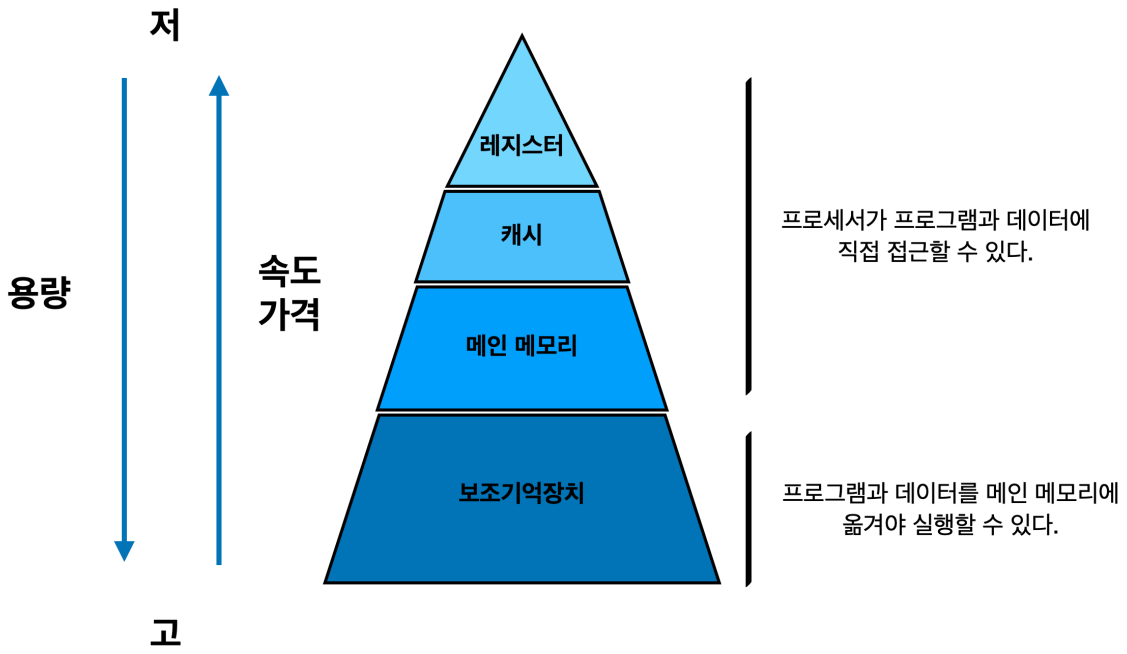
Memory

메모리

- **메모리는 컴퓨터 성능과 밀접**하여 많은 사용자들은 크고 빠르면서 비용이 저렴한 메모리를 요구한다.
    - 그러나, 속도가 빠른 메모리는 가격이 비싸 보통 **메모리 계층 구조**를 구성해 비용, 속도, 용량, 접근시간 등을 상호 보완한다.

메모리 계층 구조



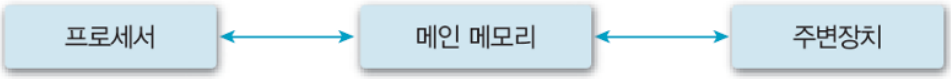
- 프로그램을 실행하거나 데이터를 참조하려면 모두 메인 메모리에 올려야 한다.
    - 그렇다고 해서, 고가의 메인 메모리를 무작정 크게 만들 수는 없는 노릇이다.
- **불필요한 프로그램과 데이터는 보조기억장치에 저장했다가, 실행 및 참조할 때만 메인 메모리로 옮기는 원리를 적용한다.**
    - 따라서, 메모리 계층 구조는 비용, 속도, 크기가 다른 **메모리를 효과적으로 사용**함으로써 **시스템 성능을 향상**시킨다.

레지스터

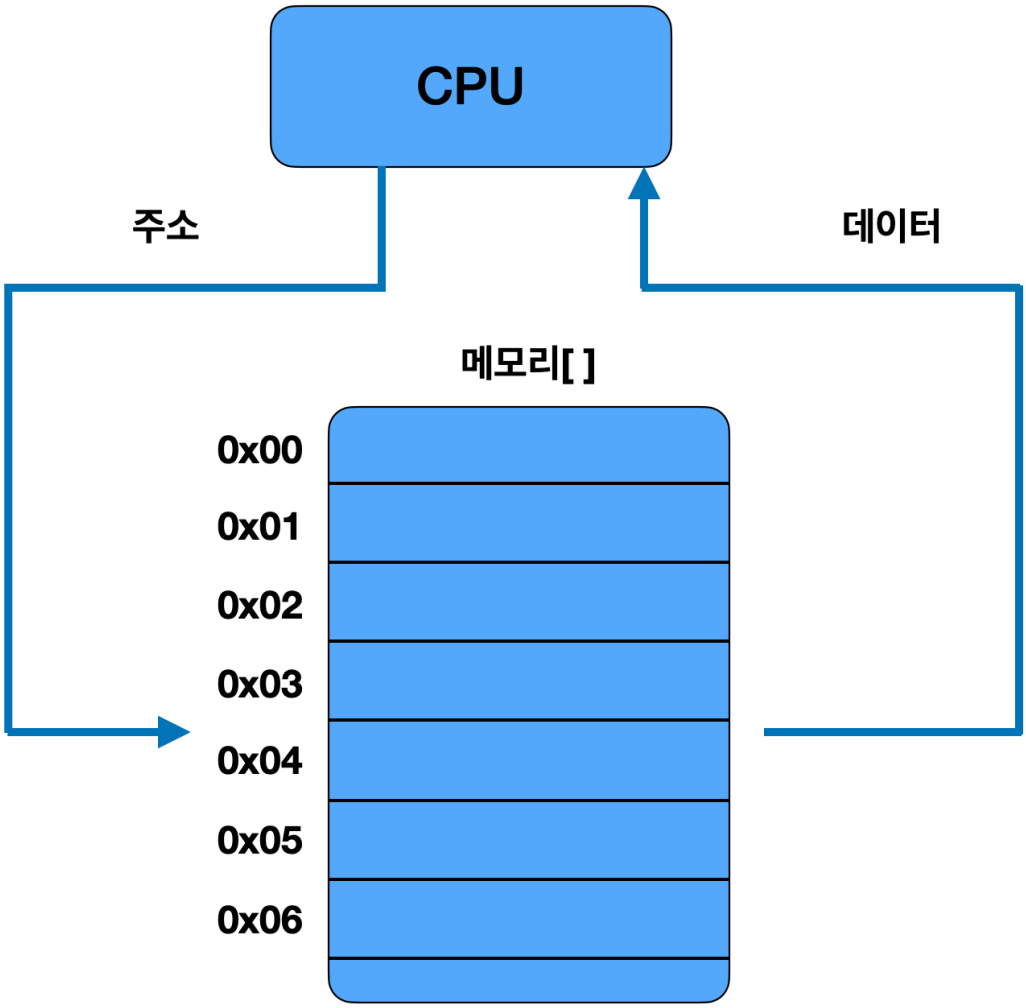
- **CPU 내부**에 있으며, **CPU가 사용할 데이터를 보관하는 가장 빠른 메모리**이다.

메인 메모리 (주기억장치)

- **CPU 외부**에 있으며, CPU에서 즉각적으로 수행할 프로그램과 데이터를 저장하거나 CPU에서 처리한 결과를 메인 메모리에 저장한다.



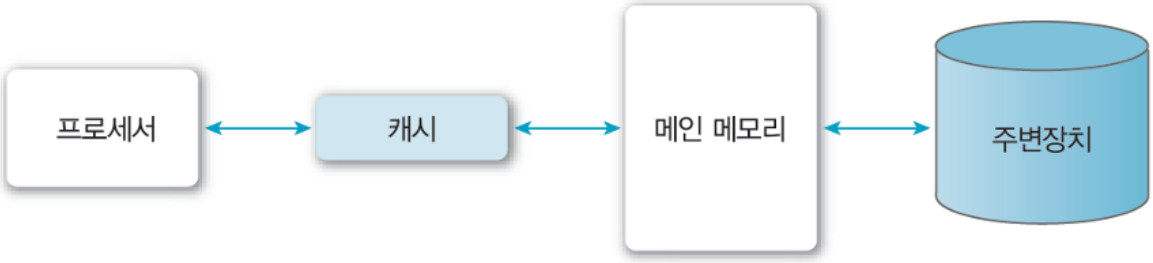
- **메인 메모리는 CPU(프로세서)와 보조기억장치 사이**에 있으며, 여기서 발생하는 **디스크 입출력 병목 현상**을 해결하는 역할도 한다.
    - 다만, CPU와 메인 메모리 간에 속도 차이가 있으므로 메인 메모리의 부담을 줄이기 위해 CPU 내부 혹은 외부에 캐시를 구현하기도 한다.



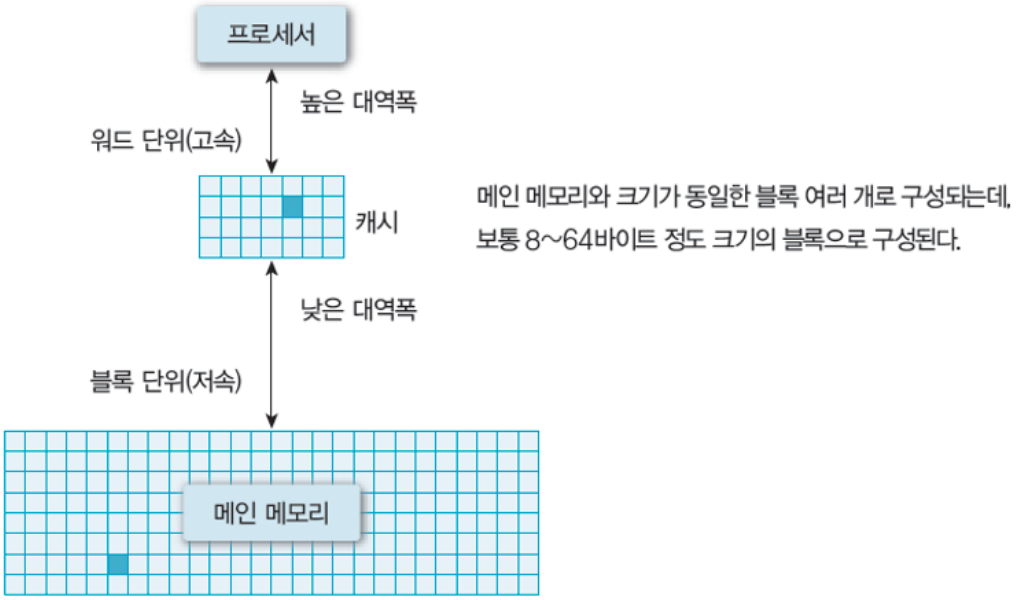
- **메인 메모리는 다수의 셀(cell)로 구성되며, 각 셀은 비트로 구성**된다.
  - 셀이 K 비트이면 셀에 2^K 값을 저장할 수 있다.
  - 메인 메모리에 데이터를 저장할 때는 **셀 한 개 혹은 여러 개에 나눠서 저장**한다.
  - **셀은 주소로 참조**하는데, n 비트라면 주소 범위는 $0 \sim 2^{n-1}$ 이다.

- 위의 내용처럼 **컴퓨터에 주어진 주소를 물리적 주소**라고 한다.
  - 그러나, 프로그래머는 **물리적 주소 대신 수식이나 변수를 사용**한다.
- 그리고 **컴파일러가 프로그램을 기계 명령어로 변환할 때 변수와 명령어에 주소를 할당**하는데, 이 주소를 **논리적 주소**라고 한다.
- **컴파일로 논리적 주소를 물리적 주소로 변환**하며, 이 과정을 **매핑** 혹은 **메모리 맵**이라고 한다.

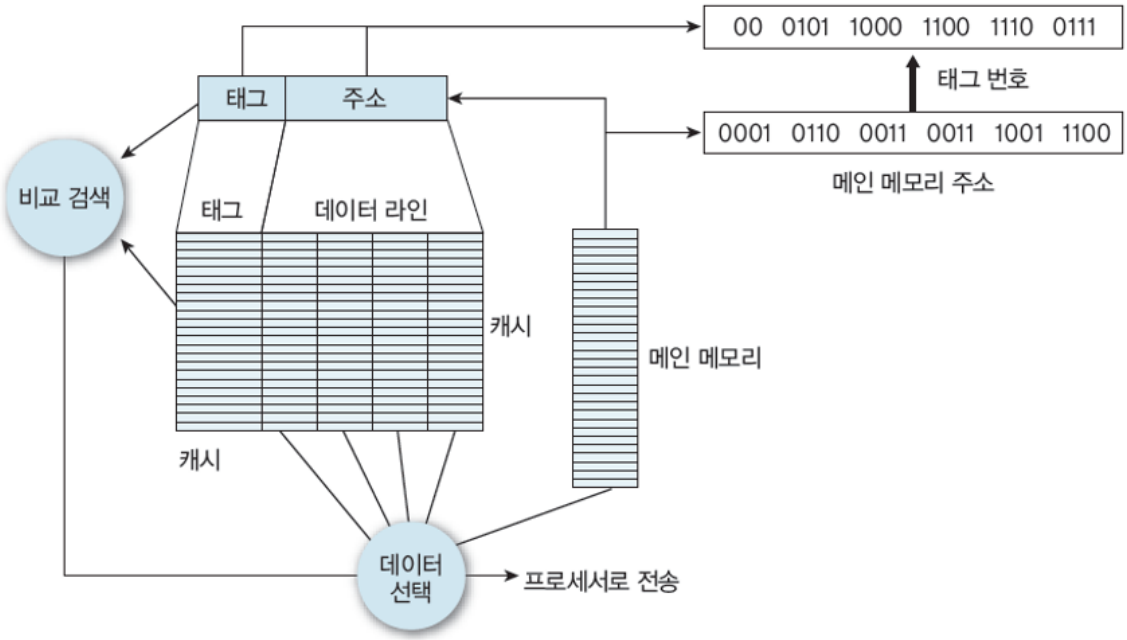
캐시



- **CPU 내부 혹은 외부**에 있으며, **처리 속도가 빠른 CPU와 상대적으로 느린 메인 메모리의 속도 차이를 보완**하는 **고속 버퍼**이다.



- 메인 메모리에서 데이터를 블록 단위로 가져와 CPU에 **워드 단위로 전달**해 속도를 높인다.
- 또한, 데이터가 이동하는 통로(대역폭)를 확대해 **CPU와 메인 메모리의 속도 차이를 줄여준다.**



- 캐시는 주소 영역을 한 번 읽어들일 수 있는 크기로 나눈 후, 각 블록에 번호를 부여해 이 번호를 **태그**로 저장한다.

- **캐시의 구체적 동작**은 아래와 같다.
  - 먼저 CPU는 메인 메모리 접근 전, 캐시에 해당 주소의 자료가 있는지 먼저 확인한다.
  - 이를 위해 접근하려는 주소 24비트 중 태그에 해당하는 첫 22비트를 캐시의 모든 라인과 비교해 일치하는 라인을 찾는다.

    `(접근하려는 주소 24비트 → 0001 0110 0011  0011 1001 1100)`

    `(태그에 해당하는 첫 22비트 → 00 0101 1000 1100 1110 0111)`
  - 일치하는 라인이 있다면, 주소의 나머지 2비트(00)을 이용해 데이터 라인의 4개 바이트 (00, 01, 10, 11) 중 해당하는 바이트를 가져온다.

- **캐시의 성능**은 작은 용량의 캐시에 **CPU가 이후 참조할 정보가 얼마나 들어있느냐**에 따라 좌우된다.
  - CPU가 참조하려는 정보가 있을 때를 **캐시 히트(cache hit)** 라고 한다.
  - 반대로, 참조하려는 정보가 없을 때를 **캐시 미스(cache miss)** 라고 한다.

- 이 때 블록의 크기는 캐시 성능으로 좌우되는데, 실제 프로그램 실행 시 참조한 메모리에 대한 **공간적 지역성**과 **시간적 지역성**이 있기 때문이다.
    - **공간적 지역성** → **대부분의 프로그램이 참조한 주소와 인접한 주소의 내용을 다시 참조**하는 특성

      `e.g.` → 프로그램이 명령어를 순차적으로 실행하는 경향이 있어 명령어가 특정 지역 메모리에 인접해 있다.
    - **시간적 지역성** → **한 번 참조한 주소를 곧 다시 참조**하는 특성

      `e.g.` → 순환으로 인해 프로그램을 반복하더라도 메모리는 일부 영역만 참조한다.

- 지역성은 **블록이 크면 캐시 히트율이 올라갈 수 있음**을 의미한다.
    - 그러나 **블록이 커지면 이에 따른 전송 부담과 캐시 데이터 교체 작업이 자주 일어나 블록 크기를 무한정 늘릴 수는 없다.**

보조기억장치

- 주변장치 중 프로그램, 데이터를 저장하는 하드웨어이다.
- 자기디스크, 광디스크, 자기테이프 등이 있다.

메모리 관리

- 모든 프로그램은 우선 메모리에 적재되어야 실행이 가능하므로, **메모리는 프로그램을 실행하는 중요 작업 공간**이다.
    - 또한, 다중 프로그래밍 환경에서는 **한정된 메모리**를 여러 프로세스가 함께 사용하므로 **이를 효율적으로 관리할 방법**이 필요하다.

- **메모리 관리**는 **프로세스들을 위해 메모리를 할당 및 제거하며 보호하는 활동**이다.
    - 프로세스의 요청에 따라 메모리 일부를 할당하고, 필요 없으면 자유롭게 재사용할 수 있도록 하는 것이다.

- 메모리 관리는 **메모리 관리자**가 담당한다.

    - **메모리 관리자**는 운영체제의 관리 모듈과 **메모리 관리장치(MMU, Memory Management Unit)가** 협력하여 관리한다.

- **메모리 관리자**는 **메모리와 관련된 여러 정책을 수립하고 정책에 따라 메모리를 관리**하는데, 주요 정책 세 가지는 아래와 같다.
    - **적재 정책**
        - **디스크에서 메모리로 프로세스를 반입할 시기를 결정**한다.
            - 요구 적재 → 참조 요청에 따라 다음에 실행할 프로세스를 메모리에 적재한다.
            - 예상 적재 → 요청을 미리 예측해 메모리에 적재한다.
    - **배치 정책**
        - **디스크에서 반입한 프로세스를 메모리 어느 위치에 저장할 것인지 결정**한다.
            - 최초 적합 → 첫 공백 분할 공간에 적재
            - 최적 적합 → 가장 작은 크기의 사용 공간을 적재
            - 최악 적합 → 가장 큰 사용 공간에 적재
    - **교체 정책**
        - **메모리가 충분하지 않을 때 현 메모리에 적재된 프로세스 중 제거할 프로세스를 결정**한다.
            - FIFO 대치 알고리즘
            - LRU 대치 알고리즘

메모리의 구조

- **메모리는 주소의 연속**으로, 이 주소는 크게 두 관점에서 해석이 가능하다.
    - **프로그래머가 프로그래밍에 사용하는 공간**으로 보는 논리적 관점의 **논리적 주소**
        - CPU에 의해 프로그램이 실행되고 있을 때 만들어진다.
        - CPU에 위치한 메모리의 물리적 주소를 가르킨다.
        - 가상 주소라고도 하며, 프로그램에서 사용하는 자료구조 등이 이에 속한다.
    - **실제 데이터나 프로그램을 저장하는 공간**으로 보는 물리적 관점의 **물리적 주소**

        - 논리적 주소에 대응하여 적재하는 실제 주소로, 메모리 칩이나 디스크 공간에서 만든다.
        - 사용자들은 직접적으로 물리적 주소로 접근하지 못하고 대신 대응되는 논리적 주소로 접근한다.

- **프로그램들은 논리적 주소를 생성하고 해당 프로그램이 이 논리적 주소에서 실행되고 있다고 상정**한다.
    - 그러나 **프로그램 실행을 위해서는 물리적 주소가 필요**하며, **논리적 주소에서 물리적 주소로의 변환**이 필요하다.

- **논리적 주소와 물리적 주소의 변환은 메모리 관리 장치(MMU)가 처리**한다.
    - 변환 과정에서 고정 분할, 동적 분할, 페이징, 세그멘테이션 등의 변환 방법을 사용할 수 있다.

- 그런데, **해당 프로세스의 논리적 주소에 대응하는 물리적 주소**를 알아야 CPU가 프로세스를 실행할 수 있을 것이다.
    - C, 자바와 같은 고급 프로그래밍 언어에서는 변수를 사용해 논리적 주소를 표시하고, 이를 실행하면 물리적 주소로 변환한다.

      ⇒ 이렇게 **두 주소를 연결시켜주는 작업을 바인딩**이라고 한다.

- 바인딩의 방식은 **프로그램이 적재되는 물리적 메모리의 주소가 결정되는 시기**에 따라 세 가지로 분류가 가능하다.

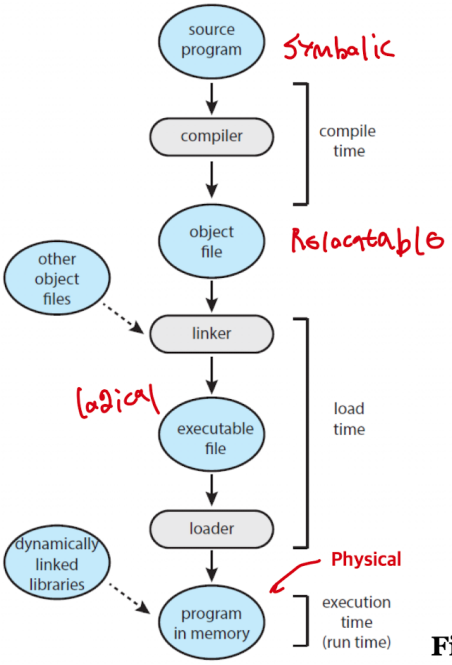


Figure 9.3 *Multistep processing of a user program.*

- **컴파일 시간**
  - 프로세스가 메모리에 적재될 위치를 컴파일 과정에서 알 수 있다면 컴파일러는 물리적 주소를 생성할 수 있다.
  - 즉, **프로그램 내부에서 사용하는 주소와 물리적 주소가 동일한 것을 의미**한다.
- **로드 시간**
  - **프로그램 실행이 시작될 때 물리적 메모리 주소가 결정되는 바인딩 방식을 의미**한다.
  - 로더의 책임하에 물리적 메모리 주소가 부여되며, **프로그램 종료까지 물리적 메모리 상 위치가 고정**된다.
  - 결과적으로 프로그램 내부에서 사용하는 주소와 물리적 메모리 주소는 다른 방식이다.
- **실행 시간**
  - **프로그램 실행 후에도 프로그램이 위치한 물리적 메모리상의 주소가 변경될 수 있는 바인딩 방식**을 의미한다.
  - 다른 방식들과 다르게 실행 시간에 바인딩이 이루어져 **MMU 의 지원이 필요**하다.

메모리 관리 관련 용어

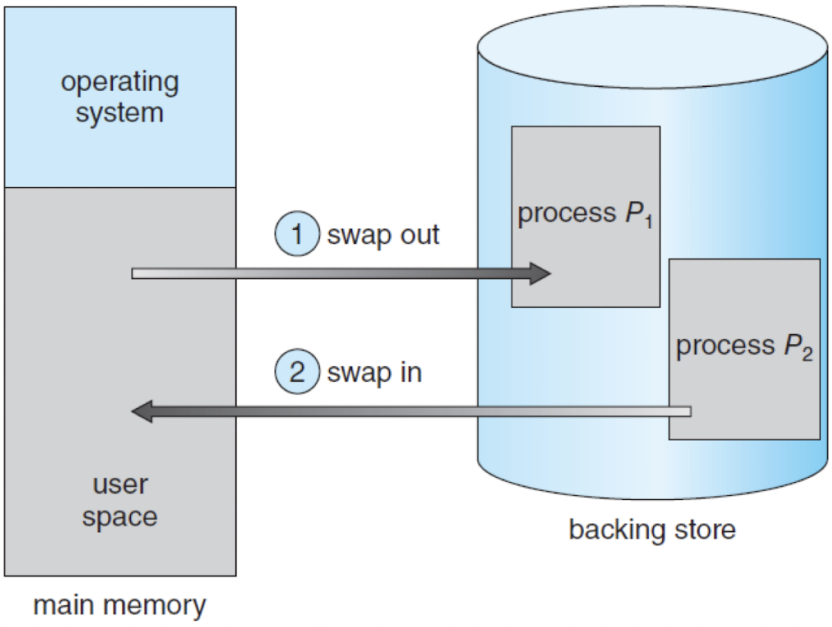
동적 로딩(Dynamic Loading)

- 프로그램, 데이터 **전체**를 물리 메모리 공간에 저장한다면, 효율이 떨어질 것이다.
- 이 때 동적 로딩을 이용하는데, 이 동적 로딩은 **모든 루틴을 한번에 로드하지 않음**으로서 메모리 공간의 효율적 사용을 도와준다.
- **즉, 오로지 필요할 때만 루틴을 로드하는 것이다.**

스와핑

- **실제 물리적 주소 공간보다 더 큰 프로세스를 실행하기 위해 사용**한다.
- 시스템 멀티 프로그래밍의 빈도를 조절하는 것으로, **메모리상의 프로세스의 수를 조절**할 수 있다.
- 프로세스의 명령어와 데이터는 메모리에 올라와 있어야 실행이 가능한데, **메모리가 부족한 경우 프로세스 혹은 프로세스의 일부를 메모리에서 백킹 스토어로 일시적으로 스왑**한다.
- 필요하면 백킹 스토어에서 다시 가져와 실행한다.

- **기본 스와핑(Standard Swapping)**



- 전체 프로세스가 백킹 스토어와 메인 메모리 사이를 스왑하는 것을 말한다.
- **전체 프로세스 스왑은 코스트가 커 시스템에 부담을 줄 수 있다.**

- **페이징 스와핑(Swapping with Paging)**
  - 전체 프로세스를 스왑하는 대신 **프로세스의 페이지를 스왑하는 방법**이다.
  - 물리적 메모리와 논리적 메모리를 분리할 수 있다.
  - **아주 작은 단위의 페이지도 스왑할 수 있다.**
  - 오늘날에는 페이징이라는 말은 스와핑 가능한 페이징을 일컫는다.
    - **page out** → 메모리에서 백킹 스토어로 페이지가 이동
    - **page in** → 백킹 스토어에서 메모리로 페이지가 이동
  - 이 방법은 **가상 메모리에서 큰 위력을 발휘**한다.

메모리 적재 방법

- **메모리에 프로세스를 적재하는 방법**으로 크게 두 가지가 있다.

  - **연속적**으로 적재하는 **연속 메모리 적재 방법**
  - **페이지** 혹은 **세그먼트** 단위로 나눠 여러 곳에 적재하는 **분산(비연속) 메모리 적재 방법**

단일 프로그래밍 환경에서 연속 메모리 할당

- 초기 컴퓨터 시스템은 사용자 한 명만 컴퓨터를 사용할 수 있었다.

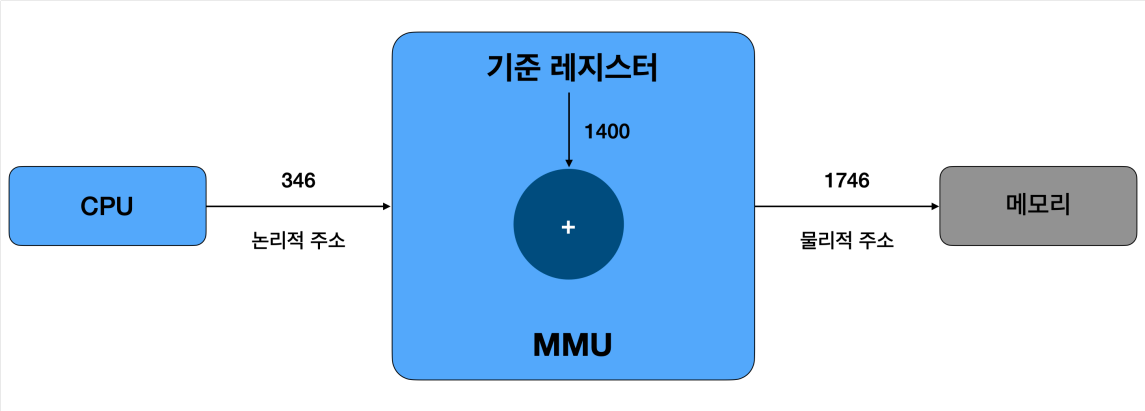
- 또한, 프로그램은 메모리보다 클 수 없고 직접 배치 과정을 수행해 항상 같은 메모리 위치에 적재되었다.
- **메모리를 제어하는 모든 권한이 사용자에게 있어 사용자가 주소를 잘못 지정하면 운영체제가 손상될 위험이 있다.**

  ⇒ 이러한 이유로, CPU에 **경계 레지스터**를 두어 **사용자 프로그램이 메모리 주소를 참조할 때마다 경계**

  레지스터를 검사한 후 실행되도록 설정하였다.

- 그러나 위 시스템의 문제는 컴퓨터 주소 공간이 0000부터 시작하더라도 **사용자 프로그램의 시작 주소는 기준 레지스터 값 이후가 된다는 단점**이 있다.
  - 만약, 기준 주소가 변경하면 처음부터 모든 프로그램을 다시 적재해야 한다.

- 위의 단점을 방지하기 위해, **주소 바인딩을 연기하는 방법**을 채택할 수 있다.



  - 기준 레지스터에 있는 값은 사용자 프로세스를 메모리로 보낼 때 생성하는 주소값에 더한다.
  - 기준 값이 1400이라면, 주소 0에서 사용자 프로세스는 동적으로 1400으로 대체되고, 주소 346의 접근은 1746으로 대체된다.

- 🤔 **기준 레지스터와 경계 레지스터**
  - 기준 레지스터는 가장 작은 물리적 주소를 저장하고, 경계 레지스터는 프로그램 영역이 저장되어 있는 크기를 저장한다.
  - 즉, 기준 레지스터는 물리적 주소이고 경계 레지스터는 논리적 주소이다.

- 단일 프로그래밍 환경에서의 연속 메모리 할당은 단순하고 이해하기 쉽다는 장점이 있다.
- 그러나 한 번에 프로그램 하나만 사용 가능하고, 메모리 효율성이 떨어진다.
    - 메모리보다 큰 프로그램은 수행할 수 없어 스와핑을 사용해 제한된 메모리를 확장한다.
    - 입출력 작업이 교대로 발생한다면 CPU가 쉬는 기간이 길어질 것이다.

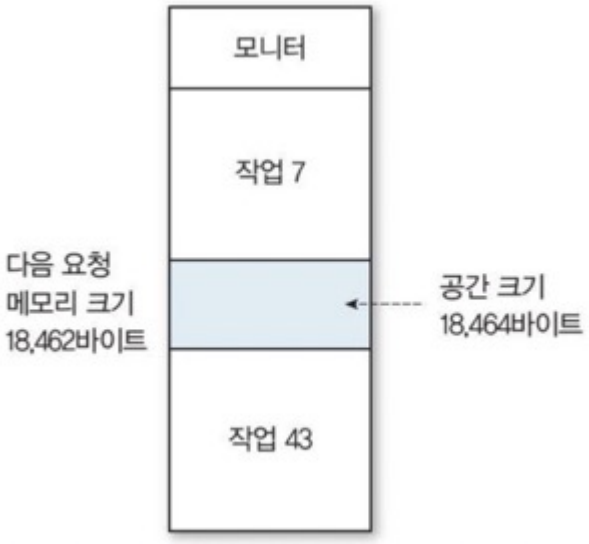
      ⇒ 자원 낭비가 심하고, 프로세스 스와핑 비용이 많이 든다.

다중 프로그래밍 환경에서 연속 메모리 할당

- 메모리를 여러 개 고정된 크기로 분할하는 고정 분할 방법, 다른 하나는 필요한 만큼만 메모리를 할당하는 가변 분할 방법으로 나눠진다.

고정 분할 방법

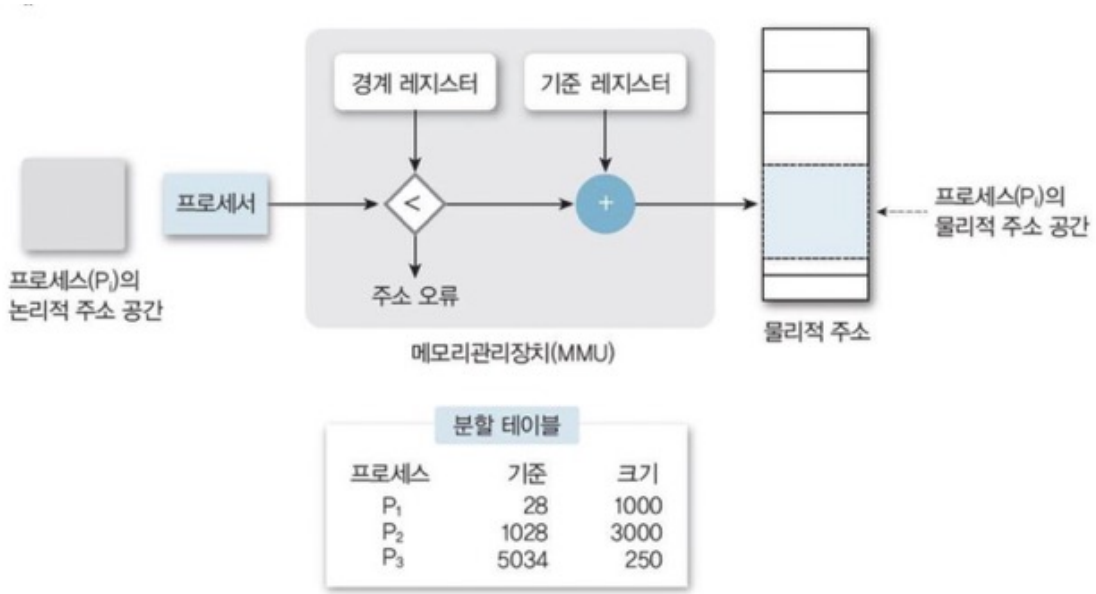
- 메모리를 여러 개의 고정된 크기로 분할하고, 분할된 각 메모리는 프로세스 하나를 실행할 수 있다.
- 물리적 주소는 분할 기준 레지스터값에 논리적 주소를 더해 생성한다.
- 고정 분할 방법에서는 논리적 주소가 분할된 메모리보다 크면 오류가 발생하고, 작으면 내부 단편화가 발생한다.
- **내부 단편화(internal fragmentation)**



- **프로세스가 필요로 하는 양보다 더 크게 메모리를 할당**해 공간이 낭비되는 상태를 의미한다.
- 위의 예시대로 메모리를 할당하면 2바이트가 남는데, 이 공간이 낭비된다.

가변 분할 방법

- 고정된 경계를 없애고 프로세스가 필요한 만큼 메모리를 할당하는 방법을 의미한다.



분할 테이블

프로세스	기준	크기
P_1	28	1000
P_2	1028	3000
P_3	5034	250

- **메모리 보호를 위해 경계 레지스터와 기준 레지스터 사이에서만 프로세스가 할당할 수 있게 한다.**

- 가변 분할 방법에서는 운영체제가 메모리의 어떤 부분을 사용하고, 사용할 수 있는지 알 수 있는 테이블을 유지해야 한다.

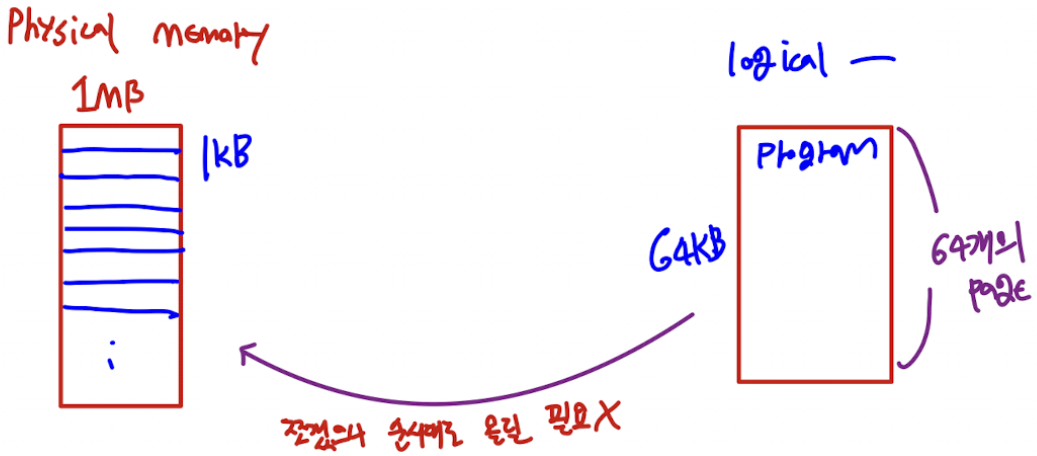
  - 위의 예로 사용 가능 공간이 메모리에 흩어져있고, 프로세스를 실행하려면 충분한 크기의 메모리 영역을 찾아야 한다는 사실을 알 수 있다.
  - 이 때 사용 가능한 블럭을 홀(hole) 이라고 칭한다.

- 홀을 어떤 식으로 할당하는 것이 좋은지 판단하기 위해 최초 적합, 최초 적합, 최악 적합 등을 들 수 있다.
  - **First-Fit** → 할당할 수 있는, 즉 크기가 맞는 홀이 보이면 바로 할당하는 방법
  - **Best-Fit** → 할당할 수 있는 가장 작은 홀에 할당하는 방법
  - **Worst-Fit** →할당할 수 있는 가장 큰 홀에 할당하는 방법

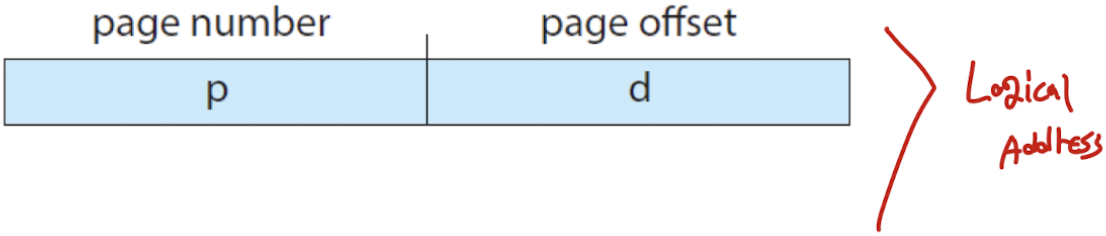
- 가변 분할에서는 외부 단편화 문제가 발생한다.
- **외부 단편화(external fragmentation)**
  - 메모리 공간이 작은 홀로 나뉘어져 있어 **충분한 메모리 공간이 있음에도 할당을 받을 수 없는 상태를 의미**한다.
  - 50MB가 할당이 가능한 상태에서, 3MB의 메모리 할당 요청이 들어와도 2MB로 메모리가 나뉘어져 있어 할당을 받지 못한다.
  - 이를 해결하기 위해서는 사용중인 메모리 영역을 한 곳으로 몰고 홀들도 한 곳으로 합쳐 큰 메모리 공간을 만드는, 즉 **압축**으로 해결한다.

분산 메모리 할당 1, 페이징

- 연속 메모리 할당과는 반대로, **프로세스의 물리적 주소 공간을 연속적이지 않게 배치하는 방법**이다.
- **외부 단편화 문제와 압축 문제를 해결할 수 있는 방법**이다.
- 운영체제와 컴퓨터 하드웨어의 도움을 받아 구현할 수 있다.
- **페이징의 기본적인 방법**
  - 물리적 메모리를 고정된 크기로 잘라 **프레임(frames)**으로 나눈다.
  - 논리적 메모리를 프레임과 같은 크기, 즉 고정된 크기로 잘라 **페이지(page)**로 나눈다.



- 연속 메모리 할당 때처럼 프로그램 전체를 한번에 올릴 필요 없이, **조각을 물리적 메모리의 위치에 맞게 할당**시키면 된다.
- 논리적 주소 공간과 물리적 주소 공간이 완전히 분리되어 물리적 주소 공간을 크게 고려하지 않아도 되고, **이 둘 사이의 매핑은 운영체제가 알아서 진행**해준다.



- CPU에 의해 생성되는, 즉 논리적 주소는 두 파트로 분류된다.
- **페이지 넘버와 페이지 간격**으로 분류된다.
  - 이는 하나의 프로세스라도 한번에 물리적 메모리 공간에 올리는 것이 아니라 **페이지 단위로 물리적 메모리에 올리는 위치가 모두 다르기 때문에** 위의 두 파트를 이용하는 것이다.
- 두 파트를 이용하여 **어떤 프로세스의 몇 번째 페이지가 물리적 메모리의 어느 공간에 위치해 있는지** 알 수 있다.

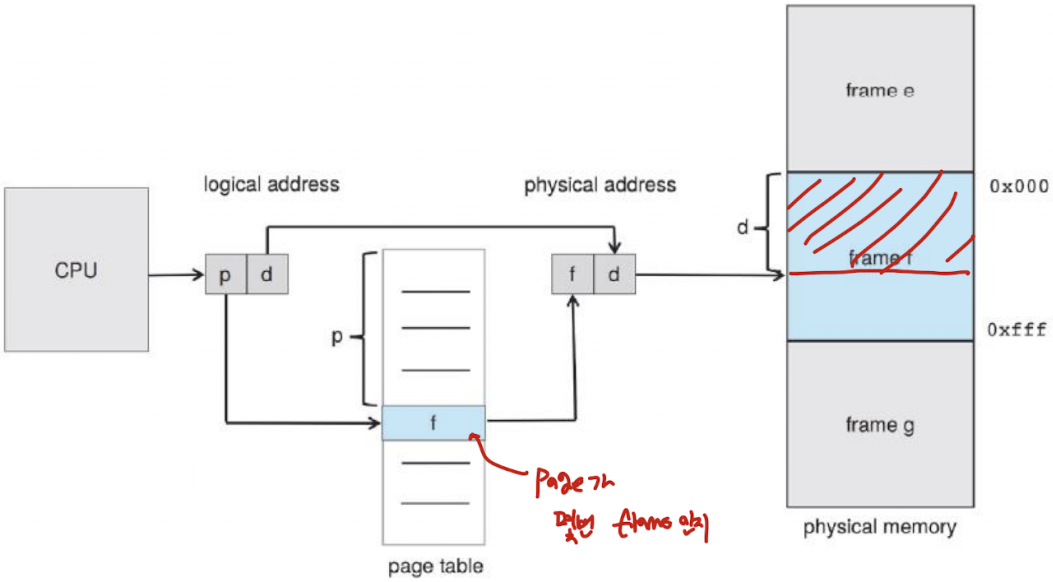


Figure 9.8 *Paging hardware.*

- 모든 프로세스는 주소 변환을 위해 페이지 테이블을 가진다.
- 페이지 넘버는 **프로세스별 페이지 테이블의 인덱스값**으로 사용된다.
  - 프로세스 1의 페이지, 프로세스 3의 페이지가 섞이면 안되니, 페이지 테이블을 통해 프로세스의 페이지들이 연달아 실행될 수 있도록 하는 것이다.

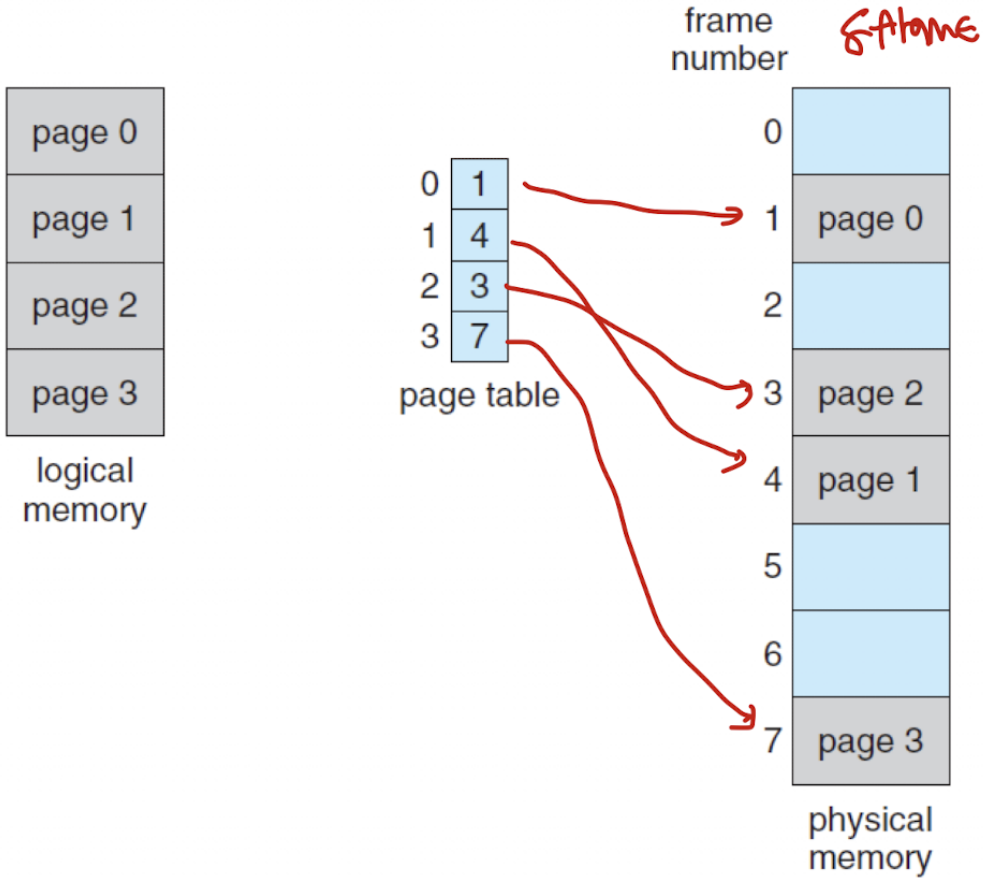


Figure 9.9 *Paging model of logical and physical memory.*

- **CPU 실행의 순서**
  1. 페이지 번호 P를 추출하여 페이지 테이블의 인덱스로 사용한다.
  2. 페이지 테이블에서 대응하는 프레임 번호 F를 추출한다.
  3. 페이지 번호 P를 프레임 번호 F로 바꾼다.

- 그렇다면, **페이지 사이즈**는 어떤 식으로 정해지는 걸까?
  - 이는 하드웨어에 의해 정해지는데, 기본적으로 **2의 배수**여야 하고 사이즈는 보통 4KB에서 1GB 정도 차지한다.

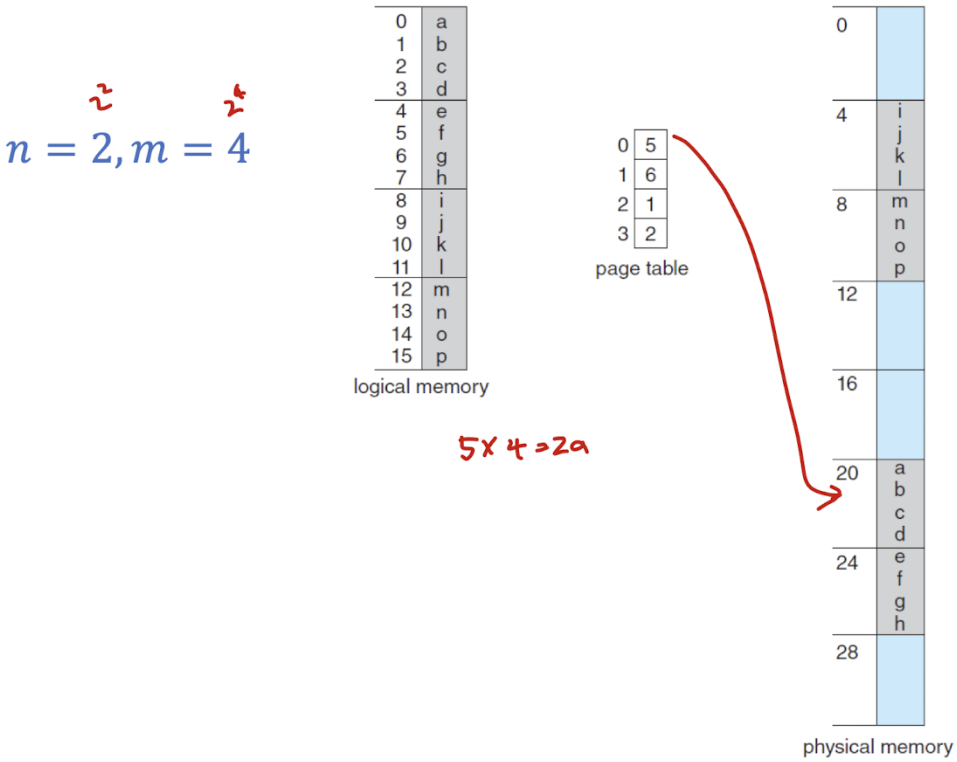
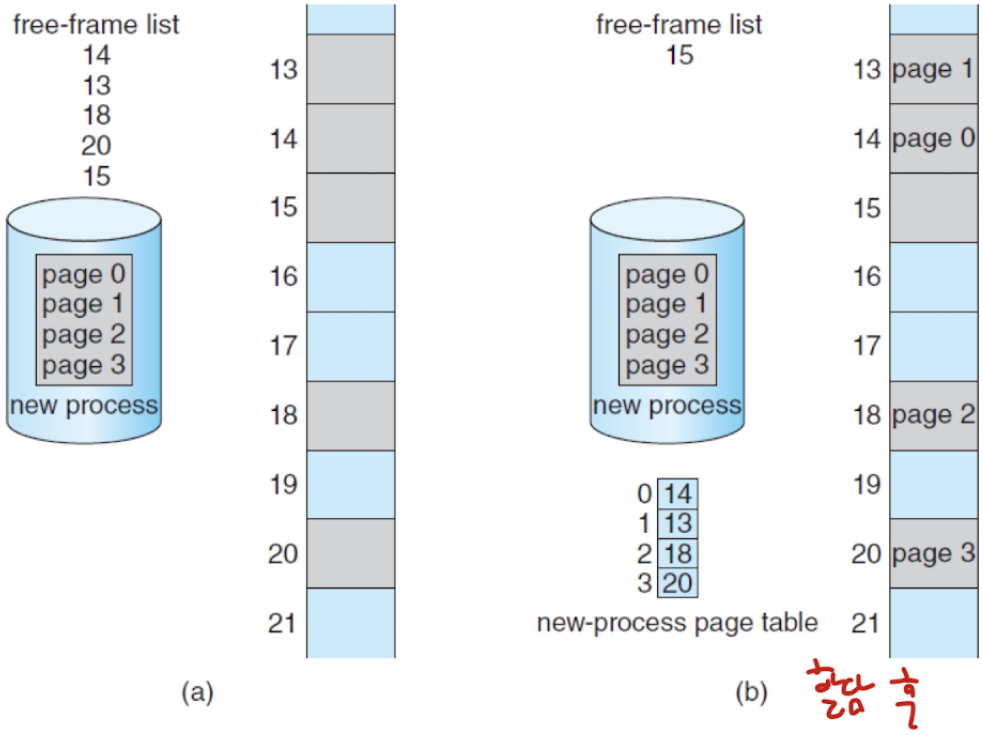


Figure 9.10 *Paging example for a 32-byte memory with a 4-byte pages.*

- 논리 주소 공간의 크기가 2^m 이고 페이지 크기가 2^n일 때, 페이지 넘버는 $m - n$, 페이지 간격은 n이 된다.
  - $2^4 = 16, 2^2 = 4$
  - 페이지 넘버 = $4 - 2, 2^2 = 4$
  - 페이지 간격 = $2, 2^2 = 4$

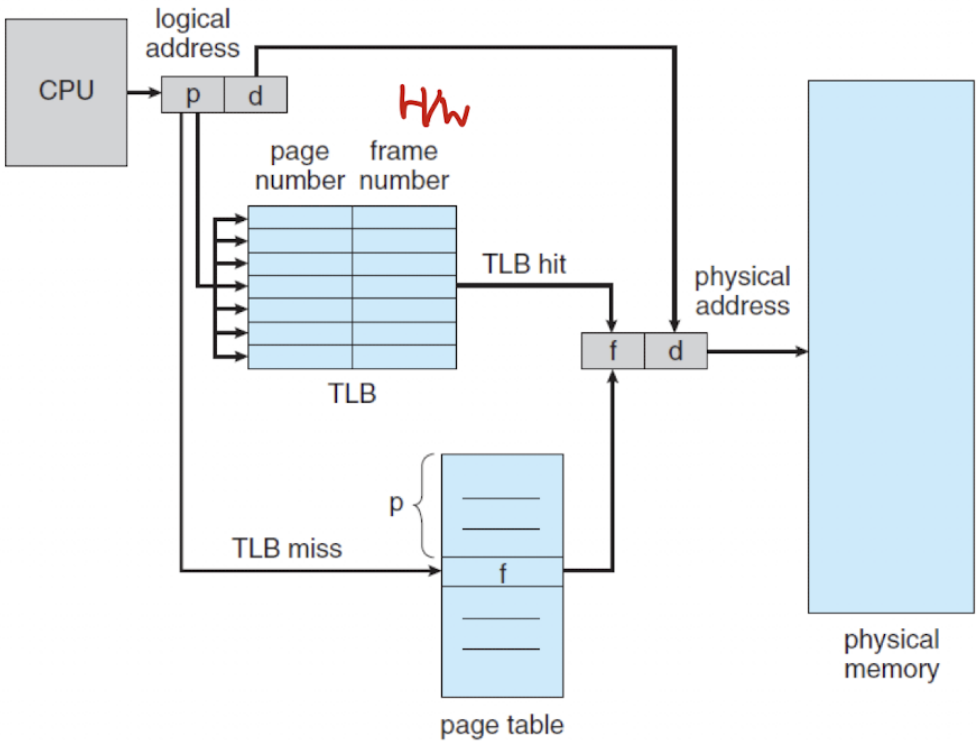


- 할당되지 않은 프레임들은 리스트로 관리되다가, 프로세스가 실행될 때 순차적으로 페이지를 메모리에 할당한다.

PTBR(page-table base register)

- 페이지 테이블의 크기가 갈수록 무거워지고 다양해지면서, **하드웨어만으로 페이지 테이블을 관리하는 것이 힘들어졌다.**
  - **PTBR은 메인 메모리에 있는 페이지 테이블의 시작 번지**를 가리킨다.
  - 컨텍스트 스위칭의 속도는 빨라졌지만, 여전히 메모리 접근 시간은 느린데, **그 이유는 두 번이나 메모리 접근을 실행하기 때문**이다.
    - 메인 메모리에 있는 페이지 테이블에 접근
    - 페이지 테이블 내에 있는 실제 데이터에 접근

TLB(Translation Look-aside Buffer)



- 그래서, **메모리 접근 시간도 향상시키기 위해 캐시 메모리, 즉 TLB를 거쳐서 페이지 테이블에 접근하는 방식**이 나오게 된다.

- **캐시 메모리**
  - 메인 메모리에서 빈번하게 사용하는 데이터를 캐시에 저장해 CPU에 더 빨리 접근할 수 있게 해주는 하드웨어이다.

- TLB에는 **빈번하게 사용되는 페이지 테이블 일부를 저장**하고 있고, CPU에서 메인 메모리의 페이지 테이블에 접근하기 전에 TLB에 접근하여 페이지 정보가 있다면 곧바로 주소 변환이 이루어진다.
  - **TLB hit** : TLB 안에 포함되어 있는 페이지 넘버
  - **TLB miss** : TLB 안에 포함되어 있지 않은 페이지 넘버
  - **hit radio** : TLB hit 발생률, 이에 따라 EAT(Effective Memory-Access Time)이 조정된다.

- 시스템의 메모리 접근 시간이 10ns 일 때 계산 방법

- 80% hit radio 의 EAT : 0.80 X 10 + 0.20 X 20 = 12ns
- 99% hit radio 의 EAT : 0.99 X 10 + 0.01 X 20 = 10.1ns

Memory Protection with Paging

- 페이징에서의 메모리 보호는 **각 프레임에 관련된 보호 비트**에 의해 이루어진다.

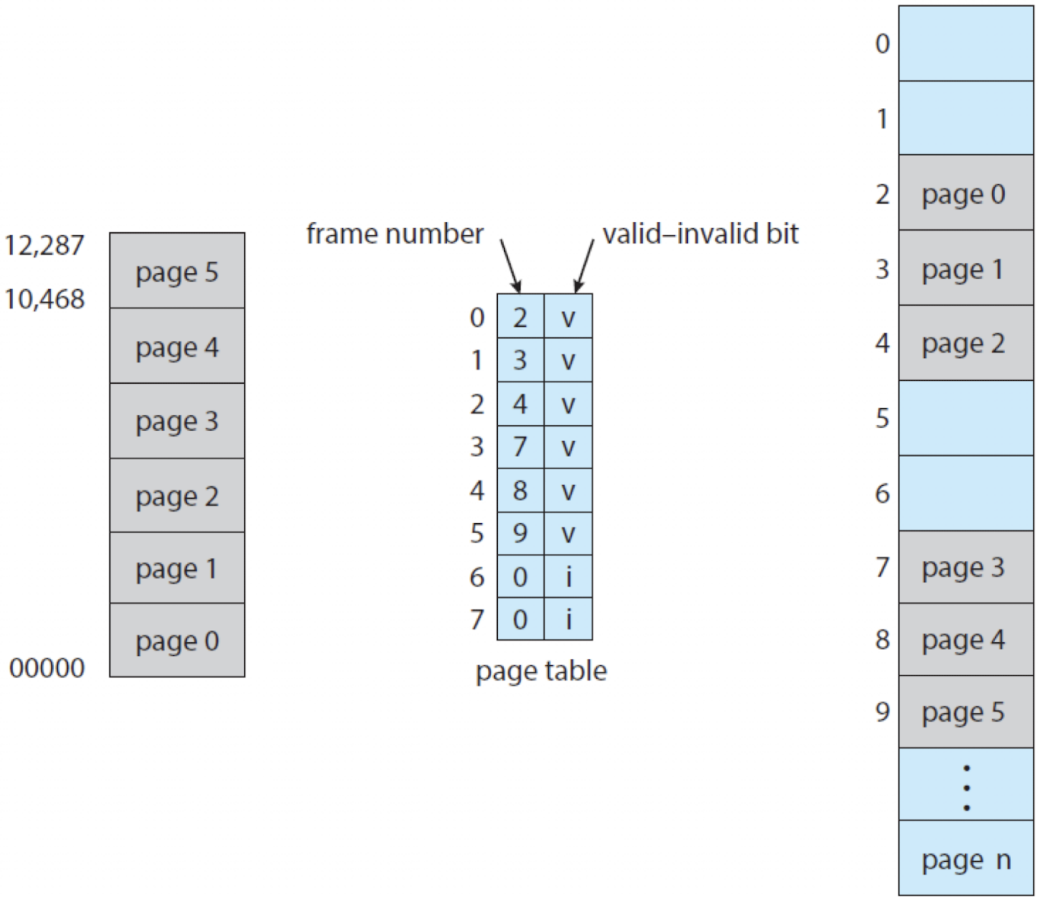


Figure 9.13 *Valid bit (v) or invalid bit (i) in a page table.*

- **유효-무효 비트(valid-invalid bit)**는 각 페이지 테이블에 모두 첨부되어 있다.
  - **비트가 유효한 경우**
    - 해당 페이지가 프로세스의 논리적 주소 공간에 포함되어 있다는 의미
  - **비트가 무효할 경우**
    - 논리적 주소 공간에 포함되어 있지 않은 페이지, 즉 프로세스가 주소 부분을 사용하지 않거나 백킹 스토어에 접근하여 권한이 없는 경우

- 불법 주소가 들어오면 유효-무효 비트를 이용해 인터럽트를 건다.

Shared Pages

- 페이징의 장점 중 하나로 **공통 코드를 공유할 수 있다.**
  - 이는 멀티 프로그래밍 환경에서 큰 장점으로 작용될 수 있다.
  - 예를 들어 C에서 `libc` 라이브러리를 각 프로세스가 복사해서 로드한다면 매우 비효율적이므로 재입력 코드(Reentrant Code), **즉 실행 중에 변할 일이 없는 코드는 프로세스끼리 공유하는 것이 효율적**이다.

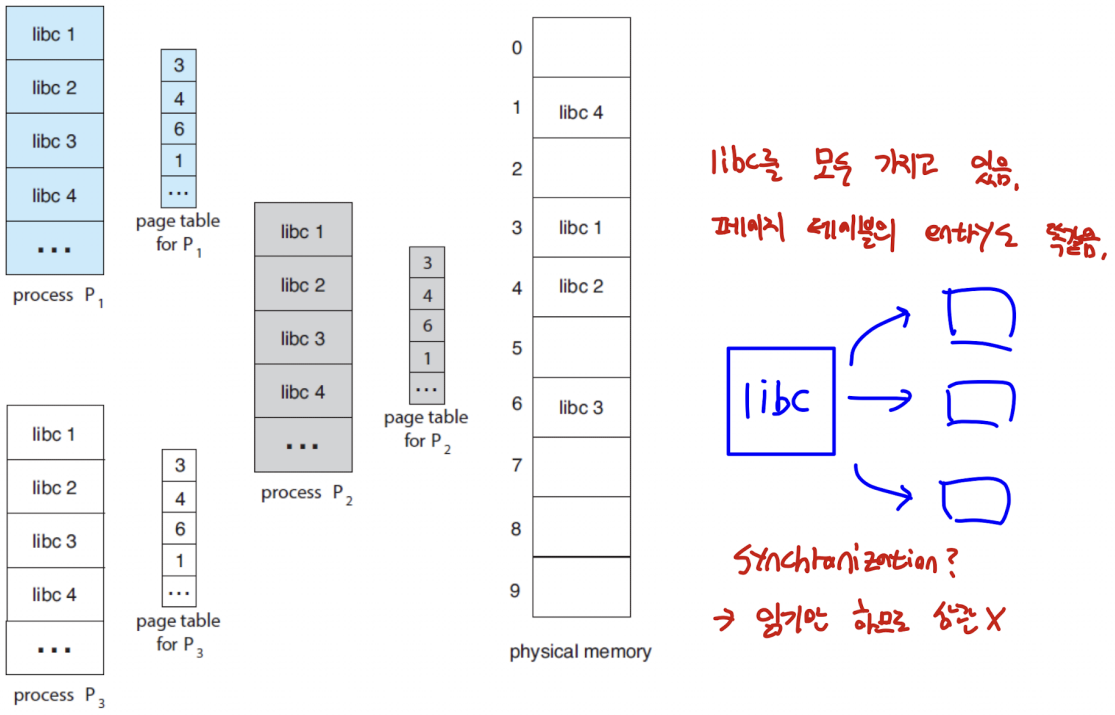


Figure 9.14 *Sharing of standard C library in a paging environment.*

- 하나의 물리 메모리를 공유한 모습을 확인할 수 있다.
- 프로세스들은 읽기만 하고 있으므로 **동기화 문제 역시 발생하지 않는다.**