자료구조 기말 과제 – 캐시 시뮬레이터 구현

1. 캐시(Cache)

캐시(cache)란 데이터를 임시로 저장해두는 장소를 말한다. 캐시의 목적은 요청된 데이터를 처리하는 데 있어서 필요한 데이터에 대한 빠른 접근성을 제공하는 것이다. 캐시는 사용되는 위치에 따라 다양한 형태로 존재한다. 예를 들어 CPU와 메모리사이에는 CPU cache가 존재하고, 우리가 흔히 L1, L2 캐시라 불리는 것이 이에 해당한다. 또한 메모리와 디스크사이에서 디스크에 저장된 블록을 임시로 저장하는 메모리 공간인 페이지(버퍼) 캐시가 존재한다. 하나 더 예를 들면 웹서버와 클라이언트 사이에서 기존에 읽은 웹페이지나 그림 같은 데이터를 저장하는 웹 캐시도 있다. 캐시는 이렇게 다양한 형태로 사용되지만 위에 언급한대로 그 본질적인 목적은 데이터에 대한 접근 시간을 줄임으로써 프로그램의 수행시간을 줄이는데 있다.

캐시는 일정한 크기로 분할되어 데이터를 저장한다(웹캐시인 경우 오브젝트 단위). 통상적으로 이 일정한 크기로 분할된 저장단위를 블록이라고 한다. 그리고 그 데이터에 대한 정보(예: 블록번호, 접근시간 등)를 즉, 메타데이터를 캐시 관리를 위해 유지한다. 프로그램이 특정 데이터를 필요로 하면 우선 캐시에서 존재하는지를 유지하고 있던 메타데이터를 통해 검색한다. 검색 후 그 데이터가 캐시에 존재하면 그 데이터를 접근한다. 이 경우를 캐시 히트(cache hit)라고 한다. 반대의 경우로 데이터가 존재하지 않는다면 해당 데이터가 저장되어 있는 저장소로부터 데이터를 요청한다. 그리고 그 데이터를 캐시에 저장하기 위해 기존의 데이터들 중 하나를 선택하여 제거하고 새로 읽은 데이터를 캐시에 저장한 후 그 데이터를 접근하게 된다. 이 경우는 캐시 미스(cache miss)이다. 또한 저장소로부터 읽은 데이터를 저장하기 위해 캐시에 저장되어 있는 기존의 데이터들 중 하나를 선택해서 버리고 새로 읽은 데이터를 캐시에 저장하는 정책 – 교체될 블록을 선택하는 정책 - 을 캐시 교체 정책 또는 캐시 교체 알고리즘이라고 한다.

캐시 접근 속도, 메모리 접근 속도, 디스크 접근 속도, 캐시 히트율(cache hit ratio) 등을 알면 평균적인 데이터 접근 속도를 계산할 수 있다. 그리고 통상적으로 캐시 히트율이 높을수록 데이터 접근 속도는 개선된다. 따라서 캐시를 교체할 때 기존에 캐시에 저장되어 있던 데이터들 중 어떤 데이터를 교체할 것인가는 굉장히 중요한 문제가 된다. 왜냐하면 교체되어 캐시에 사라진 데이터가 다시 참조되면 저장소로부터 데이터를 다시 읽어와야 하기 때문이다. 따라서 실질적으로 캐시의 설계는 cache hit ratio를 높이는 데 초점을 둔다.

2. 캐시(Cache) 교체 알고리즘의 예

- FIFO(First In First Out): 오래된 캐시 데이터를 먼저 비우고 새로운 데이터를 추가하는 방식이다. 이 정책에서 각 블록의 우선순위는 큐에 들어온 순서가 된다. 따라서 가장 먼저 큐에 들어온 블록이 교체 대상이 된다. 이렇게 교체하려면 큐에는 들어온 순서대로 캐시 블록이 정렬되어 있어야 한다.

- LRU(Least Recently Used): 최근에 사용되지 않는 순서대로 캐시를 교체한다. 즉, 가장 오랫동안 사용되지 않은 캐시 블록이 교체되며 일반적으로 널리 사용되는 방식이다. 이 정책은 참조된 순서대로 정렬되어 있어야 하며 가장 오래전에 참조된 블록을 교체한다.

3. 캐시 시뮬레이터 구현

본 과제에서는 FIFO와 LRU 교체 알고리즘을 구현하고 그 성능(캐시 히트율, cache hit ratio)을 비교한 결과를 소스코드와 함께 제출해야 한다. 구현에 대한 조건은 아래와 같다.

① FIFO, LRU 리스트의 유지는 이진탐색트리 또는 이중연결리스트/해싱 중 하나로 구현한다.

② 이진탐색트리의 종류는 중간과제에서 조사했던 것들 중에 선택한다.

③ 이중연결리스트만 단독으로 사용하여 구현한다면 시뮬레이션 시간이 오래 걸리므로 반드시 해싱과 함께 사용해야 하며 해싱함수는 모듈로(%)나 다른 함수를 사용해도 된다. 단 구현을 용이하기 위해 충돌 해결은 체이닝 기법을 추천한다.

④ 캐시 크기(캐시 블록의 개수)는 8192개의 데이터를 저장할 수 있으며 배열 또는 메모리 할당을 이용한다.

⑤ 이중연결리스트/해싱을 활용한 구현에서 캐시 블록의 선언은 다음과 같이 선언할 수 있다.

- struct cache_buffer {
  unsigned long blkno;
  struct cache_buffer *next, *prev;
  struct cache_buffer *hash_next, *hash_prev;
  }
- 이진탐색트리에서도 위의 구조체에서 hash_next/hash_prev를 제거하여 사용할 수 있다.

⑥ 시뮬레이터이므로 실제 데이터를 기록하지 않고 참조 스트림에서 데이터 번호를 캐시 블록에 기록한다 (위 구조체의 멤버인 blkno에 저장).

- 참조 스트림의 예: 1, 2, 3, 4, 9, 2, 5, 10, 102, 6, 5, 3, … (파일로 제공(test_trace.txt)되며, 각 라인에 쓰여 있는 번호가 필요한 데이터 번호임)

⑦ 구현 후 시뮬레이션은 파일로 제공된 참조 스트림을 입력으로 받아 수행하고 출력된 결과를 아래와 같이 정리한다.

- 결과 예시

	FIFO	LRU
Hit ratio	85%	90%

4. 제출물 및 제출 방법

① 아래의 내용이 포함된 시뮬레이션 결과 보고서 (PDF로 제출, PDF 파일이름은 학번으로 할 것, 페이지수 제한없음)

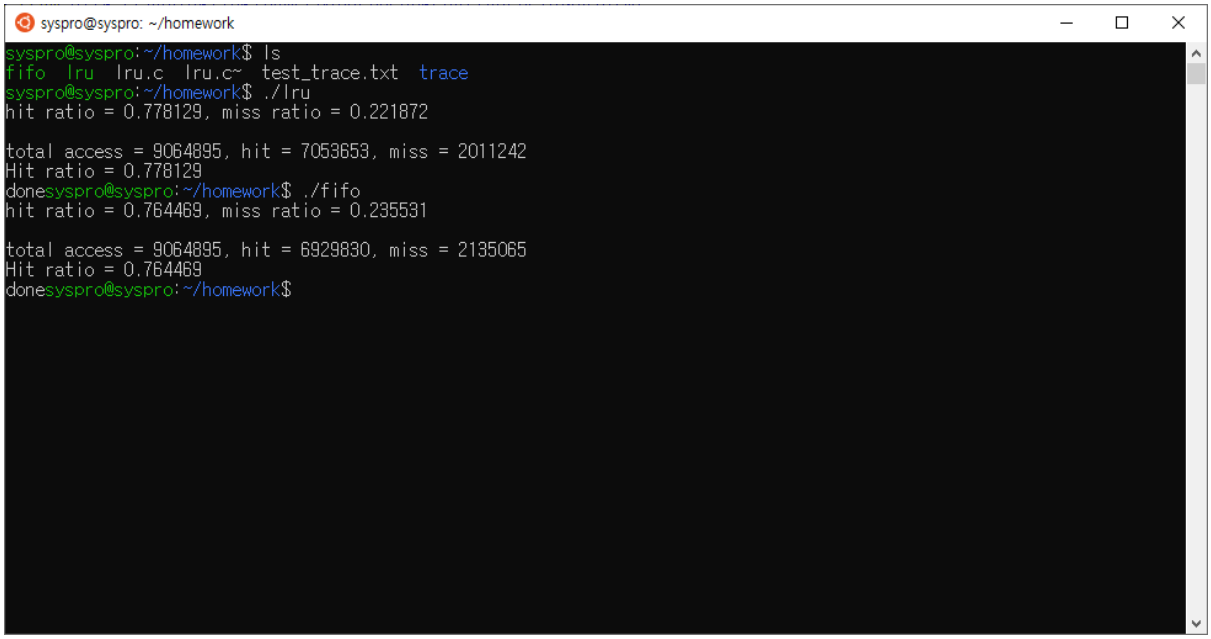
A. 소스코드 전체와 설명

B. 시뮬레이션 결과 화면 캡처

C. LRU/FIFO 구현 및 시뮬레이션 결과보고

② 제출시간은 2021년 12월 23일 밤 11시까지 온라인 강의실 과제제출을 통하여 제출해야 함(반드시 제출기한을 지켜야 하며 이메일 제출은 받지 않음)

5. 실험 결과



위 그림은 실제 시뮬레이터를 구현하고 실행한 결과이다. 실험 결과 FIFO의 히트율(hit ratio)는 0.764469이고 LRU의 히트율은 0.778129이다. 통상적으로 LRU 교체 알고리즘이 FIFO의 히트율보다 높은 결과를 보인다. 시뮬레이터의 구현 방법은 다를 수 있지만 과제에서 제공한 트레이스를 시뮬레이션하면 위와 정확히 같은 히트/미스율을 보여야 한다. 그렇지 않다면 구현이 잘못된 것으로 생각하고 디버깅을 진행해야 한다.