**챕터8 Deadlocks; 교착 상태**

**8.3 교착상태 특성**

**8.3.1 프로세스**

교착 상태는 한 시스템에서 다음 네 가지 조건이 동시에 성립될 때 발생할 수 있다.

1. 상호 배제(mutual exclusion): 최소한 하나의 자원이 비공유 모드로 점유되어야 한다. 비공유 모드에서는 한 번에 한 스레드만이 그 자원을 사용할 수 있다. 다른 스레드가 그 자원을 요청하면, 요청 스레드는 자원이 방출될 때까지 반드시 지연되어야 한다.

2. 점유하며 대기(hold-and-wait): 스레드는 최소한 하나의 자원을 점유한 채 현재 다른 스레드에 의해 점유된 자원을 추가로 얻기 위해 반드시 대기해야 한다.

3. 비선점(no preemption): 자원들을 선점할 수 없어야 한다. 즉, 자원이 강제적으로 방출될 수 없고, 점유하고 있는 스레드가 태스크를 종료한 후, 그 스레드에 의해 자발적으로만 방출될 수 있다.

4. 순환 대기(circular wait): 대기하고 있는 스레드의 집합 {T0, T1, …, Tn}에서 T0는 T1이 점유한 자원을 대기하고, T1은 T2가 점유한 자원을 대기하고, …, Tn-1은 쑤이 점유한 자원을 대기하며, Tn은 T0가 점유한 자원을 대기한다.

**8.3.2 자원 할당 그래프**

이 그래프는 정점(vertex) V의 집합과 간선(edge) E의 집합으로 구성된다.

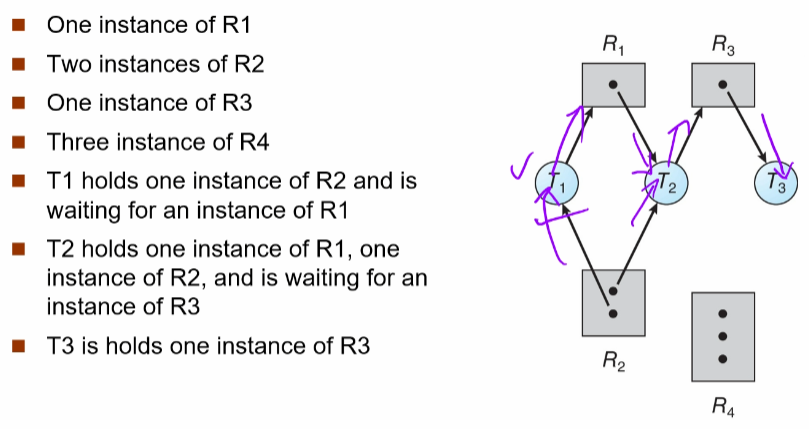
정점 V의 집합은 시스템 내의 모든 활성 스레드의 집합인 T = {T1, T2, …, Tn}과

시스템 내의 모든 자원 유형의 집합인 R = {R1, R2, …, Rm}의 두 가지 유형으로 구별된다.

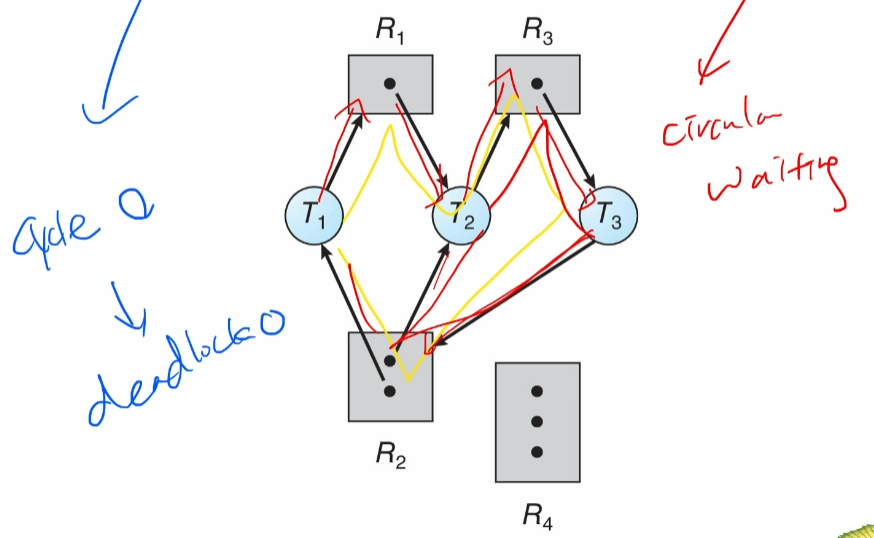
요청 간선(request edge): 스레드 Ti가 자원 유형 Rj의 인스턴스를 하나 요청하는 것으로 현재 이 자원을 기다리는 상태이다.

할당 간선(assignment edge): 자원 유형 Rj의 한 인스턴스가 스레드 Ti에 할당된 것을 의미한다.

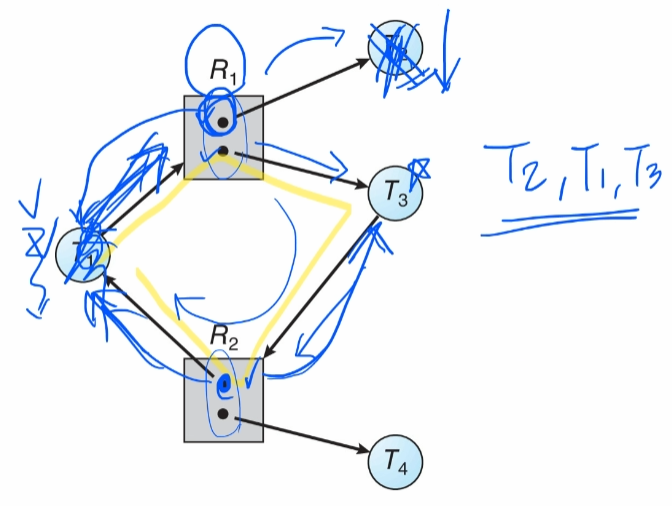
자원 할당 그래프



교착 상태를 갖는 자원 할당 그래프



사이클이 있으면서 교착 상태가 아닌 자원 할당 그래프



자원 할당 그래프에 사이클이 없다면, 시스템은 교착 상태가 아니다.

반면에, 사이클이 있다면 시스템은 교착 상태일 수도 있고 아닐 수도 있다.

**8.4 교착 상태 처리 방법**

원칙적으로 교착 상태 문제를 처리하는 데는 다음과 같은 세 가지 다른 방법이 있다.

1. 시스템이 결코 교착 상태가 되지 않도록 보장하기 위하여 교착 상태를 예방(Prevention)하거나 회피(Avoidance)하는 프로토콜을 사용한다.

2. 시스템이 교착 상태가 되도록 허용한 다음에 복구시키는 방법(Deadlock detection and recover)이 있다.

3. 문제를 무시(Ignorance)하고, 교착 상태가 시스템에서 절대 발생하지 않는 척한다.

**8.5 Deadlock Prevention; 교착 상태 예방**

교착 상태의 네 가지 조건 중 최소한 하나가 성립하지 않도록 보장함으로써, 우리는 교착 상태의 발생을 예방할 수 있다.

**8.5.1 Mutual Exclusion; 상호 배제**

상호 배제 조건이 성립되어야 한다. 즉 적어도 하나의 자원은 공유가 불가능한 자원이어야 한다.

반면에, 공유 가능한 자원들은 배타적인 접근을 요구하지 않으며, 따라서 교착 상태에 관련될 수 없다.

그러나 일반적으로 상호 배제 조건을 거부함으로써 교착 상태를 예방하는 것은 불가능하다.

어떤 자원들은 근본적으로 공유가 불가능하기 때문이다.

**8.5.2 Hold and Wait; 점유하며 대기**

시스템에서 점유하며 대기 조건이 발생하지 않도록 하려면 스레드가 자원을 요청할 때마다 다른 자원을 보유하지 않도록 보장해야 한다.

우리가 사용할 수 있는 하나의 프로토콜은 각 스레드가 실행을 시작하기 전에 모든 자원을 요청하고 할당해야 한다.

물론 이것은 자원 요청의 동적 특성으로 인해 대부분의 응용 프로그램에서는 실용적이지 않다.

한 대안 프로토콜은 스레드가 자원을 전혀 갖고 있지 않을 때만 자원을 요청할 수 있도록 허용한다. 스레드는 일부 자원을 요청하고 사용할 수도 있다. 스레드가 추가의 자원을 요청하려면, 자신에게 할당된 모든 자원을 반드시 먼저 방출해야 한다.

이 두 프로토콜은 두 가지 주요 단점이 있다.

첫째, 자원이 할당되었지만 장기간 사용되지 않을 수 있기 때문에 자원 이용률이 낮을 수 있다.

둘째, 기아가 발생할 수 있다. 인기 있는 여러 개의 자원이 필요한 스레드는 필요한 자원 중 적어도 하나는 항상 다른 스레드에 할당되므로 무한정 대기할 수 있다.

**8.5.3 No Preemption; 비선점**

만일 어떤 자원을 점유하고 있는 스레드가 즉시 할당할 수 없는 다른 자원을 요청하면(즉, 스레드가 반드시 대기해야 하면), 현재 점유하고 있는 모든 자원이 선점된다.

즉, 이들 자원들이 묵시적으로 방출된다.

선점된 자원들은 그 스레드가 기다리고 있는 자원들의 리스트에 추가된다.

스레드는 자신이 요청하고 있는 새로운 자원은 물론, 이미 점유하였던 옛 자원들을 다시 획득할 수 있을 때만 다시 시작될 것이다.

**8.5.4 Circular Wait; 순환 대기**

교착 상태 예방을 위해 지금까지 제시된 세 가지 옵션은 대부분 상황에서 일반적으로 실용적이지 않다.

그러나 교착 상태에 대한 네 번째이자 마지막 조건인 순환 대기 조건은 필요한 조건 중 하나를 무효화하여 실용적인 해결책을 제공할 수 있는 기회를 제공한다.

순환 대기 조건이 성립되지 않도록 하는 한 가지 방법은 모든 자원 유형에 전체적인 순서를 부여하여, 각 프로세스가 열거된 순서대로 오름차순으로 자원을 요청하도록 요구하는 것이다.

**8.6 Deadlock Avoidance; 교착 상태 회피**

교착 상태 회피 알고리즘은 시스템에 순환 대기 상황이 발생하지 않도록 자원 할당 상태를 검사한다.

**8.6.1 Safe State; 안전 상태**

시스템 상태가 안전(Safe)하다는 말은 시스템이 어떤 순서로든 스레드들이 요청하는 모든 자원을(최대 요구 수를 요구하더라도) 교착 상태를 야기시키지 않고 차례로 모두 할당해 줄 수 있다는 것을 뜻한다.

즉, 시스템이 안전 순서(safe sequence)를 찾을 수 있다면 시스템은 안전하다고 말한다.

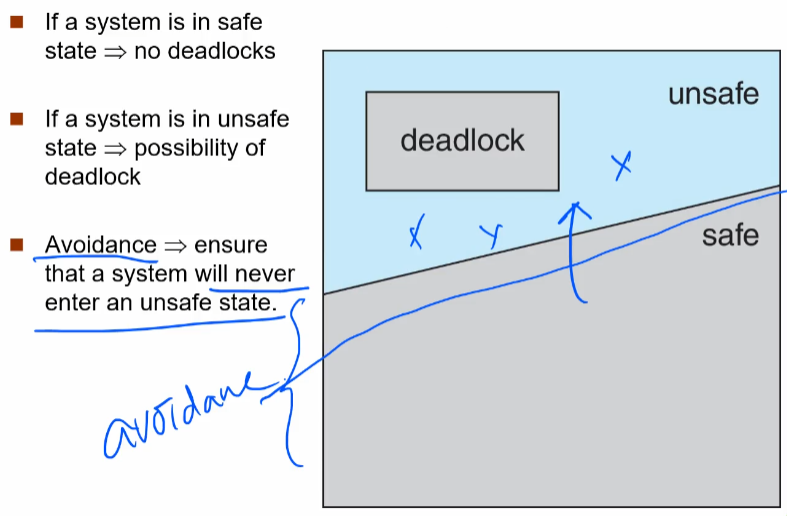
모든 스레드를 무사히 마칠 수 있는 시퀀스를 찾을 수 없으면 불안전(unsafe)하다고 한다.

시스템의 상태가 안전하다면 물론 교착 상태가 아니다. 반대로 교착 상태에 있는 시스템은 불안전한 상태에 있다.

그렇지만 시스템 상태가 불안전하다고 해서 반드시 교착 상태로 간다는 것을 뜻하는 것은 결코 아니다.

시스템이 불안전하다는 말은 앞으로 교착 상태로 가게 될 수도 있다는 뜻이다.

그러므로 시스템이 안전 상태에 머무는 한 운영체제는 불안전 상태나 교착 상태 모두를 예방할 수 있다.



**8.6.2 자원 할당 그래프 알고리즘**

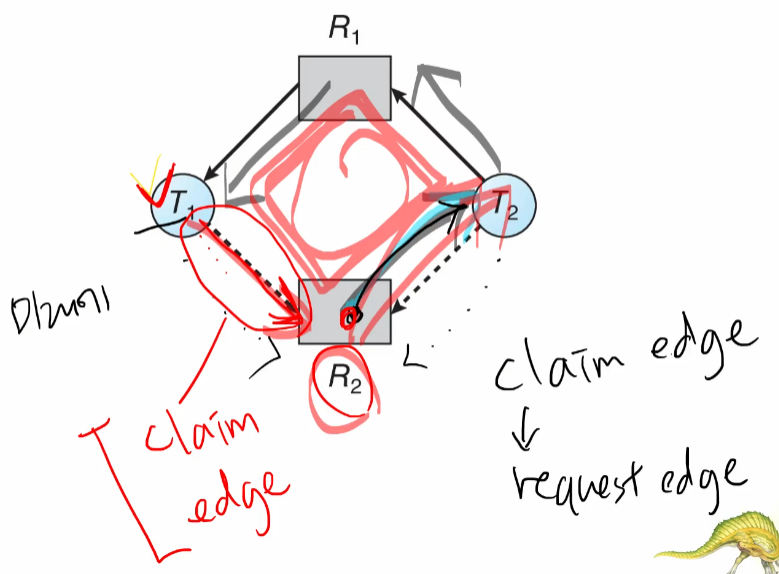
만약 각 자원 유형마다 단지 하나의 인스턴스를 갖는 자원 할당 시스템을 갖고 있다면, 우리는 교착 상태 회피를 위해 8.3.2절에서 정의한 자원 할당 그래프의 변형을 사용할 수 있다.

요청 간선과 할당 간선에 추가하여, 우리는 예약 간선(claim edge)이라는 새로운 타입의 간선을 도입한다.

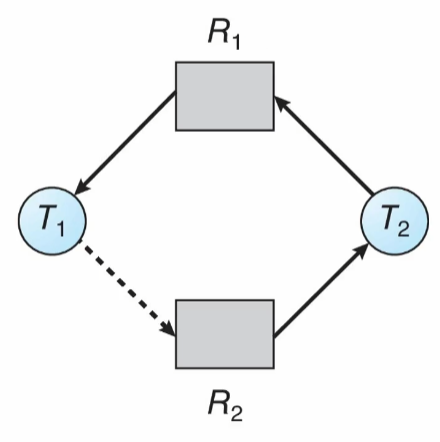
예약 간선 Ti -> Rj는 Ti가 미래에 자원 Rj를 요청할 것이라는 의미이다.

Ti가 자원 Rj를 요청하면, 예약 간선 Ti -> Rj는 요청 간선으로 변환된다.

교착 상태 회피를 위한 자원 할당 그래프

.

불안전 상태의 자원 할당 그래프



**8.6.3 Banker’s Algorithm; 은행원 알고리즘**

자원 할당 그래프 알고리즘은 종류마다 자원이 여러 개씩 있게 되면 사용할 수 없다. 아래에서 기술하는 알고리즘은 이와 같은 상황에서 사용할 수 있는 것이다.

하지만 앞의 자원 할당 그래프 알고리즘보다 아래 알고리즘은 효율성이 다소 떨어진다.

이 알고리즘은 통상 은행원(Banker’s) 알고리즘이라고 불린다.

이 시스템에서는 스레드가 시작할 때 스레드가 가지고 있어야 할 자원의 최대 개수를 자원 종류마다 미리 신고하여야 한다.

물론 이 숫자가 자원의 총 보유 수를 넘어서면 안 된다.

스레드가 자원들을 요청하면 시스템은 그것을 들어주었을 때 시스템이 계속 안전 상태에 머무르게 되는지 여부를 판단해야 한다.

계속 안전하게 된다면 그 요청을 들어준다. 그렇지 않다면 이 스레드의 요청은 허락이 안 된 채 다른 스레드가 끝날 때까지 기다리게 된다.

은행원 알고리즘을 구현하려면 몇 가지의 자료구조가 필요하다.

n이 스테드의 수이고 m이 자원의 종류 수라고 하자.

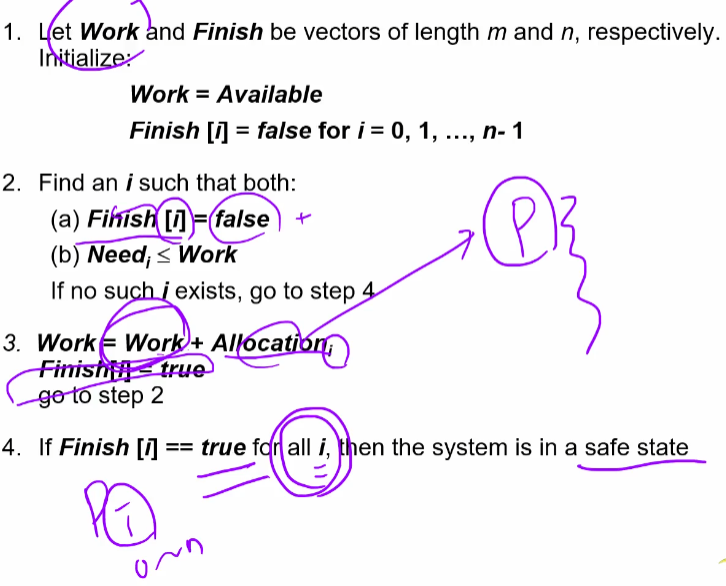
- **Available**: 각 종류별로 가용한 자원의 개수를 나타내는 벡터

- **Max**: 각 스레드가 최대로 필요로 하는 자원의 개수를 나타내는 행렬

- **Allocation**: 각 스레드에 현재 할당된 자원의 개수를 나타내는 행렬

- **Need**: 각 스레드가 향후 요청할 수 있는 자원의 개수를 나타내는 행렬

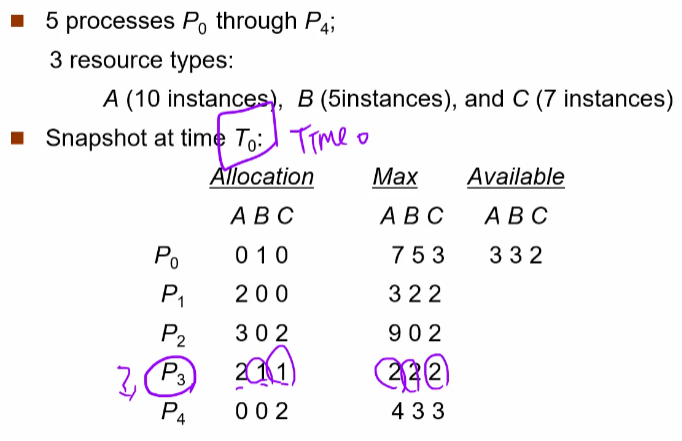
**8.6.3.1 Safety Algorithm; 안전성 알고리즘**



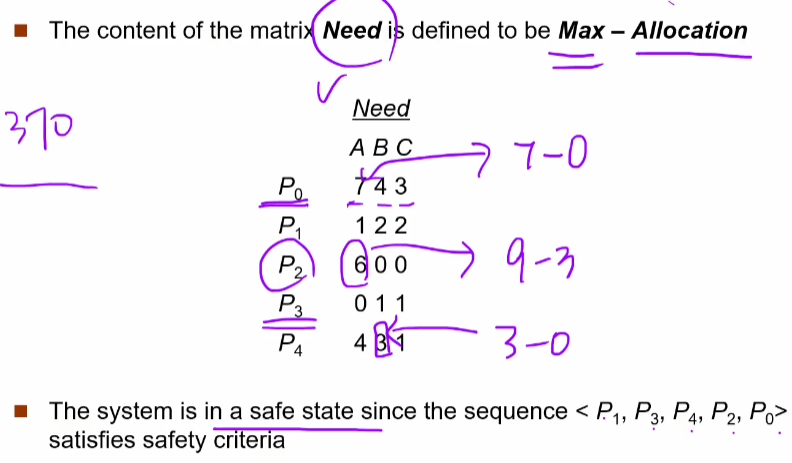
**8.6.3.3 예시**

다섯 개의 프로세스 T0부터 T4까지 있고, A, B, C 세 가지 종류의 자원이 있다고 가정하자.

시스템에는 A 자원이 10개, B 자원이 5개, C 자원이 7개가 있다.



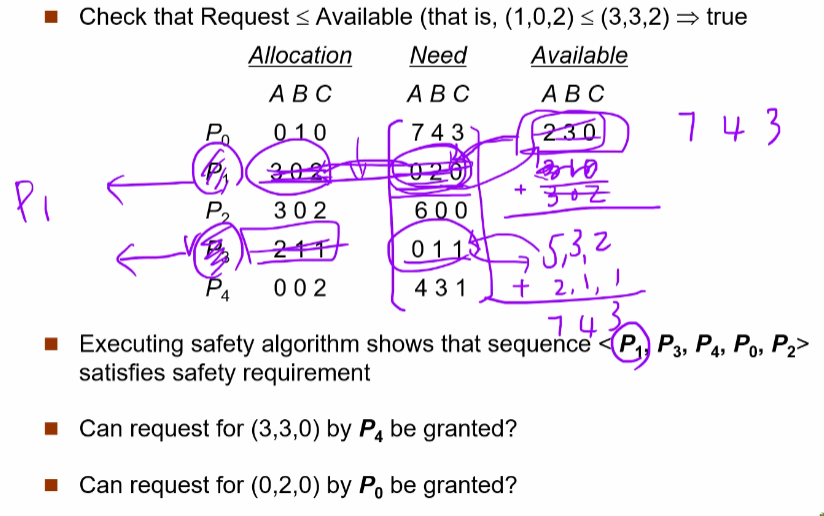
Need 행렬의 값은 (Max – Allocation)으로부터 얻어진다.



이 시스템은 안전하다.

<T1, T3, T4, T2, T0> 순서는 안전성 기준을 만족시키기 때문이다.

T1이 A 자원 한 개와 C 자원 두 개를 추가로 요청한다고 가정해 보자.



<T1, T3, T4, T0, T2>가 안전성 조건을 만족시킨다. 따라서 T1의 요청을 즉시 들어줄 수 있다.

이러한 상태에서 T4가 (3, 3, 0)을 요청하면 자원이 모자라므로 들어줄 수 없다는 것을 알 수 있다.

또 T0가 (0, 2, 0)을 요청한다면 자원은 충분히 있지만 상태를 불안전 상태로 만드므로 역시 그 요청을 즉시 들어줄 수 없다는 것을 알 수 있다.

**8.7 Deadlock Detection; 교착 상태 탐지**

만일 시스템이 교착 상태 예방이나 교착 상태 방지 알고리즘을 사용하지 않는다면, 교착 상태가 발생할 수 있다.

이러한 환경에서는 다음 알고리즘들을 반드시 지원해야 한다.

- 교착 상태가 발생했는지 검사하기 위해 시스템의 상태를 검사하는 알고리즘

- 교착 상태로부터 회복하는 알고리즘

**8.7.1 각 자원 유형이 한 개씩 있는 경우**

모든 자원이 한 개의 인스턴스만 있다면, 대기 그래프(wait-for graph)라고 하는, 자원 할당 그래프의 변형을 사용해 교착 상태 탐지 알고리즘을 정의할 수 있다.

우리는 자원 할당 그래프로부터 자원 유형의 노드를 제거하고 적절한 간선들을 결합함으로써 대기 그래프를 얻을 수 있다.

앞에서와 마찬가지로, 대기 그래프가 사이클을 포함하는 경우에만 시스템에 교착 상태가 존재한다.

교착 상태를 탐지하기 위해 시스템은 대기 그래프를 유지할 필요가 있고, 주기적으로 그래프에서 사이클을 탐지하는 알고리즘을 호출한다.

(a) 자원 할당 그래프 (b) 대응되는 대기 그래프



**8.7.2 각 유형의 자원을 여러 개 가진 경우**

대기 그래프는 종류마다 자원이 여러 개씩 존재하는 상황에서는 사용할 수 없다.

아래의 기법은 이러한 상황에서 교착 상태를 탐지할 수 있다.

이 알고리즘은 은행원 알고리즘과 마찬가지로 시시각각 그 내용이 달라지는 자료구조를 사용한다.

- **Available**: 각 종류의 자원이 현재 몇 개가 가용한지를 나타내는 벡터

- **Allocation**: 각 스레드에 현재 할당된 자원의 개수를 나타내는 행렬

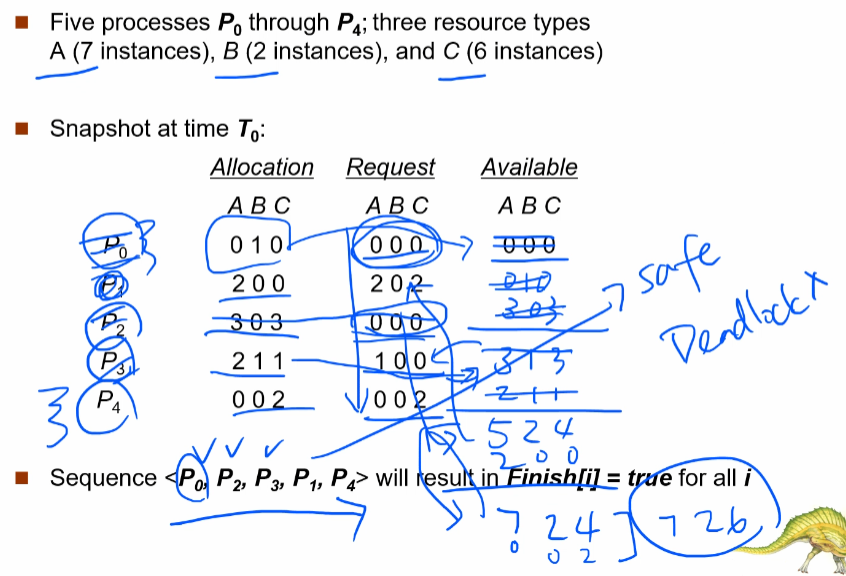
- **Request**: 각 스레드가 현재 요청 중인 자원의 개수를 나타내는 행렬

탐지 알고리즘의 예시

다섯 개의 스레드 T0부터 T4까지 있고

A, B, C 세 가지 종류의 자원이 있다고 가정한다

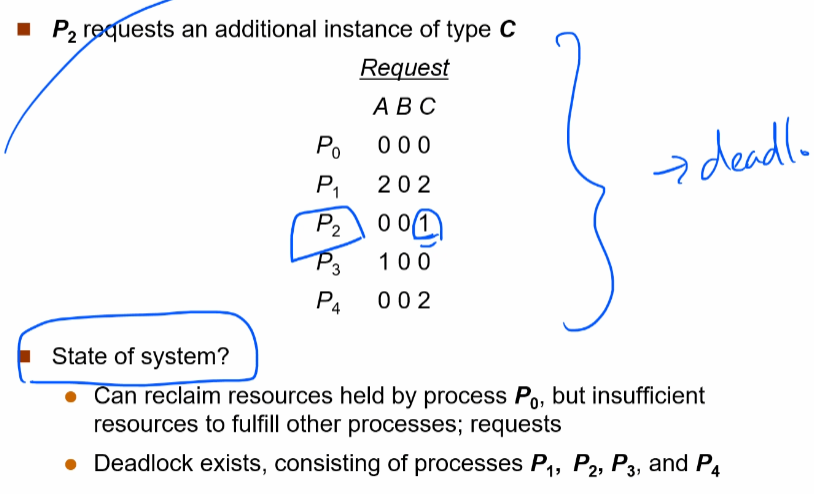
A 자원이 7개, B 자원이 2개, C 자원이 6개가 있다



이 시스템은 교착 상태에 처해있지 않음을 알 수가 있다.

알고리즘을 돌려 보면 <T0, T2, T3, T1, T4> 순서와 같이 작업을 다 끝낼 수 있고 이때 모든 i에 대해서 Finish[i] == true가 됨을 확인할 수가 있다.

그렇지만 T2가 C 자원을 한 개 더 요청하면 Request 행렬은 아래처럼 된다



이제는 시스템이 교착 상태에 빠지게 된다.

T0의 자원을 회수한다고 하더라도 다른 프로세스들이 요구하는 자원을 충족시켜줄 방법이 없기 때문이다.

**8.8 Recover from Deadlock; 교착 상태로부터 회복**

탐지 알고리즘이 교착 상태가 존재한다고 결정하면, 여러 대안의 처리 방법이 있다.

한 가지 방법은 교착 상태가 발생한 것을 운영자(operator)에게 통지해, 운영자가 수작업으로 처리하게 하는 것이다.

다른 방법은 시스템이 자동으로 교착 상태로부터 회복하게 하는 것이다.

교착 상태를 깨뜨리는 데는 두 가지 방법이 있다.

한 가지 방법은 순환 대기를 깨뜨리기 위해 단순히 한 개 이상의 스레드를 중지(abort)시키는 것이다.

두 번째 방법은 교착 상태에 있는 하나 이상의 스레드들로부터 자원을 선점(preempt)하는 것이다.

**8.8.1 Process and Thread Termination; 프로세스와 스레드의 종료**

- **교착 상태 프로세스를 모두 중지**: 이 방식은 확실하게 교착 상태의 사이클을 깨뜨리지만, 그 비용이 크다. 왜냐하면, 이들 프로세스가 오랫동안 연산했을 가능성이 있으며, 이들 부분 계산의 결과들을 반드시 폐기해야 하며, 아마도 후에 다시 계산해야 하기 때문이다.

- **교착 상태가 제거될 때까지 한 프로세스씩 중지**: 이 방법은 각 프로세스가 중지될 때마다 교착 상태 탐지 알고리즘을 호출해 프로세스들이 아직도 교착 상태에 있는지 확인해야 하므로 상당한 오버헤드를 유발한다.

어느 프로세스를 선택할지 결정하는 데는 다음과 같은 많은 결정 요인이 있다.

1. 프로세스의 우선수위가 무엇인지

2. 지금까지 프로세스가 수행된 시간과 지정된 일을 종료하는 데 더 필요한 시간

3. 프로세스가 사용한 자원 유형과 수(예를 들어, 자원들을 선점하기가 단순한지 여부)

4. 프로세스가 종료하기 위해 더 필요한 자원의 수

5. 얼마나 많은 수의 프로세스가 종료되어야 하는지

**챕터9 Main Memory; 메인 메모리**

**9.1 Background; 배경**

메인 메모리와 각 처리 코어에 내장된 레지스터들은 CPU가 직접 접근할 수 있는 유일한 범용 저장장치이다

각 CPU 코어에 내장된 레지스터들은 일반적으로 CPU클록의 1사이클 내에 접근이 가능하다

메인 메모리의 접근을 완료하기 위해서는 많은 CPU 클록 틱 사이클이 소요되며, 이 경우 CPU가 필요한 데이터가 없어서 명령어를 수행하지 못하고 지연되는(stall) 현상이 발생하게 된다

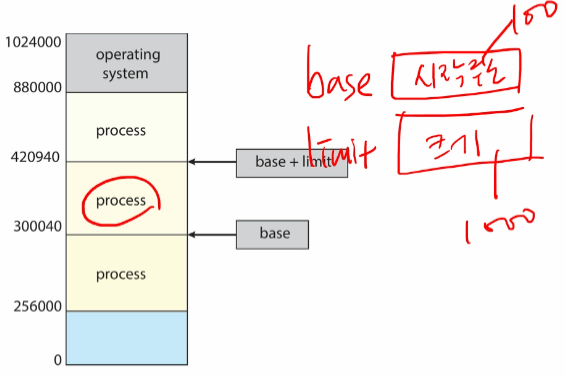
캐시는 CPU와 메인메모리 사이에 위치한다

물리 메모리의 상대적인 접근 속도의 차이를 고려하는 것에 추가로 올바른 동작을 보장해야만 한다

**9.1.1 기본 하드웨어**

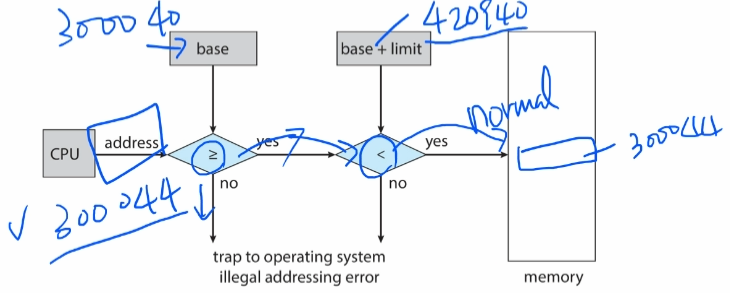
각각의 프로세스가 독립된 메모리 공간을 가지도록 보장해야 한다

기준(base)과 상한(limit)이라고 불리는 두 개의 레지스터들을 사용하여 이러한 보호 기법을 제공한다

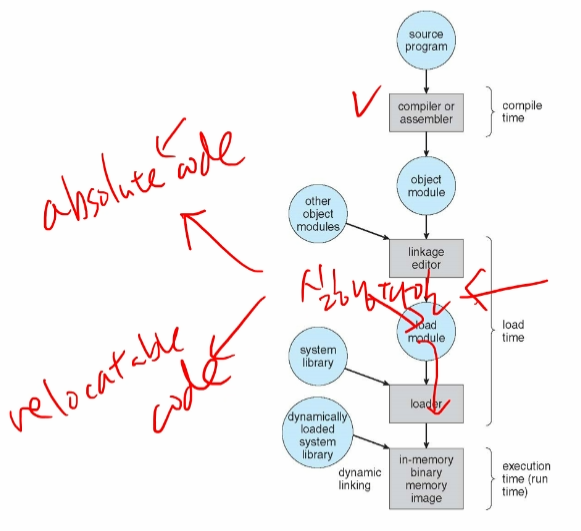


메모리 공간의 보호는 CPU 하드웨어가 사용자 모드에서 만들어진 모든 주소와 레지스터를 비교함으로써 이루어진다

기준과 상한 레지스터는 여러 가지 특권 명령을 사용하는 운영체제에 의해서만 적재(load)된다



**9.1.2 주소의 할당**



컴파일 시간 바인딩: 만일 프로세스가 메모리 내에 들어갈 위치를 컴파일 시간에 미리 알 수 있으면 컴파일러는 절대 코드를 생성할 수 있다. 그러나 만일 이 위치가 변경되어야 한다면 이 코드는 다시 컴파일되어야 한다.

적재 시간 바인딩: 만일 프로세스가 메모리 내 어디로 올라오게 될지를 컴파일 시점에 알지 못하면 컴파일러는 일단 이진 코드를 재배치 가능 코드로 만들어야 한다

실행 시간 바인딩: 만약 프로세스가 실행하는 중간에 메모리 내의 한 세그먼트로부터 다른 세그먼트로 옮겨질 수 있다면 우리는 바인딩이 실행 시간까지 허용되었다고 이야기한다

**9.1.3 논리 대 물리 주소 공간**

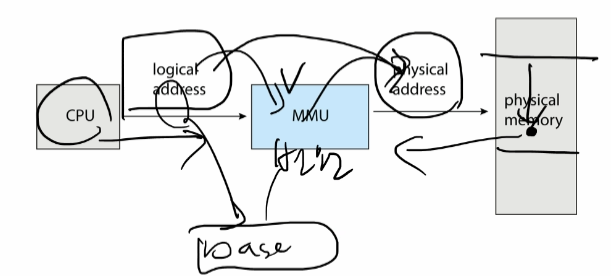
CPU가 생성하는 주소를 일반적으로 논리 주소(logical address)라 하며

반면에 메모리가 취급하게 되는 주소[즉, 메모리 주소 레지스터(MAR)에 주어지는 주소)는 일반적으로 물리 주소(physical address)라 한다

프로그램에 의해 생성된 모든 논리 주소 집합을 논리 주소 공간(logical address space)이라 하며

이 논리 주소와 일치하는 모든 물리 주소 집합을 물리 주소 공간(physical address space)이라 한다

프로그램의 실행 중에는 이와 같은 가상 주소를 물리 주소로 바꾸어줘야 하는데 이 변환(mapping) 작업은 하드웨어 장치인 메모리 관리 장치(memory management unit, MMU)에 의해 실행된다

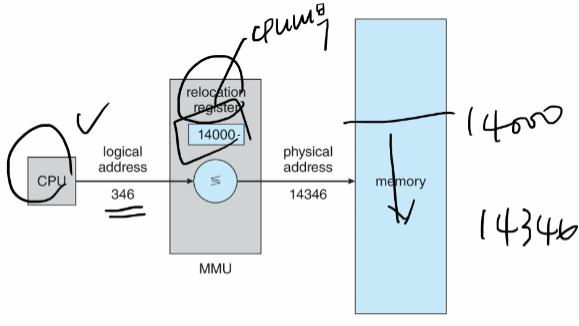


기준 레지스터를 재배치(relocation) 레지스터라고 부른다

재배치 레지스터 속에 들어있는 값은 주소가 메모리로 보내질 때마다 그 모든 주소에 더해진다

예를 들어, 재배치 레지스터 값이 14000이라면 프로세스가 346번지에 액세스할 때, 실은 메인 메모리의 14346번지에 액세스하게 된다

사용자 프로그램은 논리 주소를 사용한 것이고, 메모리 하드웨어는 논리 주소를 실제 주소로 바꾼 것이다



**9.2 연속 메모리 할당**

메인 메모리는 운영체제뿐만 아니라 여러 사용자 프로세스도 수용해야 한다

메모리는 일반적으로 두 개의 부분으로 나뉘어지는데, 하나는 운영체제를 위한 것이며 다른 하나는 사용자 프로세스를 위한 것이다

**9.2.2 메모리 할당**

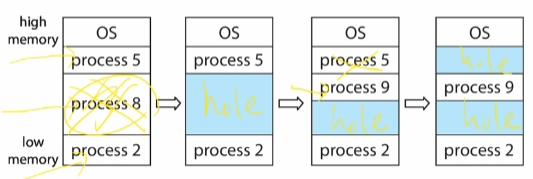
메모리를 할당하는 가장 간단한 방법 중 하나는 프로세스를 메모리의 가변 크기 파티션에 할당하는 것이다

각 파티션에는 정확히 하나의 프로세스만 적재될 수 있다

이 가변 파티션 기법에서 운영체제는 사용 가능한 메모리 부분과 사용 중인 부분을 나타내는 테이블을 유지한다

처음에는 모든 메모리가 사용자 프로세스에 사용 가능하며, 하나의 큰 사용 가능한 메모리 블록인 hole로 간주한다

프로세스가 시스템에 들어오면, 운영체제는 각 프로세스가 메모리를 얼마나 요구하며, 또 사용 가능한 메모리 공간이 어디에 있는지를 고려하여 공간을 할당한다



동적 메모리 할당 문제

최초 적합: 첫 번째 사용 가능한 가용 공간을 할당한다. 충분히 큰 가용 공간을 찾았을 때 검색을 끝낼 수 있다

최적 적합: 사용 가능한 공간 중에서 가장 작은 것을 택한다. 이 방법은 아주 작은 가용 공간을 만들어 낸다

최악 적합: 가장 큰 가용 공간을 택한다. 이 방식에서 할당해 주고 남게 되는 가용 공간은 충분히 커서 다른 프로세스들을 위하여 유용하게 사용될 수 있다

**9.2.3 단편화**

최초 적합 젼략과 최적 적합 전략 모두 외부 단편화(external ftagmentation)로 인해 어려움을 겪는다

프로세스들이 메모리에 적재되고 제거되는 일이 반복되다 보면, 어떤 가용 공간은 너무 작은 조각이 되어 버린다

외부 단편화는 이처럼 유휴 공간들을 모두 합치면 충분한 공간이 되지만 그것들이 너무 작은 조각들로 여러 곳에 분산되어 있을 때 발생한다

최초 적합의 경우 통계적인 분석을 해보면 N개의 블록이 할당되었을 때 0.5N개의 블록이 단편화 때문에 손상될 수 있다는 것을 알 수 있다

즉, 메모리의 3분의 1이 쓸 수 없게 될 수 있다는 것이다

이 현상은 50% 규칙(50-percent rule)으로 알려져 있다

할당된 공간은 요구된 공간보다 약간 더 클 수 있다

이들 두 크기 사이의 남는 부분이 바로 내부 단편화(internal fragmentation)이고, 이 내부 단편화 역시 사용이 못 되는 부분이다

외부 단편화 문제를 해결하는 다른 방법으로는 압축(compaction)이 있다

**9.3 페이징**

페이징은 연속 메모리 할당을 괴롭히는 두 가지 문제인 외부 단편화와 관련 압축의 필요성을 피한다

**9.3.1 기본 방법**

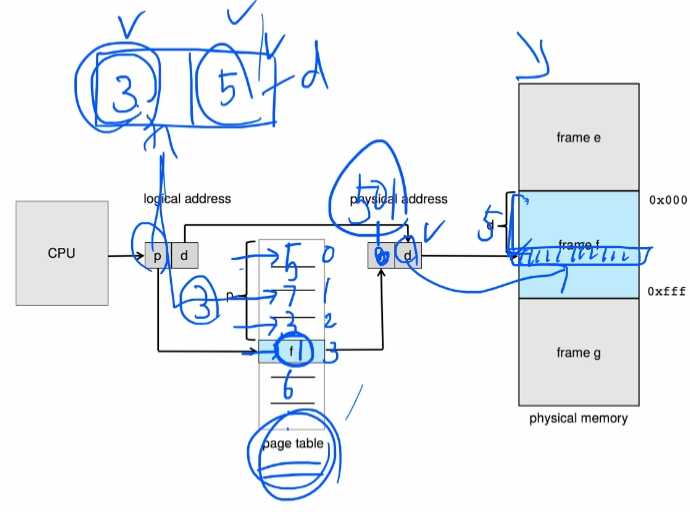
물리 메모리는 프레임(frame)이라 불리는 같은 크기 블록으로 나누어진다

논리 메모리는 페이지(page)라 불리는 같은 크기의 블록으로 나누어진다

CPU에서 나오는 모든 주소는 페이지 번호(p)와 페이지 오프셋(d: offset) 두 개의 부분으로 나누어진다

페이지 번호는 프로세스 페이지 테이블(page table)을 액세스할 때 사용된다

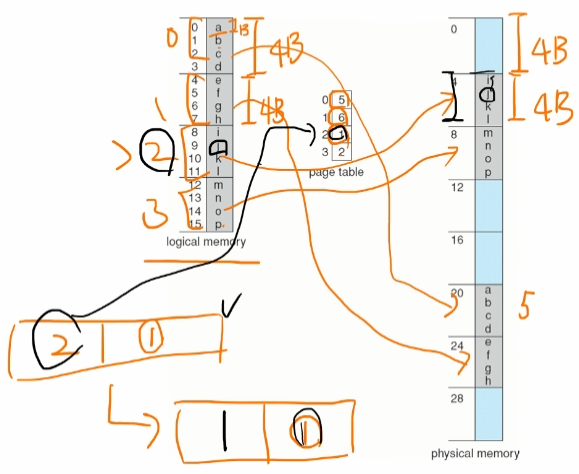
오프셋 d는 변하지 않기 때문에 대체되지 않으며, 프레임 번호와 오프셋은 이제 물리 주소를 구성한다



페이지 번호 p를 추출하여 페이지 테이블의 인덱스로 사용한다

페이지 테이블에서 해당 프레임 번호 f를 추출한다

논리 주소의 페이지 번호 p를 프레임 번호 f로 바꾼다



페이징 기법을 사용하면 외부 단편화가 발생하지 않는다. 모든 놀고 있는 프레임이 프로세스에 할당될 수 있기 때문이다

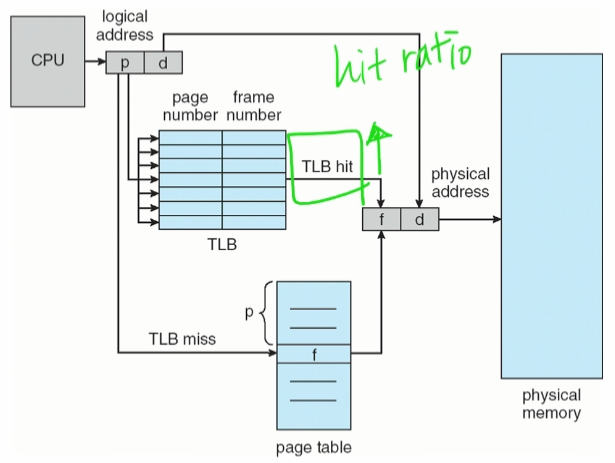
그러나 이제는 내부 단편화가 발생한다

예를 들어, 페이지 크기가 2,048 B이고 프로세스가 72,766 B를 요구한다면 35개의 페이지 프레임을 할당하고 1,086 B가 남는다

36번째로 할당되는 페이지 프레임은 2,048 – 1,086 = 962 B의 내부 단편화가 발생하게 된다

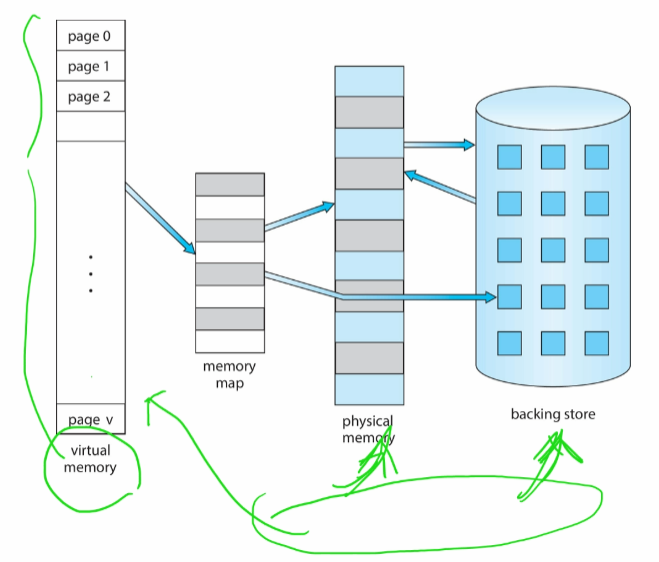
최악의 경우 프로세스가 n개의 페이지와 추가로 1B를 요구할 수 있다

그 결과 n+1번째 프레임은 거의 모든 프레임이 내부 단편화가 된다



**챕터10 가상 메모리**

**10.1 배경**



만일 프로그램을 일부분만 메모리에 올려놓고 실행할 수 있다면 다음과 같은 많은 이점이 있다

프로그램은 물리 메모리 크기에 의해 더는 제약받지 않게 된다

각 프로그램이 더 작은 메모리를 차지하므로 더 많은 프로그램을 동시에 수행할 수 있게 된다

-> 이에 따라 응답시간은 늘어나지 않으면서도 CPU 이용률과 처리율이 높아진다

프로그램을 메모리에 올리고 스왑하는 데 필요한 I/O 횟수가 줄어들기 때문에 프로그램들이 보다 빨리 실행된다

가상 메모리는 실제의 물리 메모리 개념과 개발자의 논리 메모리 개념을 분리한 것이다

이렇게 함으로써 얻어지는 장점은 작은 메모리를 가지고도 얼마든지 큰 가상 주소 공간을 프로그래머에게 제공할 수 있다는 점이다

**10.2 요구 페이징**

프로그램 실행 시작 시 프로그램의 전부를 물리 메모리에 적재할 수도 있다

다른 전략은 필요한 페이지만 적제하는 것이다

이 기법을 요구 페이징(demand paging)이라고 하며 가상 메모리 시스템에서 일반적으로 사용된다

적은 I/O 필요성, 불필요한 I/O

적은 메모리 요구

빠른 응답

더 많은 유저

**10.2.1 기본 개념**

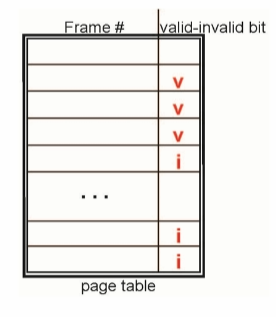
위에서 언급한 것처럼 요구 페이징의 기본 개념은 필요할 때만 페이지를 메모리에 적재하는 것이다

결과적으로 프로세스가 실행되는 동안 일부 페이지는 메모리에 있고 일부는 보조저장장치에 있다

유효-무효 기법

이 비트가 유효(valid)하다고 설정되면 해당 페이지가 메모리에 있다는 의미이고

무효(unvalid)하다고 설정되면 해당 페이지가 유효하지 않거나(즉, 가상 주소 공간상에 정의되지 않았거나) 유효하지만 보조저장장치에 존재한다는 것을 의미한다



페이지 폴트를 처리하는 과정

1. 프로세스에 대한 내부 테이블을 검사해서 그 메모리 참조(reference)가 유효인지 무효인지를 알아낸다

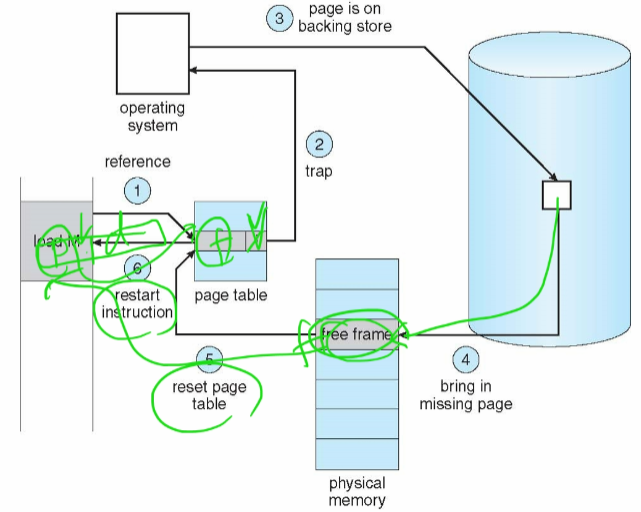
2. 만약 무효한 페이지에 대한 참조라면 그 프로세스는 중단된다. 만약 유효한 참조인데 프로세스가 아직 올라오지 않았다면, 그것을 보조저장장치로부터 가져와야 한다

3. 빈 공간, 즉 가용 프레임(free frame)을 찾는다

4. 보조저장장치에 새로이 할당된 프레임으로 해당 페이지를 읽어 들이도록 요청한다

5. 보조저장장치 읽기가 끝나면, 이 페이지가 이제는 메모리에 있다는 것을 알리기 위해 페이지 테이블을 갱신하며, 프로세스가 유지하고 있는 내부 테이블을 수정한다

6. 트랩에 의해 중단되었던 명령어를 다시 수행한다. 이제 프로세스는 마치 그 페이지가 항상 메모리에 있었던 것처럼 해당 페이지에 접근할 수 있다



극단적인 경우에는 메모리에 페이지가 하나도 안 올라와 있는 상태에서도 프로세스를 실행시킬 수 있다

운영체제에서 명령 포인터의 값을 프로세스의 첫 명령으로 설정하는 순간, 이 명령이 메모리에 존재하지 않는 페이지에 있으므로, 페이지 폴트를 발생시킨다

페이지가 적재되고 나면 프로세스는 수행을 계속하는데 프로세스가 사용하는 모든 페이지가 메모리에 올라올 때까지 필요할 때마다 페이지 폴트가 발생한다

일단 필요한 모든 페이지가 적재되고 나면 더 폴트가 발생하지는 않는다

이것이 순수 요구 페이징(pure demand paging)이다

즉, 어떤 페이지가 필요해지기 전에는 결코 그 페이지를 메모리로 적재하지 않는 방법이다

프로그램들은 한 명령어에서도 여러 개의 페이지 폴트를 일으킬 수 있다

이렇게 되면 시스템 성능의 저하를 초래할 것이다

다행히 실행 중인 프로세스들을 분석해 보면 이러한 경우는 거의 발생하지 않는다는 것을 알 수 있다

모든 프로그램은 참조의 지역성(locality of reference)이라는 성질이 있어서 프로그램의 어느 한 특정 작은 부분만 한동안 집중적으로 참조하는데, 이러한 성질 때문에 요구 페이징은 만족할 만한 성능을 보인다

**10.2.2 가용 프레임 리스트**

페이지 폴트가 발생하면 운영체제는 요청한 페이지를 보조저장장치에서 메인 메모리로 가져와야 한다

페이지 폴트를 해결하기 위하여 대부분의 운영체제는 이러한 요청을 충족시키기 위해 사용하기 위한 가용 프레임 풀인 가용 프레임 리스트를 유지한다

운영체제는 일반적으로 zero-fill-on-demand라는 기법을 사용하여 가용 프레임을 할당한다

Zero-fill-on-demand 프레임은 할당되기 전에 “0으로 모두 채워져” 이전 내용이 지워진다

프레임을 다시 할당하기 전에 프레임의 내용을 지우지 않을 경우 잠재적인 보안 취약점이 될 수도 있다는 것을 고려하라

시스템이 시작되면 모든 가용 메모리가 가용 프레임 리스트에 넣어진다

가용 프레임이 요청되면 가용 프레임 리스트의 크기가 줄어든다

어떤 시점에서, 리스트의 크기는 0으로 떨어지거나 특정 임계값 밑으로 떨어지며, 이 시점에서 다시 채워져야 한다

**10.2.3 요구 페이징의 성능**

페이지 폴트 처리 시간은 다음 3개의 주요 작업 요소로 이루어져 있다

1. 인터럽트의 처리

2. 페이지 읽기

3. 프로세스 재시작

TLB hit 80%

TLB miss 18%

Page fault 2%

Access time 1microsecond

Average access and transfer time 20milliseconds

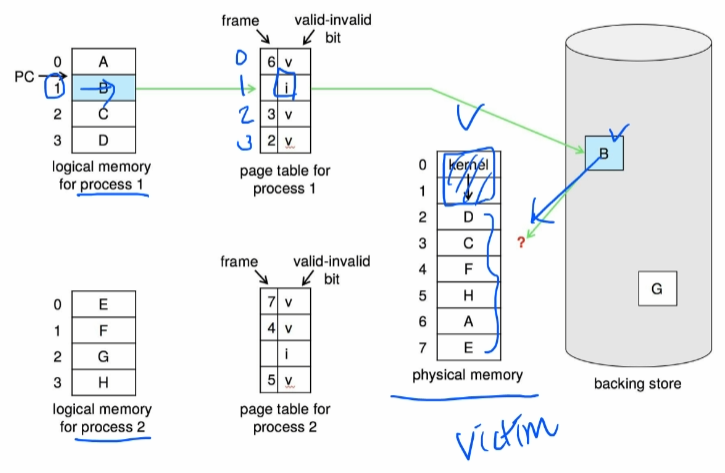
80% \* 1mus = 0.8 \* 1 = 0.8mus

18% \* 0.18mus = 0.18 \* 2 = 0.36mus

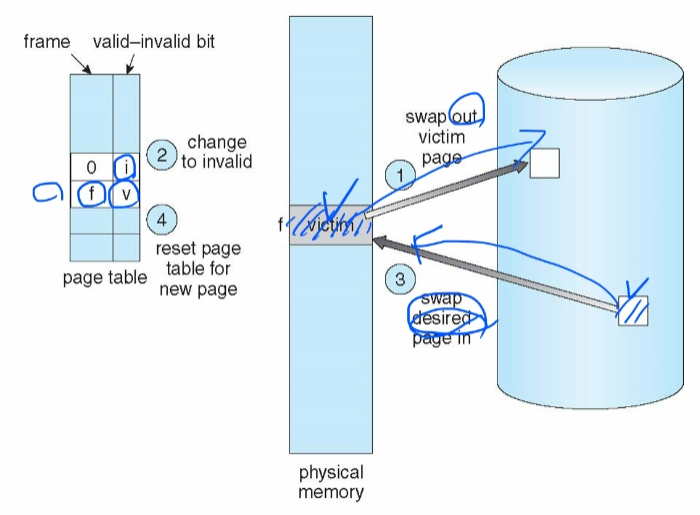
2% \* 20002mus = 0.02 \* 20002 = 400.04mus

= 401.2mus = 0.4012ms

**10.4 페이지 교체**



**10.4.1 기본적인 페이지 교체**



요구 페이징 시스템은 두 가지 중요한 문제를 해결해야 한다

프레임 할당(frame-allocation) 알고리즘: 여러 프로세스가 존재하는 경우 각 프로세스에 얼마나 많은 프레임을 할당해야 하는지 결정해야 한다

페이지 교체(page-replacement) 알고리즘: 페이지 교체가 필요할 때 어떤 페이지를 교체해야 할지 결정해야 한다

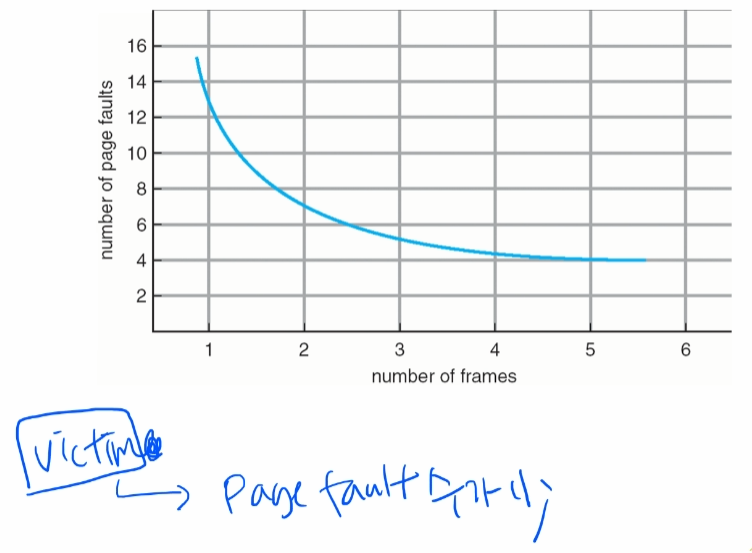
페이지 교체 알고리즘의 성능은 특정 메모리 참조 나열에 대해 알고리즘을 적용하여 페이지 폴트 발생 횟수를 계산하여 평가한다

이러한 메모리 주소의 나열을 참조열(reference string)이라 부른다

페이지 교체 알고리즘을 설명하기 위하여 3개의 프레임을 가진 메모리를 가정하고 페이지 참조열은 아래와 같다고 가정한다

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

프레임의 수가 증가함에 따라 페이지 폴트 수는 어떤 최솟값으로 떨어진다

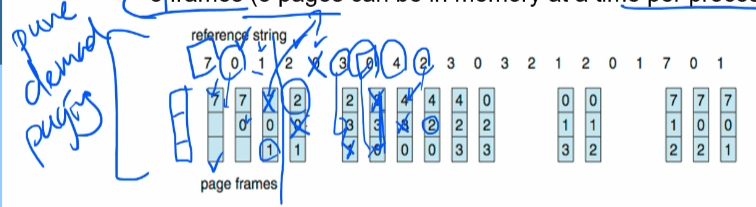


**10.4.2 FIFO 페이지 교체**

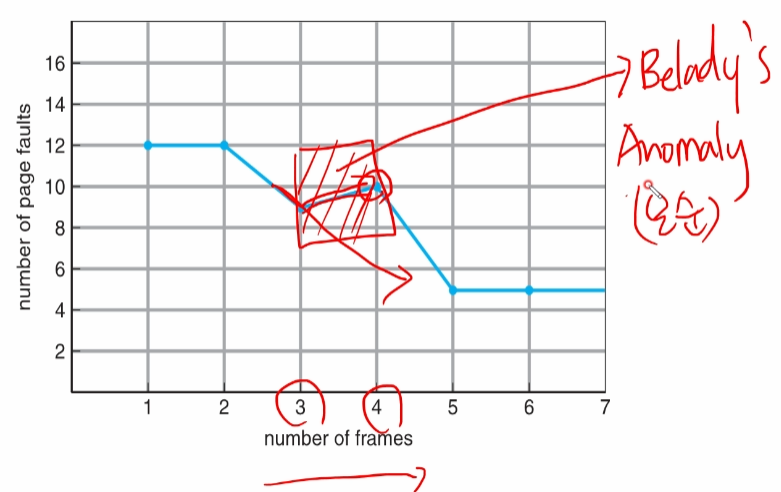
메모리에 올라온 지 가장 오래된 페이지를 내쫓는다

FIFO 페이지 교체 알고리즘은 이해하기도 쉽고, 프로그램하기도 쉽다

그러나, 성능이 항상 좋지는 않다



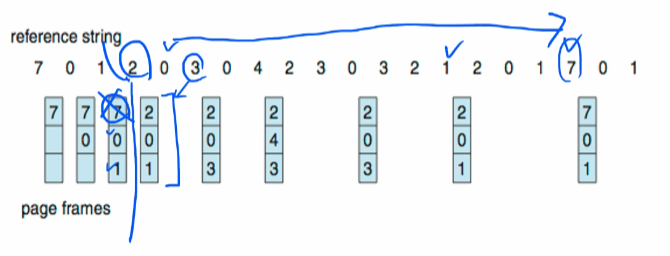
Belady의 모순은 프로세스에 프레임을 더 주었는데 오히려 페이지 폴트율은 더 증가하는 현상을 일컫는다



**10.4.3 최적 페이지 교체 Optimal Page Replacement**

최적 교체 정책이란 모든 알고리즘보다 낮은 페이지 폴트율을 보이며 Belady의 모순이 발생하지 않는 것이다

앞으로 가장 오랫동안 사용되지 않을 페이지를 찾아 교체한다



안타깝게도 이 알고리즘의 실제 구현은 어렵다

왜냐하면, 이 방시기은 프로세스가 앞으로 메모리를 어떻게 참조할 것인지를 미리 알아야 하기 때문이다

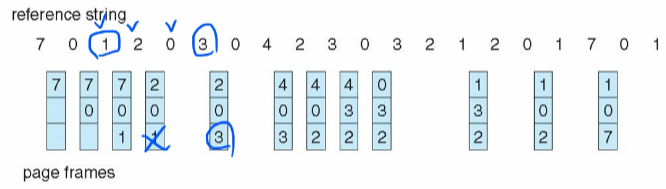
따라서 최적 페이지 교체 알고리즘은 주로 비교 연구 목적을 위해 사용한다

**10.4.4 LRU 페이지 교체**

가장 오랜 기간 동안 사용되지 않은 페이지를 교체할 수 있다

LRU 알고리즘은 페이지마다 마지막 사용 시간을 유지한다

페이지 교체 시에 LRU는 가장 오랫동안 사용되지 않은 페이지를 선택한다



LRU 정책은 페이지 교체 알고리즘으로 자주 사용되며 좋은 알고리즘으로 인정받고 있다

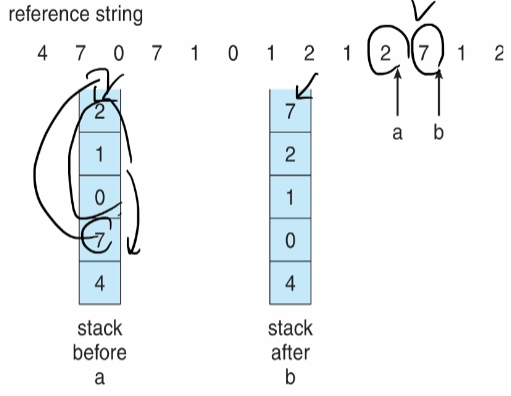
문제는 어떻게 이 알고리즘을 구현하느냐 하는 것이다

LRU 페이지 교체 알고리즘은 하드웨어의 지원이 필요하다

두 가지의 구현 방법이 가능하다

계수기: CPU에 논리적인 시계나 계수기를 추가한다

스택: 꼭대기는 항상 가장 최근에 사용된 페이지이고, 밑바닥은 가장 오랫동안 사용되지 않은 페이지이다



OPT와 마찬가지로 LRU 교체는 Belady의 모순 현상을 야기하지 않는다

이러한 알고리즘들을 스택 알고리즘(stack algorithm)이라고 부른다

**10.4.5 LRU 근사 페이지 교체**

LRU 페이지 교체 지원을 충분히 할 수 있는 하드웨어는 많지 않다

그러나 많은 시스템은 참조 비트(reference bit)의 형태로 어느 정도의 지원은 하고 있다

처음에 모든 참조 비트는 운영체제에 의해 0으로 채워진다

프로세스가 실행되면서 참조되는 페이지의 비트는 하드웨어가 1로 세팅한다

**10.4.5.2 2차 기회 알고리즘**

2차 기회 알고리즘의 기본은 FIFO 교체 알고리즘이다

그러나 페이지가 선택될 때마다 참조 비트를 확인한다

참조 비트가 0이면 페이지를 교체하고 1이면 다시 한번 기회를 주고 다음 FIFO 페이지로 넘어간다

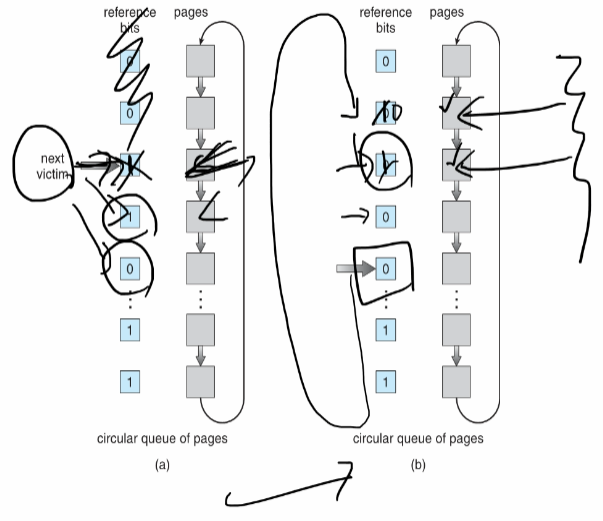
어떤 프레임을 빼앗아야 할 일이 필요해지면, 포인터는 0 값의 참조 비트를 가진 페이지를 발견할 때까지 큐를 뒤진다

포인터가 돌아가면서 참조 비트 값들이 1인 것은 0으로 바꾼다

희생될 페이지(0을 가지고 있는 페이지)가 발견되면, 그 페이지는 교체되고 새로운 페이지는 순환 큐의 해당 위치에 삽입한다

최악의 경우, 모든 비트가 1 값을 가지고 있었으면 포인터는 큐를 완전히 한 바퀴 돈다(그러면서 0 값으로 세트)

두 번째 돌 때는 사실상 FIFO와 같은 것이 된다



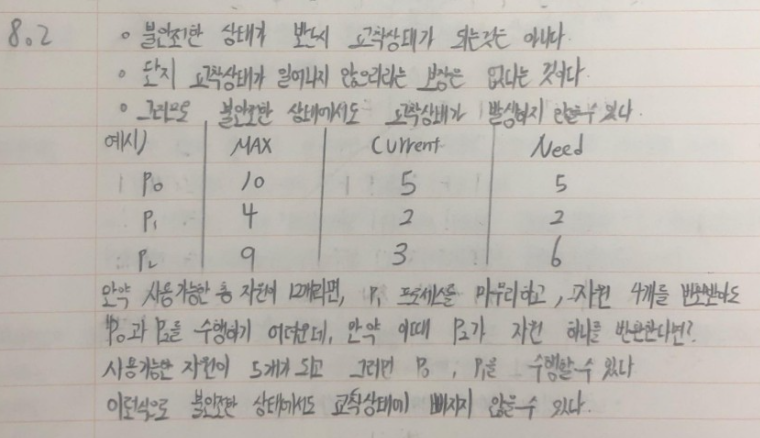
**10.4.6 계수-기반 페이지 교체 Counting Algorithms**

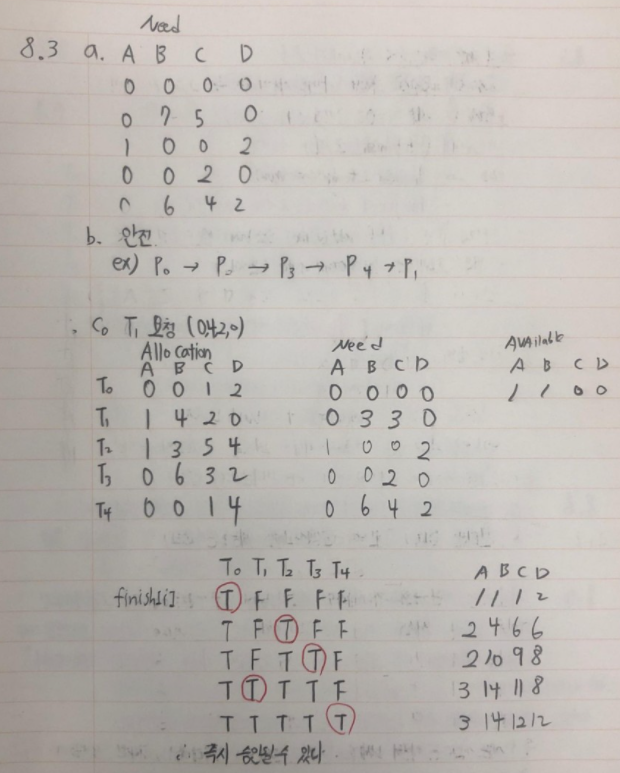
각 페이지를 참조할 때마다 계수를 하여 다음과 같은 두 가지의 기법을 만들 수 있다

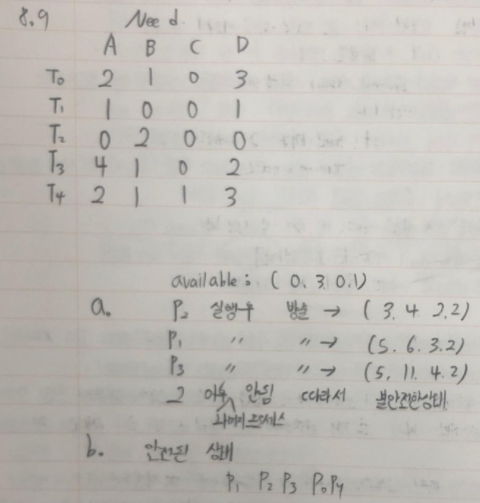
LFU 알고리즘: 참조 횟수가 가장 작은 페이지를 교체하는 방법이다

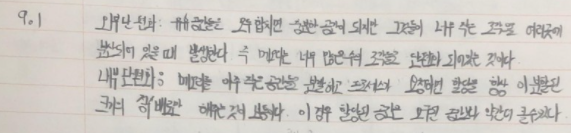
MFU 알고리즘: 가장 작은 참조 회수를 가진 페이지가 가장 최근 참조된 것이고 앞으로 사용될 것이라는 판단에 근거한 것이다

이들은 구현하는 데에 상당히 비용이 많이 들고 OPT 페이지 교체 정책을 제대로 근사하지 못하기 때문에 일반적으로 잘 쓰이지는 않는다

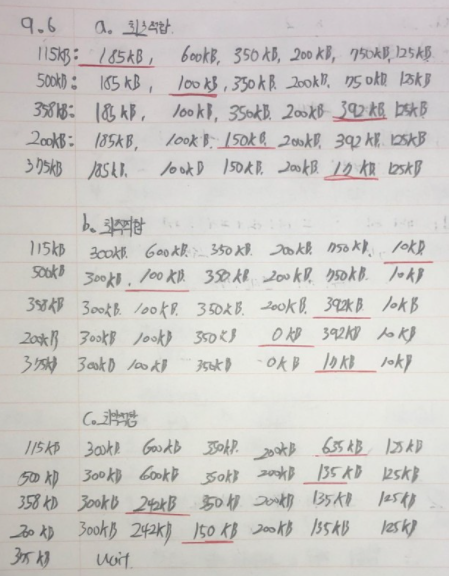


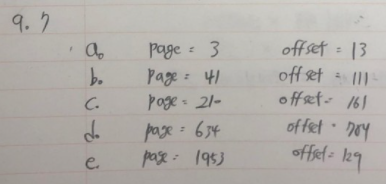


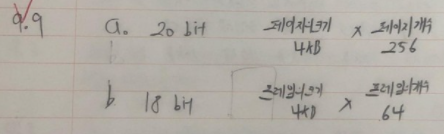


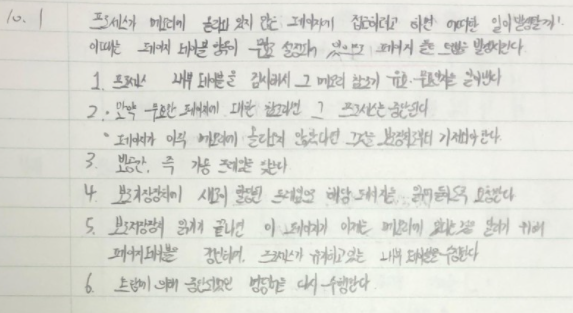


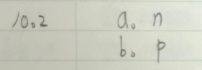




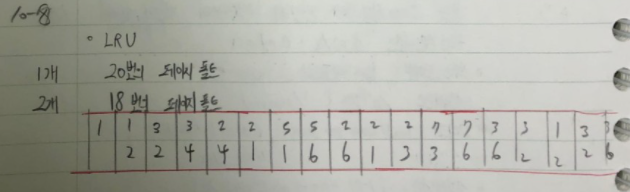


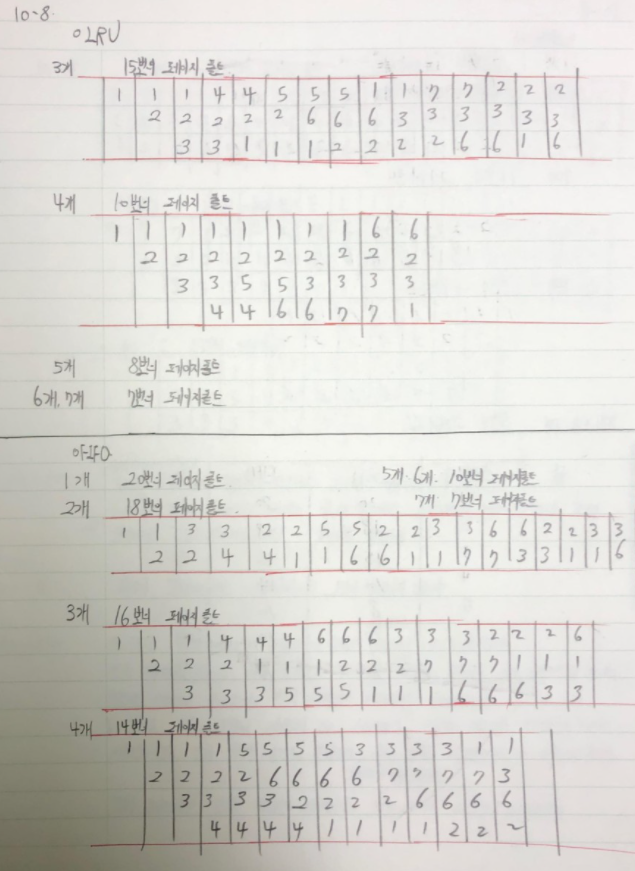


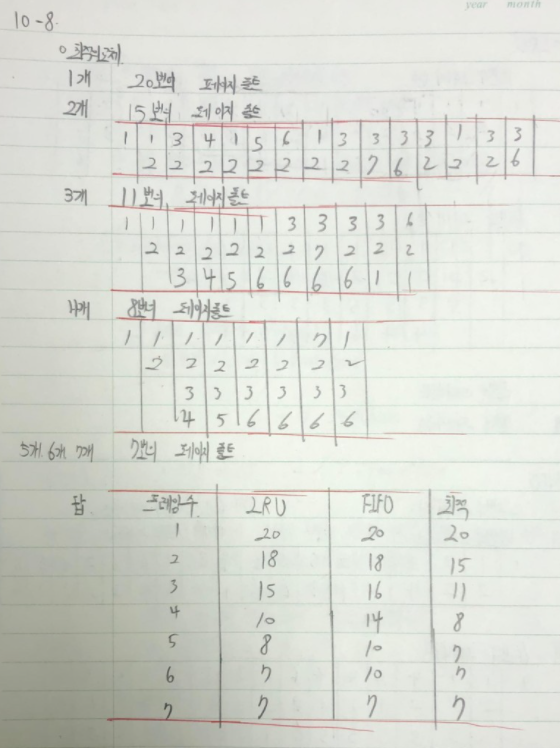


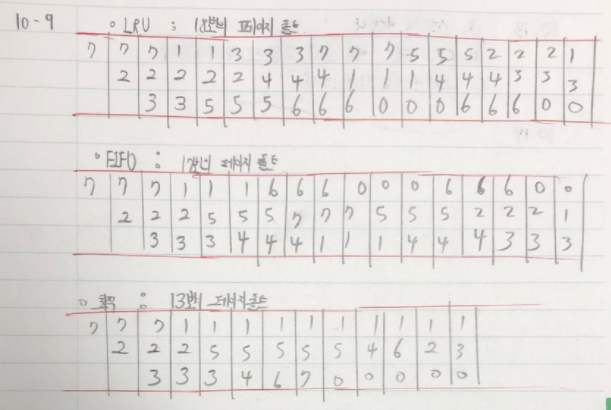












TLB hit 80%

TLB miss 18%

Page fault 2%

Access time 1microsecond

Average access and transfer time 20milliseconds

80% \* 1mus = 0.8 \* 1 = 0.8mus

18% \* 0.18mus = 0.18 \* 2 = 0.36mus

2% \* 20002mus = 0.02 \* 20002 = 400.04mus

= 401.2mus = 0.4012ms