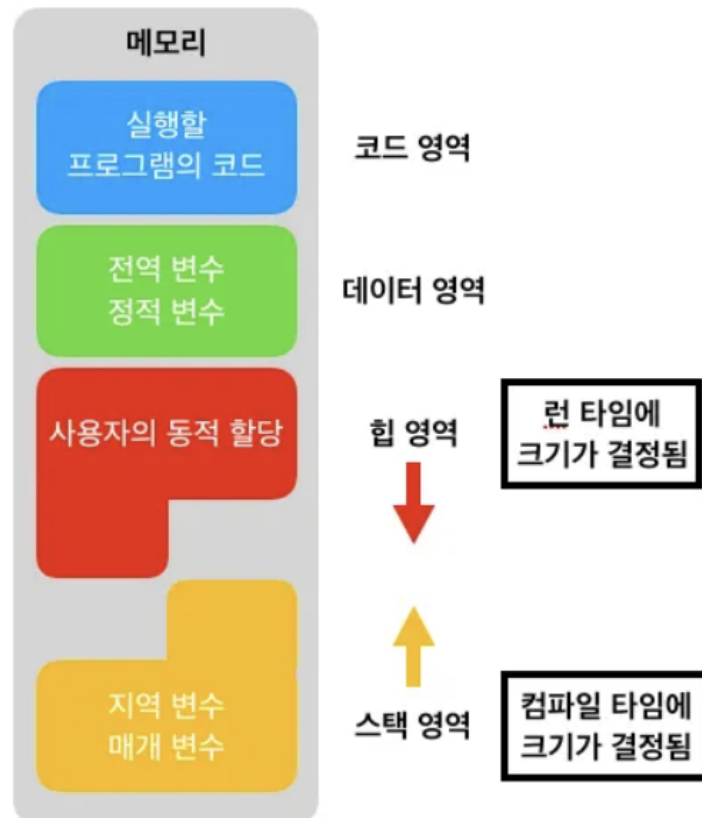


메모리&캐시

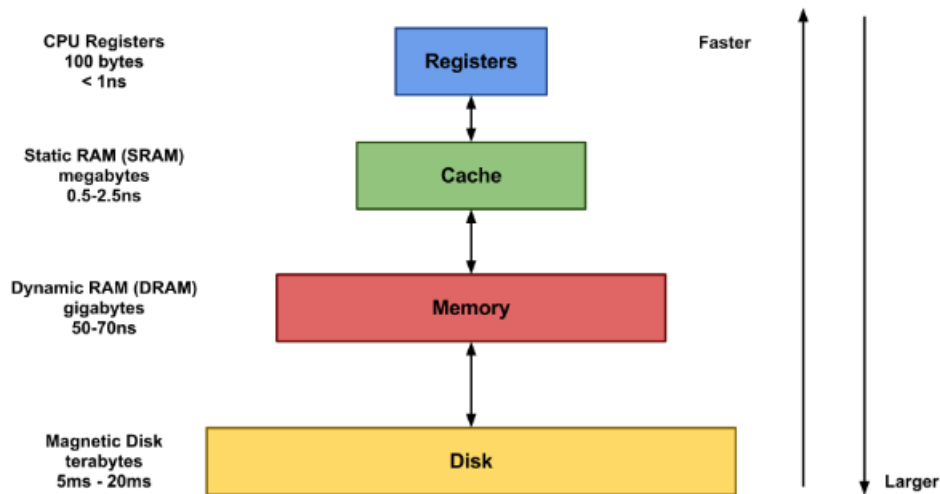
1. 메모리

- 메모리 구조



2. 캐시 (Cache)

: 자주 사용하는 데이터나 값을 미리 복사해놓는 임시 장소



캐시는 저장 공간이 작고 비용이 비싸지만 성능이 빠르다.

캐시의 주요 목적은 더 느린 기본 스토리지 계층에 액세스해야 하는 필요를 줄임으로써 데이터 검색 성능을 높이는 것이다.

- 캐시 데이터는 일반적으로 램(RAM)처럼 빠르게 액세스할 수 있는 하드웨어에 저장된다. 소프트웨어 구성 요소와 함께 사용될 수도 있음
- 속도를 위해 용량을 절충하는 캐시는 일반적으로 데이터의 하위 집단을 일시적으로 저장한다. 보통 완전하고 영구적인 데이터가 있는 데이터베이스와 대조적
 - 접근 시간에 비해 원래 데이터를 접근하는 시간이 오래 걸리는 경우 (서버의 균일한 API 데이터)
 - 반복적으로 동일한 결과를 돌려주는 경우 (이미지/ 썸네일)

3. 캐시 적중률 (Hit Rate)

- 캐시 적중 (Cache Hit), 캐시 누락 (Cache Miss)

: 캐시 적중이란 캐시에서 파일이 요청되고 캐시가 해당 요청을 이행할 수 있을 때 발생

ex) 사용자가 고양이 사진을 표시해야 하는 웹페이지를 방문하는 경우 브라우저에서는 이 사진에 대한 요청을 웹페이지의 **CDN(콘텐츠 전송 네트워크, Content Delivery Network)**에 보낼 수 있다.



CDN이란 사용자가 물리적으로 위치한 곳으로 콘텐츠를 더 빠르게 전달하기 위해 함께 작동하는 연결된 캐시 서버의 글로벌 네트워크

→ CDN의 스토리지에 그림 사본이 있는 경우 요청은 캐시 적중으로 이어지고 그림이 브라우저로 전송됨

→ 반대로 현재 CDN 캐시에 그림 사본이 없으면, 캐시 누락이 발생하고 요청은 원본 서버로 전달된다.

4. 캐시 적중률

: 수신한 요청 수와 비교하여 캐시가 성공적으로 채울 수 있는 콘텐츠 요청 수를 측정한 것

$$\frac{\text{Number of cache hits}}{(\text{Number of cache hits} + \text{Number of cache misses})} = \text{Cache hit ratio}$$



캐시 적중률이 높다 ≠ CDN 성능이 좋다

다른 요소들도 CDN의 효과를 평가하는데 아주 중요하다.

예를 들어,

콘텐츠가 제공되는 위치도 중요하다. 이상적으로 CDN은 최종 사용자와 가장 가까운 CDN 서버의 콘텐츠를 제공

만약 그렇지 않으면 CDN의 성능이 최적이지 않음

캐싱은 CDN이 하는 일의 중요한 부분이지만, 주요 목적은 일반적으로 웹 자산을 더 빠르고 안정적으로 만드는 것이다. 다양한 성능 메트릭은 CDN이 웹/앱 사이트의 속도를 높이는데 얼마나 도움이 되었는지 측정하는데 도움을 준다.

5. 캐시의 지역성 (시간적, 공간적)

- 캐시가 효율적으로 작동하기 위해서는 캐시의 적중률을 극대화시켜야 함

→ 즉, **캐시에 저장할 데이터가 지역성**을 가져야 함



지역성이란?

: 기억 장치 내의 정보를 균등하게 액세스 하는 것이 아닌 어느 한 순간에 특정 부분을 집중적으로 참조하는 특성

◦ 시간적 지역성

: 최근에 참조된 데이터가 곧 다시 참조되는 특성

→ 메모리 상의 같은 주소에 여러 차례 읽기 쓰기를 수행하는 경우, 상대적으로 작은 크기의 캐시를 사용해도 효율성을 꺾할 수 있다.

◦ 공간적 지역성

: 최근에 참조된 데이터와 인접한 데이터가 참조될 가능성이 높은 특성

- CPU 캐시나 디스크 캐시의 경우, 한 메모리 주소에 접근할 때 그 주소 뿐만 아니라 해당 블록을 전부 캐시에서 가져오게 된다.
- 이때 메모리 주소를 오름차순이나 내림차순으로 접근하면, 캐시에 이미 저장된 같은 블록의 데이터를 접근하게 되므로 캐시의 효율성이 크게 향상됨

5. 가상 메모리

: 가상 메모리란 프로세스가 혼자 메모리를 사용하는 것처럼 메모리를 가상화 한것을 말한다.



각 프로세스마다 메모리를 할당하기에는 메모리 크기에 한계가 있다.

가상 메모리는 적은 메모리로 여러 프로세스를 실행시키기 위한 시스템이다. 이는 메모리가 실제 메모리 보다 많아 보이게 하는 기술이다.

프로세스 크기 자체는 엄청나지만, 실제로 프로세스가 사용하는 메모리는 작다는 점에서 착안하고 고안 되었다. 프로세스간 공간 분리로 프로세스 이슈가 전체 시스템에 영향을 주지 않게 할 수 있다.

• 작동 매커니즘

: 프로세스는 가상 주소를 사용해서 실제 해당 주소에서 데이터를 읽고 쓸 때만 물리주소로 바꿔준다.

- 가상 주소 : 프로세스가 참조하는 주소
- 물리 주소 : 실제 메모리 RAM 주소
- Memory Management Unit (MMU)



가상 메모리 시스템을 작동하기 위해 MMU에서 가상 메모리 주소와 물리 주소를 관리하고, CPU는 가상 메모리를 다룬다.

MMU는 하드웨어 장치인데, 하드 웨어 장치가 소프트 웨어 방식보다 주소 변환이 더 빠르다.

MMU는 RAM을 여러 부분(pages)로 나누어 각 페이지를 하나의 독립된 항목으로 처리

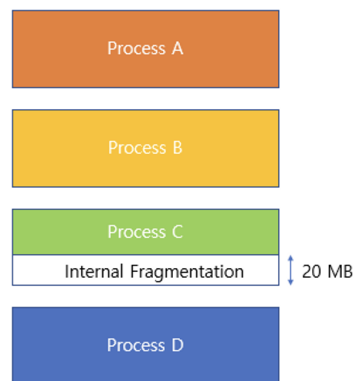
6. 메모리 단편화 (Memory Fragmentation)

: RAM에서 메모리의 공간이 작은 조각으로 나뉘어 사용 가능한 메모리가 충분히 존재하지만 할당(사용)이 불가능한 상태

- 내부 단편화 (Internal Fragmentation)



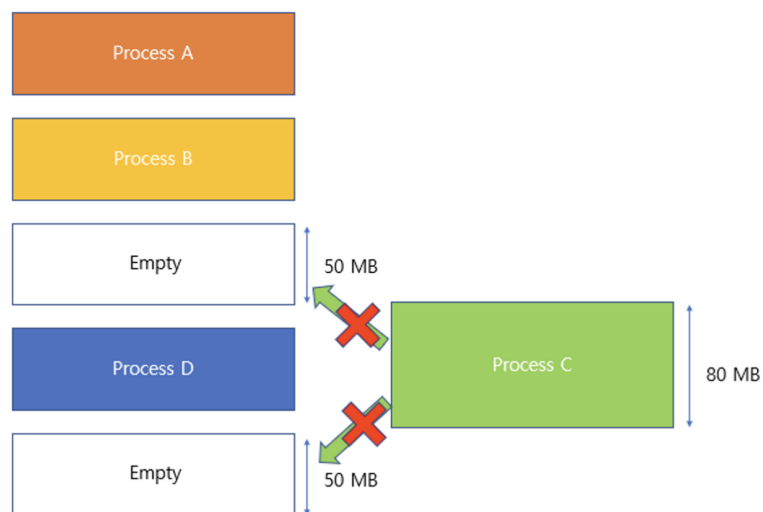
메모리를 할당할 때 프로세스가 필요한 양보다 더 큰 메모리가 할당되어 프로세스에서 사용하는 메모리 공간이 낭비되는 상황



- 외부 단편화 (External Fragmentation)

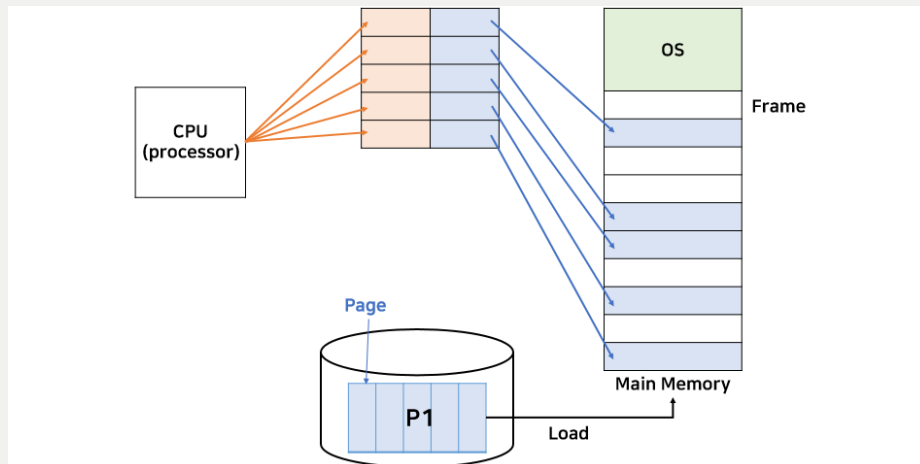


메모리가 할당되고 해제되는 작업이 반복될 때 작은 메모리가 중간중간 존재하는데, 이 때 중간중간에 생긴 사용하지 않는 메모리가 많아 총 메모리 공간은 충분하지만 실제로 할당할 수 없는 상황



7. 단편화 해결 방법

- 페이징(Paging) 기법 - 가상 메모리 사용, 외부 단편화 해결, 내부 단편화 존재



- 프로세스의 주소 공간을 고정된 사이즈의 페이지 단위로 나누어 물리적 메모리에 불연속적으로 할당하는 방식
- 메모리 Frame이라는 고정 크기로 분할되고, 프로세스는 Page라는 고정 크기로 분할됨
- 페이지와 프레임은 크기가 같음
- 페이지와 프레임을 대응시키는 page mapping 과정이 필요하여 paging table을 생성해야 함
- 연속적이지 않은 공간도 활용할 수 있기 때문에 외부 단편화 문제 해결
- 페이지 테이블에는 각 페이지 번호와 해당 페이지가 할당된 프레임의 시작 물리 주소를 저장

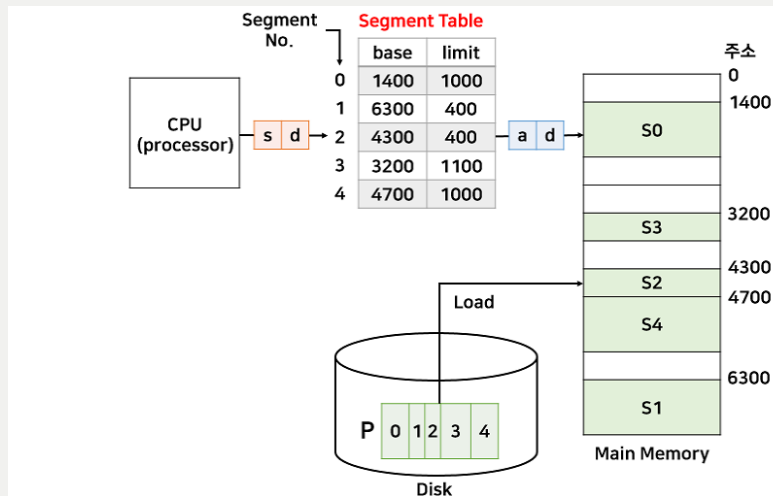
• 내부 단편화 문제가 존재하는 이유:

1. **페이지 크기와 프로세스 요구량 불일치:** 페이징에서는 페이지 크기가 일정하게 정해져 있습니다. 하지만 프로세스가 필요로 하는 메모리 양이 페이지 크기보다 작은 경우, 마지막 페이지는 일부만 사용되고 나머지 부분은 낭비될 수 있습니다. 이는 내부 단편화로 이어집니다.

• 페이징 기법이 외부 단편화를 해결하는 이유:

1. **고정 크기 페이지 할당:** 페이징 기법에서는 물리적 메모리와 논리적 주소 공간을 고정 크기의 페이지로 나누어 관리합니다. 이 페이지들은 동일한 크기를 갖기 때문에 메모리 할당이 단편화 없이 이루어질 수 있습니다. 프로세스가 필요한 메모리 양을 페이지 단위로 요청하고 할당받기 때문에, 작은 조각들로 메모리가 나뉘는 외부 단편화 문제를 해결할 수 있습니다.

- 세그멘테이션(Segmentation)기법 - 가상 메모리 사용, 내부 단편화 해결, 외부 단편화 존재



- 프로세스를 서로 크기가 다른 논리적인 블록 단위인 세그먼트(Segment)로 분할하여 메모리에 할당
- 각 세그먼트는 연속적인 공간에 저장
- 세그먼트들의 크기가 서로 다르기 때문에 프로세스가 메모리에 적재될 때 빈 공간을 찾아 할당하는 기법
- 페이징과 마찬가지로 mapping을 위한 segment table 필요

< 내부 단편화가 해결되는 이유 >

1. **가변 크기 할당:** 세그멘테이션을 사용하면 논리적인 주소 공간을 세그먼트라는 여러 조각으로 나눌 수 있습니다. 각 세그먼트는 프로그램의 특정 부분을 나타내며, 이러한 세그먼트의 크기는 해당 프로그램의 요구에 따라 가변적으로 설정할 수 있습니다. 따라서 필요한 크기에 맞춰 세그먼트를 할당하므로 내부 단편화가 줄어듭니다.
2. **크기 조정:** 세그먼트를 더 이상 사용하지 않을 때 해당 세그먼트의 메모리 공간을 반환할 수 있습니다. 이렇게 되면 메모리 내부에 불필요한 조각이 남지 않고, 메모리 공간을 효율적으로 재사용할 수 있습니다.

- 메모리 풀(Memory Pool)기법



고정된 크기의 블록을 할당하여 메모리 동적 할당을 가능하게 해준다. memory pool이라고 불리는 동일한 사이즈의 **메모리 블록들을 미리 할당해놓고 프로세스들이 필요할 때마다 사용하고 반납하는 기법이다.**

→ 미리 할당하기 때문에 메모리 누수가 있다.

미리 공간을 할당해놓고 사용한 다음 반납하기 때문에 이로 인한

외부 단편화는 발생하지 않는다. 또한 필요한 크기만큼 할당을 하기 때문에 **내부 단편화도 존재하지 않는다.**

메모리의 할당, 해제가 빈번할 때 메모리 풀 방식이 효과적이다.

< 내부 단편화가 일어나지 않는 이유 >

1. **고정 크기 블록 할당:** 메모리 풀은 미리 일정 크기의 메모리 블록을 할당합니다. 이 블록의 크기는 주로 프로그램에서 자주 사용되는 작은 크기로 선택됩니다. 모든 할당은 미리 정의된 이러한 고정 크기의 블록 단위로 이루어지므로, 내부 단편화가 발생할 여지가 줄어듭니다.
2. **동일한 크기의 할당과 해제:** 메모리 풀에서는 동일한 크기의 메모리 블록만을 할당하고 해제합니다. 따라서 할당된 블록을 해제하면 그 블록은 다시 재활용될 수 있습니다. 크기가 일정하고 동일한 블록을 재사용하므로 내부 단편화가 발생하지 않습니다.

8. 동적 메모리 할당 vs 정적 메모리 할당



동적 메모리 할당

프로그램 실행 중에 사용할 메모리 공간을 할당하는 것을 말한다. 사용이 끝나면 운영체제가 쓸 수 있도록 반납하고 다음에 요구가 오면 재할당을 받을 수 있다.

↔ 정적 메모리 할당의 경우, 프로그램이 실행하는 순간 메모리 할당이 이루어진다.

동적으로 할당된 메모리 공간은 **프로그래머가 명시적으로 해제하거나, 쓰레기 수집(garbage collection)이 일어나기 전까지 그대로 유지된다.**

C/C++ 와 같이 garbage collection이 없는 언어의 경우, 동적 할당을 하면 사용자가 해제하기 전까지는 메모리 공간이 계속 유지된다.



정적 메모리 할당

정적 메모리 할당은 메모리의 크기가 하드 코딩되어 있기 때문에 프로그램이 실행될 때 이미 해당 메모리의 크기가 결정되는 것을 말한다.

장점은 할당을 해제하지 않음으로 인한


메모리 누수와 같은 문제를 신경쓰지 않아도 된다. 정적 할당된 메모리는 실행 도중에 해제되지 않고, **프로그램이 종료할 때 알아서 운영체제가 회수한다.**

단점은 메모리의 크기가 하드 코딩되어 있어서 나중에 조절할 수 없다. 스택에 할당된 메모리이므로 동적 할당에 비해 할당받을 수 있는 최대 메모리에 제약을 받는다.

< 참고 >

[CS] 캐시, 가상 메모리

캐시와 가상메모리

 <https://velog.io/@kang1221/CS-캐시>

CS