# 1. 물리 메모리의 한계

## 1.1 주소 공간과 물리 메모리

> 1개의 주소가 가리키는 크기는 메모리 한 바이트이므로, 32비트 CPU가 액세스할 수 있는 물리 메모리의 최대 크기는 $2^{32}$ 바이트 = 4GB

32비트 CPU 시스템에서 프로세스가 실행 중에 닿을 수 있는 최대 주소 범위는 $2^{32}$=4GB 크기이다. 이것을 **프로세스의 주소 공간**이라 한다. 프로세스의 주소 공간은 고정 크기이지만 물리 메모리의 크기는 설치하기에 달려 있으며, 대부분의 시스템에서 물리 메모리는 비용 때문에 프로세스의 주소 공간보다 작게 설치된다.

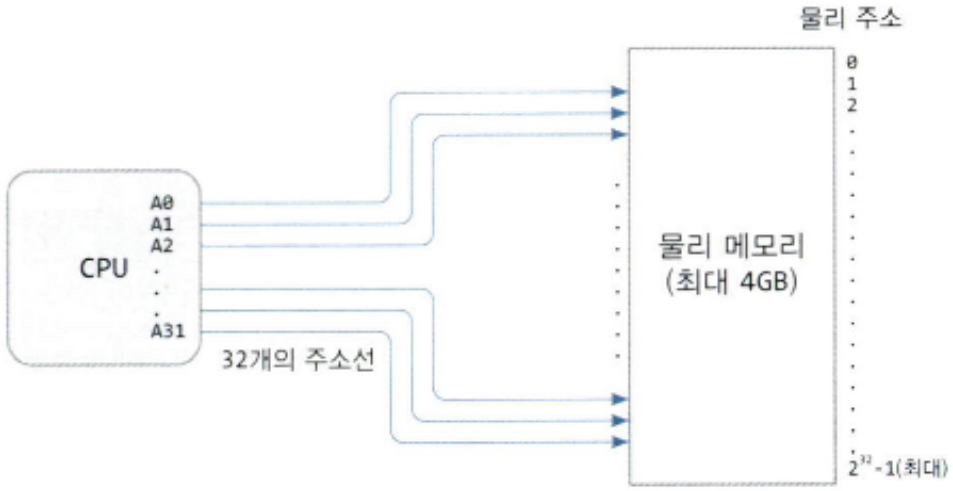
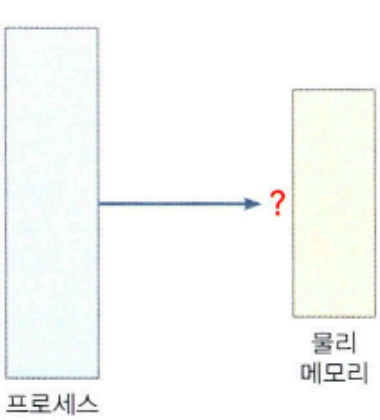


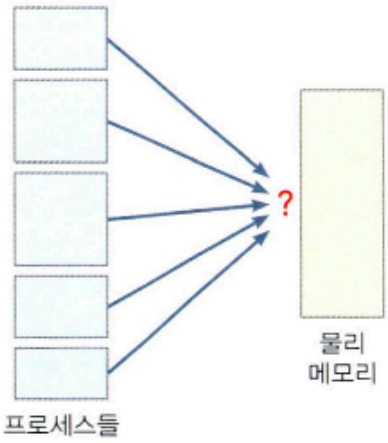

그림 10-1 32비트 CPU에서 물리 메모리의 최대 크기는 4GB

- 운영체제는 물리 메모리보다 큰 프로세스를 실행할 수 있는가?

  예를 들어, 컴퓨터에는 2GB RAM이 설치되어 있는데, 프로세스가 실행 중에 동적 할당받아 프로세스가 사용하는 메모리량이 2GB보다 크게 되면 이 프로세스의 실행이 가능한가 하는 문제이다.




(a) 프로세스가 물리 메모리보다 큰 경우




(b) 여러 프로세스들을 합친 크기가 물리 메모리보다 큰 경우

- 운영체제는 여러 프로세스를 합쳐 물리 메모리보다 클 때 이들을 동시에 실행시킬 수 있는가?

  예를 들어, 5개의 프로세스가 동시에 실행된다. 처음에 이들은 메모리를 많이 활용하지 않았지만, 스택과 힙이 계속 늘여가서 이들이 사용하는 메모리가 설치된 물리 메모리보다 커지게 될 때, 정상적으로 실행시킬 수 있는가하는 문제이다.

# 2. 가상 메모리 개념

## 2.1 가상 메모리 개요

**가상 메모리(virtual memory)**는 물리 메모리보다 큰 프로세스나 여러 개의 프로세스를 동시에 실행시켜, 사용자나 응용프로그램에게 무한대의 메모리가 있다고 느끼도록 하는 메모리 관리 기법이다.

가상 메모리 기법은 '프로세스가 실행되기 위해서는 코드, 데이터, 스택, 힙 등 모든 영역들이 물리 메모리에 적재되어 있어야 한다'는 전제를 깨고, 프로세스의 적재 공간을 메모리에서 보조 기억 장치의 영역으로 확장한다.

- 물리 메모리를 디스크 공간으로 확장
- 스와핑(swapping)

### 가상 메모리 개념

첫째, 운영체제는 물리 메모리의 영역을 하드 디스크까지 연장하고, 프로세스를 물리 메모리와 하드 디스크에 나누어 저장한다. 이렇게 함으로써 물리 메모리의 크기 한계를 극복한다.

둘째, 프로세스가 실행될 때 전체가 물리 메모리에 적재되어 있을 필요는 없다. 실행에 필요한 부분만 메모리에 적재하고 나머지는 하드 디스크에 저장해두고 실행에 필요할 때 물리 메모리로 이동시킨다. 실행 속도 저하가 있겠지만 물리 메모리가 부족하니 어쩔 수 없는 선택이다.

셋째, 물리 메모리에 빈 영역이 부족하게 되면, 운영체제는 프로세스를 구분하지 않고 물리 메모리의 일부분을 하드 디스크에 옮겨 빈 영역을 확보한다. 이렇게 되면, 극한 경우 모든 프로세스에 있어 당장 실행에 필요한 최소한의 부분만 물리 메모리에 남기고 나머지는 하드 디스크에 존재하게 된다. 이런 방식은, 가능한 많은 프로세스들을 메모리에 적재함으로써 다중프로그래밍 정도를 높여 CPU 활용률을 높이고 시스템의 처리율을 높인다.

넷째, 물리 메모리를 확장하여 사용하는 디스크 영역을 스왑 영역(swap area)라고 부른다. 물리 메모리에 빈 영역을 만들기 위해 물리 메모리의 일부를 디스크로 옮기는 작업을 스왑-아웃(swap-out), 스왑 영역으로부터 물리 메모리로 적재하는 과정을 스왑-인(swap-in)이라고 부른다.

다섯째, 가상 메모리 기법이 사용되면, 사용자는 무한대에 가까운 물리 메모리가 있어 물리 메모리의 크기를 걱정하지 않고 큰 프로그램을 작성할 수 있고, 걱정없이 여러 프로그램을 동시에 실행시킬 수 있다. 한편, 프로세스는 0번지부터 연속적으로 존재한다고 생각하며, 어떤 부분이 물리 메모리에 있고, 어떤 부분이 하드 디스크에 있는지 알지 못한다. 운영체제는 프로세스별로 어떤 부분이 물리 메모리에 적재되어 있고 어떤 부분이 하드 디스크에 있는지 유지 관리하고 프로세스에게 최소한의 물리 메모리를 할당하여 최대한 많은 프로세스를 실행시키는데 집중한다.

여섯째, 가상 메모리는 운영체제마다 구현 방법이 다르다.

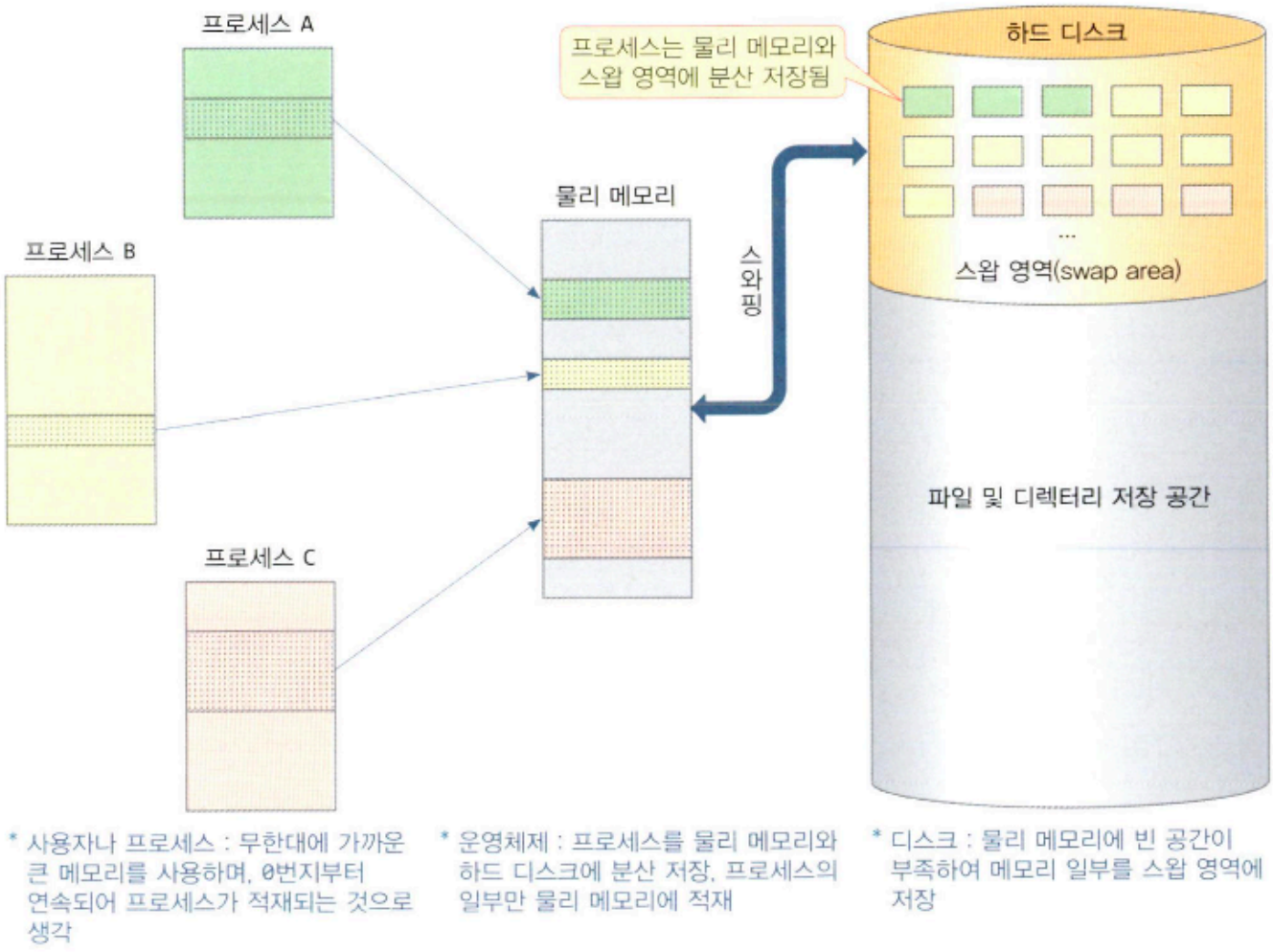



그림 10-3 가상 메모리를 사용하는 시스템에서 프로세스와 물리 메모리, 하드 디스크의 관계

## 논리 주소와 가상 주소

프로그램/프로세스 내에서 사용하는 주소를 논리 주소라고 불렀는데, 논리 주소를 **가상 주소(virtual address)**라고도 한다. 컴파일러 입장에서 보면 프로그램 내에서 코드와 변수의 주소는 프로그램의 내에서의 상대적인 주소이므로 논리 주소라고 부르는 것이 적합하지만, 가상 메모리를 다루는 운영체제 입장에서는 프로세스의 가상 주소 공간 내에서 사용되는 주소를 가상 주소라고 부르는 것이 더 적합하다.

**malloc()이 메모리가 부족하여 null을 리턴하는 경우, 물리 메모리가 부족한 것인가?**

> 가상 메모리를 사용하면, 프로세스는 설치된 물리 메모리보다 많은 메모리를 사용할 수 있다고 했다. malloc()이 null을 리턴하는 것은 프로세스의 사용자 주소 공간 내에 힙 영역이 부족하다는 것이다. malloc()을 호출한 프로세스가 자신의 사용자 주소 공간을 최대로 사용하여 더 이상 사용할 주소 공간이 없다는 것이다.

## 2.2 가상 메모리 구현

가상 메모리 기법은 운영체제에서 다음 2가지 방법으로 구현되고 있다.

- 요구 페이징(demand paging)
- 요구 세그먼테이션(demand segmentation)

> 요구 페이징은 페이징 기법을 토대로 프로세스의 일부 페이지들만 메모리에 적재하고 나머지는 하드 디스크에 두며, 페이지가 필요할 때 메모리를 할당받고 페이지를 적재시키는 메모리 관리 기법이다. 즉, 요구페이징 = 페이징 + 스와핑

한편, 요구 세그먼테이션은 세그먼테이션 기법을 토대로 하며, 프로세스를 구성하는 일부 세그먼트들만 메모리에 적재해두고, 다른 세그먼트가 필요할 때 메모리를 할당받아 세그먼트를 적재하는 메모리 관리 기법이다. 현대 운영체제는 대부분 요구 페이징 기법을 사용한다.

# 3. 요구 페이징(demand paging)

## 3.1 요구 페이징 개념

요구 페이징(demand paging)은 페이징 기법을 기반으로 만들어진 가상 메모리 기법이다. 프로세스의 페이지들을 물리 메모리와 하드 디스크에 분산 할당하는 방식으로, 현재 실행에 필요한 일부 페이지들만 물리 메모리에 적재하고 나머지는 하드 디스크에 둠으로써 제한된 물리 메모리에 많은 프로세스를 실행시킨다.

요구 페이징을 사용하는 운영체제는 프로세스를 실행시킬 때 실행에 필요한 첫 페이지만 물리 메모리에 적재하여 실행시키고, 실행 중 물리 메모리에 없는 페이지를 참조하게 되었을 때, 물리 메모리의 빈 프레임을 할당받고 이곳에 페이지를 적재한다.

> 요구 페이징에서 '**요구(demand)**'는 페이지가 필요할 때까지 물리 메모리에 적재하지 않고 두었다가, **페이지가 필요할 때 물리 메모리를 할당받고 디스크에서 읽어 적재**시킨다는 의미이다.

프로세스가 실행되고 시간이 흐르게 되면 실행에 필요한 페이지들이 물리 메모리에 하나씩 적재되고, 물리 메모리가 부족하게 되면 다시 일부 페이지들이 디스크로 쫓겨나게 된다.

## 3.2 요구 페이징 구성

그림 10-4는 5개 페이지로 구성되는 프로세스의 사례로, 현재 3개의 페이지가 물리 메모리에 적재되어 있고 2개의 페이지는 하드 디스크의 스왑 영역에 저장된 상태이다. 처음에 프로세스의 페이지들이 실행 파일로부터 하나씩 메모리에 적재되어 5개의 페이지가 모두 적재되었지만, 운영체제가 다른 프로세스의 페이지들을 메모리에 적재하기 위해 페이지 2와 페이지 3이 적재되었던 프레임들을 비우고 이들을 스왑 영역에 저장한 상태이다.

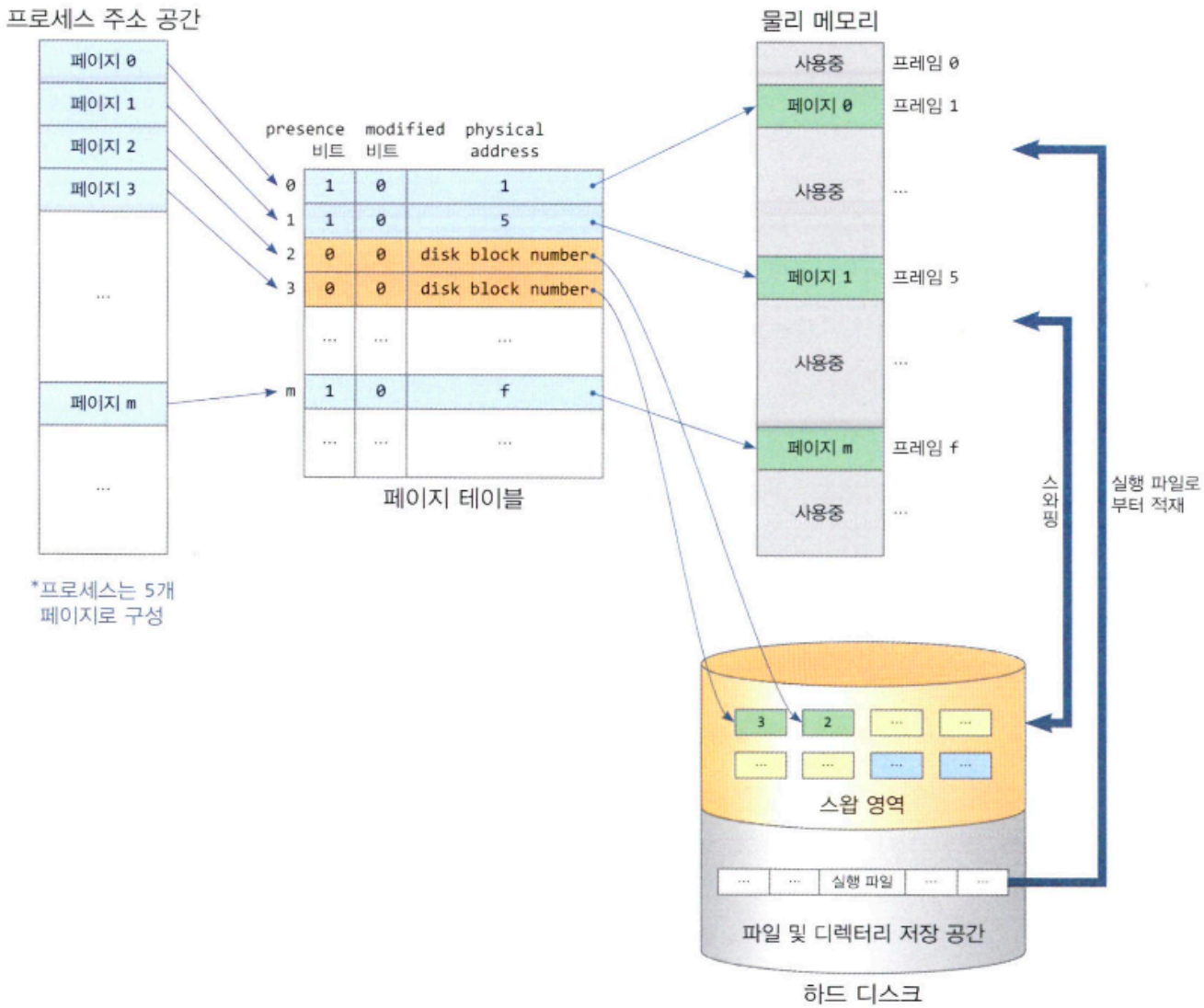



그림 10-4 요구 페이징의 개념과 구성

### 디스크의 스왑 영역

스왑 영역은 메모리에서 쫓겨난 페이지들이 저장되는 영역이다. 스왑 영역은 보통 운영체제 설치 시 시스템 관리자가 위치와 크기를 결정한다.

> **스왑 영역의 크기는 얼마가 적당할까?**
> 최근 들어 컴퓨터의 메모리 량이 16GB, 32GB, 64GB 등으로 매우 커졌기 때문에 스왑 영역을 물리 메모리의 $1/2$ 정도로 정할 것을 추천하고 있다. 대체로 **2GB~32GB** 정도 설정하는 추세이다.

## 페이지 테이블

요구 페이징을 위해 페이지 테이블에 다음 3개의 필드가 사용된다.

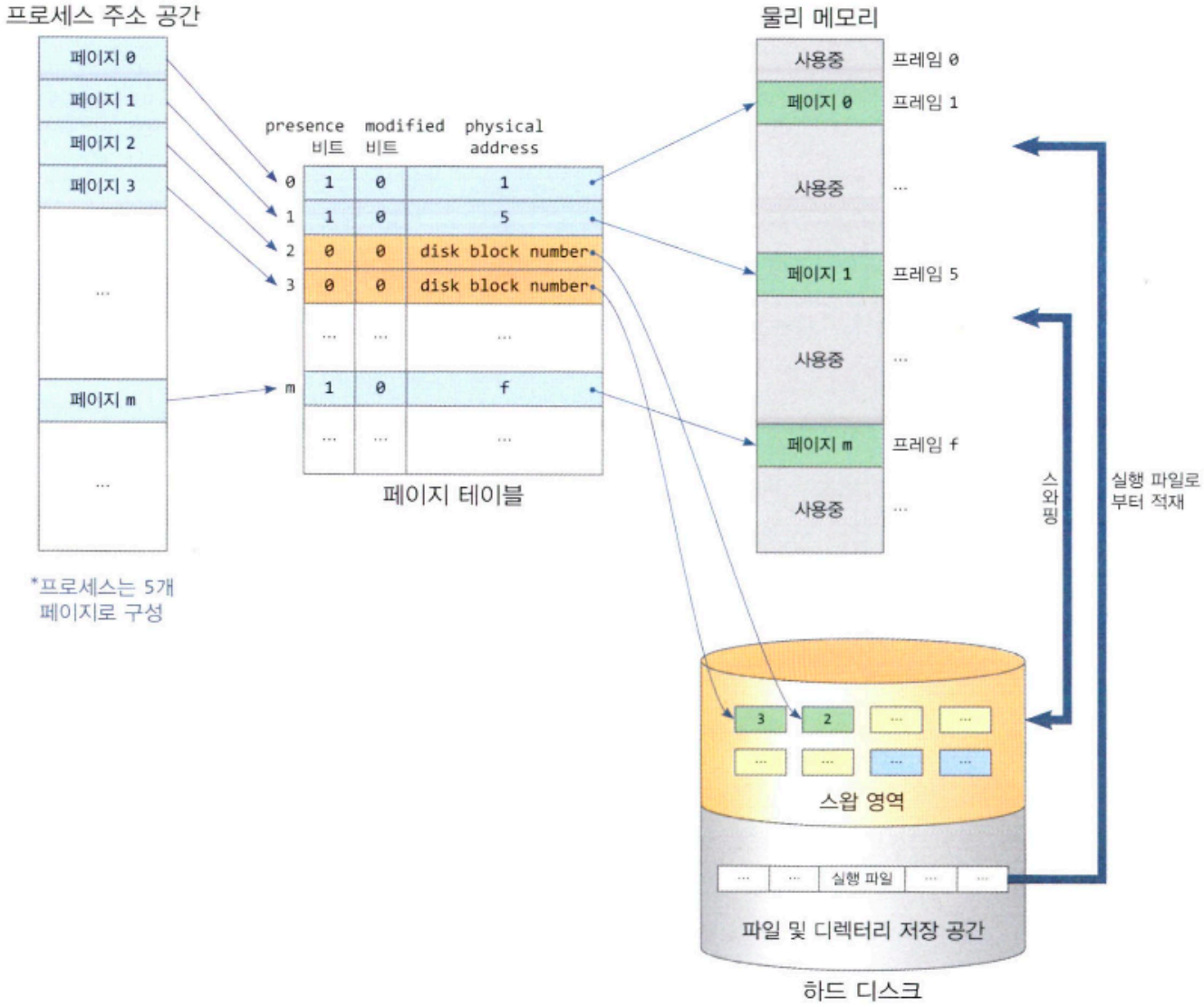
- presence/valid 비트 - 페이지가 메모리에 적재되어 있는지를 나타내는 비트
  - 1: 물리 메모리에 있음, 0: 디스크에 있음
- modified/dirty 비트 - 페이지가 메모리에 적재된 후 수정되었는지를 나타내는 비트
  - 1: 물리 메모리에 적재된 후 수정됨, 0: 수정된 적 없음
  - 1이면, 페이지가 물리 메모리에서 쫓겨날 때 스왑 영역에 저장(flush)되어야 한다.
- physical address 필드
  - 페이지가 메모리에 적재되어 있는 경우(presence bit=1) 이 필드에 **물리 프레임의 번호**가 기록
  - 페이지가 메모리에 없는 경우(presence bit=0) 디스크 주소가 기록, 이 주소는 스왑 영역의 주소이거나 실행 파일에 대한 주소이다.

## 페이지 폴트

페이지 폴트는 가상 메모리의 개념을 이해하는데 매우 중요한 요소이다.

> CPU가 가상 주소를 발생시켜 액세스하려는 페이지가 현재 물리 메모리에 없을 때 페이지 폴트(page fault)라고 한다.

페이지 폴트가 발생하면 폴트가 발생한 페이지를 메모리에 적재하는 과정이 진행된다.




위 그림에서 CPU가 `페이지 2`을 액세스하기 위해 가상 주소를 출력하면, MMU는 페이지 폴트를 발생시킨다. 왜냐하면 페이지 테이블의 `페이지 2` 항목에 presence bit가 0으로 되어 있기 때문이다. 즉 `페이지 2`가 메모리에 존재하지 않고 디스크의 스왑 영역에 존재한다.

페이지 폴트가 발생하면 페이지 폴트 핸들러(page fault handler) 코드가 실행되어, 물리 메모리에서 빈 프레임을 할당받고 요청된 페이지를 하드 디스크에서 읽어 적재하고 페이지 테이블을 수정한다. 만약 물리 메모리에 빈 프레임이 없는 경우 프레임 중 하나를 **희생 프레임(victim frame)**으로 선택(페이지 교체 알고리즘)한다. 희생 프레임에 저장된 페이지가 적재된 후 수정되었다면(modified bit=1), 이 페이지를 스왑 영역에 저장한다.

페이지를 스왑 영역으로부터 프레임에 적재하는 것을 스왑-인(swap-in) 또는 페이지-인(page-in)이라고 부르고, 프레임에 적재된 페이지를 스왑 영역에 저장하는 것을 스왑-아웃(swap-out) 또는 페이지 아웃(page-out)이라고 부른다. 한편, 페이지 폴트와 반대로 CPU가 발생한 가상 주소의 페이지가 메모리 프레임에 있을 때 페이지 히트(page hit)라고 한다.

## 3.3 페이지 폴트 자세히 알기

요구 페이징에서 가장 중요한 것이 페이지 폴트를 처리하는 과정이다. 페이지 폴트는 CPU가 메모리를 액세스할 때 가상 주소를 물리 주소로 바꾸는 과정에서 발생한다.  main() 함수와 전역 변수 n을 가진 프로그램이 적재된 프로세스가 그림 10-6과 같다고 하자.

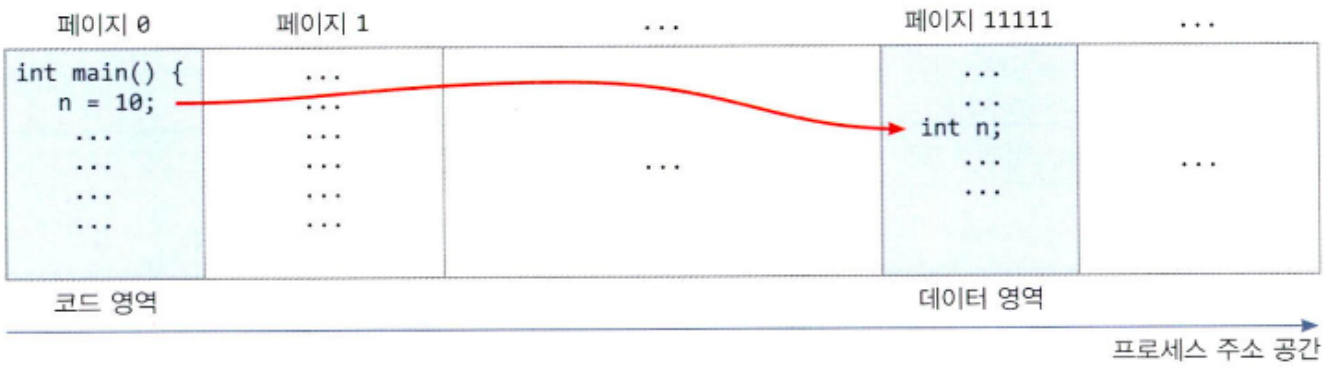



그림 10-6 샘플 프로세스의 구성

`n = 10;`을 포함하는 main() 함수의 코드는 `페이지 0`에 있다. 전역 변수들은 `0x11111번 페이지`에 있으며, 전역 변수 n의 가상 주소는 `0x11111234`번지이다.

```
; n = 10을 컴파일한 코드
mov eax, 10             ; eax 레지스터에 10 저장
mov [11111234], eax     ; eax 레지스터 값을 0x11111234 번지에 저장
```

페이지 폴트가 발생하고 처리되는 과정을 그림 10-7과 함께 설명한다.

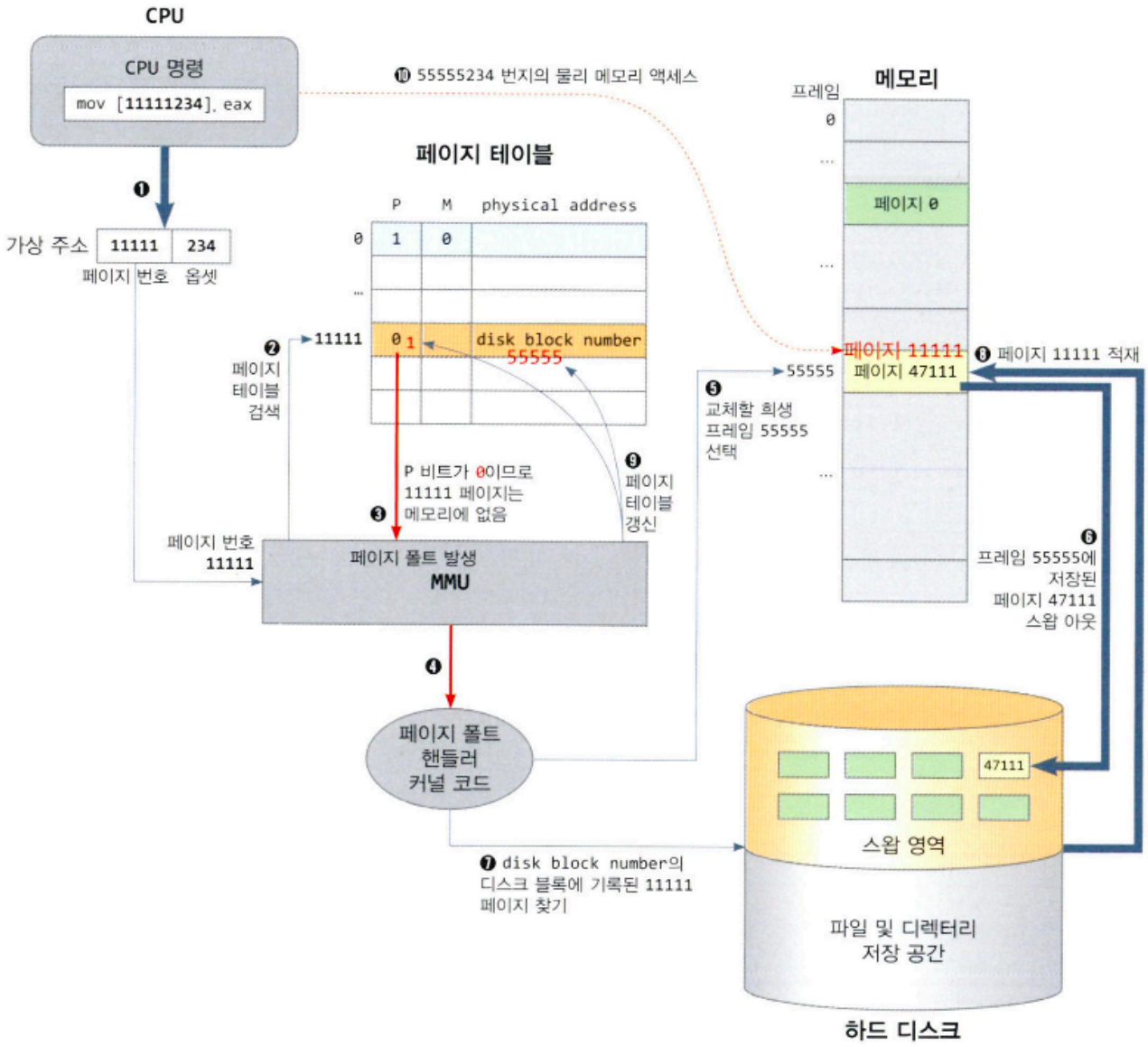


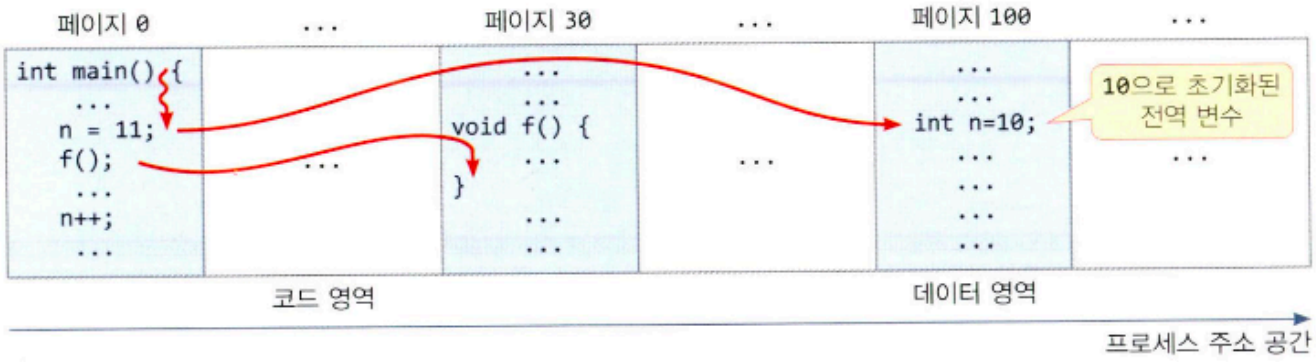

그림 10-7 페이지 폴트 처리 과정

1. CPU는 `mov [11111234], eax` 명령을 실행하기 위해 가상 주소 `11111234`를 출력한다. `11111234`는 페이지 번호가 `11111`이고 offset이 `234`인 가상 주소이다.
2. MMU는 페이지 테이블에서 `11111번 항목`을 검사한다.(TLB 생략...)
3. 페이지 테이블의 `11111번 항목`의 P(presence) bit가 0이므로 `11111번 페이지`는 메모리에 적재되어 있지 않다. 그러므로 MMU는 **페이지 폴트 예외(page fault exception)**를 발생시킨다. 만일 P 비트가 1이었다면 페이지 테이블 항목에 기록된 프레임 번호로 물리 주소가 완성되어 메모리 액세스가 진행되었을 것이다.
4. 페이지 폴트 예외가 발생하면 커널의 페이지 폴트 핸들러가 실행된다.
5. 페이지 폴트 핸들러는 `11111번 페이지`를 적재할 **희생 프레임**을 선택한다. 희생 프레임으로 `55555번 프레임`이 선택되었다고 가정
6. `55555번 프레임`에 적재된 `47111번 페이지`을 디스크에 스왑-아웃시킨다. 만일 `47111번 페이지`가 이미 스왑 영역에 저장되어 있고, 적재된 후 수정된 것이 없다면 스왑-아웃 없이 그냥 `55555번 프레임`을 비운다.
7. 페이지 폴트 핸들러는 스왑 영역이나 실행 파일로부터 `11111번 페이지`를 찾는다. 이 페이지가 전에 메모리에 적재된 적이 있으면 스왑 영역에 있을 것이다. 아니면 실행 파일 내에서 이 페이지의 위치를 찾는다.

8. 디스크로부터 `11111번 페이지`를 `55555번 프레임`에 적재한다.
9. 페이지 폴트 핸들러는 페이지 테이블의 `11111번 항목`에 할당된 프레임 번호를 기록하고 P 비트를 1로 설정한다.
10. 페이지 폴트 핸들러가 종료되고 CPU는 `mov [11111234], eax` 명령 처리를 재개한다. 가상 주소 `11111234` 번지가 물리 주소 `55555234`로 바뀌어 출력되고 명령 처리의 나머지 과정이 진행된다. 그러고 나면 `11111번 페이지 테이블 항목`의 M 비트는 1로 변경된다.

## 3.4 요구 페이징 시스템에서 프로세스 실행

요구 페이징을 시스템에서 운영체제가 프로세스를 생성하고 실행 파일로부터 코드와 데이터 등을 적재하고 실행시키는 과정에서 메모리가 어떻게 관리되는지 사례를 통해 알아보자.




* main() 함수가 실행되는 동안 페이지0, 페이지 100, 페이지 30의 순서로 메모리가 필요함

그림 10-8 프로세스의 실행 과정을 설명하기 위한 샘플 프로세스

1. **프로세스의 시작 페이지 적재**

운영체제는 그림 10-9(a)와 같이 메모리 프레임 1개를 할당받고 실행 파일로부터 프로세스의 실행이 시작될 첫 페이지(여기서는 페이지 0으로 가정)를 적재한 후 프로세스를 실행시킨다.  CPU는 페이지 0에 들어 있는 main() 함수에서 실행을 시작한다.

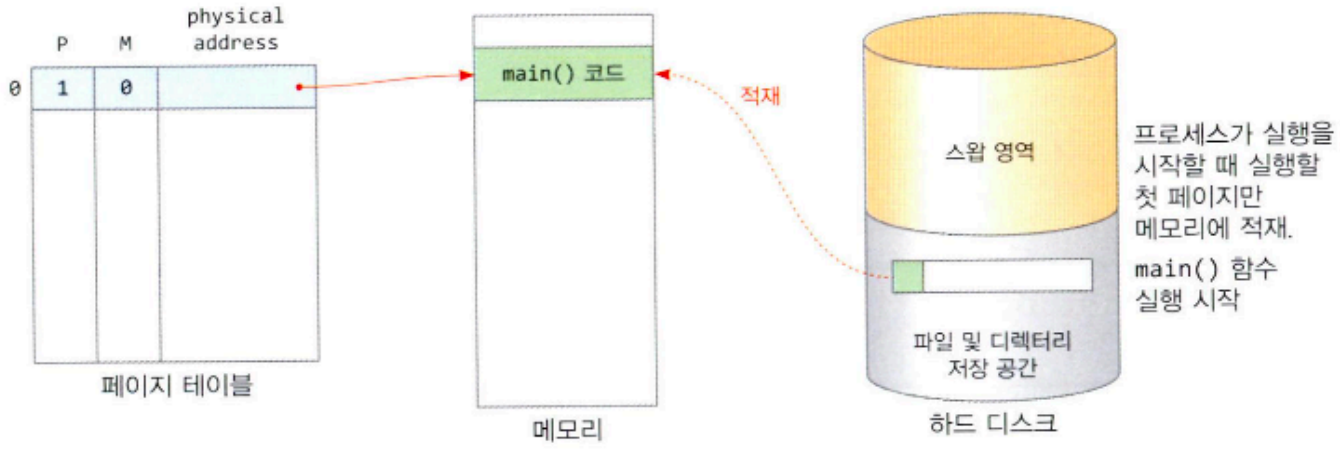



그림 10-9(a) 프로세스가 실행을 시작할 때 첫 페이지를 메모리에 적재

2. **여러 번의 페이지 폴트를 통해 실행 파일로부터 페이지들 적재**

프로세스의 실행 초기에는 전역 변수가 담긴 페이지나 스택 페이지가 메모리에 없기 때문에 페이지 폴트가 연이어 발생한다. 폴트가 발생한 페이지는 실행 파일이나 스왑 영역에 있다.

프로세스가 처음 액세스하는 코드나 데이터 페이지는 실행 파일에서 찾아 메모리에 적재한다. 한 번 적재된 페이지가 메모리에 없다면 분명 스왑 영역에 있을 것이므로 스왑 영역에서 적재한다. 코드가 들어있는 페이지는 스왑 영역으로 스왑-아웃시키지 않으므로 항상 실행 파일로부터 적재한다. 실행 초기에는 계속된 페이지 폴트를 통해 페이지들이 실행 파일로부터 메모리에 적재되지만, 시간이 지나 필요한 페이지들이 거의 적재되고 나면 스왑 영역으로부터 주로 적재되게 된다.

다시 사례로 돌아와서, main() 함수에서 `n=11;`을 실행할 때 전역 변수 n을 액세스하기 위해서 `100번 페이지`가 필요하다. 하지만 메모리에 없기 때문에 페이지 폴트가 발생하고 페이지 폴트 핸들러는 실행 파일로부터 `100번 페이지`를 읽어 들인다. 메모리에 빈 프레임이 있어서 적재하였다. 이때 실행 파일에 들어있는 변수 n의 초기 값 10이 적재되며 `M비트=0`으로 설정된다. 그리고 `n=11;`을 실행하면 `100번 페이지`이 수정되기 때문에 `M비트=1`로 수정된다.

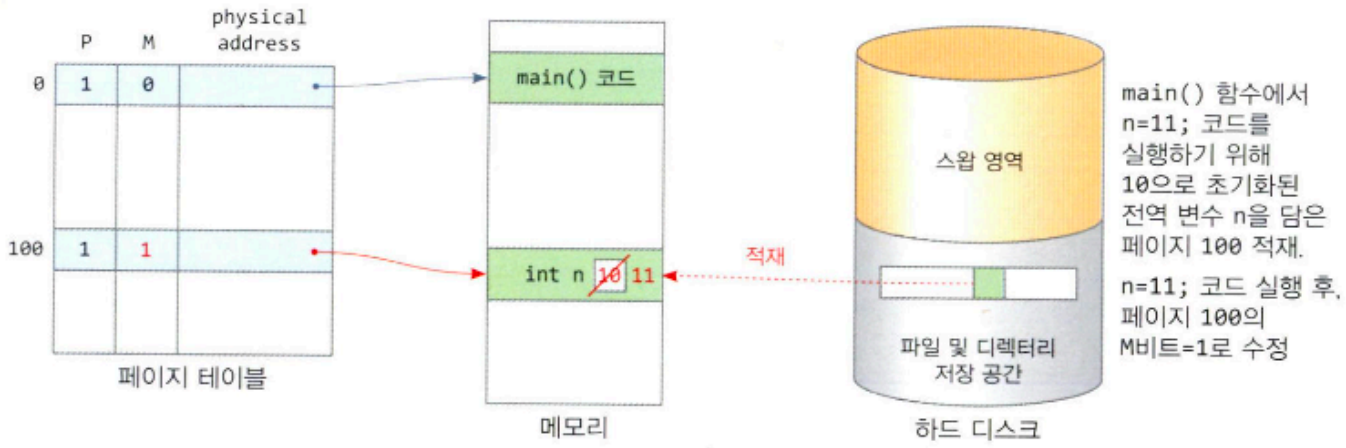



그림 10-9(b) 전역 변수 n이 들어 있는 페이지 100 적재 후 n=11 실행

그러고 나서 `f()`가 호출되는데 `f()`의 코드가 들어 있는 `30번 페이지`도 메모리에 존재하지 않아 페이지 폴트가 발생한다. 페이지 폴트 핸들러에 의해 빈 프레임을 할당받고 실행 파일에서 `30번 페이지`를 찾아 적재한다.

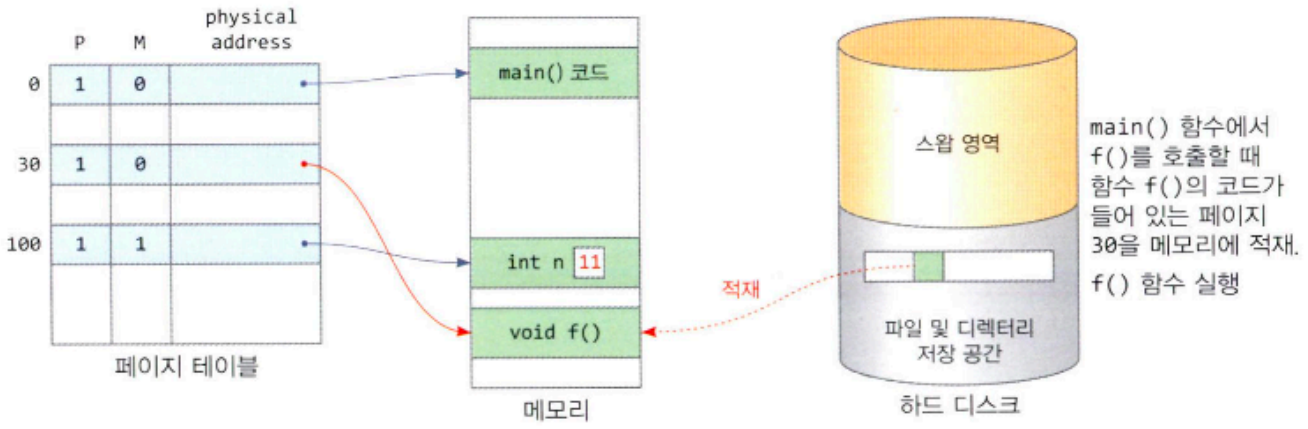



그림 10-9(c) 함수 f()의 코드가 들어 있는 페이지 30 적재

3. **메모리가 부족하면 스왑-아웃/스왑-인**

이 프로세스가 중단되고 다른 프로세스로 컨텍스트 스위칭되었다. 그리고 시간이 지나 메모리가 부족해지고 급기야 빈 프레임이 없는 상황이 발생하여 운영체제는 `100번 페이지`가 들어 있는 프레임을 희생 프레임으로 선택하였다. 희생 프레임에 들어 있는 `100번 페이지`는 디스크로 **스왑-아웃**되고 `P비트=0`이 되며, 희생 프레임에는 다른 프로세스가 요청한 페이지가 적재되었다. 전역 변수 n이 든 페이지는 디스크로 스왑-아웃되고 페이지 테이블의 `disk block number`에는 `100번 페이지`가 저장된 디스크 블록 번호가 저장된다.

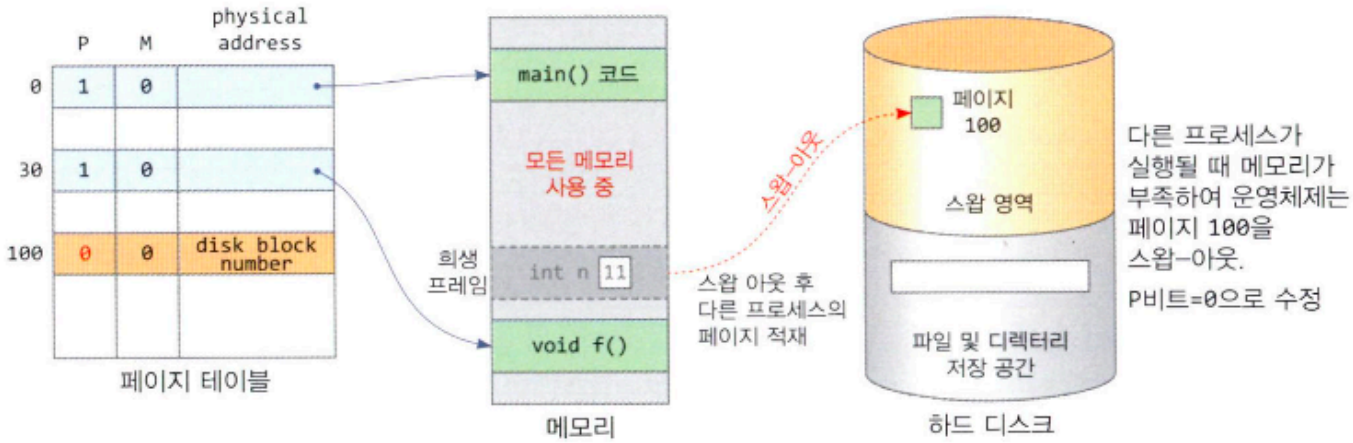



그림 10-9(d) 메모리가 부족하여 전역 변수 n이 들어 있는 페이지 100의 스왑-아웃

4. **스왑-아웃된 페이지를 다시 스왑-인**

중단된 프로세스가 실행을 재개하여 `main()`함수에서 `n++;` 코드를 실행하고자 한다. 하지만, 변수 n이 들어 있는 페이지가 메모리에 존재하지 않아 또 페이지 폴트가 발생한다. 페이지 폴트 핸들러에 의해 빈 프레임을 할당받고 스왑 영역에서 `100번 페이지`를 메모리에 적재하고 `P비트=1`, `M비트=0`으로 설정한다. `n++;` 코드를 실행하면 `M비트=1`로 수정된다.

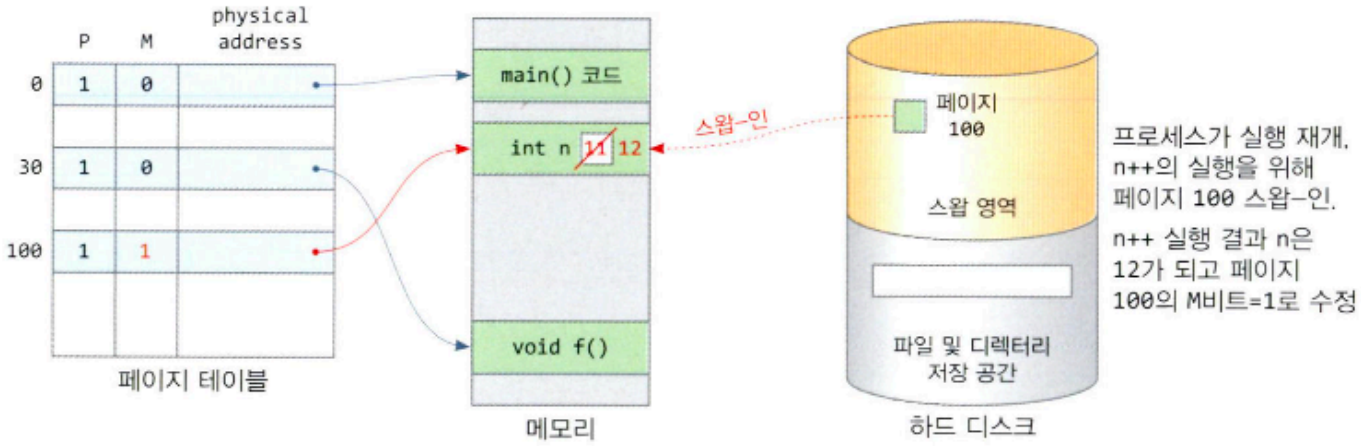



그림 10-9(e) 페이지 100의 스왑-인 후 n++ 실행

5. **수정된 페이지는 스왑 영역에 쓰기**

시간이 흐르고 운영체제가 빈 프레임을 만들기 위해 `f()` 함수의 코드가 들어있는 `30번 페이지`를 희생 페이지로 선택하였다면 다음과 같이 처리한다.

- `M비트=1`이라면, 페이지가 들어 있는 프레임을 스왑 영역에 다시 기록(flush)한다.
- `M비트=0`이라면, flush할 필요가 없다. 프레임에 새 페이지의 내용을 적재하여 덮어쓰면 된다.

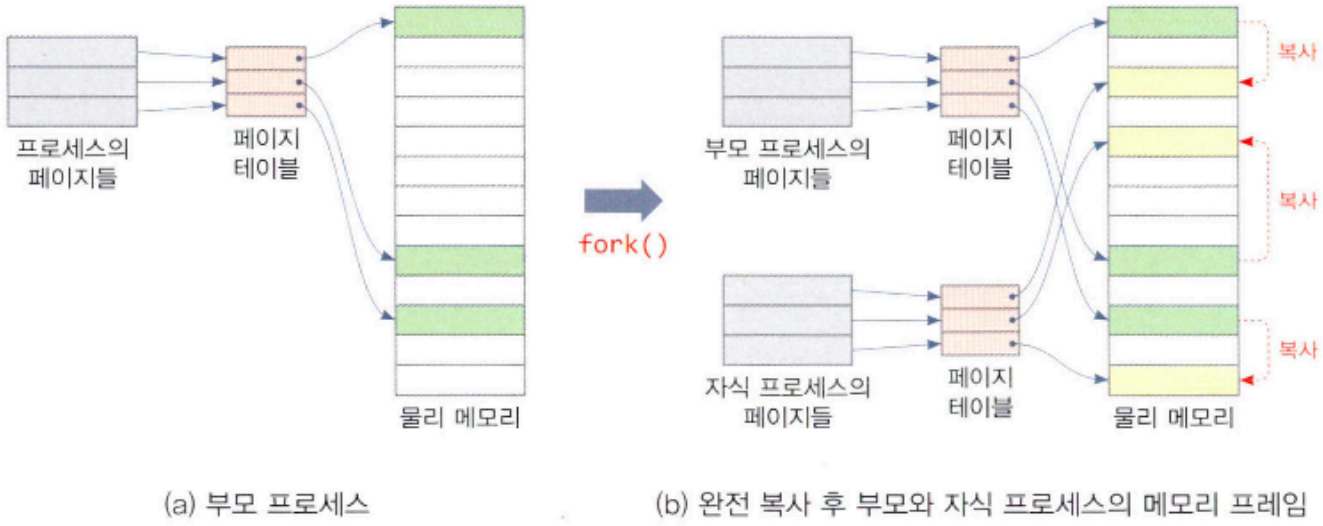
> **순수 요구 페이징**
> 아무 페이지도 적재하지 않은 채 프로세스를 실행시키고, 실행 첫 순간부터 페이지 폴트를 통해 첫 페이지를 적재시키는 방법을 **순수 요구 페이징(pure demand paging)**이라고 부른다. 오늘날 운영체제는 순수 요구 페이징을 사용하기도 한다.

> **가상 메모리에서 스왑 영역은 꼭 필요할까?**
> 스왑 영역이 필요한 이유는 첫째, 메모리가 부족하여 빈 메모리를 만들기 위해 디스크 어딘가에 저장해두었다가 나중에 다시 적재해야 하는데, 이를 위한 저장소가 바로 스왑 영역이다. 둘째, 스왑 영역을 읽는 것은 일반 파일 시스템에서 읽고 쓰는 속도보다 훨씬 빠르다.

## 3.5 쓰기 시 복사(COW, Copy On Write)

### 완전 복사

프로세스는 부모 프로세스에 의해 생성되며 시스템 호출을 통해서만 이루어진다. `fork()` 시스템 호출을 사용하여 현재 프로세스를 복사한 자식 프로세스를 생성할 수 있다. 쉽게 생각하면 부모 프로세스의 메모리를 **모두 복사**하여 자식 프로세스를 만드는 것이다. 이러한 완전 복사는 문제가 없을까?




(a) 부모 프로세스 (b) 완전 복사 후 부모와 자식 프로세스의 메모리 프레임

그림 10-10 부모 프로세스의 메모리를 완전 복사하여 자식 프로세스 생성하는 방법

## 완전 복사의 비효율성

결론부터 말하면, 부모 프로세스의 메모리를 모두 복사하여 자식 프로세스를 만드는 방법은 매우 비효율적이다. `fork()`로 자식 프로세스를 생성한 후 자식 프로세스가 `execlp()`을 이용하여 곧 바로 다른 프로그램을 실행시키도록 작성되기 때문이다. 쉘 명령어 `ls`을 실행하는 사례를 보자.

```
int childPid = fork();
if(childPid == 0) {
  execlp("/bin/ls", "ls", NULL); // /bin/ls 파일을 적재하여 실행
}
```

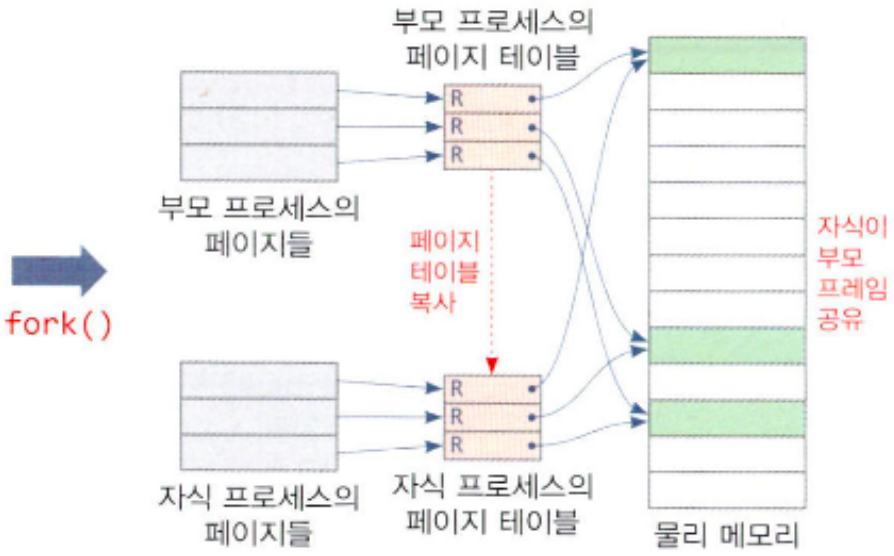
쉘은 자신을 복제한 자식 프로세스를 만들고, 자식 프로세스는 곧 바로 자신의 메모리에 `/bin/ls`을 적재하여 실행시킨다. 이처럼 자식 프로세스로 사용할 프로그램을 미리 실행 파일로 만들어 놓고, 쉘과 같은 방식으로 실행시킨다.

그러므로 부모 프로세스의 메모리를 완전 복사하여 자식 프로세스를 생성해도, 곧바로 자식 프로세스는 `execlp()`를 호출하여 할당받은 메모리를 모두 반환하고 실행 파일로부터 다시 페이지를 적재하기 때문에, 완전 복사 작업이 허사가 되어 버린다.

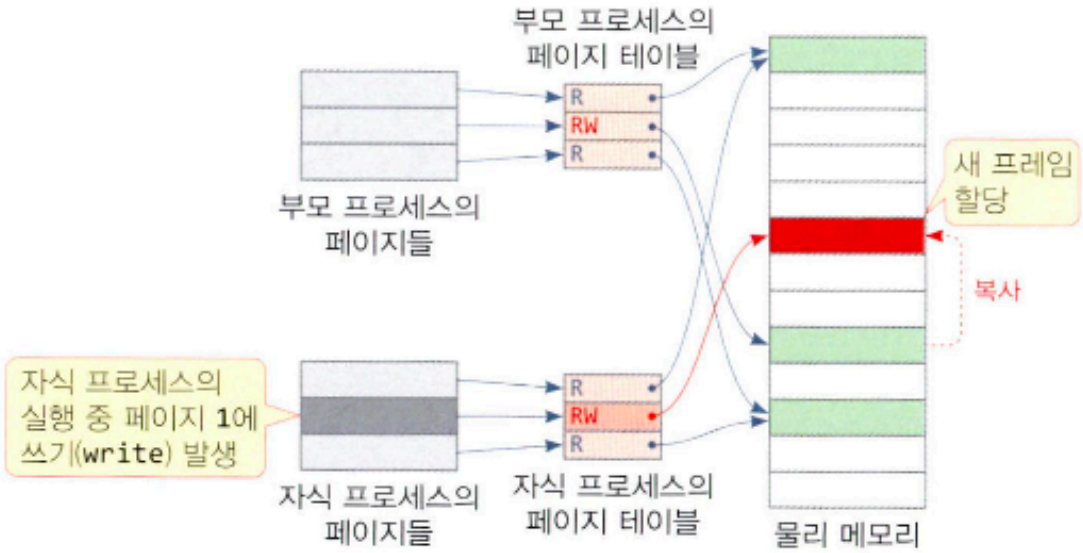
## 쓰기 시 복사(copy on write, COW)로 완전 복사 문제 해결

`copy on write`는 부모 프로세스의 메모리를 복사하지 않고 자식 프로세스가 부모 프로세스의 메모리를 완전히 **공유**하도록 하고 둘 다 실행되도록 내버려둔다. 그 후 부모든 자식이든 실행 중 메모리 쓰기가 발생하면, 그때 운영체제는 쓰기가 발생한 페이지만 새 프레임을 할당받아 복사한다. 읽는 경우에는 복사하지 않는다.

`copy on write` 기법을 위해서는 페이지 테이블 항목에 보호 필드(protection 비트)가 추가된다. 값이 `R`이면 읽는 것만 허용되는 페이지이며, `RW`는 읽기 쓰기 모두 가능한 페이지임을 나타낸다. 자식 프로세스가 막 생성될 당시 부모와 자식의 페이지 테이블 모두 protection 비트를 `R`로 표시한다.




(b) '쓰기 시 복사' 후 부모와 자식 프로세스




(c) 자식 프로세스가 페이지1에 쓰기를 실행할 때, 운영체제는
새 프레임을 할당하여 쓰기가 발생한 페이지 1을 복사

자식이나 부모 중 누가 페이지에 쓰기를 실행해도 마찬가지이다.

## 쓰기 시 복사의 장점

- **프로세스 생성 시간 절약**
- **메모리 절약**

# 3.6 페이지 폴트와 스래싱(thrashing)

## 페이지 폴트와 디스크 I/O

페이지 폴트가 발생하면 디스크의 실행 파일로부터 페이지를 메모리에 적재하거나, 스왑-인, 스왑-아웃시키기 때문에 필연적으로 디스크 입출력이 동반된다. 그렇기 때문에 페이지 폴트의 횟수도 줄일 필요가 있다.

## 스래싱

한 프로세스의 페이지를 적재하기 위해 다른 프로세스가 사용 중인 페이지를 스왑-아웃시키는 일이 도미노처럼 벌어지면, 계속 발생하는 페이지 폴트로 인해 시스템 내에 디스크 입출력이 증가하고 CPU는 계속 대기하여 활용률이 떨어지는 **스래싱(thrashing)**이 생긴다.

> 스래싱은 페이지 폴트가 계속 발생하여 메모리 프레임에 페이지가 반복적으로 교체되고 디스크 입출력이 심각하게 증가하여 CPU 활용률이 대폭 감소하는 현상이다.

## 스래싱의 원인

- 다중프로그래밍 정도
  - 메모리 량에 비해 실행중인 프로세스의 개수가 과도한 경우
  - 실행되는 프로세스가 많아질수록 각 프로세스에게 할당되는 프레임의 개수가 작아짐 → 페이지 폴트 가능성 증가
- 메모리 할당 정책이나 페이지 교체 알고리즘이 잘못된 경우
- 설치된 메모리가 절대적으로 작은 경우
- 특정 시간대에 너무 많은 프로세스를 실행한 경우

## 스래싱 현상 관찰

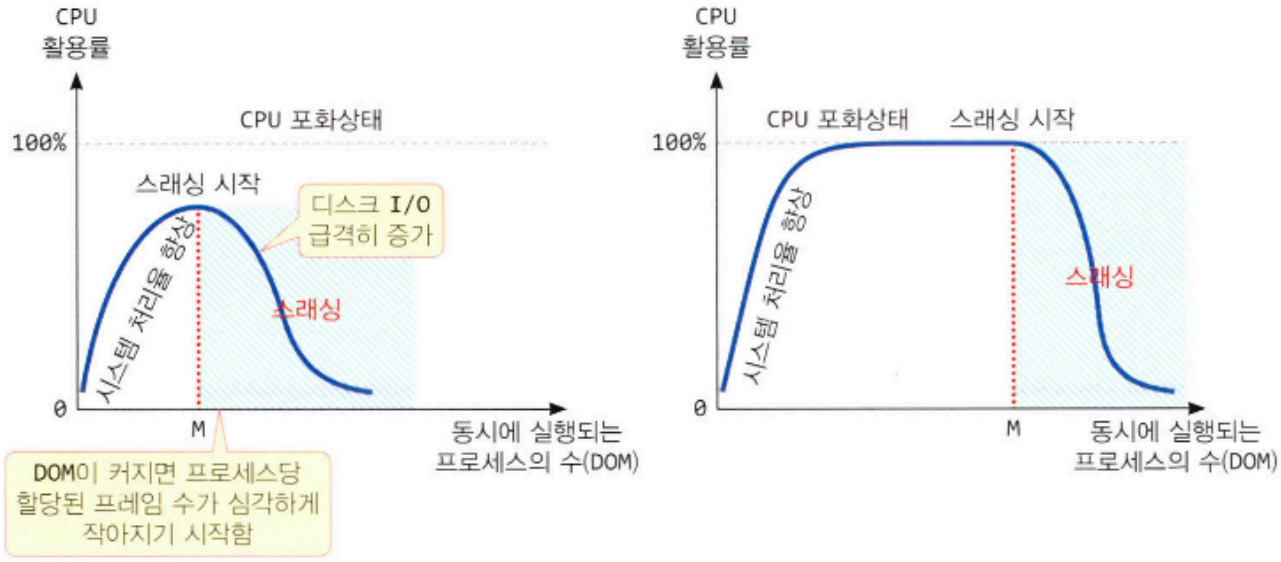



**그림 10-13** 다중프로그래밍 정도가 임계점 M을 넘어가면 스래싱 발생

(참고: Denning의 1980년 논문 Working Sets Past and Present)

그럼 10-13은 시스템에서 스래싱이 발생하는 시점을 보여준다. 동시에 실행되는 프로세스의 개수(DOM)가 늘어날수록, CPU idle time이 줄어들기 때문에 CPU 활용률이 증가되는 것은 자연스러운 현상이다. 그러다가 DOM이 임계점 M을 넘어갈 때부터 CPU 활용률이 떨어지는 현상을 볼 수 있다. 메모리에 적재된 프로세스의 개수가 늘어나면, 상대적으로 각 프로세스에게 할당되는 프레임의 개수가 줄어들어 페이지 폴트 가능성이 높아진다. 빈번한 스왑-아웃/스왑-인으로 인해 disk i/o가 증가하여 cpu 활용률이 급감하는 스래싱이 시작되는 것이다.

> 동시에 실행되는 프로세스의 개수(DOM)가 많음에도 불구하고 오히려 CPU 활용률이 갑자기 떨어질 때 스래싱이 발생하기 시작한 것으로 판단할 수 있다.

> **스래싱 검사 기법**
> 실행 중인 프로세스의 개수는 많지만 CPU의 활용률이 낮고, 메모리 활용률과 디스크 I/O 비율, 그리고 스왑-인/스왑-아웃의 비율이 높은지 검사하면 된다.

## 스래싱 해결 및 예방

- 다중프로그래밍 정도의 시스템 허용치를 낮추어 설정
- RAM 늘리기
- HDD 대신 SSD 사용