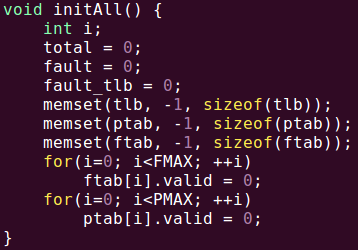
1) 구조체

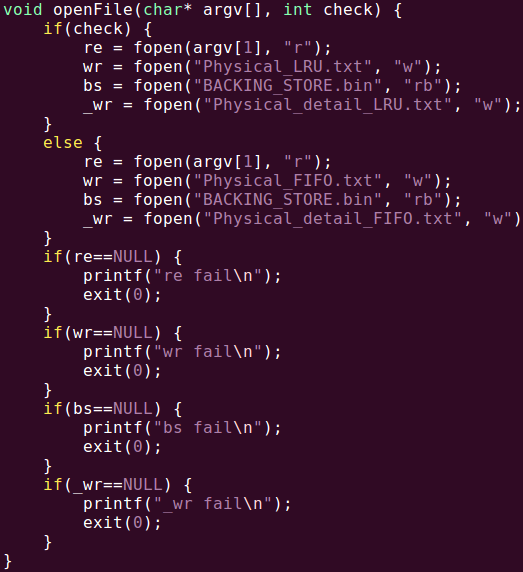
TLB는 구조체는 말그대로 MMU인 TLB를 구현하기 위한 것으로 page number와 frame number를 갖는다. 또, LRU를 하기 위해 필요한 count값을 저장하는 변수 역시 필요했다. Ptab은 page table로 page number를 index로 frame number와 valid bit를 확인할 수 있게끔 구현하였다. Ftab은 frame table로 frame number를 index로 하고, page number와 valid 값, 그리고 LRU와 FIFO를 위한 count와 start 값을 갖는다.

2) main 함수

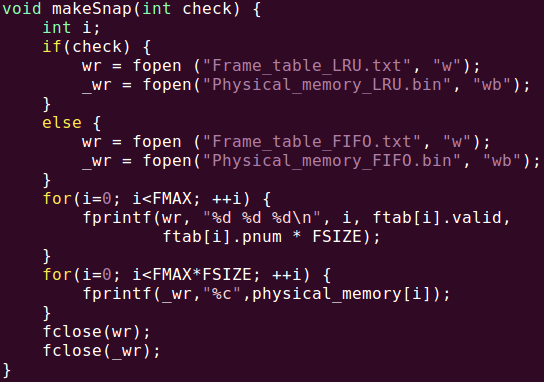
main함수는 다음과 같다. initAll()을 통해서 각종 변수들을 초기화 해주고 필요한 file에 접근하기 위하여 openFile()을 호출한다. File을 읽어서 logical address를 하나씩 받고 page number와 offset 값을 갖게 된다. TLB에 해당 page number가 없으면 TLB fault를 증가시키고 page table을 확인한다. Page table에도 해당 page가 존재하지 않을 시 page fault를 증가시키고 replacement algorithm을 통하여 해당 page를 swap in 한 후에 다시 동작 시키도록 하였다. FIFO의 경우 해당 code와 유사하기 때문에 screenshot에서 제외하였다.

3) initAll 함수

initAll함수는 앞서 설명한 구조체와 구현에 필요한 각종 변수들을 초기화하는 함수이다. Valid bit를 -1로 사용하여도 상관은 없으나, 가독성을 높이기 위하여 0으로 초기화 하였다. Total은 page를 읽은 총 수이고 fault는 page fault가 일어난 횟수, fault\_tlb는 TLB에서 miss가 난 횟수를 저장하기 위한 변수이다.

4) openFile 함수

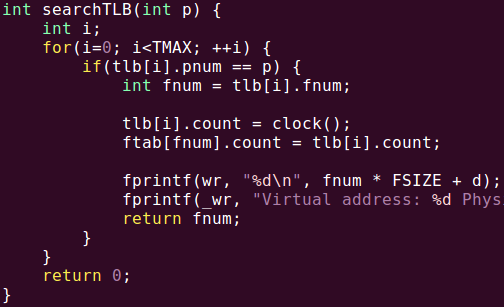
openFile함수는 각종 file들을 읽거나 쓰기 위해 만든 함수로 re는 logical address들이 저장되어 있는 Addresses.txt file을 읽는다. wr은 읽은 logical address를 physical address로 만든 값을 쓰기 위해 필요하다. bs는 BACKING\_STORE.bin을 읽어서 memory에 없는 page를 swap in하기 위하여 필요하다. 마지막으로 \_wr은 Physical.txt를 읽기 좋은 형태로 쓰기 위해서 필요한 변수이다.



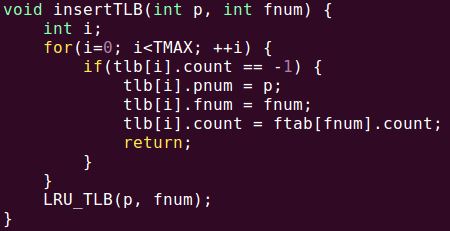
5) makeSnap 함수

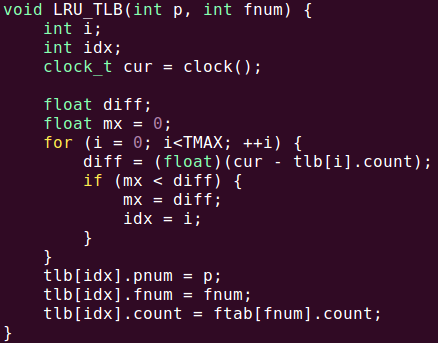
makeSnap함수는 연산을 다 마친 후에 마지막에 남아있는 Physical memory와, frame table의 snapshot을 저장하기 위한 함수이다. 이것을 통하여 작성한 코드가 적절하게 짜여 있는지를 확인할 수 있었다.

6) searchTLB 함수

 searchTLB함수는 해당 page가 TLB에 저장되어 있는지를 확인하기 위한 함수로 TLB의 크기만큼 전부 확인하며 일치하는 page가 존재하는지를 확인하는 방식으로 작성하였다. Hit가 나면 logical address에 해당하는 physical address를 Physical.txt에 작성하여야 하기 때문에 fprintf를 이용하였다.

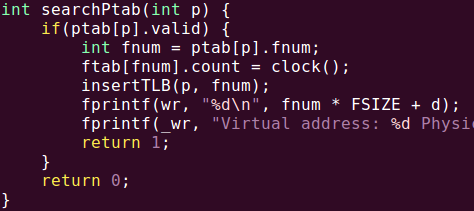
7) insertTLB 함수

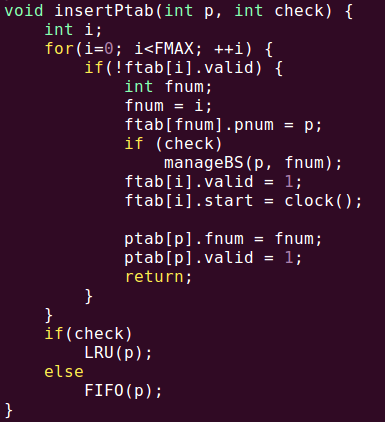
TLB에 해당 page가 존재하지 않을 경우에 TLB로 가져오기 위하여 만들어진 함수로 아직 아무것도 들어 있지 않은 cell이 존재 할 시에는 해당 cell을 이용하지만 모든 cell이 사용 중일 경우에는 LRU방식을 이용하여 victim을 선택하도록 구현하였다.

8) LRU\_TLB 함수

TLB의 모든 cell이 사용 중이어서 victim을 결정하기 위하여 사용된 함수로 LRU 방식으로 동작하도록 하였다. 현재의 clock 값에서 이전에 저장된 clock 값을 뺀 것이 최대가 될 경우 해당 cell을 가장 오랫동안 사용하지 않은 것이기 때문에, 최대가 된 cell을 victim으로 선택하여 replacement하도록 구현하였다.

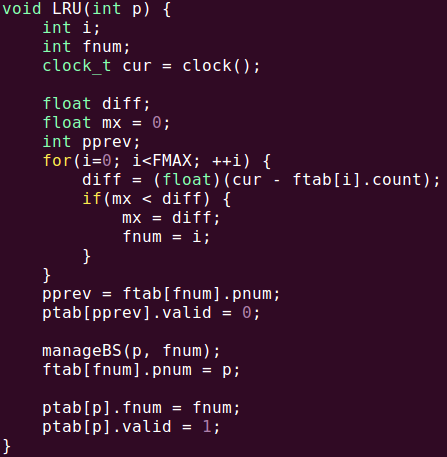
9) searchPtab 함수

해당 page가 valid한 상태일 경우 page table에 해당 page에 mapping된 frame이 존재하다는 의미이므로 reference를 한 후 clock을 갱신하고, 발견을 했다는 표시로 1을 return 하였다. 그렇지 않은 경우 0을 return하여 page를 swap in하도록 하였다.

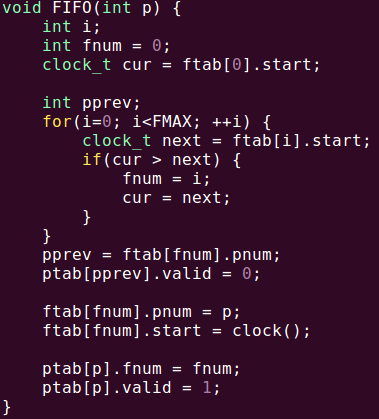
10) insertPtab 함수

이 함수는 해당 page가 frame에 할당되지 않았을 때 호출되는 함수로, 만약 사용되지 않은 frame이 존재할 시에 해당 frame을 이용하고 종료하게 되고, 그렇지 않을 시에 LRU방식이나 FIFO방식으로 victim을 선택하도록 구현하였다.

Check를 통하여 LRU 방식을 사용할 지 FIFO 방식을 사용하지를 결정하였다. Page가 swap in 되는 것을 보이기 위하여 manageBS함수를 통하여 backing store를 관리하였다.

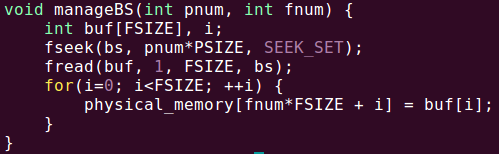
11) LRU 함수

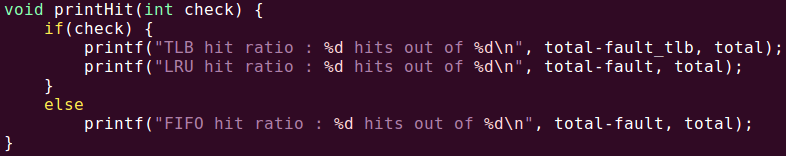
LRU 방식으로 victim을 선택하기 위하여 작성된 함수로 TLB에서의 LRU와 유사하다. 앞서 가지고 있던 count 값을 현재 clock과 비교하여 가장 예전에 사용되었던 frame을 결정하고 victim으로 선정하는 방식을 채택하였다. 이 때 victim이 정해지면 원래 frame에 존재하였던 page를 page table에서 갱신 해주어야 하기 때문에 pprev를 이용하였다.

12) FIFO 함수

FIFO함수는 FIFO방식으로 victim을 선택하기 위하여 작성한 함수로 처음 해당 page가 frame으로 swap in 되었을 때의 시점을 count에 저장하여 그 값이 가장 오래된 frame을 victim으로 선택하는 방식으로 이것도 앞선 방식과 마찬가지로 page table을 다시 갱신해줘야 하기 때문에 pprev 변수를 이용하였다. 이 방식의 경우 기준전에서 떨어진 거리가 필요 없고 단순히 가장 작은 값을 고르면 되기 때문에 구현이 조금 더 수월하였다.

13) manageBS 함수

backing store에서 해당 page를 가져오기 위한 함수로 여기서는 data의 수정이 이뤄지지 않기 때문에 바로 덮어씌워주는 방식을 사용하였다.

14) printHit 함수

hit ratio를 출력하기 위하여 사용된 함수이다.

15) 실행결과

C:\Users\wooji\AppData\Local\Microsoft\Windows\INetCache\Content.Word\15.png

4. 결과

추상적으로만 이해하였던 page를 직접 구현하는 기회를 가지게 되어 매우 유익하였다. 직접 구현을 하니 내부적인 동작과정을 명확하게 이해할 수 있는 시간을 가질 수 있었다.

구현 중 여러가지 문제점이 있었는데 첫번째는 searchPtab과 searchTLB에서 page를 발견 시 1이 아닌 page number를 return 한 경우였다. 이 경우 page number가 0일 경우 page를 발견하지 못한 것으로 간주되었는데, input이 많다 보니 발견하기가 매우 어려웠었다.

두번째 문제는 처음 과제가 주어졌을 때 TLB를 LRU가 아닌 MRU로 구현한 것이다. 이렇게 될 경우 할당되지 않은 page가 TLB에 남아있는 경우가 생겨서 TLB의 size에 따라 hit ratio가 달라지는 현상이 발생한다. 그 이유는 page table과 TLB의 replacement policy가 다르기 때문에 page table에 존재하지 않는 page가 TLB에 남아있게 되기 때문이다. FIFO의 경우에도 마찬가지인 이유로 TLB와 다른 방식으로 replacement policy가 이루어지기 때문에, FIFO를 진행할 시에는 아예 TLB를 제외하고 동작하도록 하였다.

마지막으로는 LRU를 구현하고 FIFO가 동작할 때 LRU에서 얻은 result file들을 다시 덧씌워버리는 문제였다. 이 때에는 FIFO의 출력 결과를 txt파일로 만들지 않고 진행을 해서 어느 부분이 문제인지를 찾기가 어려웠다. FIFO의 출력 결과를 만드는 과정에서 발견하게 되었고 수정을 할 수 있었다.