

Project 2

멀티 프로세스 with 가상메모리

32201345 나웅철

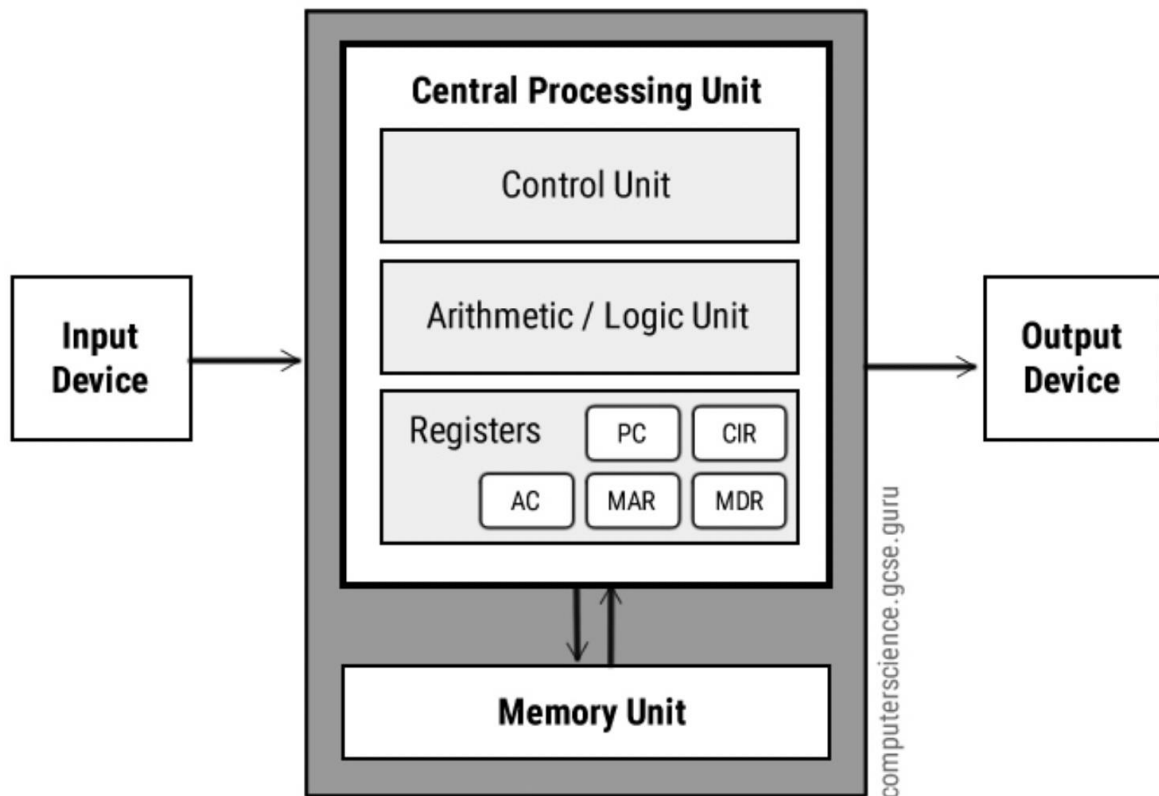
32217259 문서영

<시작하며>

이번 과제는 저번 과제에 이어 가상메모리 개념을 더한 스케줄링 프로그램 시뮬레이션을 구현하는 것이 목표이다. 가상메모리라는 단어 하나만 더해졌지만 매우 긴밀히 유기적으로 연결되어 있는 여러가지 개념들이 더해졌다고 봐야 한다. 컴퓨터 부팅 후 운영체제 실행은 어떻게 되는지, 어떻게 RAM보다 큰 용량을 가진 프로그램들이 RAM에서 실행될 수 있는지, 가상 메모리와 물리 메모리는 왜 나뉘는지, 프로그램마다 32bit 운영체제와 64bit 운영체제 별로 따로 존재하는지 등등에 대한 내용을 담을 계획이다. 또한 마지막으로 실제로 가상메모리 개념을 더한 스케줄링 시뮬레이터를 구현해볼 계획이며 CPU가 다루는 데이터 크기에 따라 설계를 어떻게 해야 했는지, 페이지 선택 알고리즘에 따른 성능평가 등을 다뤄보겠다.

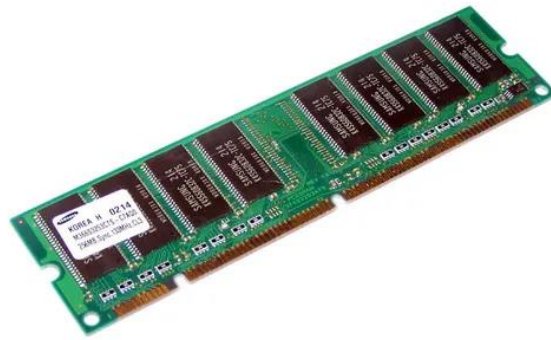
<물리 메모리>

메모리 기본

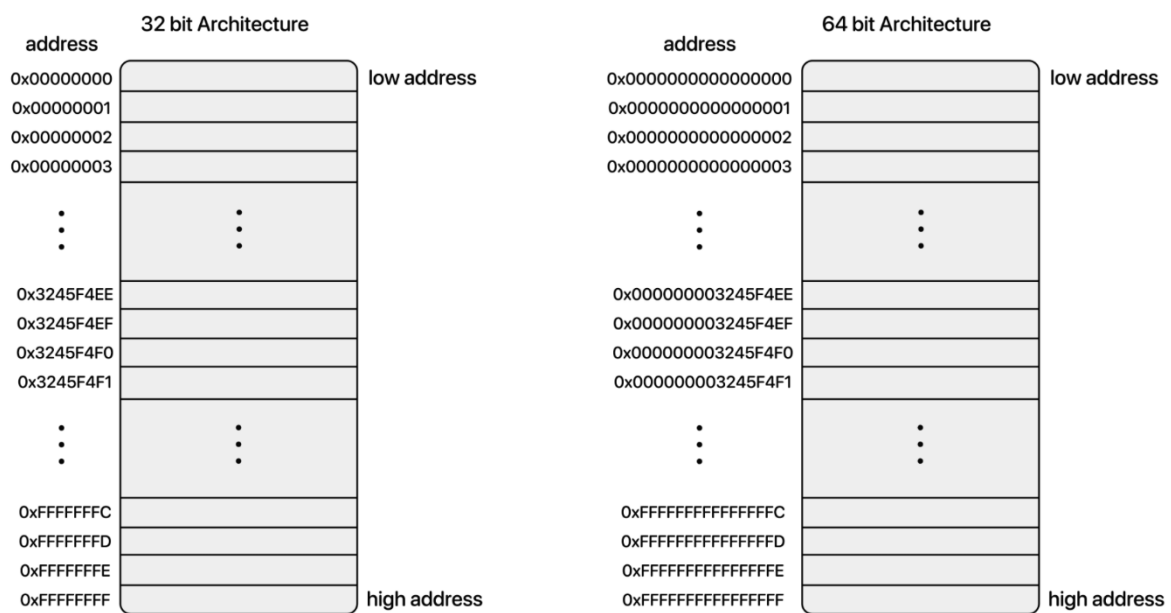


[그림 1] 폰노이만 구조

폰노이만 구조에서 메모리는 CPU 와 디스크 사이에서 CPU 의 작업공간으로 사용된다. 모든 프로그램은 하드디스크와 같은 보조 기억장치에 저장되어 작업 대상이 되었을 때 비로소 메모리에 올라와 실행된다. 이는 곧 이전의 과제들에서 살펴보았듯 프로그램이 프로세스가되어 실행됨을 의미한다. 그렇다면 프로그램은 메모리(메인 메모리)에 어떠한 상태로 저장되며 운영체제에 의해 어떻게 다뤄지며 CPU 는 어떻게 이를 사용하여 계산할까?

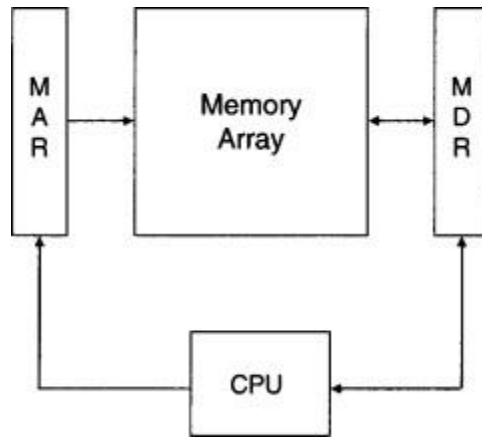


[그림 2] 삼성전자 PC133 SDR SDRAM



[그림 3] 32bit 운영체제 용 메모리, 64bit 운영체제 용 메모리

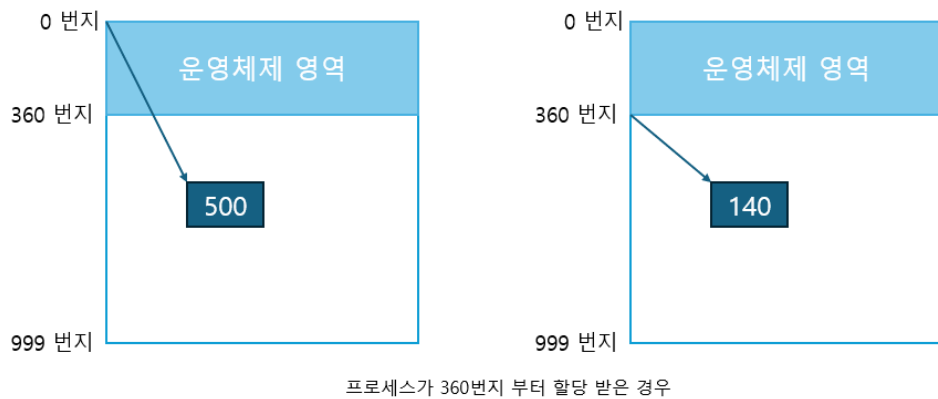
메모리의 구조는 [그림3]에서 볼 수 있듯이 각 영역은 0번지로부터 시작되는 주소로 구분되며 각 주소에 할당되는 영역의 크기는 1바이트의 크기로 이루어진다. CPU는 메모리에 있는 데이터를 가져와 연산하여 다시 메모리에 저장한다. 이 때 CPU가 몇 비트 CPU이냐에 따라 접근 가능한 메모리의 범위가 정해진다. 예를 들어 32비트 CPU는 CPU가 작업하는 데이터의 크기가 32비트로 이루어져 있다는 의미다. 마찬가지로 64비트의 경우에는 CPU가 작업하는 데이터의 크기가 64비트로 이루어져 있다는 의미이다. CPU의 데이터 처리 단위는 많은 것을 결정짓는다. CPU 안에 들어 있는 메모리 주소 레지스터, 버스의 크기, 논리 주소 공간 등등이 그러하다.



[그림 4] Memory access mechanism (출처 : www.sciencedirect.com)

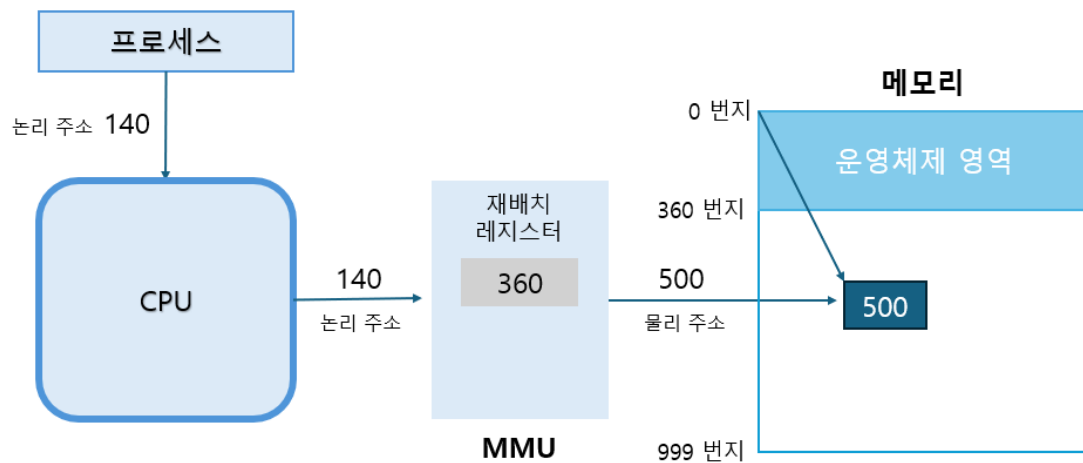
CPU에서는 프로세스의 흐름에 따라 적절한 메모리 공간으로부터 값을 받아와 계산하고 다시 적절한 메모리 공간에 결과값을 저장해야 한다. CPU는 MAR(Memory Address Register)에 들어있는 값을 통해 다음에 접근해야 할 메모리 주소를 알고, 해당 주소로 버스를 통해 접근한다. 32bit CPU를 사용하는 경우 표현 가능한 수의 범위는 $0 \sim 2^{32}-1$ 번지로 표현할 수 있으며 64bit CPU의 경우 $0 \sim 2^{64}$ 번지의 범위로 표현할 수 있다. 따라서 이렇게 MAR이 저장할 수 있고 CPU가 받아들일 수 있는 공간의 범위를 CPU가 사용할 수 있는 메모리의 최대 크기라고 볼 수 있다. 좀 더 이해하기 쉬운 단위로 환산할 경우 32bit CPU는 최대 4GB 정도의 메모리 주소를 표현할 수 있고, 64bit CPU의 경우 약 16,777,216TB의 메모리 주소를 표현할 수 있으니 사실상 거의 무한한 메모리에 접근할 수 있다고 볼 수 있다.

어떤 CPU이던 간에 메모리가 설치된 경우 각 메모리는 주소 공간이 있으며 실제 주소 공간을 '물리 주소 공간'이라고 하며, 사용자가 바라본 주소 공간은 '논리 주소 공간'이라고 한다. **사용자가 바라본**이라는 문장이 어색하게 들릴 수 있으나 절대 주소와 상대 주소에 비유하면 쉽게 이해할 수 있다. 물리 주소 공간은 실제 메모리 0 번지부터 시작하며 논리 주소 공간은 해당 프로세스가 할당 받은 메모리 공간을 기준으로 첫번째 공간을 0 번지로 생각하는 것이다. 이를 통해 각 프로세스는 자신만의 독립적인 메모리 공간을 갖고 있는 것처럼 존재할 수 있다.



[그림 5] 물리 주소와 논리 주소

프로세스는 논리 주소를 기준으로 실행하므로 컴퓨터에서는 이 논리 주소를 물리 주소로 변환하는 작업이 필요하다. 이 작업은 CPU 안에 존재하는 메모리 관리 유닛이 담당한다.



[그림 6] 360번지부터 시작하는 메모리 공간을 할당 받은 프로세스가 140번지 논리 주소를 요청

메모리 관리 유닛(MMU)은 프로세스가 할당 받은 메모리 공간을 찾기 위해 재배치 레지스터에 할당된 값을 기준으로 한다. [그림 6]을 예로 들면 프로세스가 요청한 논리 주소는 140 번지이지만 재배치 레지스터가 360 번 주소를 가지고 있으므로 이는 곧 360 번지로부터 140 만큼 떨어진 500 번지에 대한 데이터를 요청하는 것과 같다. 재배치 레지스터는 프로세스가 할당 받은 메모리 공간의 시작 주소를 갖는다.

메모리 관리

메모리 관리 작업은 프로세스가 실행되기 위해 필요한 메모리를 효율적으로 할당하고 관리하는 과정으로, 크게 세 가지로 나눌 수 있다:

1. 메모리 가져오기: 실행할 프로세스와 데이터를 메모리로 가져오는 단계로, 디스크에 저장된 프로그램과 데이터를 물리 메모리로 로드하여 실행 환경을 준비한다.
 - 요구 페이징 : 응답 속도를 향상시키고 메모리를 효율적으로 관리하기 위해 사용자가 요구할 때 해당 페이지(프로세스의 일부)를 메모리로 가져오는 것
 - 캐시 : 사용자가 사용할 것이라고 예측되는 페이지를 미리 가져오는 방식
2. 메모리 배치: 가져온 프로세스와 데이터를 메모리의 어느 위치에 배치할지 결정하는 단계이다. 이 과정에서는 메모리를 어떤 크기로 나누고, 프로세스를 어떻게 배치할 것인지가 중요하다.
 - 배치 전 메모리 분할: 메모리를 효율적으로 사용하기 위해 메모리를 미리 나누는 방식(고정 분할)과 필요한 만큼만 동적으로 나누는 방식(가변 분할)이 있다.
3. 메모리 재배치: 메모리가 부족하여 새로운 프로세스를 가져올 공간이 없는 경우, 기존에 메모리에 있던 프로세스를 하드 디스크로 옮기고(스왑 아웃) 그 공간을 사용하는 단계이다.
 - 교체 알고리즘: 어떤 프로세스를 디스크로 내보낼지 결정하는 알고리즘이 메모리 재배치의 효율성에 중요한 역할을 한다. 대표적인 교체 알고리즘에는 FIFO, LRU 등이 있다.

먼저 메모리 배치 정책에 대해 살펴보고, 이후 메모리 재배치에 대해 다뤄보자.

메모리 배치 정책의 경우 가져온 프로세스 또는 프로세스들을 메모리에 어떻게 올릴 것인지에 대한 정책이다. 선택지는 단순하다. 먼저 각각을 통째로 올리는 방법이 있을 수 있고, 각각을 쪼개어 일부분씩 올리는 방법이 있을 수 있다. 전자에 해당하는 방식을 “가변 분할 방식”, 후자에 해당하는 방식을 “고정 분할 방식”이라고 부른다.

메모리 배치 정책

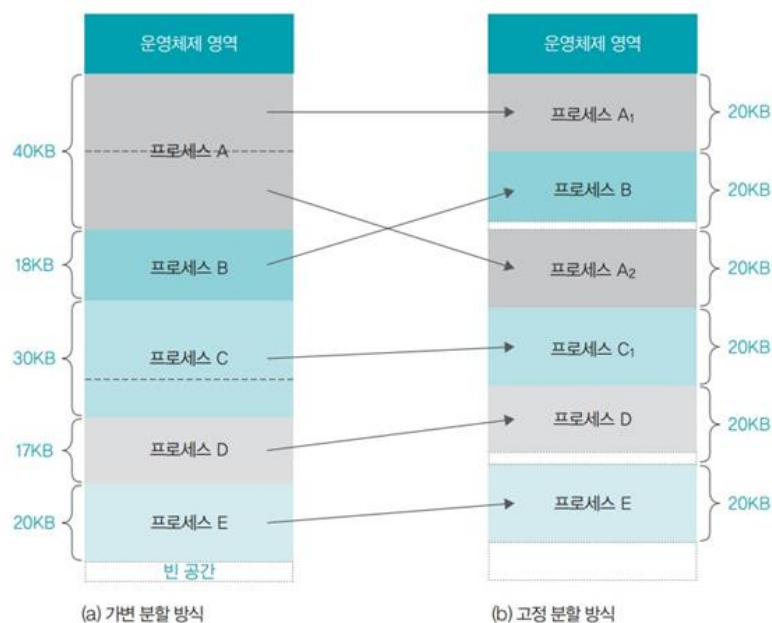
1. 가변 분할 방식:

- 프로세스의 크기에 맞게 동적으로 메모리를 나누어 배치하는 방식이다. 메모리 사용량에 따라 크기가 변동되므로, **외부 단편화(External Fragmentation)**가 발생할 가능성이 있다.
- **세그먼테이션(Segmentation)** 방식이 이에 해당하며, 프로세스를 논리적으로 구분된 세그먼트(예: 코드, 데이터, 스택) 단위로 메모리에 배치한다. 각 세그먼트는 크기가 다르며, 논리적 구조를 반영하여 배치되므로 메모리 사용의 유연성이 높다.

- 고정 분할 방식:

- 메모리를 미리 고정된 크기의 블록으로 나누어 배치하는 방식이다. 프로세스의 크기와 상관없이 정해진 크기로 나뉜 블록(프레임)에 프로세스를 적재한다.
- **페이징(Paging)** 방식이 이에 해당하며, 가상 주소 공간과 물리 메모리를 동일한 크기의 페이지와 프레임으로 나누어 사용한다. 이는 외부 단편화를 제거하지만, **내부 단편화(Internal Fragmentation)**가 발생할 수 있다.

가변 분할 방식은 유연성을 제공하지만 관리가 복잡하며, 고정 분할 방식은 관리가 단순하지만 메모리 낭비가 생길 수 있다. 운영체제는 상황에 따라 두 방식을 혼합하여 사용하는 경우도 있다.



[그림 7] 메모리 배치 정책

가변 분할 방식은 프로세스를 통째로 배치할 수 있다는 장점이 있지만 매번 달라지는 크기의 단위로 인해 메모리 관리가 복잡하다는 단점이 있다. 예를 들어 15kb의 프로세스를 실행하기

위해선 15kb 이상의 빈공간을 찾아 연속적인 공간을 배치해야 한다. 이를 위해선 이미 존재하는 프로세스들의 위치를 옮겨 떨어져 있는 빈 공간을 합치는 등의 복잡한 메모리 관리가 필요하다. 반면에, 고정 분할 방식의 경우 메모리를 일정한 크기로 나누어 각각의 메모리 조각들에 프로세스들이 나뉘어 배치되기 때문에 가변 분할 방식처럼 프로세스가 들어갈 수 있는 공간을 만들기 위한 작업을 하지 않아도 된다. 이런 이유로 현대 운영체제에서는 가변 분할 방식 보다 고정 분할 방식의 배치 정책이 사용된다. 고정 분할 방식을 사용하여 물리 메모리를 나누어 사용하는 방식을 페이징 메모리 기법이라고 한다.

메모리 재배치

메모리 재배치는 기존 프로세스를 내보내고 새로운 프로세스를 적재하는 과정으로, 스왑(Swap) 과정을 통해 이루어진다. 스왑은 메모리 부족 상황에서 덜 사용되는 프로세스를 디스크로 내보내고(스왑 아웃), 필요한 프로세스를 메모리에 다시 적재(스왑 인)하는 방식이다. 이 과정에서 교체 알고리즘이 중요한 역할을 한다.

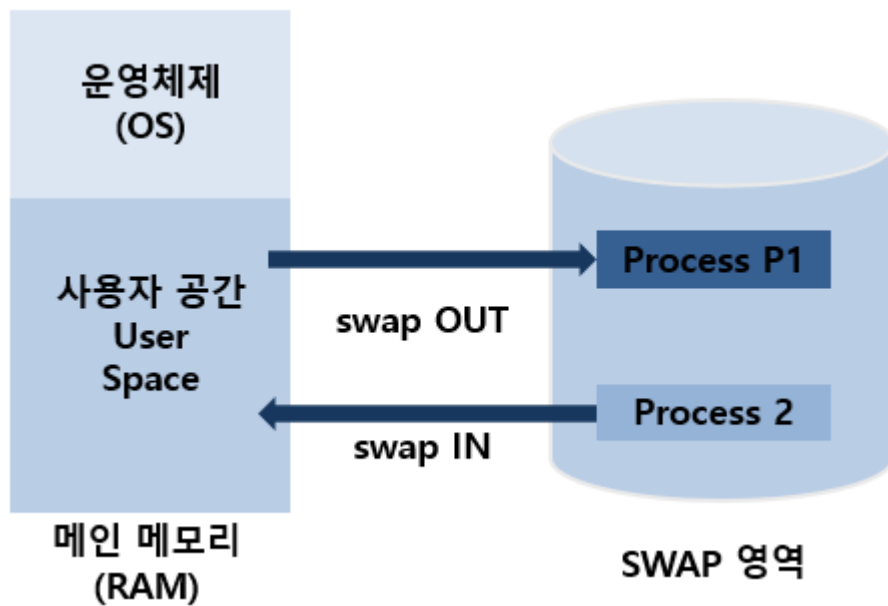
- 교체 알고리즘의 종류:

1. FIFO(First-In-First-Out): 가장 먼저 메모리에 적재된 페이지를 먼저 교체하는 방식으로, 구현이 간단하지만 효율성이 떨어질 수 있다.
2. LRU(Least Recently Used): 가장 오랫동안 사용되지 않은 페이지를 교체하여 참조 지역성(Locality of Reference)을 반영한 방식으로, 효율성이 높다.
3. Clock 알고리즘: LRU 를 효율적으로 구현한 방식으로, 페이지를 순환하며 참조 비트를 이용해 교체할 페이지를 선택한다.

메모리 재배치는 성능 저하를 초래할 수 있으므로, 적절한 교체 알고리즘과 메모리 배치 정책을 조화롭게 설계하는 것이 중요하다.

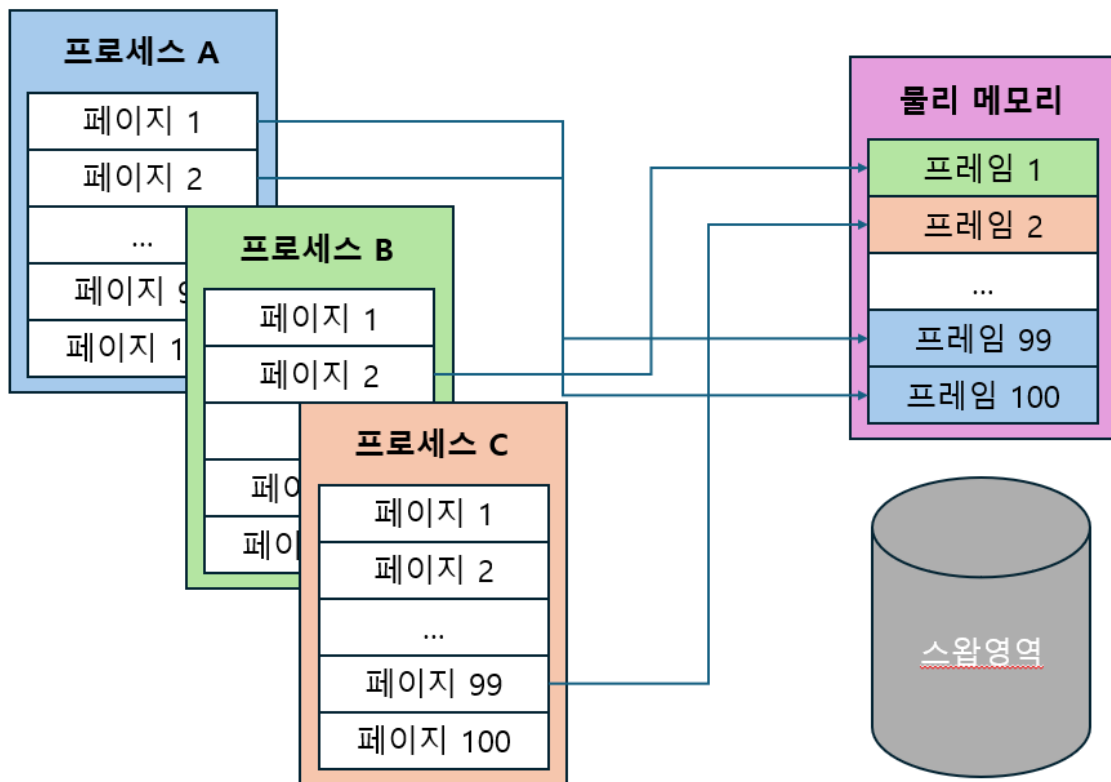
마지막의 실습에서 필자도 좀 더 유리한 메모리 관리를 위해 페이징 방식을 사용할 계획이다. 따라서 페이징 기법에 대해 좀 더 자세한 기술을 하고자 한다. 하지만 그 전에 먼저 알아 두어야 할 개념 몇 가지를 소개한다.

스왑 영역, 가상 메모리 시스템, 페이지 테이블



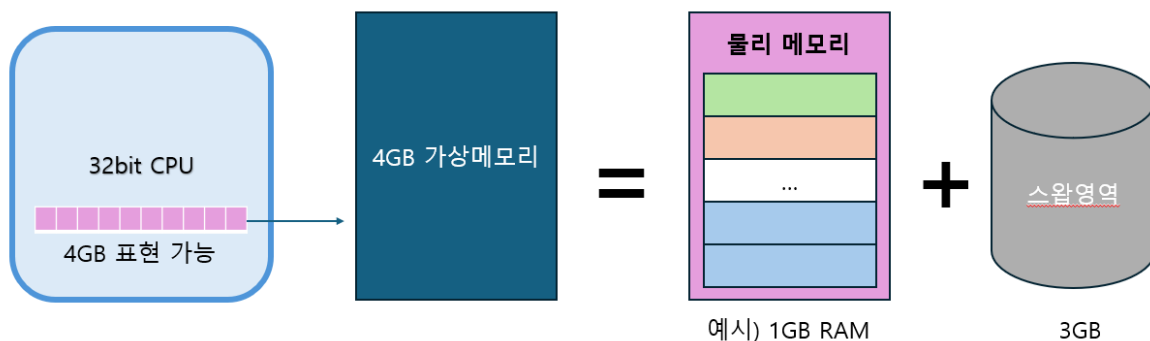
[그림 8] 스왑 영역

스왑 영역은 운영체제가 물리 메모리가 부족할 때 디스크의 일부를 활용하여 데이터를 임시로 저장하는 영역이다. 스왑 영역에서 메모리로 데이터를 가져오는 작업을 스왑인, 메모리에서 스왑 영역으로 데이터를 내보내는 작업을 스왑아웃이라고 한다. 비록 스왑 영역이 하드디스크에 존재하지만 메모리의 연장선으로서 동작하기 때문에 메모리 관리자에 의해 관리된다는 특징이 있다.



[그림 9] 가상 메모리 시스템 개요 _ 다중 프로그래밍 환경의 메모리 할당 with 페이징 기법

모든 프로세스는 메모리의 위치와 상관없이 0번부터 시작하는 연속된 메모리 공간을 갖는다. 좀 더 자세히 말하자면 물리 메모리의 위치와 상관없이 연속된 메모리 공간을 갖는 것처럼 작동한다. 또한 프로세스의 개수가 1개이던 여러 개이던 상관없이 프로세스(들)의 총 크기가 물리 메모리를 넘어서서 스왑 영역에 일부 존재하더라도 연속된 프로세스는 메모리 공간을 갖는 것처럼 동작한다. 이처럼 가상 메모리 시스템에서 메모리 관리자는 물리 메모리와 스왑 영역을 합쳐서 프로세스가 사용하는 가상 주소를 물리 주소로 변환하여 사용한다. 이를 동적 주소 변환(Dynamic Address Transformation)이라고 한다.



[그림 10] 가상메모리의 최대 크기 _ 주소 공간 전체의 크기

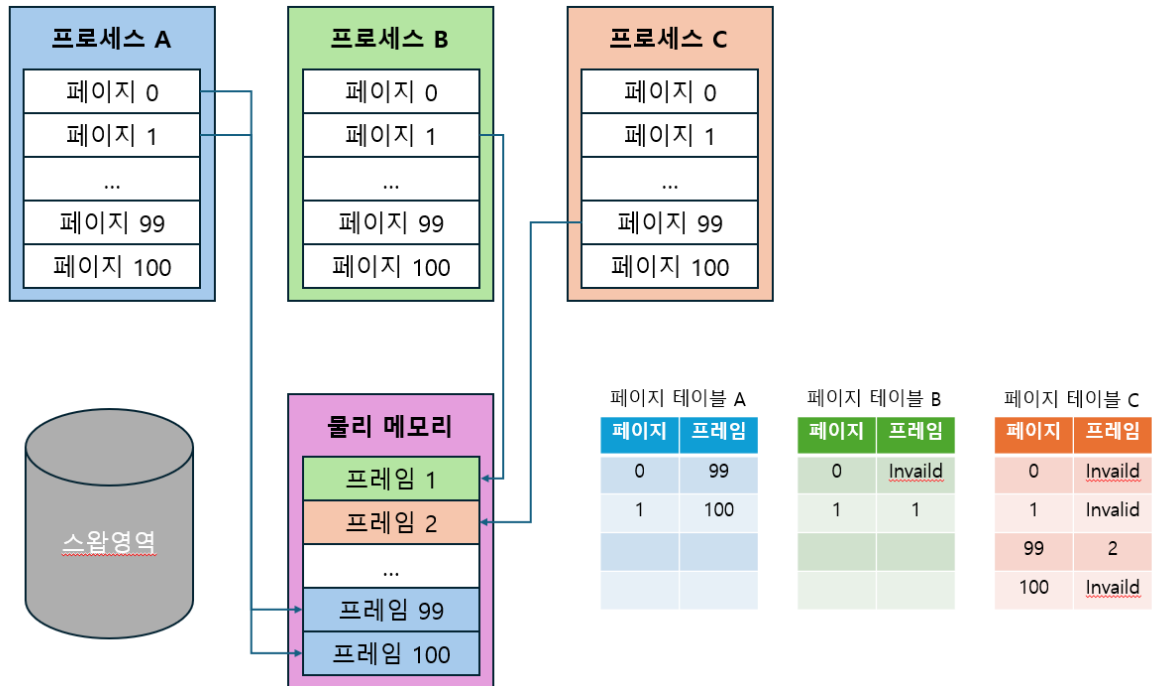
프로세스마다 각각 할당 받는 가상 메모리는 주소 공간으로 표현할 수 있는 만큼의 용량을 가지며 실질적으로는 메모리 크기와 스왑 영역의 합에 따른 제약을 받는다.



[그림 11] 매핑 테이블

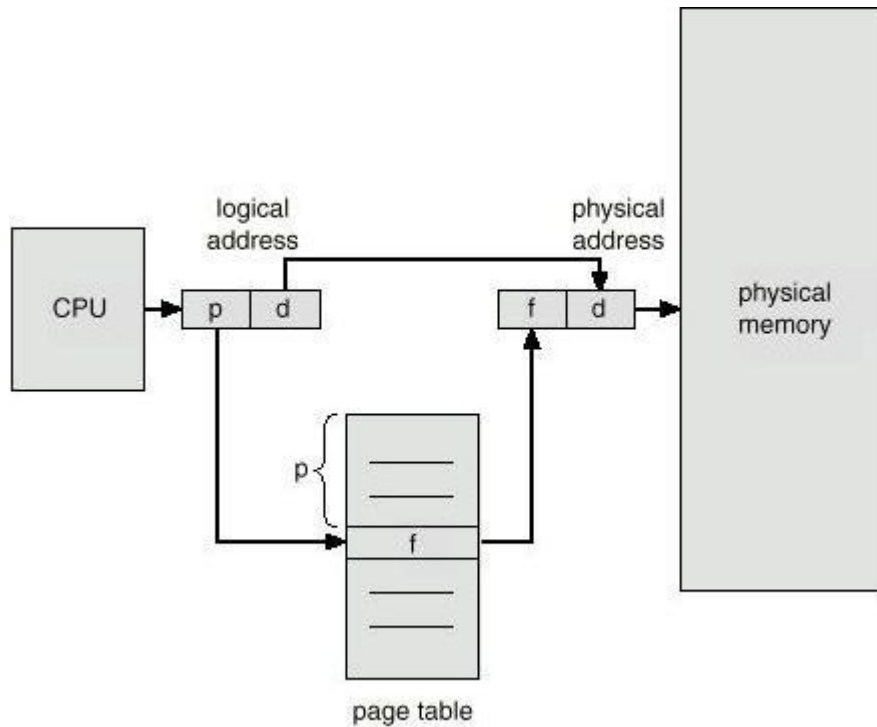
이제는 이런 의문이 든다. 각각의 프로세스는 독립적인 가상 메모리를 갖고 그 안에서 논리 주소로 진행이 되면 메모리 관리자 유닛은 어떻게 실제 물리 메모리와 연결 지을까? 그에 대한 해답은 그냥 연결 구조를 나타내는 매핑 테이블을 만드는 것이다. 프로세스와 물리주소를 연결한 표를 만들고 MMU가 그때 그때 변환해주면 되는 것이다.

페이징 기법



[그림 12] 페이징 기법의 구현

이전에 다뤘듯이 페이징 기법은 고정 분할 방식을 이용한 가상 메모리 관리 기법으로, 가상 주소 공간을 페이지라는 단위로 일정한 크기로 나누어 사용하며, 같은 크기의 프레임이라는 단위로 물리 주소 공간을 나누어 사용한다. 모든 페이지와 프레임은 페이지 테이블이라는 매핑 테이블을 통해 매핑되며, MMU(Memory Management Unit)는 이를 사용하여 가상 주소를 물리 주소로 변환하는 작업을 수행한다. 페이지 테이블은 가상 주소의 페이지 번호를 물리 메모리의 프레임 번호로 변환하기 위해 사용되며, 페이지 번호와 프레임 번호로 구성된 각각의 한 줄은 페이지 테이블 엔트리(PTE, Page Table Entry)라고 한다. 이 PTE 는 페이지의 메모리 존재 여부(Valid Bit), 수정 여부(Dirty Bit), 접근 권한(Read/Write Permission)과 같은 추가 정보를 포함하여 메모리 관리의 효율성과 보호를 강화한다. 즉, 페이지 테이블은 페이지 엔트리의 집합으로 이루어진 데이터 구조이며, 이를 통해 물리 메모리를 효율적으로 관리하고 외부 단편화를 방지할 수 있다.



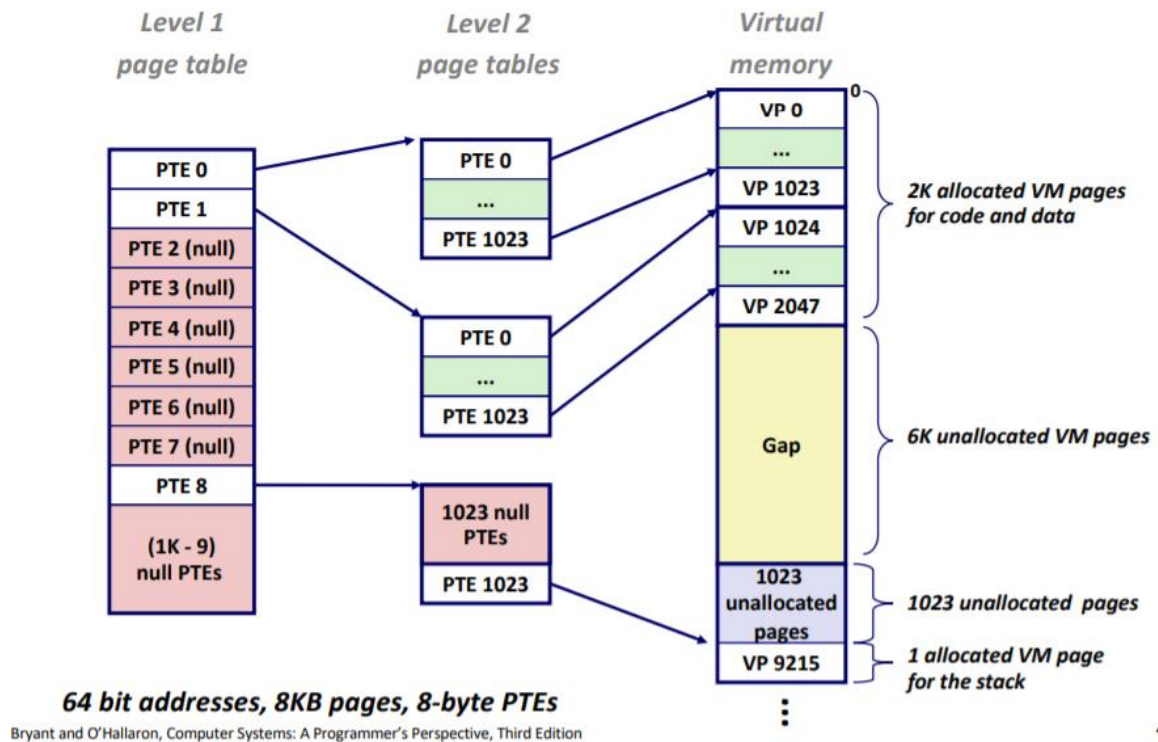
[그림 13] 주소 변환 과정

페이징의 주소 변환 과정은 가상 주소를 물리 주소로 변환하는 작업으로, 가상 주소는 페이지 번호(Page Number)와 페이지 내 오프셋(Offset)으로 나뉜다. 프로세스가 특정 가상 주소에 접근하면, 메모리 관리 장치(MMU, Memory Management Unit)는 페이지 테이블을 참조하여 해당 페이지 번호가 매핑된 물리 메모리의 프레임 번호(Frame Number)를 찾는다. 변환된 프레임 번호에 오프셋을 더해 최종 물리 주소가 결정된다. 주소 변환 과정은 다음과 같이 정형화 하여 표현할 수 있다.

$$VA = \langle P, D \rangle \rightarrow PA = \langle F, D \rangle$$

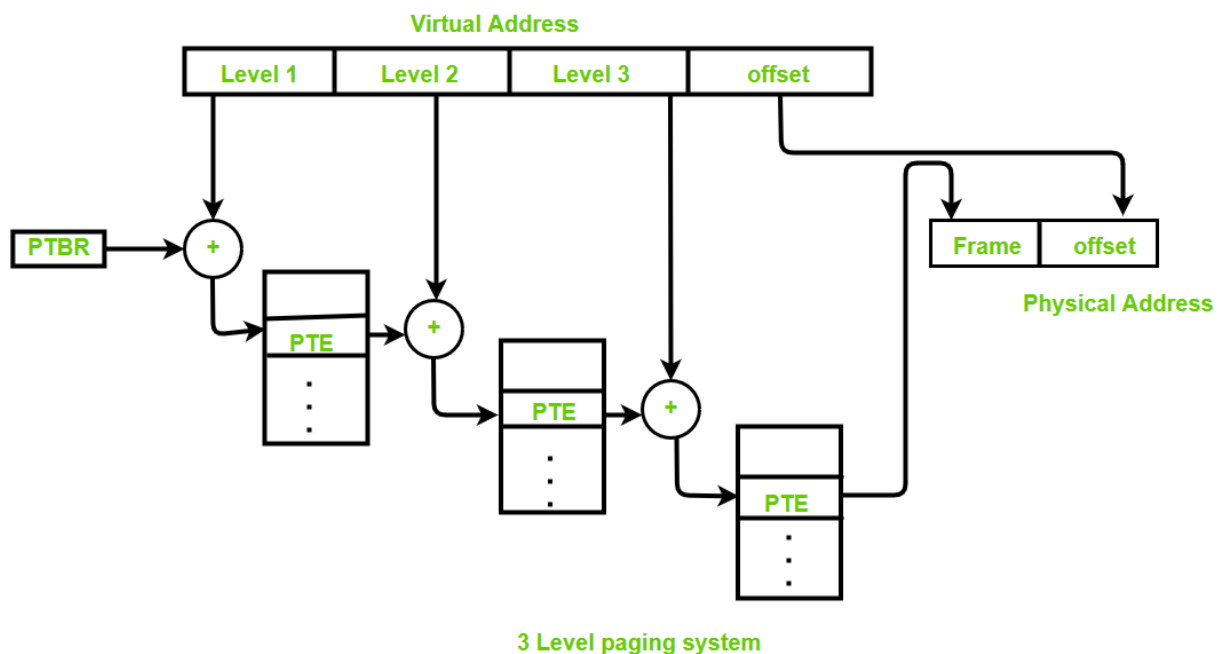
P 에 해당하는 페이지는 현재 가상 주소를 페이지 크기로 나눈 몫에 해당하고, D 에 해당하는 오프셋은 현재 가상 주소를 페이지 크기로 나눈 나머지에 해당한다.

A Two-Level Page Table Hierarchy



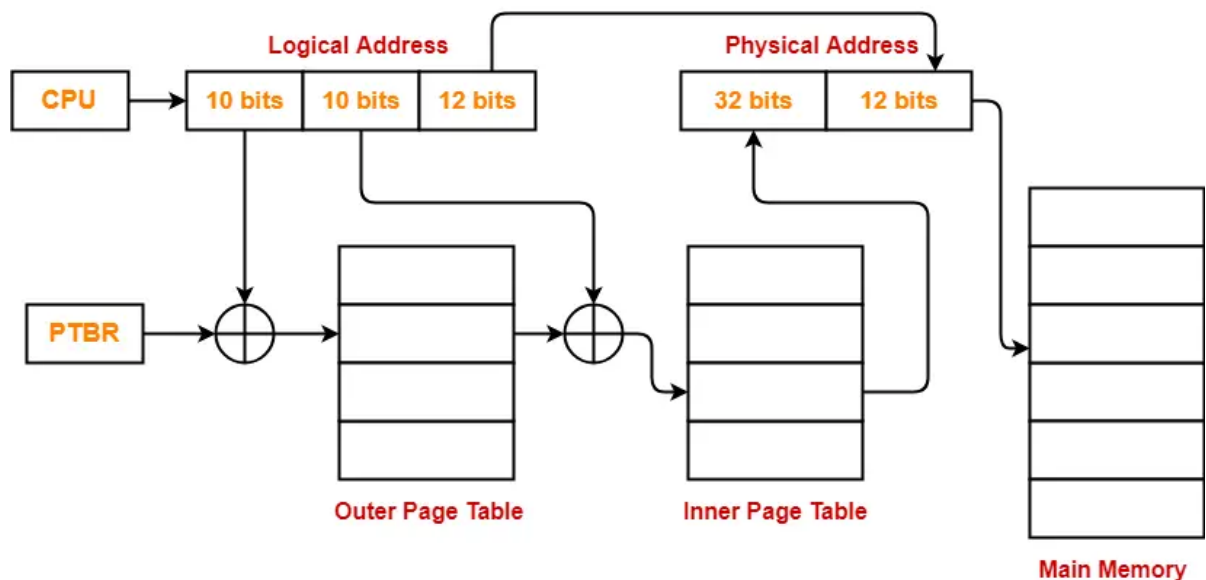
43

[그림 14] 2단계 페이지 테이블 시스템



[그림 15] 3단계 페이지 테이블 시스템

다단계 페이지 테이블(Multilevel Page Table)은 가상 주소 공간이 크고 페이지 테이블이 거대해지는 문제를 해결하기 위해 페이지 테이블을 계층 구조로 나눈 방식이다. 가상 주소를 상위 비트와 하위 비트로 나누어, 상위 비트를 통해 상위 페이지 테이블에서 하위 페이지 테이블의 위치를 찾고, 하위 비트를 이용해 실제 데이터가 저장된 물리 메모리의 프레임 번호를 찾는다. 이러한 구조는 페이지 테이블 전체를 메모리에 유지하지 않고, 필요한 하위 테이블만 메모리에 로드하여 메모리 사용량을 크게 줄일 수 있다. 다단계 페이지 테이블은 가상 주소 공간이 매우 큰 64 비트 시스템에서 필수적으로 사용되며, 메모리 관리의 효율성을 높이는 중요한 기법으로 활용된다. 필자는 시뮬레이터 버전 1에서는 1 단계 페이지 테이블을, 버전 2에서는 2 단계 페이지 테이블을 활용하여 구현할 계획이다. 버전 1에서는 최대한 간단히 만드는 것이 목적이어서 16bit CPU 구조를 가지만 버전 2에서는 2 단계 페이지 테이블을 구현하기 위해 32bit CPU 구조를 갖도록 설계하였음을 덧붙인다. 따라서 32bit 를 기준으로 2 단계 페이지 테이블에서의 주소 변환에 대해 설명을 이어 나가보겠다.



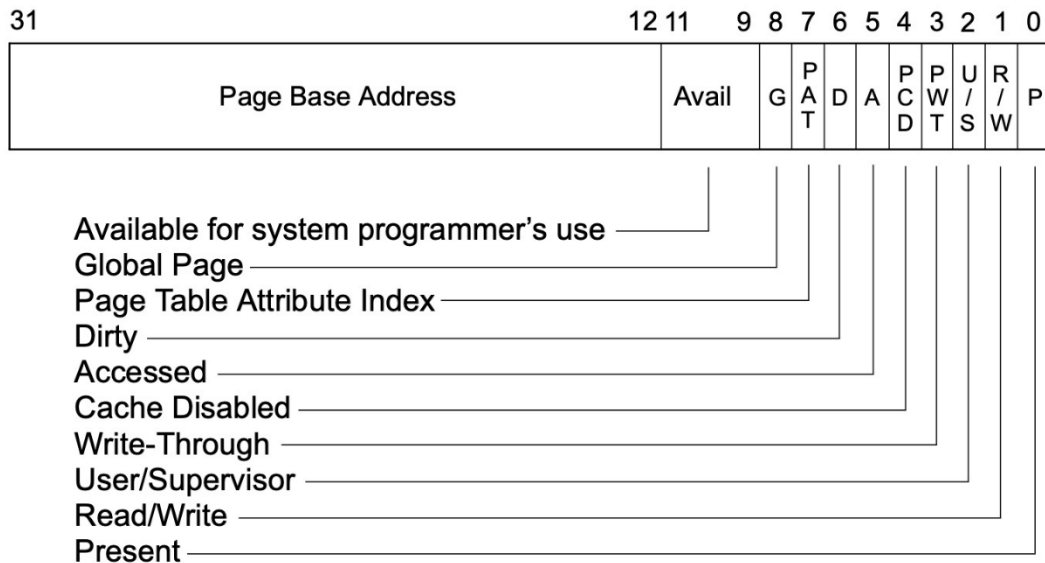
[그림 16] 2 level page table system

2 단계 페이지 테이블에서의 주소 변환 과정은 가상 주소를 상위 10 비트, 중간 10 비트, 하위 12 비트로 나누어 이루어진다. 상위 10 비트는 상위 페이지 디렉토리(Page Directory)의 인덱스로 사용되며, 이를 통해 하위 페이지 테이블의 위치를 찾는다. 중간 10 비트는 하위 페이지 테이블의 인덱스로 사용되어 해당 페이지가 매핑된 물리 메모리의 프레임 번호(Frame Number)를 찾는다. 마지막으로, 하위 12 비트는 페이지 내 오프셋(Offset)으로 사용되어 프레임 내 데이터의 정확한 위치를 결정한다. 이 과정은 페이지 디렉토리와 하위 페이지 테이블을 계층적으로 접근하여 주소 변환에 필요한 메모리를 줄이고 효율적인 가상 메모리 관리를 가능하게 한다.

페이지 테이블 엔트리의 구조

지금까지 다뤘던 내용들을 토대로 볼 때 페이지 테이블은 해당 페이지의 상태 정보를 담고 있는 자료 구조라고 할 수 있다. 그렇다면 이 자료구조를 이루는 구성성분이라 할 수 있는 엔트리는 과연 어떤 구조로 되어있을까?

Page-Table Entry (4-KByte Page)



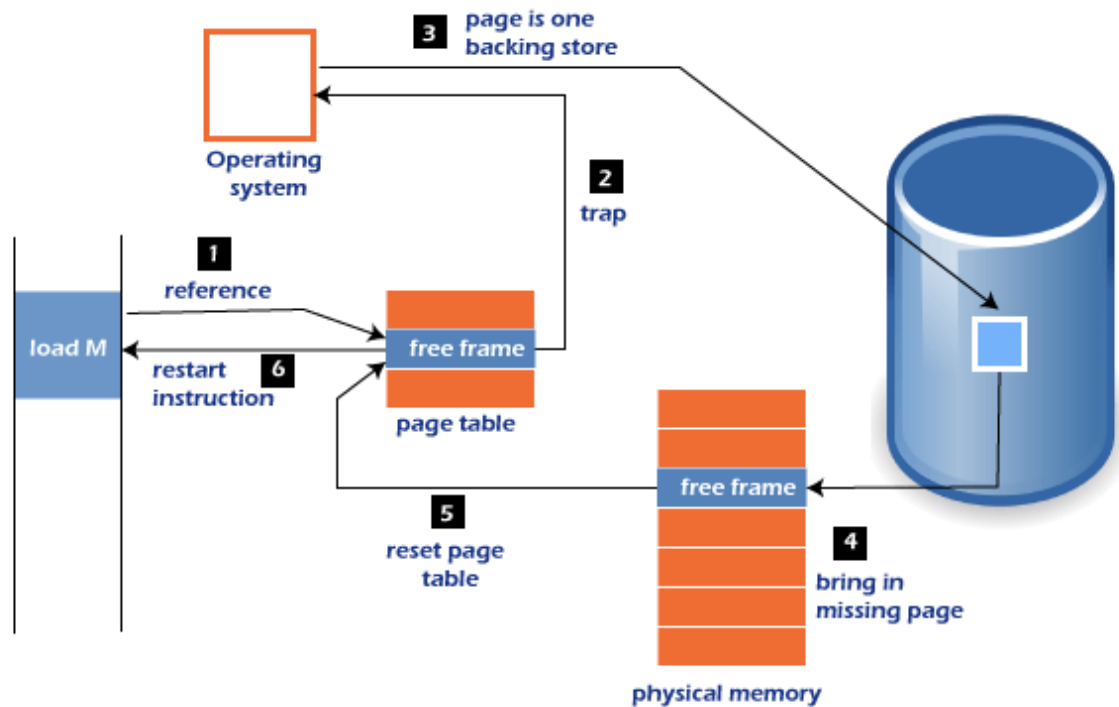
[그림 17] Page-Table Entry (4kb Page)

[그림 17]이 바로 엔트리의 구조이다. 이 경우 32비트를 엔트리로 사용하는 모습이며 플래그 비트와 주소 필드로 구성되어 있으며, 플래그 비트의 경우 다시 여러 항목들로 이루어져 있다. 그 중 눈여겨볼만한 플래그로는 P(present bit)가 있으며 이는 valid bit 즉 유효 비트를 의미한다. 유효비트는 해당 페이지가 스왑 영역에 있는지 메모리에 있는지 표시해주는 아주 중요한 비트이다. 현재 메모리에 해당 페이지가 있을 경우 유효비트는 0, 스왑 영역에 있을 경우 1을 표시한다.

Page Fault

프로세스가 요청하는 가상 주소에 해당하는 프레임을 찾기 위해 페이지 테이블을 살펴보았을 때 만약 대응하는 프레임이 아직 없이 Invalid 상태라면 page fault가 발생하고 스왑 영역에서 필요한 프레임을 가져다 메모리에 저장하고 다시 instruction을 실행한다. 이때 판단 기준이 되는 요소가 바로 엔트리의 유효 비트이다. 필요한 프레임이 메모리에 저장되면 유효비트는 1에서 0으로 변경된다. 유효비트가 0인 경우 엔트리의 주소 필드에는 해당 프레임의 메모리 주소가 기록되지만 유

효비트가 1인 경우 주소 필드에는 스왑 영역내 주소가 저장된다는 점도 중요하다.



[그림 18] Page Fault Mechanism

```
[MMU] Translating virtual address 0xb257
└─ Page Number: 11
└─ Offset: 0x257
[MMU] Address translation failed: Page Fault: Invalid Page
[Page Fault] Process 0, Page 11
[Memory] Allocated frame 1. Remaining free frames: 14
[Page Fault] Resolved: Page 11 mapped to frame 1

[MMU] Translating virtual address 0xb257
└─ Page Number: 11
└─ Offset: 0x257
[Memory Access] Successful
└─ Virtual Address: 0xb257
└─ Physical Address: 0x1257
└─ Frame Number: 1
└─ Offset: 0x257
```

[그림 19] Page Fault 처리 후 해당 주소 다시 접근 (Ver1 시뮬레이터 中)

페이지 폴트가 발생했을 때 상태는 두 가지로 나뉜다. 물리 메모리에 빈 공간이 남아 있는 상태에서 발생했을 경우, 그리고 물리 메모리에 빈 공간이 남아 있지 않고 모든 프레임이 차 있는 경우가 그렇다. 첫 번째 빈 공간이 남아 있는 상태라면 스왑 영역에서 가져온 데이터를 빈 프레임

에 스왑인하면 되겠지만 만약 모든 프레임이 차 있는 경우라면, 여러 개의 이미 존재하는 프레임들 중 하나를 골라 스왑 영역으로 내보내야 한다. 이를 두고 페이지 교체 알고리즘이라고 한다. 또한 교체의 대상이 되는 페이지를 victim page라고 한다.

페이지 교체 알고리즘

페이지 교체 알고리즘(Page Replacement Algorithm)은 가상 메모리 시스템에서 새로운 페이지를 메모리에 적재하기 위해 기존 페이지를 교체해야 할 때, 어떤 페이지를 내보낼지를 결정하는 기법이다. 이는 물리 메모리가 제한된 환경에서 시스템 성능을 유지하기 위해 필수적이며, 효율적인 알고리즘 선택은 페이지 폴트(Page Fault) 발생률과 디스크 I/O를 최소화하는 데 중요하다. 대표적인 알고리즘으로는 FIFO(First-In-First-Out), LRU(Least Recently Used), Optimal Algorithm 등이 있다. FIFO는 가장 먼저 메모리에 들어온 페이지를 교체하는 방식으로 구현이 간단하지만 성능 저하를 초래할 수 있으며, LRU는 가장 오랫동안 사용되지 않은 페이지를 교체하여 참조 지역성(Locality of Reference)을 반영한 효율적인 방식이다. Optimal Algorithm은 이론적으로 가장 적은 페이지 폴트를 보장하지만, 미래의 참조를 예측해야 하므로 현실적으로 구현이 불가능하다.

현대 운영체제는 시스템 성능을 극대화하기 위해 다양한 알고리즘을 변형하거나 혼합하여 사용한다. 예를 들어, Linux는 LRU를 기반으로 한 Clock 알고리즘을 활용하고, Windows는 워킹 셋 개념을 포함한 WSClock 알고리즘을 사용해 메모리 자원의 효율성을 높인다. 이러한 알고리즘은 특정 메모리 접근 패턴에 따라 성능이 달라질 수 있으며, 시스템 환경에 맞춰 최적화된다. 페이지 교체 알고리즘은 메모리 관리에서 중요한 요소로, 운영체제의 안정성과 성능을 보장하는 데 핵심적인 역할을 한다.

표 9-1 페이지 교체 알고리즘의 종류

종류	알고리즘	특징
간단한 알고리즘	무작위	무작위로 대상 페이지를 선정하여 스왑 영역으로 보낸다.
	FIFO	처음 메모리에 올라온 페이지를 스왑 영역으로 보낸다.
이론적 알고리즘	최적	미래의 접근 패턴을 보고 대상 페이지를 선정하여 스왑 영역으로 보낸다.
최적 근접 알고리즘	LRU	시간적으로 멀리 떨어진 페이지를 스왑 영역으로 보낸다.
	LFU	사용 빈도가 적은 페이지를 스왑 영역으로 보낸다.
	NUR	최근에 사용한 적이 없는 페이지를 스왑 영역으로 보낸다.
	FIFO 변형	FIFO 알고리즘을 변형하여 성능을 높인다.

페이지 교체 알고리즘 성능 평가 기준

페이지 교체 알고리즘의 성능은 주로 페이지 폴트의 발생 빈도를 기준으로 평가된다. 페이지 폴트는 필요한 페이지가 물리 메모리에 존재하지 않아 디스크에서 해당 페이지를 로드해야 하는 상황을 의미하며, 디스크 접근은 메모리 접근보다 훨씬 느리기 때문에 페이지 폴트는 시스템 성능에 큰 영향을 미친다. 따라서, 페이지 교체 알고리즘의 성능은 **페이지 폴트율(Page Fault Rate)**을 낮추는 데 얼마나 효과적인지에 따라 평가된다. 또한, 알고리즘이 **참조 지역성(Locality of Reference)**을 얼마나 잘 반영하는지도 중요한 성능 평가 기준이다. 참조 지역성은 프로그램이 메모리를 사용하는 패턴을 나타내며, 최근에 참조된 페이지나 가까운 주소가 다시 참조될 가능성이 높다는 특성을 기반으로 한다. 필자의 경우 시뮬레이터를 만들면서 처음엔 10개의 무작위 주소를 생성하였었는데 메모리 공간에 비해 주소 공간이 컸기에 상당히 높은 확률로 페이지 폴트가 발생하여서 곤란했던 경험을 하였다. 이는 CPU비트 수를 작게 하여 주소 공간을 줄이고 메모리를 키워 저장되는 페이지 수를 늘리고 아예 무작위로 주소를 생성하는 것이 아니라 어느 정도 지역성을 고려하여 인접한 주소를 생성하도록 설계하는 것으로 문제를 해결하였다.

이외에도 알고리즘의 실제 구현 가능성과 오버헤드 역시 성능 평가에서 고려해야 할 요소이다. 예를 들어, Optimal Algorithm은 가장 낮은 페이지 폴트율을 보장하지만 미래의 메모리 접근을 예측해야 하므로 실제 시스템에서는 사용할 수 없다. 반면, LRU(Least Recently Used)는 참조 지역성을 잘 반영하지만, 페이지 참조 시간을 기록하거나 관리하는 데 추가적인 자원이 필요하다. 또한, 알고리즘이 다양한 워크로드에 대해 얼마나 안정적인 성능을 제공하는지도 중요하다. 따라서, 페이지 교체 알고리즘의 성능은 페이지 폴트율뿐만 아니라 구현 복잡성, 오버헤드, 다양한 환경에서의 일관된 성능을 기준으로 종합적으로 평가해야 한다.

설계 및 구현

필자는 이번 프로젝트를 진행하면서 페이지 테이블의 레벨과 페이지 교체 알고리즘이라는 두 가지의 기준을 가지고 분석할 것을 목표로 설계를 시작하였다. 따라서 Ver1 은 1 레벨 페이지 테이블 시스템으로 구현하였고, Ver2 는 2 레벨 페이지 테이블 시스템이면서 FIFO 알고리즘과 LRU 알고리즘을 사용하도록 구현하였다.

버전 1

시스템 설계

- 16bit CPU
- 물리 메모리 : 64KB
- 페이지(프레임) 크기 : 4KB
- 페이지 테이블 엔트리 : 2Byte
- 프로세스 수 : 10 개

주요 기능

- 1 레벨 페이징 시스템
- FIFO 페이지 교체 알고리즘
- Free Frame List 기반 프레임 관리
- MMU 를 통한 주소 변환
- 실시간 성능 모니터링 (터미널 출력)

[Ver 1 함수 및 변수 정리]

VARIABLES	DATA TYPE	DEFINITION
TIME_TICK	100000	SIGALRM의 인터벌 시간 (0.1sec)
RUN_TIME	60	프로그램의 총 실행 시간
MSG_KEY	0x12345 * 2	Message queue에 대한 키 값
MAX_CPROC	10	자식 프로세스 개수
QUEUE_SIZE	10	큐에 들어갈 수 있는 프로세스의 총 개수
PAGE_TABLE_SIZE	16	페이지 테이블의 엔트리 개수
FRAME_SIZE	4096	프레임 크기 ($2^{12} = 4KB$)
NUM_FRAMES	16	프레임 개수 ($64KB/4KB = 16$)
MAX_VIRT_ADDR	65536	16bit 주소 공간 ($2^{16} = 64KB$)
FILE	FILE *	결과값이 작성되는 파일 지정
RUN_TIME	int	실제 실행 시간으로, RUN_TIME과 같아지면 종료됨
TIME_QUANTUM	Int	Round Robin(RR) 방식의 time quantum(tq) 설정
COUNT	Int	실제 실행되는 tq 값으로, time_quantum과 같아지면 종료됨
RUNQ	Queue	cpu 동작을 위한 큐
WAITQ	Queue	io 작업 동작을 위한 큐
CUR_PROC	Process *	현재 시그널에서 동작하고 있는 자식 프로세스의 포인터
MSG_QUEUE	int	자식과 부모 프로세스 간 통신을 위한 메시지 큐 생성 값
PAGE_TABLES	PageTable[]	프로세스 당 페이지 테이블 설정
PMEM	PhysicalMemory	물리 메모리 구조체 지정
FRAME_LIST	free_frame_list	비어있는 프레임 리스트 구조체
MMU	MMU	MMU 구조체
CURRENT_TIME	int	프레임에 할당하기 위한 시간 지정(run_time과 같은 값)
METRICS	PerformanceMetrics	성능 분석을 위한 구조체

Module	Function	Definition
Queue	initQueue	큐 초기화 함수
	isFull	큐가 가득 찼는지 확인하는 함수
	isEmpty	큐가 비어있는지 확인하는 함수
	enqueue	큐에 프로세스 추가하는 함수
	dequeue	큐에서 프로세스 제거하는 함수
	getProcessAtIndex	큐 내의 특정 프로세스의 인덱스 리턴
Paging	print_qstate	큐의 상태 출력 함수
	init_physical_memory	물리 메모리 초기화 함수
	init_free_frame_list	free frame list 초기화 함수
	init_mmu	mmu 초기화 함수
	get_free_frame	free frame을 찾아 가져오는 함수
	return_frameget	frame 반환 함수
	mmu_translate_address	mmu의 주소 변환 함수
	handle_page_fault	page fault 처리 함수
Main	handle_memory_access	mmu의 valid bit에 따른 메모리 접근 처리 함수
	print_memory_status	메모리 상태 출력 함수
	print_performance_metrics	성능 지표 출력 함수
	signal_handler	SIGALRM을 받아 큐에 들어간 자식 프로세스를 처리하는 함수
	waitq_burst	waitq의 I/O burst 처리 함수

[Ver 1 구현 상세]

버전 1의 경우 버전 2를 만들기 전 간단히 먼저 시뮬레이션을 돌려보는 것을 목표로 하여 적은 용량의 메모리와 FIFO 방식의 페이지 교체 알고리즘을 적용하여 구현하였다.

<메모리 관리 파트>

```
// Free Frame List 노드
typedef struct frame_node {
    int frame_number;
    struct frame_node* next;
} frame_node;

// Free Frame List
typedef struct {
    frame_node* front;
    frame_node* rear;
    int count;
} free_frame_list;
```

```
#define FRAME_SIZE 4096 // 4KB
#define NUM_FRAMES 16 // 64KB / 4KB = 16

// 물리 메모리 관리 구조체
typedef struct {
    char memory[NUM_FRAMES][FRAME_SIZE]; // 실제 물리 메모리
    frame_info frames[NUM_FRAMES]; // 각 프레임의 정보
} PhysicalMemory;
```

[그림 21] 물리 메모리 구조

[그림 22] Free Frame List 관리

프레임 16개 중 1~15까지는 사용자 영역으로서 연결리스트로 구성하고 0번은 커널용으로 구성하였다. 또한 free_frame_list의 count로 가용 프레임 수를 추적하도록 설계하였다.

<페이징 시스템>

```
// 페이지 테이블 엔트리 구조체
typedef struct page_table_entry {
    unsigned int frame_number:4; // 16개 프레임
    unsigned int valid_bit:1; // 유효 비트
    unsigned int dirty_bit:1; // 변경 비트
    unsigned int referenced_bit:1; // 참조 비트
    unsigned int protection_bits:3; // 접근 권한
} page_table_entry;

// 페이지 테이블 구조체
typedef struct {
    page_table_entry entries[PAGE_TABLE_SIZE];
} PageTable;
```

[그림 23] 페이지 테이블 엔트리

프로세스당 16개의 페이지 테이블 엔트리를 가지고 있으며 각 엔트리는 2바이트로 구성된다. 2바

이트 중 10비트만 사용하고 있으며 낭비되는 6비트를 개선할 경우 더 효율적인 시스템을 설계할 수 있을 듯 싶지만 버전1이므로 넘어가도록 하자.

```
// MMU의 주소 변환
int mmu_translate_address(unsigned int virtual_addr) {
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);

    metrics.total_memory_accesses++;
    metrics.process_stats[mmu.ptbr].memory_accesses++;
}
```

```
// 페이지 폴트 처리
void handle_page_fault(int process_idx, unsigned int page_number) {
    metrics.page_faults++;
    metrics.process_stats[process_idx].page_faults++;

    printf("[Page Fault] Process %d, Page %d\n", process_idx, page_number);
}
```

```
// 자식 프로세스
else if (pid == 0) { // 자식 프로세스
    // 초기 프로세스 정보 전송
    struct msgbuf init_msg;
    init_msg.mtype = i + 1;
    init_msg.idx = i;
    init_msg.pid = getpid();
    init_msg.cpu_burst = cpu_burst[i];
    init_msg.io_burst = io_burst[i];

    if (msgsnd(msg_queue, &init_msg, sizeof(init_msg) - sizeof(long), 0) == -1) {
        perror("msgsnd failed");
        exit(1);
    }

    printf("[Child] Process %d (PID: %d) created\n", i, getpid());

    // 실행 중 메모리 접근 요청
    while(1) {
        kill(getpid(), SIGSTOP);

        // 새로운 10개의 가상 주소 생성 및 전송
        struct mem_msgbuf mem_msg;
        mem_msg.mtype = i + 1;
        mem_msg.pid = getpid();
        mem_msg.process_index = i;

        for(int j = 0; j < 10; j++) {
            mem_msg.virtual_addresses[j] = rand() & 0xFFFF;
        }

        if (msgsnd(msg_queue, &mem_msg, sizeof(mem_msg) - sizeof(long), 0) == -1) {
            perror("Memory access request failed");
        }
    }
    exit(0);
}
```

[그림 24] 자식프로세스의 페이지 접근 요청

<실행 결과>

```
Time Quantum: 30
[Memory] Physical memory initialized
[Memory] Free Frame List initialized with 15 frames
[MMU] MMU initialized
[System] Removed old message queue
[System] Created new message queue: 9
[Parent] Received initial info from Process 0
[Child] Process 0 (PID: 6716) created
  PID: 6716
  CPU Burst: 12
  I/O Burst: 33
[Parent] Process 0 added to run queue
[Parent] Received initial info from Process 1
[Child] Process 1 (PID: 6717) created
  PID: 6717
  CPU Burst: 27
  I/O Burst: 110
[Parent] Process 1 added to run queue
[Parent] Received initial info from Process 2
[Child] Process 2 (PID: 6718) created
  PID: 6718
  CPU Burst: 44
  I/O Burst: 33
[Parent] Process 2 added to run queue
[Parent] Received initial info from Process 3
[Child] Process 3 (PID: 6719) created
  PID: 6719
  CPU Burst: 50
  I/O Burst: 16
[Parent] Process 3 added to run queue
[Parent] Received initial info from Process 4
[Child] Process 4 (PID: 6720) created
  PID: 6720
  CPU Burst: 33
  I/O Burst: 131
```

[그림 25] 자식 프로세스 생성

```
[Initial State] All processes created

System Status at Time 6
RUN QUEUE | P0[CPU:12] → P1[CPU:27] →
WAIT QUEUE | Empty

===== Memory Status =====
Frame 0: Free
Frame 1: Free
Frame 2: Free
Frame 3: Free
Frame 4: Free
Frame 5: Free
Frame 6: Free
Frame 7: Free
Frame 8: Free
Frame 9: Free
Frame 10: Free
Frame 11: Free
Frame 12: Free
Frame 13: Free
Frame 14: Free
Frame 15: Free
Free Frames: 15
=====
```

[그림 26] 메모리 초기화

```
System Status at Time 1
RUN QUEUE | P0[CPU:12] → P1[CPU:27] → P2[CPU:44] → P3[CPU:50]
WAIT QUEUE | Empty

[Timer] Time quantum: 30, Count: 29

[Memory] Processing memory access requests for Process 0

[MMU] Translating virtual address 0xb257
  Page Number: 11
  Offset: 0x257
[MMU] Address translation failed: Page Fault: Invalid Page
[Page Fault] Process 0, Page 11
[Memory] Allocated frame 1. Remaining free frames: 14
[Page Fault] Resolved: Page 11 mapped to frame 1

[MMU] Translating virtual address 0xb257
  Page Number: 11
  Offset: 0x257
[Memory Access] Successful
  Virtual Address: 0xb257
  Physical Address: 0x1257
  Frame Number: 1
  Offset: 0x257

[MMU] Translating virtual address 0x35ec
  Page Number: 3
  Offset: 0x5ec
[MMU] Address translation failed: Page Fault: Invalid Page
[Page Fault] Process 0, Page 3
[Memory] Allocated frame 2. Remaining free frames: 13
[Page Fault] Resolved: Page 3 mapped to frame 2

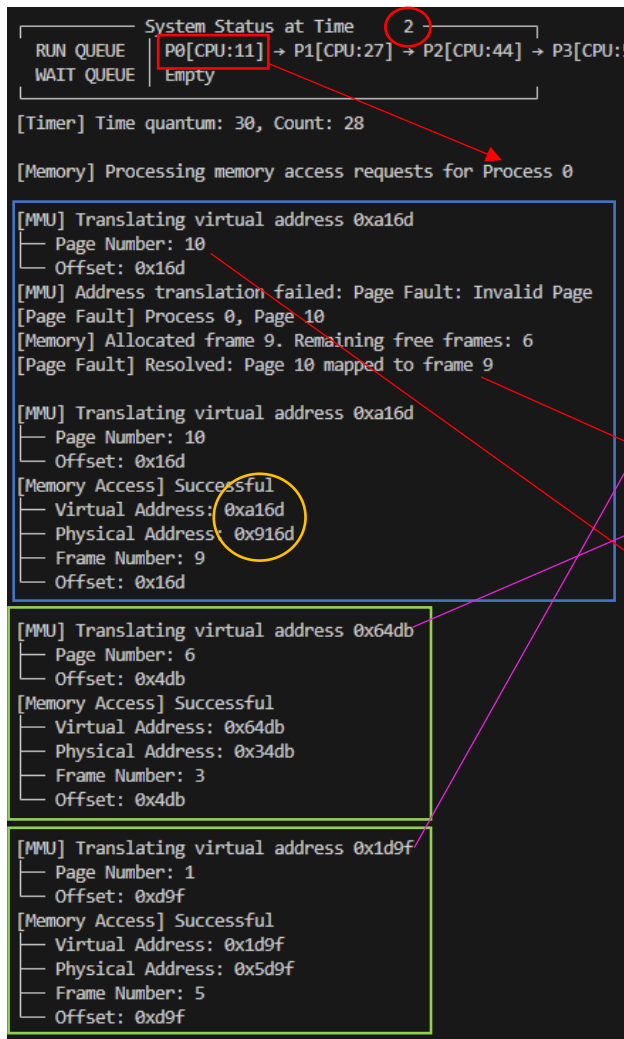
[MMU] Translating virtual address 0x35ec
  Page Number: 3
  Offset: 0x5ec
[Memory Access] Successful
  Virtual Address: 0x35ec
  Physical Address: 0x25ec
  Frame Number: 2
  Offset: 0x5ec
```

[그림 27] 스케줄링 시뮬레이션 시작_0번 프로세스

0번 프로세스 진행중 : 페이지 폴트 처리 후 재 접근

[페이지->프레임] :

[11->1],[3->2],[6->3],[8->4],[1->5],[13->6],[5->7],[9->8]...



페이지	프레임
0	Invalid
1	5
2	Invalid
3	2
4	Invalid
5	7
6	3
7	Invalid
8	4
9	8
10	Invalid
11	1
12	Invalid
13	6
14	Invalid
15	11

← 커널영역으로 사용

← 9

[그림 29] 생성된 페이지 테이블

[그림 28] 0번 프로세스 진행 중

[그림 28]은 [그림 27]에 이어 바로 다음 상황인데, 이전 단계에서 만들어진 페이지 테이블을 토대로 6번 페이지는 3번 프레임에서, 1번 페이지는 5번 프레임에서 찾을 수 있었던 반면 10번 페이지는 아직 데이터가 없어 페이지 폴트 발생 후 요구 페이지징 정책을 통해 원하는 페이지를 메모리상에 가져온 것을 확인할 수 있다.

가상메모리주소



16bit = 4bit + 12bit

a 16d
↓ ↓
페이지 오프셋

페이지	프레임
0	Invalid
1	5
2	Invalid
3	2
4	Invalid
5	7
6	3
7	Invalid
8	4
9	8
10	Invalid
11	1
12	Invalid
13	6
14	Invalid
15	11

← 커널영역으로 사용

물리 메모리주소



16bit = 4bit + 12bit

9 16d
↓ ↓
프레임 오프셋
(동일)

[그림 30] 가져오기 정책 _ 요구 페이징

60초 실행 후 결과

메모리 상태와 성능 평가용 통계 출력

```
===== Memory Status =====
Frame 0: Free
Frame 1: Used by Process 9, Page 6 (Time: 586)
Frame 2: Used by Process 9, Page 12 (Time: 586)
Frame 3: Used by Process 9, Page 11 (Time: 587)
Frame 4: Used by Process 9, Page 1 (Time: 588)
Frame 5: Used by Process 9, Page 3 (Time: 588)
Frame 6: Used by Process 9, Page 10 (Time: 590)
Frame 7: Used by Process 9, Page 4 (Time: 592)
Frame 8: Used by Process 9, Page 14 (Time: 596)
Frame 9: Used by Process 9, Page 5 (Time: 597)
Frame 10: Used by Process 9, Page 0 (Time: 599)
Frame 11: Used by Process 9, Page 13 (Time: 595)
Frame 12: Used by Process 9, Page 15 (Time: 595)
Frame 13: Used by Process 9, Page 7 (Time: 585)
Frame 14: Used by Process 9, Page 2 (Time: 586)
Frame 15: Used by Process 9, Page 8 (Time: 586)
Free Frames: 0
=====
```

[그림 31] 메모리 상태 _ 0번 프레임은 커널영역

```
[System] Simulation finished after 600 ticks

===== Performance Metrics =====
Total Memory Accesses: 6741
Total Page Faults: 741
Page Fault Rate: 10.99%
Page Replacements: 726
Average Address Translation Time: 15338.88 ns

Per-Process Statistics:
Process 0:
├─ Memory Accesses: 583
├─ Page Faults: 73
└─ Fault Rate: 12.52%
Process 1:
├─ Memory Accesses: 1118
├─ Page Faults: 118
└─ Fault Rate: 10.55%
Process 2:
├─ Memory Accesses: 1251
├─ Page Faults: 131
└─ Fault Rate: 10.47%
Process 3:
├─ Memory Accesses: 613
├─ Page Faults: 73
└─ Fault Rate: 11.91%
Process 4:
├─ Memory Accesses: 864
├─ Page Faults: 84
└─ Fault Rate: 9.72%
Process 5:
├─ Memory Accesses: 325
├─ Page Faults: 45
└─ Fault Rate: 13.85%
Process 6:
├─ Memory Accesses: 892
├─ Page Faults: 82
└─ Fault Rate: 9.19%
Process 7:
├─ Memory Accesses: 34
├─ Page Faults: 14
└─ Fault Rate: 41.18%
Process 8:
├─ Memory Accesses: 502
├─ Page Faults: 52
└─ Fault Rate: 10.36%
Process 9:
├─ Memory Accesses: 559
├─ Page Faults: 69
└─ Fault Rate: 12.34%
=====
```

[그림 32] 통계 출력

버전 2

시스템 설계

✓ 시스템 구조

- 부모 프로세스 : 커널/스케줄러 역할
- 자식 프로세스 : 10개의 사용자 프로세스
- 프로세스 간 통신 : 메시지 큐 사용
- Round Robin 스케줄링 기법 사용

✓ 메모리 관리

■ 물리 메모리

- ◆ 물리 메모리 크기 : 44MB(4KB 프레임 * 일단11264개) → 변경 가능
 - 프레임 개수가 페이지 테이블 필요량과 스와핑 발생에 영향을 주어 10300개, 10400개, 10752개, 11264개, 16384개의 여러 시도를 하였다.

- ◆ 프레임 크기 : 4KB

- ◆ 페이지 테이블에 할당되는 메모리 : 40MB(10240프레임)

- ◆ 데이터용 메모리 : 4MB(1024 프레임) → 변경 가능

■ 가상 메모리

- ◆ 프로세스당 1MB 가상 주소 공간

- 32bit CPU 시스템을 상정하였으나 시뮬레이션 특성상 굳이 4GB의 주소공간을 모두 사용하지 않아도 페이징 시스템의 동작을 구현하는데 문제가 없다고 판단하여 1MB로 제한함

- ◆ 스왑 공간 4MB

- 프로세스당 1MB의 가상 주소 공간을 가지기 때문에 총 10개의 프로세스는 10MB의 가상 주소 공간을 필요로 한다고 쳤을 때 40퍼센트 정도를 스왑 공간으로 설정하는 것이 스와핑 현상을 관찰하기에 부족하지 않다고 판단하였음

✓ 주소 변환 _ 2레벨 페이징 시스템

- 32비트 가상 주소 -> 32비트 물리 주소
 - ◆ 상위 10 비트 : 디렉토리 인덱스
 - ◆ 중간 10 비트 : 페이지 테이블 인덱스
 - ◆ 하위 12 비트 : 오프셋

✓ 페이지 관리

- 요구 페이징
- 페이지 폴트 처리
- 페이지 교체 알고리즘 : FIFO와 LRU 중 사용자가 명령줄 인수로 선택 가능
- Ver1에서 프리 프레임 리스트를 사용한 반면 Ver2에서는 프레임배열을 순차 검색하여 사용하지 않는 프레임을 사용

✓ 스왑 관리

- 스왑 영역할당/해제
- 페이지 스왑인/아웃
- 스왑 상태 추적

✓ 프로세스 관리 (←저번 과제 코드 재활용)

- 프로세스별 독립적인 페이지 테이블
- Run Queue와 Wait Queue관리
- I/O 버스트 처리

✓ 모니터링 및 통계

- 실시간 메모리 상태 출력
- 페이지 폴트 통계
- 스왑 사용량 추적
- 프레임 사용 현황
- 평균 메모리 접근 시간 측정

✓ 페이지 교환 알고리즘

■ FIFO(First-In-First-Out)

◆ 가장 기본적인 알고리즘 : 가장 오래된 페이지 교체

■ LRU(Least Recently Used)

◆ 가장 오래전 접근한 페이지를 교체

✓ 주요 데이터 구조

■ 페이지 디렉토리

■ 페이지 테이블

■ 프레임 정보 배열

■ 스왑 영역

■ 프로세스 큐

✓ 성능 최적화

■ 지역성을 고려한 메모리 접근 패턴

◆ 이 패턴은 스와핑 현상에 영향을 주기에 아예 랜덤하게 주소를 생성한 방법과

◆ 지역성을 고려하여 어느정도 인접한 주소를 생성하게 하는 방법을 사용해왔다.

■ 효율적인 페이지 테이블 관리

■ 메모리 상태 주기적 모니터링

Ver2_ver1 이외, 추가/변경된 변수 및 함수

Variables	Data Type	Definition
RUN_TIME	600	프로그램의 총 실행 시간
MSG_KEY	0x12345 * 3	Message queue에 대한 키 값
MAX_VIRT_ADDR	0xFFFFFFFF	32bit 주소 공간 ($2^{32} = 4GB$)
PAGE_SIZE	4096	페이지 사이즈 ($2^{12} = 4KB$)
FRAME_SIZE	4096	프레임 사이즈 (페이지 사이즈와 동일)
NUM_FRAMES	10752	물리 메모리에서 할당 가능한 프레임의 개수
MAX_VIRT_PAGES	1048576	가상 메모리에서 사용할 수 있는 페이지의 수
SWAP_SIZE	PAGE_SIZE * 1024	4MB의 스왑 공간
PAGE_DIR_SIZE	1024	첫 번째 페이징 비트로, 페이지 디렉토리 사이즈 (2^{10})
PAGE_TABLE_SIZE	1024	두 번째 페이징의 페이지 테이블 사이즈 (2^{10})
OFFSET_BITS	12	offset 비트 (12bit)
mmu32	MMU32[]	32bit의 MMU 구조체
frames	FrameInfo[]	프레임 구조체
swap_space	SwapEntry *	스왑 공간의 엔트리 구조체 포인터
global_clock	0 (uint64_t)	lru 알고리즘에서 실제 시간을 count하기 위한 변수
process_page_directories	PageDirectory *[]	첫 번째 페이징을 위한 페이지 디렉토리 구조체 배열
fifo_next_victim	1 (static int)	FIFO 알고리즘에서의 다음 프레임의 사용되지 않은 인덱스 (0번째 프레임 제외)
current_algorithm	ReplacementAlgorithm	현재 사용하고 있는 알고리즘 (0: FIFO, 1: LRU)
stats	VMStats	메모리 통계를 위한 변수 구조체

Module	Function	Definition
Paginng	get_frame_address	메모리에서 프레임 주소 가져와 리턴하는 함수
	find_fifo_victim	FIFO 알고리즘을 사용하여 사용하지 않는 프레임 찾는 함수
	find_lru_victim	LRU 알고리즘을 사용하여 사용하지 않는 프레임 찾는 함수
	init_paging_system	페이징에 사용되는 변수(mmu32, frame, swap_space) 초기화 함수
	init_process_memory	첫 번째 페이징 할 때 사용되는 페이지 디렉토리 초기화 함수
	handle_memory_request	페이징 기법 실제로 수행하는 함수
Swap	find_free_swap_slot	비어있는 스왑 공간 찾는 함수
	find_in_swap	swap in 할 수 있는 공간 찾는 함수
	swap_out_page	paging 알고리즘을 사용하여 swap out 할 프레임 리턴 함수
	swap_in_page	paging 알고리즘을 사용하여 swap in 할 프레임 리턴 함수
	print_memory_status	프레임 및 스왑에 대한 정보 출력 함수
Status	update_stats	메모리 통계를 위한 시간 업데이트 함수
	update_memory_access	update_stats() 함수 호출 함수
	update_page_fault	page fault 개수 카운트 함수
	update_page_replacement	page 교체 카운트 함수
	update_swap_stats	swap in/out 카운트 함수
	print_vm_stats	메모리 전체 통계 출력 함수

[Ver 2 구현 상세]

```
// System Parameters
#define PAGE_SIZE 4096 // 4KB
#define FRAME_SIZE 4096 // 4KB
#define NUM_FRAMES 11264//10300//10400//10752//11264//16384 // 6
#define MAX_VIRT_PAGES 1048576 // 4GB / 4KB
#define SWAP_SIZE (PAGE_SIZE * 1024) // 4MB swap space

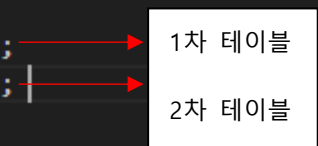
// Page Table Constants
#define PAGE_DIR_SIZE 1024 // 10 bits
#define PAGE_TABLE_SIZE 1024 // 10 bits
#define OFFSET_BITS 12 // 12 bits for 4KB pages

// Two-Level Page Table Entry
typedef struct {
    uint32_t frame_number:14; // Physical frame number
    uint32_t present:1; // Present in memory == valid bit
    uint32_t dirty:1; // Modified
    uint32_t accessed:1; // Recently accessed (for LRU)
    uint32_t user:1; // User/supervisor mode
    uint32_t writable:1; // Read/write permissions
    uint32_t reserved:13; // Reserved bits
} page_table_entry;

// Page Directory Entry
typedef struct {
    uint32_t pt_frame:4; // Frame containing page table
    uint32_t present:1; // Page table present in memory
    uint32_t user:1; // User/supervisor mode
    uint32_t writable:1; // Read/write permissions
    uint32_t reserved:25; // Reserved bits
} page_dir_entry;

// Two-Level Page Table
typedef struct {
    page_dir_entry directory[PAGE_DIR_SIZE];
    page_table_entry *tables[PAGE_DIR_SIZE];
} PageDirectory;

// Swap Space Entry
```



[그림 33] 주요 자료 구조

```

uint32_t translate_address(uint32_t virtual_addr) {
    update_memory_access();
    if (!mmu32.current_pd) {
        mmu32.translation_result.valid = 0;
        strcpy(mmu32.translation_result.error_msg, "No active page directory");
        return 0;
    }

    uint32_t dir_index = (virtual_addr >> 22) & 0x3FF;
    uint32_t page_index = (virtual_addr >> 12) & 0x3FF;
    uint32_t offset = virtual_addr & 0xFFF;

    page_dir_entry *dir_entry = &mmu32.current_pd->directory[dir_index];
    if (!dir_entry->present) {
        mmu32.translation_result.valid = 0;
        strcpy(mmu32.translation_result.error_msg, "Page directory entry not present");
        return 0;
    }

    page_table_entry *page_table = mmu32.current_pd->tables[dir_index];
    page_table_entry *page_entry = &page_table[page_index];

    if (!page_entry->present) {
        mmu32.translation_result.valid = 0;
        strcpy(mmu32.translation_result.error_msg, "Page not present");
        return 0;
    }

    frames[page_entry->frame_number].last_access_time = ++global_clock;
    page_entry->accessed = 1;

    uint32_t physical_addr = (page_entry->frame_number << 12) | offset;
    mmu32.translation_result.physical_addr = physical_addr;
    mmu32.translation_result.valid = 1;

    return physical_addr;
}

```

[그림 34] 주소 변환 함수

```

int find_fifo_victim(void) {
    int initial_position = fifo_next_victim;

    do {
        // 페이지 테이블로 사용되지 않는 프레임 찾기
        if (!frames[fifo_next_victim].is_page_table) {
            int victim = fifo_next_victim;
            fifo_next_victim = (fifo_next_victim + 1) % NUM_FRAMES;
            if (fifo_next_victim == 0) fifo_next_victim = 1; // 0번 프레임 건너뛰기
            return victim;
        }
        fifo_next_victim = (fifo_next_victim + 1) % NUM_FRAMES;
        if (fifo_next_victim == 0) fifo_next_victim = 1; // 0번 프레임 건너뛰기
    } while (fifo_next_victim != initial_position);

    return -1; // 사용 가능한 프레임을 찾지 못함
}

int find_lru_victim(void) {
    uint64_t oldest_time = global_clock;
    int victim_frame = -1;

    for (int i = 1; i < NUM_FRAMES; i++) {
        if (!frames[i].is_page_table && frames[i].last_access_time < oldest_time) {
            oldest_time = frames[i].last_access_time;
            victim_frame = i;
        }
    }

    return victim_frame;
}

```

[그림 35] 페이지 교체 시스템

<실행 결과>

```
Time Quantum: 30
Selected Algorithm: LRU
[VMM] Initialized paging system
[System] Removed old message queue
[System] Created new message queue: 5
[Child] Process 0 (PID: 74882) created
[Parent] Process 0 initialized and added to run queue
[Child] Process 1 (PID: 74883) created
[Parent] Process 1 initialized and added to run queue
[Child] Process 2 (PID: 74884) created
[Parent] Process 2 initialized and added to run queue
[Child] Process 3 (PID: 74886) created
[Parent] Process 3 initialized and added to run queue
[Child] Process 4 (PID: 74889) created
[Parent] Process 4 initialized and added to run queue
[Child] Process 5 (PID: 74893) created
[Parent] Process 5 initialized and added to run queue
[Child] Process 6 (PID: 74894) created
[Parent] Process 6 initialized and added to run queue
[Child] Process 7 (PID: 74895) created
[Parent] Process 7 initialized and added to run queue
[Child] Process 8 (PID: 74896) created
[Parent] Process 8 initialized and added to run queue
[Child] Process 9 (PID: 74897) created
[Parent] Process 9 initialized and added to run queue
```

[그림 36] 시뮬레이션 시작 시 _ 프로세스 생성

```
[MMU] Translating virtual address 0x001e9ace
Page Number: 489
Offset: 0xace
[Memory Access] Successful
Virtual Address: 0x001e9ace
Physical Address: 0x0282bace
Frame Number: 10283
Offset: 0xace

[MMU] Translating virtual address 0x001ed928
Page Number: 493
Offset: 0x928
Page fault occurred: Page not present
[Page Fault] Process 1, Page 493
[Page Fault] Resolved: Page 493 mapped to frame 10292

[MMU] Translating virtual address 0x001ed928
Page Number: 493
Offset: 0x928
[Memory Access] Successful
Virtual Address: 0x001ed928
Physical Address: 0x02834928
Frame Number: 10292
Offset: 0x928
```

[그림 37] 시뮬레이션 동작 _ 10개의 가상 주소에 접근

```

Frame 11245: Process 9, Page 2313, Last Access: 13704
Frame 11246: Process 9, Page 2374, Last Access: 13707
Frame 11247: Process 9, Page 2523, Last Access: 13713
Frame 11248: Process 9, Page 2506, Last Access: 14520
Frame 11249: Process 9, Page 2495, Last Access: 13727
Frame 11250: Process 9, Page 2476, Last Access: 13732
Frame 11251: Process 9, Page 2331, Last Access: 14343
Frame 11252: Process 9, Page 2325, Last Access: 15423
Frame 11253: Process 9, Page 2501, Last Access: 13742
Frame 11254: Process 9, Page 2455, Last Access: 15461
Frame 11255: Process 2, Page 645, Last Access: 15650
Frame 11256: Process 2, Page 548, Last Access: 13757
Frame 11257: Process 2, Page 597, Last Access: 15664
Frame 11258: Process 2, Page 592, Last Access: 15634
Frame 11259: Process 2, Page 637, Last Access: 14562
Frame 11260: Process 2, Page 535, Last Access: 14579
Frame 11261: Process 2, Page 745, Last Access: 13777
Frame 11262: Process 2, Page 721, Last Access: 13781
Frame 11263: Process 2, Page 687, Last Access: 15680

```

Swap Space Usage: 555/4194304 pages

=====

[System] Simulation finished

===== Virtual Memory Statistics =====

Page Replacement Algorithm: FIFO

Total Memory Accesses: 4971

Page Faults: 1791 (36.03%)

Page Replacements: 768

Swap-ins: 213

Swap-outs: 768

Average Memory Access Time: 0.13 ms

Memory Usage:

Used Frames: 11263/11264 (100.0%)

Page Table Frames: 10240 (90.9%)

=====

[System] Cleanup completed

```

Frame 11245: Process 5, Page 1510, Last Access: 16100
Frame 11246: Process 5, Page 1398, Last Access: 16229
Frame 11247: Process 5, Page 1468, Last Access: 16232
Frame 11248: Process 5, Page 1438, Last Access: 16115
Frame 11249: Process 5, Page 1294, Last Access: 16117
Frame 11250: Process 5, Page 1360, Last Access: 16158
Frame 11251: Process 5, Page 1449, Last Access: 16128
Frame 11252: Process 8, Page 2169, Last Access: 15679
Frame 11253: Process 5, Page 1491, Last Access: 16146
Frame 11254: Process 5, Page 1378, Last Access: 16148
Frame 11255: Process 5, Page 1452, Last Access: 16151
Frame 11256: Process 5, Page 1516, Last Access: 16203
Frame 11257: Process 9, Page 2544, Last Access: 15052
Frame 11258: Process 5, Page 1326, Last Access: 16157
Frame 11259: Process 5, Page 1496, Last Access: 16161
Frame 11260: Process 5, Page 1461, Last Access: 16176
Frame 11261: Process 5, Page 1364, Last Access: 16190
Frame 11262: Process 9, Page 2308, Last Access: 16302
Frame 11263: Process 5, Page 1349, Last Access: 16181

```

Swap Space Usage: 558/4194304 pages

=====

[System] Simulation finished

===== Virtual Memory Statistics =====

Page Replacement Algorithm: LRU

Total Memory Accesses: 5185

Page Faults: 1835 (35.39%)

Page Replacements: 812

Swap-ins: 254

Swap-outs: 812

Average Memory Access Time: 0.14 ms

Memory Usage:

Used Frames: 11263/11264 (100.0%)

Page Table Frames: 10240 (90.9%)

=====

[System] Cleanup completed

[그림38] 실행 종료 좌: FIFO 방식 우: LRU 방식 **각 프로그램 1분씩 진행 **

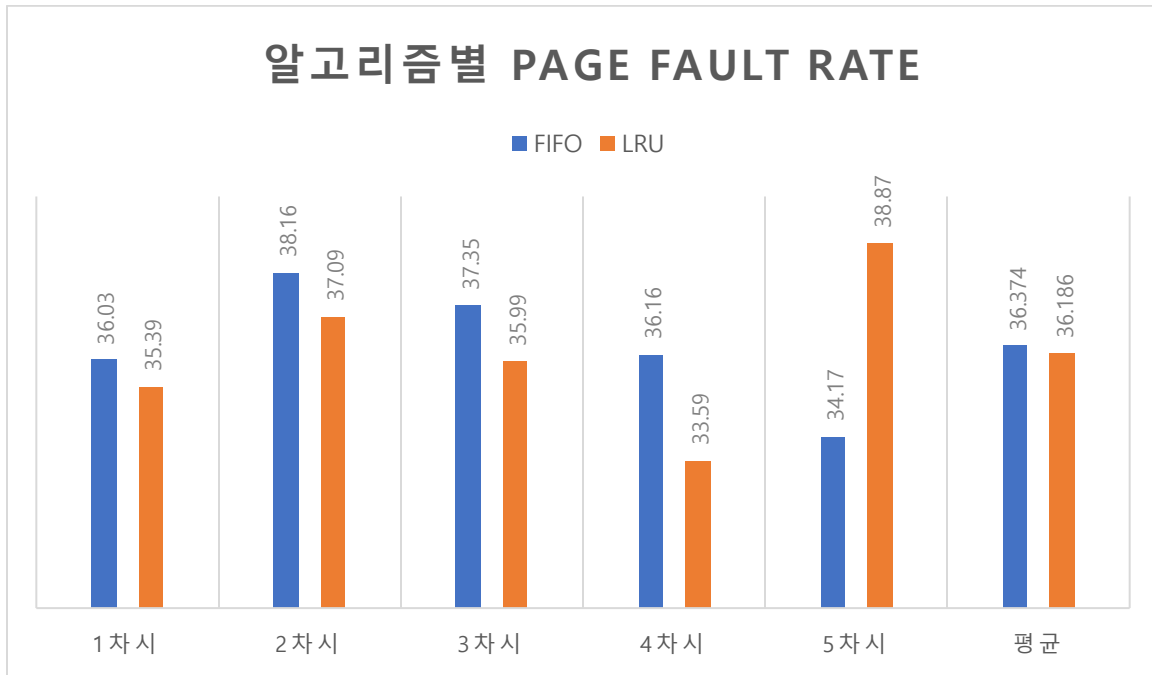
Virtual Memory Statistics	Virtual Memory Statistics
Page Replacement Algorithm: FIFO	Page Replacement Algorithm: FIFO
Total Memory Accesses: 5191	Total Memory Accesses: 4964
Page Faults: 1981 (38.16%)	Page Faults: 1854 (37.35%)
Page Replacements: 958	Page Replacements: 831
Swap-ins: 331	Swap-ins: 175
Swap-outs: 958	Swap-outs: 831
Average Memory Access Time: 0.16 ms	Average Memory Access Time: 0.08 ms
Memory Usage:	Memory Usage:
Used Frames: 11263/11264 (100.0%)	Used Frames: 11263/11264 (100.0%)
Page Table Frames: 10240 (90.9%)	Page Table Frames: 10240 (90.9%)

Virtual Memory Statistics	Virtual Memory Statistics
Page Replacement Algorithm: FIFO	Page Replacement Algorithm: FIFO
Total Memory Accesses: 4809	Total Memory Accesses: 5089
Page Faults: 1739 (36.16%)	Page Faults: 1739 (34.17%)
Page Replacements: 716	Page Replacements: 716
Swap-ins: 197	Swap-ins: 266
Swap-outs: 716	Swap-outs: 716
Average Memory Access Time: 0.00 ms	Average Memory Access Time: 0.01 ms
Memory Usage:	Memory Usage:
Used Frames: 11263/11264 (100.0%)	Used Frames: 11263/11264 (100.0%)
Page Table Frames: 10240 (90.9%)	Page Table Frames: 10240 (90.9%)

[그림 39] 성능 비교를 위해 각 알고리즘별 4번 추가 시행 **각 프로그램 1분씩 진행 **

Virtual Memory Statistics	Virtual Memory Statistics
Page Replacement Algorithm: LRU	Page Replacement Algorithm: LRU
Total Memory Accesses: 5055	Total Memory Accesses: 5077
Page Faults: 1875 (37.09%)	Page Faults: 1827 (35.99%)
Page Replacements: 852	Page Replacements: 804
Swap-ins: 250	Swap-ins: 190
Swap-outs: 852	Swap-outs: 804
Average Memory Access Time: 0.03 ms	Average Memory Access Time: 0.08 ms
Memory Usage:	Memory Usage:
Used Frames: 11263/11264 (100.0%)	Used Frames: 11263/11264 (100.0%)
Page Table Frames: 10240 (90.9%)	Page Table Frames: 10240 (90.9%)

Virtual Memory Statistics	Virtual Memory Statistics
Page Replacement Algorithm: LRU	Page Replacement Algorithm: LRU
Total Memory Accesses: 4894	Total Memory Accesses: 5169
Page Faults: 1644 (33.59%)	Page Faults: 2009 (38.87%)
Page Replacements: 621	Page Replacements: 986
Swap-ins: 193	Swap-ins: 279
Swap-outs: 621	Swap-outs: 986
Average Memory Access Time: 0.03 ms	Average Memory Access Time: 0.11 ms
Memory Usage:	Memory Usage:
Used Frames: 11263/11264 (100.0%)	Used Frames: 11263/11264 (100.0%)
Page Table Frames: 10240 (90.9%)	Page Table Frames: 10240 (90.9%)



[그림 40] 1분 실행 결과 비교

Swap Space Usage: 1535/4194304 pages =====	Swap Space Usage: 1514/4194304 pages =====
[System] Simulation finished	[System] Simulation finished
===== Virtual Memory Statistics =====	===== Virtual Memory Statistics =====
Page Replacement Algorithm: FIFO	Page Replacement Algorithm: LRU
Total Memory Accesses: 83474	Total Memory Accesses: 82017
Page Faults: 28694 (34.37%)	Page Faults: 27757 (33.84%)
Page Replacements: 27671	Page Replacements: 26734
Swap-ins: 26136	Swap-ins: 25220
Swap-outs: 27671	Swap-outs: 26734
Average Memory Access Time: 0.00 ms	Average Memory Access Time: 0.01 ms
Memory Usage:	Memory Usage:
Used Frames: 11263/11264 (100.0%)	Used Frames: 11263/11264 (100.0%)
Page Table Frames: 10240 (90.9%)	Page Table Frames: 10240 (90.9%)
=====	=====
[System] Cleanup completed	[System] Cleanup completed

[그림41] 10분 실행 결과 비교

알고리즘 별 페이지 폴트 발생률을 비교하기 위해 먼저 시뮬레이터를 1분씩 실행한 결과를 비교해 보았다. 그 결과 5번 실행 평균 발생률이 FIFO는 36.174%, LRU는 36.186%로 LRU가 아주 근소한 차이로 더 낮은 발생률을 보이는 것으로 보였다. 그러나 적은 실행 횟수와 아주 근소한 차이 이기에 이러한 결과로부터 LRU가 FIFO보다 더 나은 페이지 정책이라는 결론을 낼 수 없을 것 같다고 판단하였다. 따라서 시뮬레이션 실행시간을 10분으로 늘려서 실행해보았다. 그러나 이번에도

1분의 실행 시간일 때와 비슷한 매우 근소한 차이로 LRU에서의 페이지 폴트 발생률이 낮은 것으로 결과가 나왔다. 이는 차이가 어느정도 벌어질 것이라고 예상했던 것과는 차이가 많이 나는 결과였다. 이러한 결과로 미루어 짐작할 때 시간을 늘리는 것이 근본적인 문제가 아니라는 생각이 들었다. 추측하건데 랜덤한 주소를 10개 생성하는 과정에서의 메모리 접근 패턴이 LRU의 장점인 '최근 사용 이력 활용'이 큰 효과를 낼 수 없는 구조인 것 같다는 생각을 하였다. 현재 필자가 구현한 랜덤 주소 생성의 경우 다음과 같이 고정 주소를 3개 설정하고 10퍼센트의 확률로 접근하게 하여 접근했던 주소를 다시 접근하게끔 설계를 하였었다. 정확히 같은 주소를 다시 접근하도록 함으로서 LRU 방식에 유리하다고 생각하였던 것이다. 나름대로 메모리 접근 패턴에 있어 지역성을 확보하기위한 조치였는데 이것이 LRU방식을 FIFO로부터 차별화하기에는 약한 조치였던 것 같다.

```
// 자주 접근할 고정 주소 3개 설정
uint32_t frequent_addresses[3];
for(int k = 0; k < 3; k++) {
    frequent_addresses[k] = proc_base + (rand() % (1 << 20));
}

while(1) {
    kill(getpid(), SIGSTOP);

    struct mem_msgbuf mem_msg;
    mem_msg.mtype = i + 1;
    mem_msg.pid = getpid();
    mem_msg.process_index = i;

    for(int j = 0; j < 10; j++) {
        if (rand() % 10 < 1) { // 10% 확률로 고정 주소 사용
            mem_msg.virtual_addresses[j] = frequent_addresses[rand() % 3];
        } else {
            uint32_t new_addr = proc_base + (rand() % (1 << 20));
            mem_msg.virtual_addresses[j] = new_addr;
        }
    }

    msgsnd(msg_queue, &mem_msg, sizeof(mem_msg) - sizeof(long), 0);
}
```

[그림 42] 메모리 접근 주소 생성 방식_1차

따라서 이 부분을 강화하기로 결정하고 다음과 같이 방식을 변경하였다. 먼저 메모리 접근 패턴에 있어 공간적 지역성을 강화하기 위해 이전에 10퍼센트의 확률로 고정주소 3개를 생성하였던 것을 30퍼센트 확률로 고정주소 5개로 변경하였다. 또한 시간적 지역성을 추가하여 40퍼센트 확률로 최근 접근한 주소 근처 주소를 접근하도록 설계를 강화하였다. 이는 last_addr변수를 이용해 이전 접근한 위치를 파악하였다. 마지막으로는 메모리 접근 패턴을 만들기 위해 완전 랜덤 접근 확률을 30퍼센트로 대폭 낮추어 제한하였다.

Swap Space Usage: 63/4194304 pages =====	Swap Space Usage: 80/4194304 pages =====
[System] Simulation finished	[System] Simulation finished
===== Virtual Memory Statistics =====	===== Virtual Memory Statistics =====
Page Replacement Algorithm: FIFO	Page Replacement Algorithm: LRU
Total Memory Accesses: 4286	Total Memory Accesses: 4365
Page Faults: 1086 (25.34%)	Page Faults: 1105 (25.32%)
Page Replacements: 63	Page Replacements: 82
Swap-ins: 0	Swap-ins: 2
Swap-outs: 63	Swap-outs: 82
Average Memory Access Time: 0.14 ms	Average Memory Access Time: 0.14 ms
Memory Usage:	Memory Usage:
Used Frames: 11263/11264 (100.0%)	Used Frames: 11263/11264 (100.0%)
Page Table Frames: 10240 (90.9%)	Page Table Frames: 10240 (90.9%)
=====	=====
[System] Cleanup completed	[System] Cleanup completed

[그림 43] 메모리 접근 주소 생성 방식 2차

[그림 43]은 강화한 메모리 접근 패턴으로 실행한 결과이다. 페이지 폴트 발생률 자체는 크게 줄었으나 알고리즘 간의 차이는 여전히 미미했다. 따라서 처음에 추측했던 메모리 접근 패턴은 LRU 알고리즘이 FIFO알고리즘에 비해 차별화되지 못한 근본적인 원인은 아니었던 것 같다는 결론이 내려졌다.

전체적인 구조에 대해 고민하던 중 메모리 자체의 크기가 작아서 그럴 수도 있지 않을까 라는 생각이 들었다. 원래 프레임 개수를 16384개로 페이지 테이블을 저장할 커널 영역으로 사용할 10240개의 프레임을 제외한 약 6천여개의 프레임을 데이터용 프레임으로 사용하려 했다가 프레임 수가 너무 많은 나머지 스와핑 현상이 전혀 발생하지 않아 프레임 개수를 줄인 것이었다. 그런데 이제 보니 스와핑 현상이 일어날 수 있도록 충분히 적은 프레임 개수(현재 1024개)를 사용하다 보니 데이터용 프레임이 너무 적어 어떤 알고리즘을 사용해도 자주 교체할 수 밖에 없는 상황이 발생한 것이었던 것이다. 아무래도 프레임 수가 적으면 알고리즘의 장점을 발휘할 여지가 줄어들 수 밖에 없을 것 같다. 따라서 이번에는 프레임 수를 스와핑 현상이 일어날 수 있도록 하면서 알고리즘간 차별화는 일어날 수 있는 수를 찾아 실행해보려고 한다.


```

===== Virtual Memory Statistics =====
Page Replacement Algorithm: FIFO
Total Memory Accesses: 4549
Page Faults: 1139 (25.04%)
Page Replacements: 0
Swap-ins: 0
Swap-outs: 0
Average Memory Access Time: 0.07 ms

Memory Usage:
Used Frames: 11379/12264 (92.8%)
Page Table Frames: 10240 (83.5%)
=====

```

```

===== Virtual Memory Statistics =====
Page Replacement Algorithm: LRU
Total Memory Accesses: 4495
Page Faults: 1055 (23.47%)
Page Replacements: 0
Swap-ins: 0
Swap-outs: 0
Average Memory Access Time: 0.06 ms

Memory Usage:
Used Frames: 11295/12264 (92.1%)
Page Table Frames: 10240 (83.5%)
=====

```

[그림44] 12264 _ 600 틱 기준

```

===== Virtual Memory Statistics =====
Page Replacement Algorithm: FIFO
Total Memory Accesses: 4381
Page Faults: 1081 (24.67%)
Page Replacements: 0
Swap-ins: 0
Swap-outs: 0
Average Memory Access Time: 0.10 ms

Memory Usage:
Used Frames: 11321/16384 (69.1%)
Page Table Frames: 10240 (62.5%)
=====

```

```

===== Virtual Memory Statistics =====
Page Replacement Algorithm: LRU
Total Memory Accesses: 4435
Page Faults: 1165 (26.27%)
Page Replacements: 0
Swap-ins: 0
Swap-outs: 0
Average Memory Access Time: 0.00 ms

Memory Usage:
Used Frames: 11405/16384 (69.6%)
Page Table Frames: 10240 (62.5%)
=====

```

[그림 45] 16384_600틱 기준

```

[System] Simulation finished

===== Virtual Memory Statistics =====
Page Replacement Algorithm: FIFO
Total Memory Accesses: 4607
Page Faults: 1167 (25.33%)
Page Replacements: 0
Swap-ins: 0
Swap-outs: 0
Average Memory Access Time: 0.12 ms

Memory Usage:
Used Frames: 11407/20000 (57.0%)
Page Table Frames: 10240 (51.2%)
=====

```

```

[System] Simulation finished

===== Virtual Memory Statistics =====
Page Replacement Algorithm: LRU
Total Memory Accesses: 4404
Page Faults: 1194 (27.11%)
Page Replacements: 0
Swap-ins: 0
Swap-outs: 0
Average Memory Access Time: 0.18 ms

Memory Usage:
Used Frames: 11434/20000 (57.2%)
Page Table Frames: 10240 (51.2%)
=====

```

[그림 46] 20000_600틱 기준

===== Virtual Memory Statistics =====	===== Virtual Memory Statistics =====
Page Replacement Algorithm: FIFO	Page Replacement Algorithm: LRU
Total Memory Accesses: 4171	Total Memory Accesses: 4445
Page Faults: 1061 (25.44%)	Page Faults: 1055 (23.73%)
Page Replacements: 0	Page Replacements: 0
Swap-ins: 0	Swap-ins: 0
Swap-outs: 0	Swap-outs: 0
Average Memory Access Time: 0.09 ms	Average Memory Access Time: 0.21 ms
Memory Usage:	Memory Usage:
Used Frames: 11301/12000 (94.2%)	Used Frames: 11295/12000 (94.1%)
Page Table Frames: 10240 (85.3%)	Page Table Frames: 10240 (85.3%)
=====	=====

[그림 47] 12000_600틱 기준

프레임 수와 알고리즘 성능의 관계를 분석하기 위해 12,000 개부터 20,000 개까지의 다양한 프레임 수에 대해 600 틱 기준으로 시뮬레이션을 진행하였다. 일반적으로 더 많은 프레임이 가용할 때 LRU 알고리즘이 시간적 지역성을 더 잘 활용할 수 있을 것이라 예상했지만, 실험 결과는 예상과 달리 뚜렷한 경향성을 보여주지 않았다. 또한 페이지 교체와 스와핑이 일어나지 않아 페이지 교체에서의 효율을 판단할 수가 없었다.

<지역성에 의한 메모리 접근 패턴과 페이지 교체 알고리즘의 상관관계>

프레임의 수가 너무 적어 알고리즘에 상관없이 페이지 교체가 일어나야 하는 상황을 피하기 위해 프레임 수를 늘렸었는데 가뜩이나 주소의 지역성을 강화하여 다양성이 줄어들어 스와핑 현상이 일어나지 않으니 페이지 교체 효율을 판단할 수 없는 상황이 발생한 것이라는 결론을 내렸다. 페이지 교체 알고리즘의 효율을 판단하기 위해서는 프레임 수를 늘릴 것이 아니라 오히려 줄여서 스와핑 현상이 일어나도록 유도해야 함을 깨달았다. 따라서 프레임 수를 줄여가며 실험 결과를 살펴보자.

===== Virtual Memory Statistics =====	===== Virtual Memory Statistics =====
Page Replacement Algorithm: FIFO	Page Replacement Algorithm: LRU
Total Memory Accesses: 4406	Total Memory Accesses: 4413
Page Faults: 1026 (23.29%)	Page Faults: 1063 (24.09%)
Page Replacements: 0	Page Replacements: 0
Swap-ins: 0	Swap-ins: 0
Swap-outs: 0	Swap-outs: 0
Average Memory Access Time: 0.05 ms	Average Memory Access Time: 0.15 ms
Memory Usage:	Memory Usage:
Used Frames: 11266/11500 (98.0%)	Used Frames: 11303/11500 (98.3%)
Page Table Frames: 10240 (89.0%)	Page Table Frames: 10240 (89.0%)
=====	=====

[그림 48] 여전히 페이지 교체가 일어나지 않은 상황

FIFO	LRU
==== Virtual Memory Statistics ====	==== Virtual Memory Statistics ====
Page Replacement Algorithm: FIFO	Page Replacement Algorithm: LRU
Total Memory Accesses: 4420	Total Memory Accesses: 4382
Page Faults: 1110 (25.11%)	Page Faults: 1172 (26.75%)
Page Replacements: 87	Page Replacements: 149
Swap-ins: 0	Swap-ins: 0
Swap-outs: 87	Swap-outs: 149
Average Memory Access Time: 0.07 ms	Average Memory Access Time: 0.20 ms
Memory Usage:	Memory Usage:
Used Frames: 11263/11264 (100.0%)	Used Frames: 11263/11264 (100.0%)
Page Table Frames: 10240 (90.9%)	Page Table Frames: 10240 (90.9%)
=====	=====

[그림 49] 페이지 교체 발생

FIFO	LRU
==== Virtual Memory Statistics ====	==== Virtual Memory Statistics ====
Page Replacement Algorithm: FIFO	Page Replacement Algorithm: LRU
Total Memory Accesses: 4474	Total Memory Accesses: 4352
Page Faults: 1184 (26.46%)	Page Faults: 1132 (26.01%)
Page Replacements: 425	Page Replacements: 373
Swap-ins: 93	Swap-ins: 70
Swap-outs: 425	Swap-outs: 373
Average Memory Access Time: 0.00 ms	Average Memory Access Time: 0.21 ms
Memory Usage:	Memory Usage:
Used Frames: 10999/11000 (100.0%)	Used Frames: 10999/11000 (100.0%)
Page Table Frames: 10240 (93.1%)	Page Table Frames: 10240 (93.1%)
=====	=====

FIFO	LRU
==== Virtual Memory Statistics ====	==== Virtual Memory Statistics ====
Page Replacement Algorithm: FIFO	Page Replacement Algorithm: LRU
Total Memory Accesses: 4446	Total Memory Accesses: 4640
Page Faults: 1236 (27.80%)	Page Faults: 1190 (25.65%)
Page Replacements: 577	Page Replacements: 531
Swap-ins: 118	Swap-ins: 84
Swap-outs: 577	Swap-outs: 531
Average Memory Access Time: 0.06 ms	Average Memory Access Time: 0.05 ms
Memory Usage:	Memory Usage:
Used Frames: 10899/10900 (100.0%)	Used Frames: 10899/10900 (100.0%)
Page Table Frames: 10240 (93.9%)	Page Table Frames: 10240 (93.9%)
=====	=====

점점 알고리즘 간 페이지 폴트 발생률 차이가 벌어지고 있다.

<pre> ===== Virtual Memory Statistics ===== Page Replacement Algorithm: FIFO Total Memory Accesses: 4484 Page Faults: 1314 (29.30%) Page Replacements: 755 Swap-ins: 161 Swap-outs: 755 Average Memory Access Time: 0.17 ms Memory Usage: Used Frames: 10799/10800 (100.0%) Page Table Frames: 10240 (94.8%) ===== </pre>	<pre> ===== Virtual Memory Statistics ===== Page Replacement Algorithm: LRU Total Memory Accesses: 4582 Page Faults: 1212 (26.45%) Page Replacements: 653 Swap-ins: 131 Swap-outs: 653 Average Memory Access Time: 0.06 ms Memory Usage: Used Frames: 10799/10800 (100.0%) Page Table Frames: 10240 (94.8%) ===== </pre>
<pre> ===== Virtual Memory Statistics ===== Page Replacement Algorithm: FIFO Total Memory Accesses: 4479 Page Faults: 1269 (28.33%) Page Replacements: 810 Swap-ins: 161 Swap-outs: 810 Average Memory Access Time: 0.20 ms Memory Usage: Used Frames: 10699/10700 (100.0%) Page Table Frames: 10240 (95.7%) ===== </pre>	<pre> ===== Virtual Memory Statistics ===== Page Replacement Algorithm: LRU Total Memory Accesses: 4434 Page Faults: 1154 (26.03%) Page Replacements: 695 Swap-ins: 147 Swap-outs: 695 Average Memory Access Time: 0.11 ms Memory Usage: Used Frames: 10699/10700 (100.0%) Page Table Frames: 10240 (95.7%) ===== </pre>
<pre> ===== Virtual Memory Statistics ===== Page Replacement Algorithm: FIFO Total Memory Accesses: 4939 Page Faults: 1579 (31.97%) Page Replacements: 1220 Swap-ins: 363 Swap-outs: 1220 Average Memory Access Time: 0.12 ms Memory Usage: Used Frames: 10599/10600 (100.0%) Page Table Frames: 10240 (96.6%) ===== </pre>	<pre> ===== Virtual Memory Statistics ===== Page Replacement Algorithm: LRU Total Memory Accesses: 4694 Page Faults: 1314 (27.99%) Page Replacements: 955 Swap-ins: 239 Swap-outs: 955 Average Memory Access Time: 0.20 ms Memory Usage: Used Frames: 10599/10600 (100.0%) Page Table Frames: 10240 (96.6%) ===== </pre>
<pre> ===== Virtual Memory Statistics ===== Page Replacement Algorithm: FIFO Total Memory Accesses: 4712 Page Faults: 1382 (29.33%) Page Replacements: 1123 Swap-ins: 264 Swap-outs: 1123 Average Memory Access Time: 0.04 ms Memory Usage: Used Frames: 10499/10500 (100.0%) Page Table Frames: 10240 (97.5%) ===== </pre>	<pre> ===== Virtual Memory Statistics ===== Page Replacement Algorithm: LRU Total Memory Accesses: 4629 Page Faults: 1289 (27.85%) Page Replacements: 1030 Swap-ins: 326 Swap-outs: 1030 Average Memory Access Time: 0.14 ms Memory Usage: Used Frames: 10499/10500 (100.0%) Page Table Frames: 10240 (97.5%) ===== </pre>

여러 번 실행 해본 결과 10600에서 10700개의 프레임을 가질 때 차이가 가장 많이 벌어졌고 따라서 이번엔 10600개와 10700개의 경우를 600틱이 아닌 3000틱 동안 실행하여 결과를 살펴보자.

3000 틱 실행

<pre> ===== Virtual Memory Statistics ===== Page Replacement Algorithm: FIFO Total Memory Accesses: 31581 Page Faults: 8431 (26.70%) Page Replacements: 7972 Swap-ins: 6278 Swap-outs: 7972 Average Memory Access Time: 0.02 ms Memory Usage: Used Frames: 10699/10700 (100.0%) Page Table Frames: 10240 (95.7%) ===== </pre>	<pre> ===== Virtual Memory Statistics ===== Page Replacement Algorithm: LRU Total Memory Accesses: 31905 Page Faults: 8155 (25.56%) Page Replacements: 7696 Swap-ins: 6071 Swap-outs: 7696 Average Memory Access Time: 0.02 ms Memory Usage: Used Frames: 10699/10700 (100.0%) Page Table Frames: 10240 (95.7%) ===== </pre>
<pre> ===== Virtual Memory Statistics ===== Page Replacement Algorithm: FIFO Total Memory Accesses: 31242 Page Faults: 9052 (28.97%) Page Replacements: 8693 Swap-ins: 6620 Swap-outs: 8693 Average Memory Access Time: 0.02 ms Memory Usage: Used Frames: 10599/10600 (100.0%) Page Table Frames: 10240 (96.6%) ===== </pre>	<pre> ===== Virtual Memory Statistics ===== Page Replacement Algorithm: LRU Total Memory Accesses: 30406 Page Faults: 8426 (27.71%) Page Replacements: 8067 Swap-ins: 6051 Swap-outs: 8067 Average Memory Access Time: 0.01 ms Memory Usage: Used Frames: 10599/10600 (100.0%) Page Table Frames: 10240 (96.6%) ===== </pre>

600 틱의 경우에는 2 퍼센트에서 3 퍼센트까지 차이가 났으나 3000 틱으로 진행한 결과 1 퍼센트에서 2 퍼센트정도 차이로 LRU 가 적은 페이지 폴트 발생률을 보인 것으로 나타났다.

<결론>

비록 현재 구현에서는 FIFO 와 LRU 알고리즘 간의 성능 차이를 명확하게 시연하지 못했지만, 이번 프로젝트를 통해 몇 가지 중요한 통찰을 얻을 수 있었다. 특히 메모리 접근 패턴이 페이지 교체 알고리즘의 성능에 결정적인 영향을 미친다는 점을 실험적으로 확인할 수 있었다. 또한 시뮬레이션 환경에서 실제 프로그램의 메모리 접근 패턴을 모사하는 것이 얼마나 어려운 과제인지도 깨달을 수 있었다.

단순히 하드웨어 자원을 늘리는 것보다는, 프로세스의 메모리 접근 패턴을 더 정확히 이해하고 이를 시뮬레이션에 반영하는 것이 중요하다는 점을 알 수 있었다. 또한 페이지 교체 알고리즘의 성능은 단순히 가용 프레임의 수보다는 워크로드의 특성과 메모리 접근의 지역성에 더 크게 영향을 받는다는 것을 확인할 수 있었다.

이번 프로젝트는 비록 당초 기대했던 알고리즘 간의 성능 차이를 명확히 보여주지는 못했지만, 가상 메모리 시스템의 복잡성과 다양한 요소들이 성능에 미치는 영향을 이해하는 데 큰 도움이 되었다고 생각한다.

사용법

Ver1

- make
- ./scheduler [타임 쿼텀]

```
Usage: ./scheduler <time_quantum>
```

명령줄 인수 : 실행파일 타임퀀텀

Ver2

- make
- ./vmm2 [타임 쿼텀 (50 이하)] [알고리즘 선택]

```
Usage: ./vmm2 <time_quantum> <algorithm>
Algorithms:
0 - FIFO (First In First Out)
1 - LRU (Least Recently Used)
```

명령줄 인수 : 실행파일 타임퀀텀 알고리즘