CS Study Day.9

Memory Hierarchy

Jiwon Son / 25.09.01

Memory Hierarchy 개념

정의 및 목적

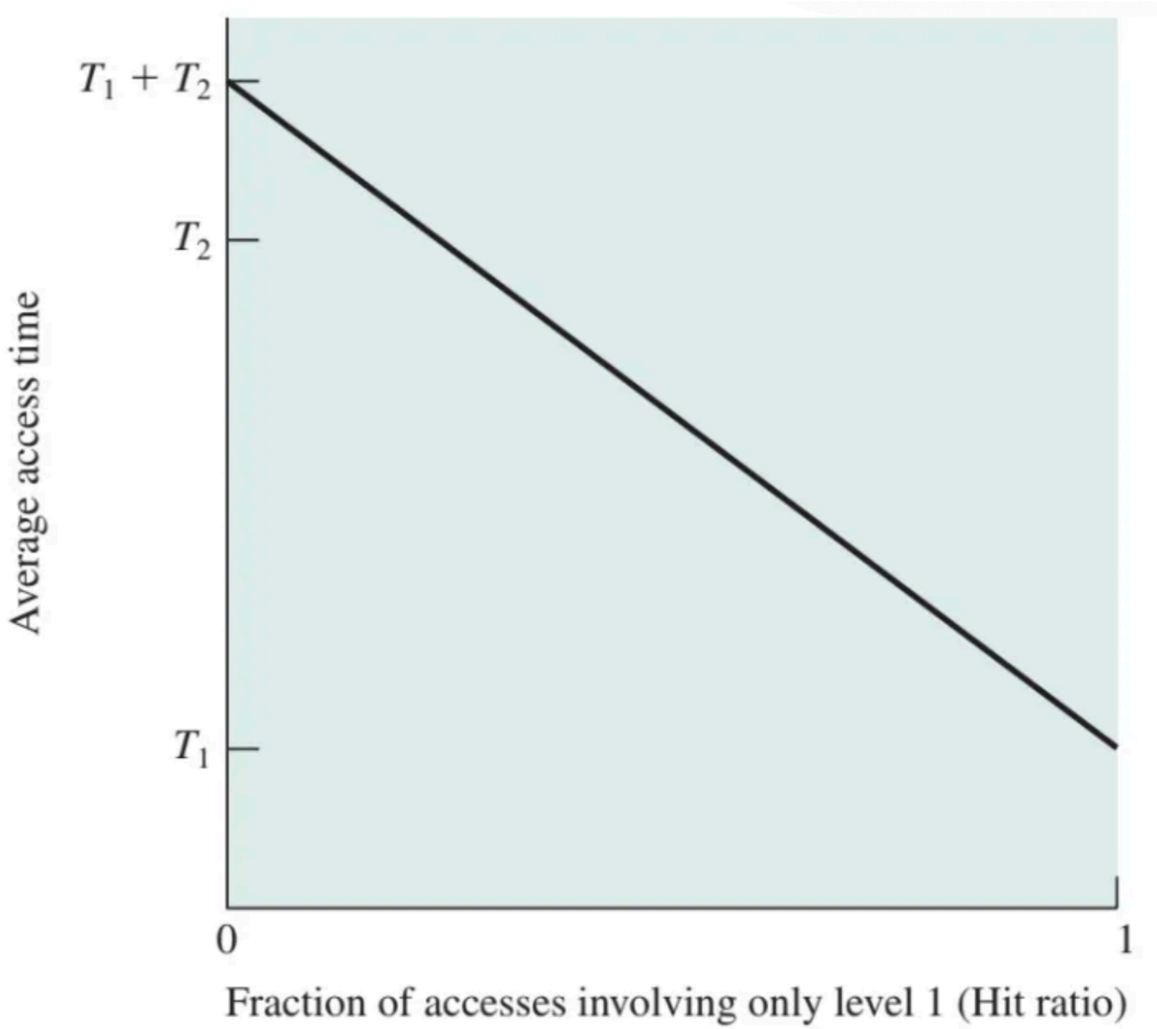
- 정의 : 데이터 저장 계층을 수직적으로 구조화한 방식
  - CPU와 메모리 거리에 비례해서 시간적, 공간적 cost가 발생함
    - 시간적 Cost(Time Cost, Latency) : 메모리에 접근하는 데 걸리는 시간
    - 공간적 Cost(Cost Per Bit) : 해당 메모리를 실제 HW로 구현하는 데 드는 단가

- 목적 : CPU와 메모리 사이에서 데이터 교환 오버헤드를 최소화하는 것
  - CPU와 데이터가 가까울수록 빨라지므로, 데이터 사용 정책에 따라 적재시킬 공간을 수직적으로 구조화하여 달성

Memory Hierarchy 개념

효율성의 수학적 증명 (참고)

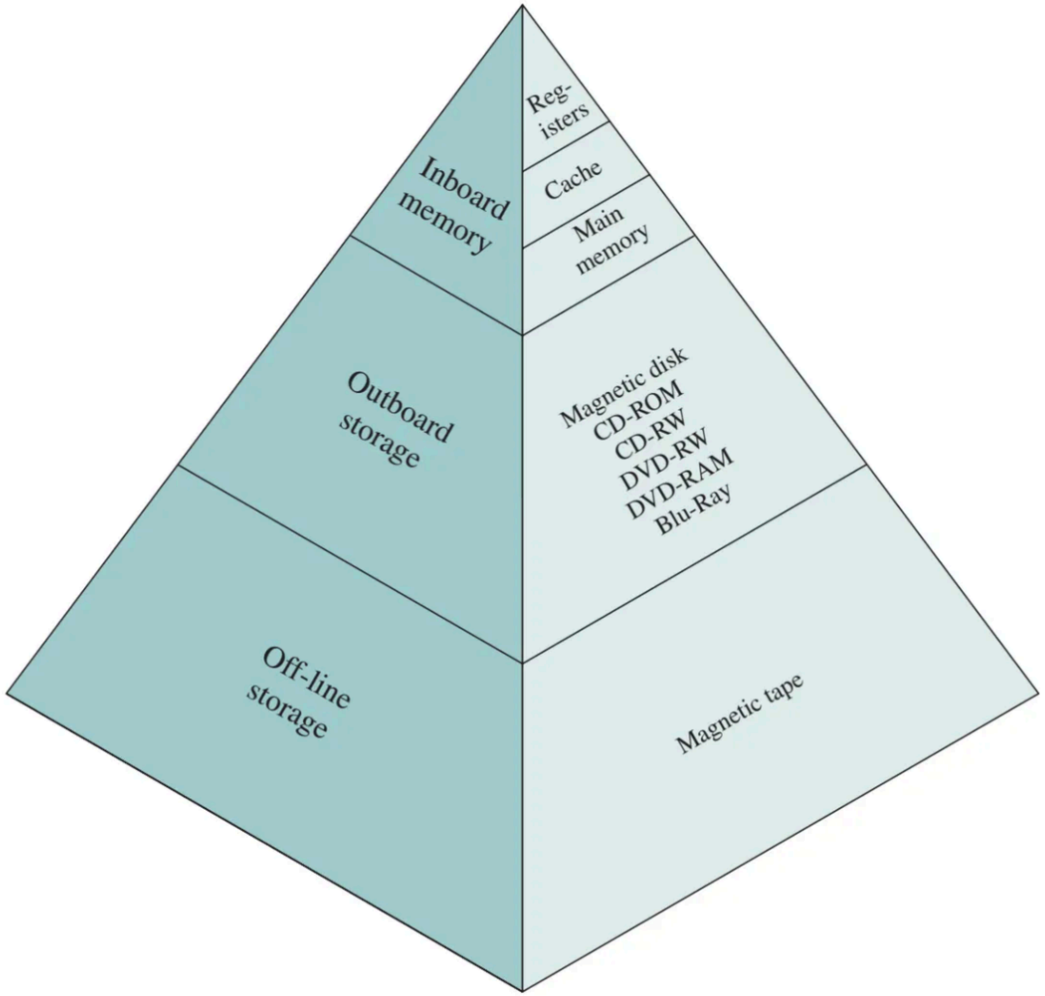
- 가정
  - L1(크기 : 1,000byte, 속도 : 0.1μs)
  - L2(크기 : 100,000byte, 속도 : 1μs)
  - 데이터가 필요할 경우 무조건 L1에 적재 후 접근
  - 어떤 데이터를 끌고올지에 대한 연산과정은 무시
  - L1 Layer Hit Ratio = 95%
- 수식
  - (0.95) * (0.1μs) + (0.05) * (0.1μs+1μs) = 0.095 + 0.055 = 0.15μs



Memory Hierarchy 개념

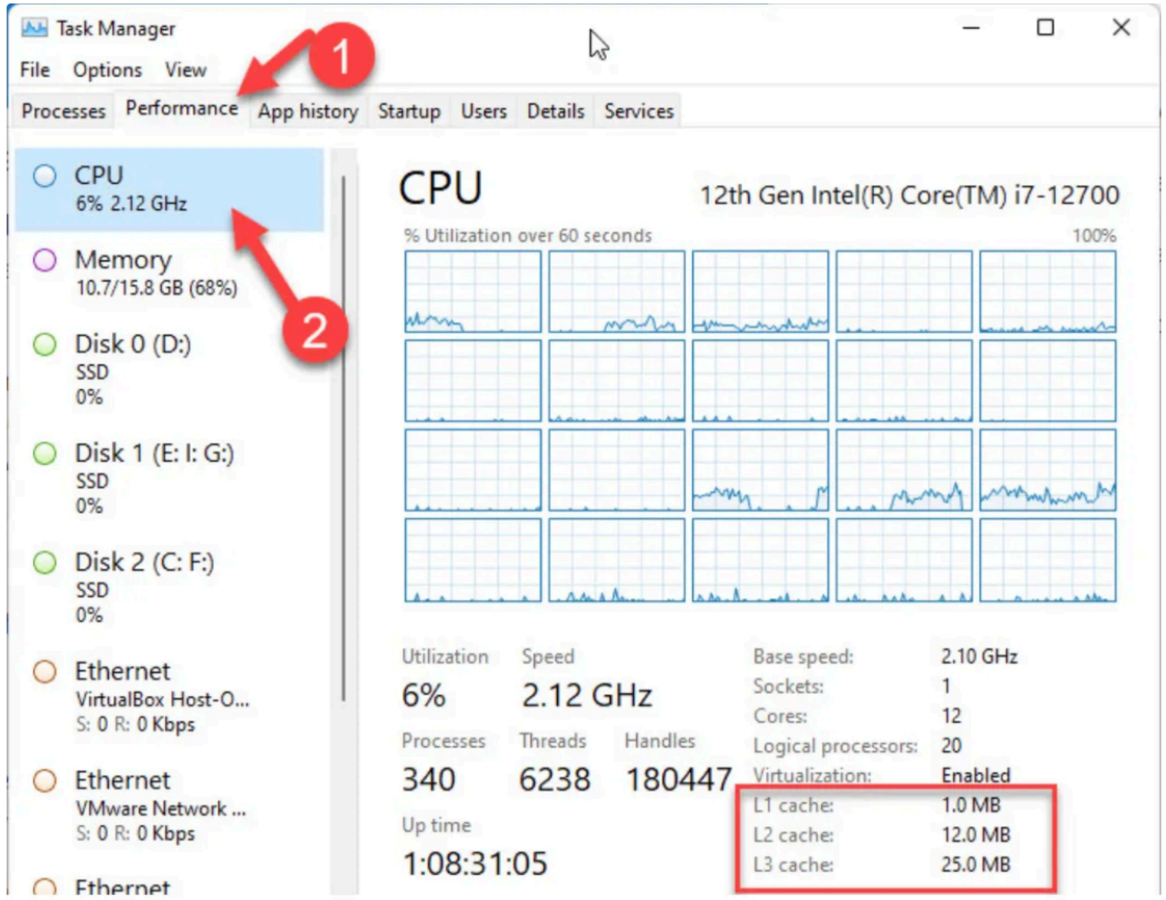
구조

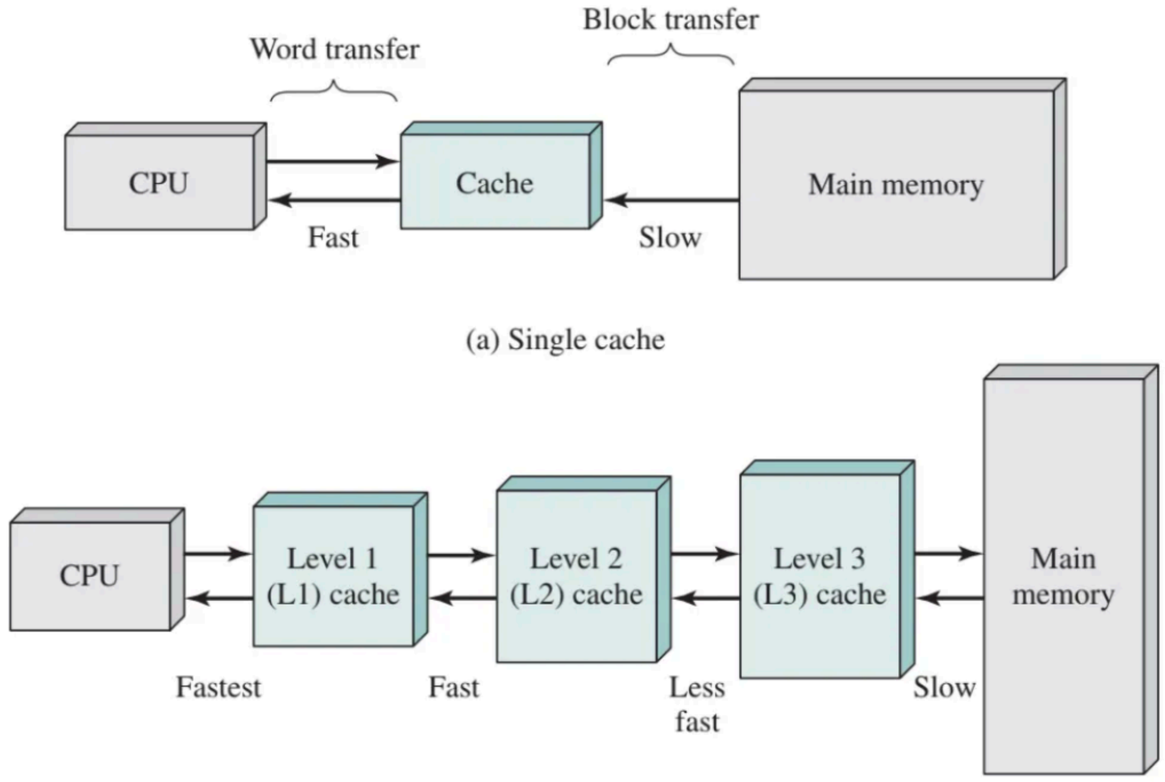
- Inboard Memory
  - 메인보드 내 탑재된 메모리
    - Register, Cache(*), RAM, etc.
- Outboard Memory
  - 메인보드와 커넥터로 이어진 전기적 메모리
    - HDD, SSD, etc.
- Offline Storage
  - 메인보드와 커넥터로 이어질 수 있는 비전기적 메모리
    - Magnetic tape, etc.



Memory Hierarchy 개념

Cache Layer (참고)





Memory Hierarchy 원리

성능 향상의 근본 원리

- 메모리 근접성(Principle of Locality)
  - 한번 사용된 메모리는 시간적, 공간적으로 인접할 경우 다시 사용될 가능성이 매우 높음
    - 시간 근접성(Temporal Locality) : 한번 접근한 데이터를 짧은 시간 내 다시 접근될 가능성이 높음
      - 가장 최근에 사용된 데이터를 캐시 메모리에 업로드
      - e.g., for (int i = 0; i < size; i++) 에서 i
    - 공간 근접성(Spatial Locality) : 메모리의 특정 위치에 접근했다면, 그 주변 위치의 데이터로 접근할 가능성이 높다는 원리
      - 최근 사용된 데이터 주변 데이터를 같이 캐시 메모리에 업로드
      - e.g., int arr[10] 에서 arr[0], arr[1], … , arr[9]

Cache 운용

Cache Hit, Miss

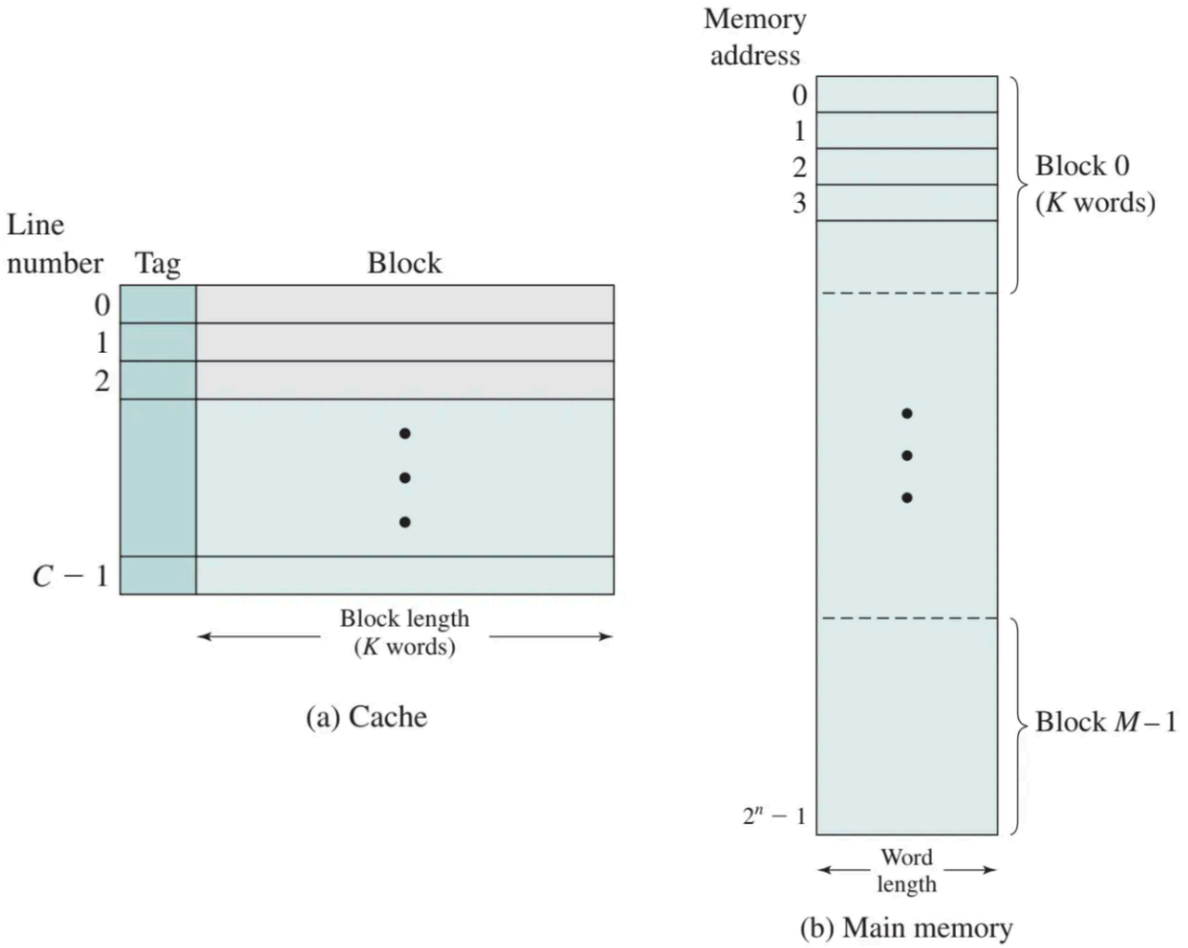
- CPU는 연산 시 가장 먼저 Cache Memory 안에서 데이터 액세스를 시도함
  - Cache 안에 이미 저장되어 있어 액세스에 성공했다면 Cache Hit
  - 접근을 시도했으나 Cache 안에 없어 메모리에서 가져와야 한다면 Cache Miss
  - Cache 자체에 적중률이 높다면 Cache Hot, 낮다면 Cache Cold

- Cache miss는 왜 발생하는가? (3C)
  - Compulsory(=Cold) : CPU가 처음으로 사용하는 데이터라 발생하는 미스
  - Capacity : 액세스하는 데이터 크기보다 Cache의 용량이 작은 경우
  - Conflict : 매핑 충돌로 인해 발생하는 미스

Cache 운용

Cache Memory Mapping(참고)

- 데이터는 메모리에 블록 단위로 저장
- 데이터가 속한 블록 번호와, 블록 자체를 Cache 구간에 복사 저장
- 이후 Cache에 액세스할 경우, Tag에 저장된 블록 번호부터 탐색
  - 블록을 찾으면, 내부에 저장된 offset을 이용해 블록 내 데이터 위치정보로 직접 액세스



Cache 운용

Cache Memory Mapping

- Main memory에서 Cache의 어느 위치에 저장할지 결정하는 방법
- Cache에 어떻게 매핑하는지에 따라 3가지 방법으로 분류
  - Direct Mapping(직접 매핑) : 특정 블록은 Cache의 “단 한 곳”에만 위치할 수 있음
    - Hash처럼 고정된 블록 번호는 고정된 Cache Address에 대응됨
  - Fully Associative Mapping(완전 연관 매핑) : Cache의 어느 곳에도 위치할 수 있음
    - 아무데나 저장될 수 있지만, 캐시에 저장된 모든 태그를 찾아야 함
  - Set-Associative Mapping(집합 연관 매핑) : 위 2가지 방식을 결합한 방법
    - Cache memory를 여러 단위의 집합(set)으로 분류하고, 각 집합에는 여러 Cache 라인이 포함됨
    - 특정 블록은 특정 집합에만 들어갈 수 있지만, 집합 내 어디든 들어갈 수 있음
    - 현대 OS에서 가장 대중적으로 사용하는 방식

Cache 운용

캐시 교환 정책(Cache Replacement Politics)

- Cache 충돌이 발생했을 경우, 기존 Cache 데이터와 새로 사용된 데이터를 교환해야 함
  - (굳이 안해도 되는 상황이 있지만, 이 경우는 후위에 설명)
  - 교환 정책은 크게 4가지로 분류
    - Direct : 방금 액세스에 실패해 충돌이 발생한 해당 위치를 교환
    - LRU(Least Recently Used) : 가장 안쓰인 cache 라인을 제거
    - FIFO : 가장 먼저 cache에 들어왔던 cache 라인을 제거
    - Random : 무작위로 라인 하나를 선정해 제거

Cache 운용

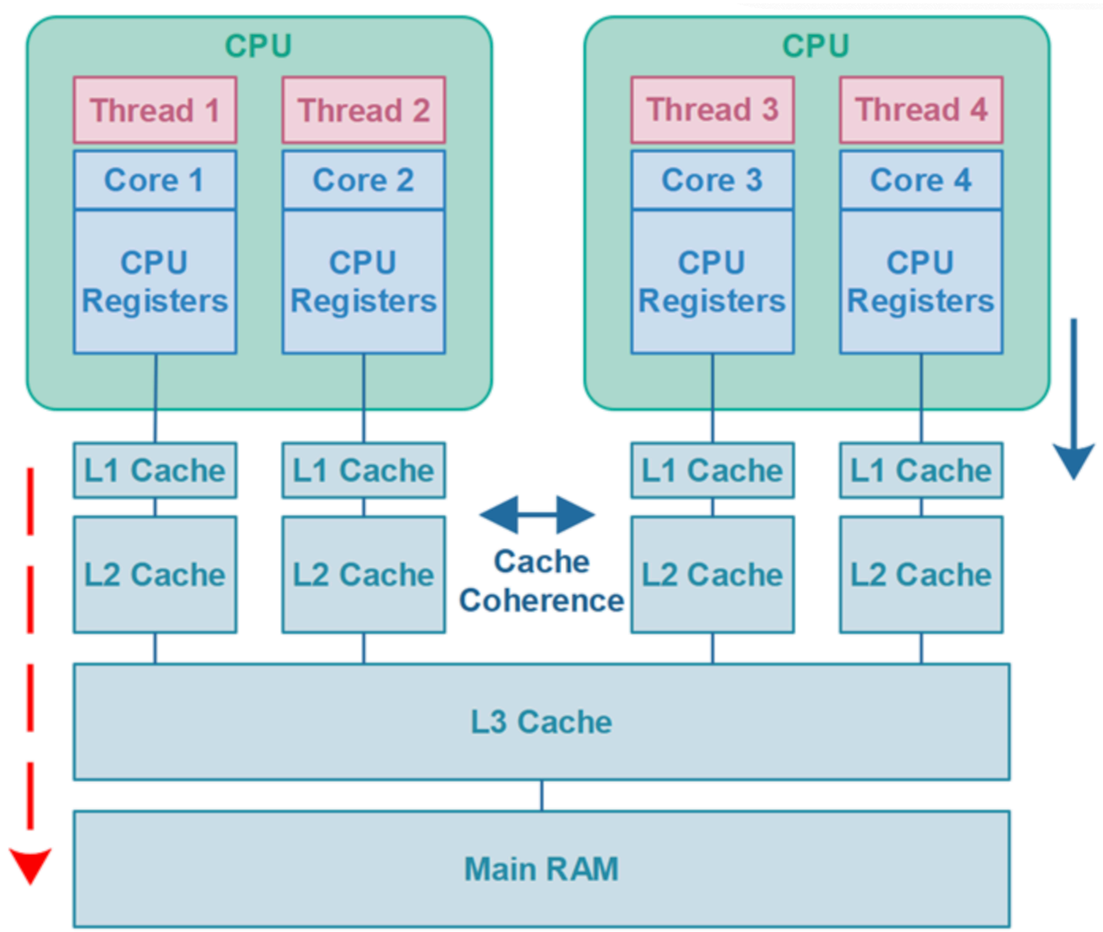
캐시 쓰기 정책(Cache Write Politics)

- Cache는 메모리 데이터의 사본이므로, 두 데이터간 일관성을 유지해주어야 함
- Cache에 저장된 데이터를 변경할 경우, 메모리에 Write하는 시점에 따라 3가지로 분류
  - Write Through : Cache에 쓰기 작업이 시작되는 즉시 원본 데이터도 수정
    - 항상 일관성이 보장되지만, 메모리 대역폭이 낭비됨
  - Write Back : Cache에만 쓰기 작업을 수행하고, 원본 데이터는 나중에 수정
    - Cache를 수정했다는 의미의 Dirty bit를 set해서 수정이 필요한지 표기
  - Write Allocate, No Write Allocate : Cache에 쓰기 작업을 수행했는데 miss인 경우 사용
    - Write Allocate(=Fetch on Write) : 해당 블록을 통째로 cache에 가져와서 쓰기작업 수행
    - No Write Allocate : 메모리에만 직접 쓰고 Cache에 올리지 않음(Cache pollution 대비)
      - Cache pollution : 재사용 가능성이 낮은 데이터가 재사용 가능성이 높은 데이터를 밀어내는 현상

Cache 운용

캐시 일관성(Cache Coherence)

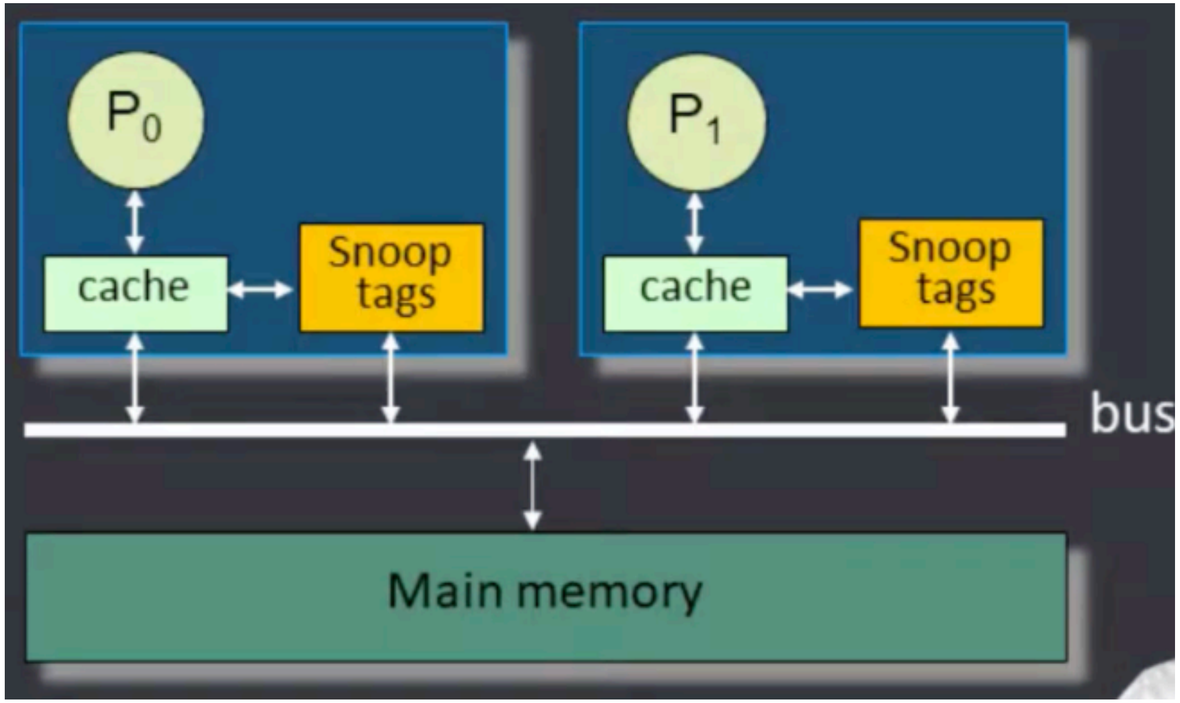
- 멀티코어 아키텍처에서, 여러 코어가 동일 메모리를 캐싱할 수 있음
  - 이 경우, 한 코어가 원본 데이터를 수정하면, 다른 코어는 stale data를 참조하고 있음
    - Stale data : 오래되거나 부정확한 데이터
  - 이를 해결하려면, 아래 조건을 만족해야 함
    - 여럿이 참조할 수 있지만, 쓰기는 혼자 해야 함
    - 어떤 시점에서도 일관성이 보장되어야 함
      - “트랜잭션 직렬화”로 해결
    - 같은 주소에 대한 쓰기는 모든 코어에서 관찰되어야 함
      - “쓰기 전파”로 해결



Cache 운용

캐시 일관성 보장 알고리즘 (1)

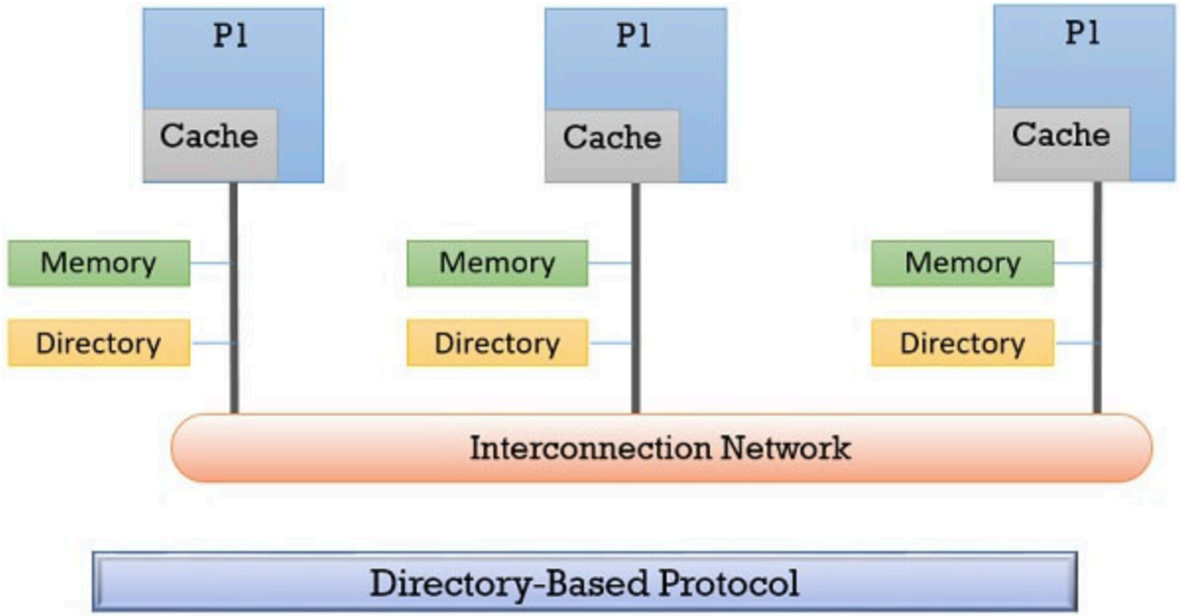
- 버스 스누핑(Snooping, Broadcast)
  - 버스를 관찰하는 스누퍼를 배치해서 변경 여부를 모니터링하는 방법
    - 모든 트랜잭션을 전부 모니터링
    - 만약 변경된 메모리를 자신이 Caching 중이라면, 수정하거나 flush하고 재작성
      - Write-update : 쓰기를 수행하는 코어가 다른 코어에게 update 메시지를 push. 수신 코어가 이 값을 caching중이라면 수정(broadcast)
      - Write-invalidate : 쓰기를 수행하는 코어가 다른 코어의 복사본 cache를 전부 무효화. 재작성 강제



Cache 운용

캐시 일관성 보장 알고리즘 (2)

- 디렉토리 기반 관리방식
  - 하드웨어적 차원으로 일관성을 보장하는 방법
  - 원본 데이터 바로 옆 누가 캐싱중인지 메모
  - 코어간 통신 시스템에 캐싱 정보를 업로드
  - 쓰기를 수행한 코어는 해당 정보를 보고 알람을 발송할 코어 결정
    - 실제 알람은 이 Metadata를 관리하는 Memory Controller, Coherence Controller가 발송



Q&A

from Github (1)

- 캐시 메모리는 어디에 위치해있나요?
  - Cache의 레벨에 따라 다르지만, 통상 CPU 코어 내부, 혹은 바로 옆에 위치
    - L1, L2 : 코어 내부에 연산장치 바로 옆에 위치. 아키텍처에서도 각 코어 단위로 할당
    - L3 : 각 코어 사이 공간에 위치. 코어들 사이에서 공유되는 레벨

Q&A

from Github (2)

- L1, L2 캐시에 대해 설명해주세요
  - L1 : 가장 작고 빠른 Cache. 자주 사용하는 명령과 데이터를 분리해서 저장
    - 최대 용량은 수십kb, 소요시간은 1ns 이내
  - L2 : L1보다 조금 크고 느린 캐시
    - 최대 용량은 수백kb에서 수mb수준, 소요 시간은 2ns 이내

Q&A

from Github (3)

- 캐시에 올라오는 데이터는 어떻게 관리되나요?
  - 블록 단위로 저장됨
    - 블록 단위 : 2^n bit 크기로 이루어진 메모리 단위
    - 내부에 Tag, Valid, Dirty Metadata를 가지고 있음
      - Tag : 해당 Cache 라인이 저장된 데이터가 주 메모리의 어느 블록에서 왔는지 명시
      - Valid : Cache가 유효한지 판단하는 정보
      - Dirty : Cache값이 현재 코어에 의해 변경되었는지를 판별하는 정보
    - 교체 정책과 쓰기 정책에 의해 수명과 동기화를 관리함

Q&A

from Github (4)

- 캐시 간 동기화는 어떻게 이루어지나요?
  - 쓰기 정책과 캐시 일관성 프로토콜에 의해 유지됨
  - Cache 안에 누군가 쓰기 작업을 수행하면, 버스 스누핑으로 각자 캐시를 업데이트하거나, 디렉토리 기반 방식을 통해 같은 데이터를 Caching중인 코어에게 알람 발송

Q&A

from Github (5)

- 캐시 메모리의 Mapping 방식에 대해 설명해주세요
  - 1. Direct Mapping
    - 메모리 블록이 Cache의 정해진 위치에만 들어가는 방식.
    - Conflict miss가 자주 발생하지만, 가장 구현이 쉬움
  - 2. Fully Associative Mapping
    - 메모리 블록이 Cache 아무데나 들어갈 수 있는 방식
    - Cache hit ratio가 가장 높지만, Cache를 전부 탐색해야 하고 구현이 복잡함
  - 3. Set Associative Mapping
    - 메모리 블록은 Cache의 정해진 집합 안에만 들어갈 수 있지만, 집합 내에서는 아무데나 위치 가능
    - 현대 가장 널리 사용되는 방식

Q&A

from Github (6)

- 캐시의 지역성에 대해 설명해주세요
  - 한번 사용된 데이터는 시간상으로 다시 사용될 확률이 높고, 공간상으로 인접한 데이터도 함께 사용될 가능성이 높다는 원리
  - 메모리를 수직적으로 구조화해서 사용할 경우 속도가 향상되는 기본 원리

Q&A

from Github (7)

- 캐시의 지역성을 기반으로, 2차원 배열을 가로/세로 탐색했을 때 성능 차이에 대해 설명해 주세요
  - C/C++에서 2차원 배열은 가로(행 단위)로 배치됨
    - e.g., int arr[10][3]이 있다면, arr[0][0] , arr[0][1], arr[0][2], arr[1][0], arr[1][1], arr[1][2], … 형태
    - 이를 일반화해서 수식으로 풀어 쓰면 아래와 같음

$$addr(i, j) = base + (i * M + j) * sizeof(elem)$$

  - 프로세서가 위 배열에 접근할 때, Cache는 공간 지역성에 의해 인접 데이터를 묶어 Cache 라인에 저장
    - 가로 탐색은 배열의 기본 형태와 일치하므로 Cache hit를 노리기 쉬움
    - 세로 탐색은 가로 행 개수만큼 떨어져있는 데이터를 탐색해야 하므로 공간 지역성이 떨어짐

Q&A

from Github (8)

- 캐시의 공간 지역성은 어떻게 구현될 수 있을까요?
  - 공간 지역성은 Cache line 단위로 구현됨
  - 프로세서가 메모리에서 데이터를 읽을 때, Cache는 해당 데이터 뿐 아니라 주변 인접 데이터도 함께 Caching 수행
    - 이때 Cache line 단위로 가져오고, 그 크기는 32byte 혹은 64byte로 고정
  - 공간 지역성을 이용해 구현하려면, 선형 연속 데이터를 사용하는 편이 효율적
    - 선형 컨테이너를 사용할 경우, 인접 데이터가 Cache line에 들어올 확률이 높아 공간 지역성 구현가능
    - 함께 사용되는 변수들을 모아 구조체로 사용하면 Cache line에 들어갈 확률이 높아짐

Q&A

from Presenter

- 캐시를 활용할 때 고려해야 할 점이 있나요?
  - 현재 머신의 특성, 환경적 요인 등을 고려해 어떤 데이터를 선정할지를 고민해야 함
  - 정리노트 부록 참고

- 캐시는 꼭 CPU 옆에 있는 비싼 저장장치에만 사용되나요?
  - HDD대비 SSD가 빠르고, SSD 대비 메모리가 빠른 특성을 이용해서, 상대적으로 빠른 장치에 데이터를 저장하는 것 자체를 Caching이라고 볼 수 있음
    - NAS에 SSD, 노트북용 메모리 슬롯이 존재하는 이유
    - 정리노트 부록 참고

End