OPERATING SYSTEMS

Proj2 :: Virtual Memory (paging)

By using C programming

손보경 32212190 ssg020910@naver.com
임석범 32203743 andylim1022@naver.com

2024.12.09.

목 차

1. Introduction

가상 메모리는 현대 운영체제의 핵심적인 기능으로, 물리적 메모리보다 큰 프로그램을 실행할 수 있도록 지원하는 기술이다. 이는 프로세스가 물리적 메모리에 완전히 적재되지 않아도 실행이 가능하게 하며, 논리적 메모리와 물리적 메모리를 분리하여 프로그래머가 메모리 용량의 제약을 걱정하지 않고 개발에 집중할 수 있도록 한다. 논리적 메모리는 프로그래머에게 단일하고 연속적인 메모리 공간으로 보이지만, 실제로는 물리적 메모리와 보조 저장 장치 간에 분산되어 관리된다. 이러한 가상 메모리의 도입으로 시스템의 메모리 활용 효율이 향상되었지만, 복잡한 구조와 알고리즘을 동반하며, 부적절한 설계나 구현은 성능 저하를 초래할 수 있다.

본 프로젝트는 이러한 가상 메모리 원리를 기반으로, 페이징 기법을 이용한 메모리 관리 시스템의 설계와 고현을 목표로 하고 있다. 프로젝트의 주요 초점은 가상 메모리의 핵심 구성 요소를 단계적으로 구현하는 데 두었다. 물리 메모리의 초기화와 할당/해제 같은 기본 메모리 관리 기능에서 시작하여, 프로세스 관리와 페이지 테이블 관리, 스와핑 기능을 차례로 개발했다. 또한, Copy-on-Write 기능을 통해 메모리 효율을 극대화하는 기술도 포함했다. 이러한 접근 방식은 가상 메모리 시스템의 안정성과 성능을 확보하고, 이를 효과적으로 구현하기 위해 논리적 순서를 따랐다.

2. Requirement

Category	Requirement
과제 설명	- 페이징을 활용한 주소 변환 및 효율적인 페이지 테이블 관리
시뮬레이터 동작 조건	- VA → PA 변환이 정상적으로 작동해야 함 - OS가 런타임 시 페이지 테이블을 업데이트
OS 초기화	- 부팅시 물리적 메모리를 페이지 크기로 나누고, 빈 페이지 프레임 리스트 관리

OS 초기화	- 총 물리 메모리 크기는 가정이 가능하며, 페이지 프레임 번호(PFN)는 0부터 시작
사용자 프로세스 초기화	- 프로세스마다 페이지 테이블 관리 - 프로세스 생성 시 비어 있는 초기화된 페이지 테이블 할당
사용자 프로세스 실행	- 프로세스는 타임슬라이스 동안 10개의 메모리 주소(페이지)를 엑세스 - 프로세스는 IPC 메시지를 통해 메모리 접근 요청 전송 - OS는 페이지 테이블을 확인하여 유효한 경우 PA 접근, 유효하지 않으면 새로운 빈 페이지 프레임 할당 후 테이블 업데이트
추가 구현1 2단계 페이지 테이블	- 메모리 사용률을 낮추기 위해 페이지 테이블 구조를 2단계로 분리 - 1단계 VM 주소는 1단계 테이블 인덱스, 2단계 VM 주소는 2단계 테이블 인덱스로 사용 - 2단계 테이블은 1단계 항목이 null인 경우 생성
추가 구현2 스와핑	- 물리 메모리 용량 부족시 LRU 알고리즘으로 메모리 페이지를 디스크로 스와핑 - 스와핑된 페이지는 디스크에 저장(PID, 페이지 번호, 데이터 포함) - 스와핑된 페이지 접근 시 디스크에서 메모리로 복원
추가 구현3 포크시 Copy-on-Write	- 메모리를 파일(simple.bin)로 초기화하여 읽기/쓰기 시뮬레이션 - 포크된 자식 프로세스는 초기엔 부모 메모리를 공유하거나 쓰기 시 새로운 페이지 프로엠 할당 및 복사 후 독립적으로 메모리 사용

출력 요구사항	- 1,000틱 동안 VA, PA, 페이지 폴트 이벤트, 페이지 테이블 변화, 읽기/쓰기 값을 로그로 출력
종료 조건	- 시뮬레이션은 시간 틱이 10,000을 초과하면 종료
평가 기준	- 구현 수준 및 데모에 따라 차등 평가 - 고유한/창의적인 접근법 시 추가 점수

3. Concepts

운영체제의 핵심적인 역할 중 하나는 메모리 관리를 효과적으로 수행하는 것이다. 프로세스가 실행되기 위해서는 프로그램 코드와 데이터가 메모리에 적재되어야 하며, 운영체제는 이러한 메모리 자원을 효율적이고 안전하게 관리해야 한다. 하지만 현대 시스템에서는 메모리 크기, 프로세스 간의 메모리 충돌, 그리고 메모리 접근 속도와 같은 다양한 문제들이 발생하기 때문에 이를 해결하기 위한 체계적인 관리 기법이 필요하다. 가상 메모리, 페이징, TLB 등의 개념은 이러한 문제를 해결하기 위해 개발된 기술로, 운영체제의 메모리 관리 체계에서 매우 중요한 요소로 작용한다. 이번 설명에서는 이러한 배경지식을 상세히 다루어, 이론적 이해를 돕고자 한다.

3-1. 메인 메모리 (Main Memory)

현대 컴퓨터 시스템에서 메인 메모리는 필수적인 역할을 담당한다. 메인 메모리는 각 바이트가 고유 주소를 가지는 큰 배열로 구성되어 있다. CPU는 명령어 실행을 위해 프로그램 카운터(PC)의 값에 따라 메인 메모리에서 명령어를 가져온다. 이 명령어는 추가 메모리 주소의 로드나 저장 작업으로 이어질 수 있다.

3.1.1. 논리적 주소 공간과 물리적 주소 공간

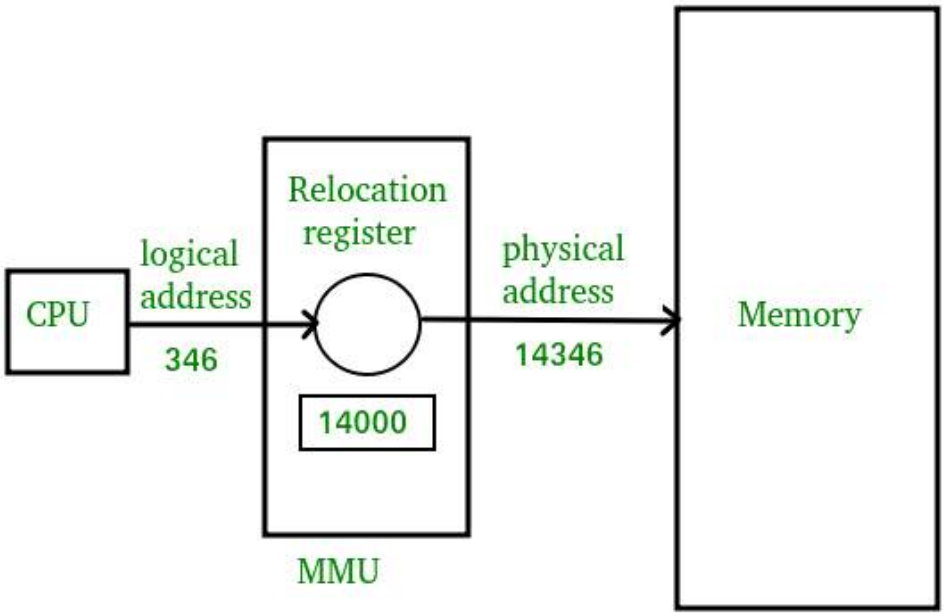


그림 1 논리적 주소 공간에서 물리적 주소 공간으로의 변환 과정 (MMU)

논리적 주소는 CPU에서 생성된 주소를 의미하며, **물리적 주소**는 메모리 장치에서 인식되는 주소로, 메모리 주소 레지스터에 로드되는 실제 주소를 가리킨다. 컴파일 시간이나 로드 시간에 주소가 바인딩될 경우 논리적 주소와 물리적 주소는 같다. 그러나 실행 시간에 주소 바인딩이 이루어지면 두 주소는 달라진다. 실행 중 논리적 주소는 가상 주소로 간주하며, 논리적 주소 공간은 프로그램이 생성하는 모든 논리적 주소의 집합, 물리적 주소 공간은 이러한 논리적 주소와 매핑되는 실제 물리적 주소의 집합이다.

주소 변환은 **메모리 관리 유닛(MMU)**을 통해 이루어진다. 논리적 주소를 물리적 주소로 변환하여 메모리에 전달하며, 이 과정은 사용자 소프트웨어에서 추상화된다.

3.1.2. 페이징

페이징(Paging)은 현대 운영체제에서 메모리를 효율적으로 관리하기 위해 사용하는 핵심 기법의 하나로, 프로세스와 물리적 메모리 간의 크기 차이를 극복하기 위해 설계되었다. 이 기법은 메모리를 고정된 크기의 블록으로

나누어 관리하며, 프로세스의 논리적 주소 공간은 **페이지**라는 단위로, 물리적 메모리는 **프레임**이라는 단위로 나뉜다. 페이지와 프레임의 크기는 같아야 하며, 보통 2의 거듭제곱 꼴의 크기로 설정된다. 이를 통해 메모리 할당 시 발생할 수 있는 외부 단편화 문제를 해결하고, 메모리 공간을 효율적으로 사용할 수 있다.

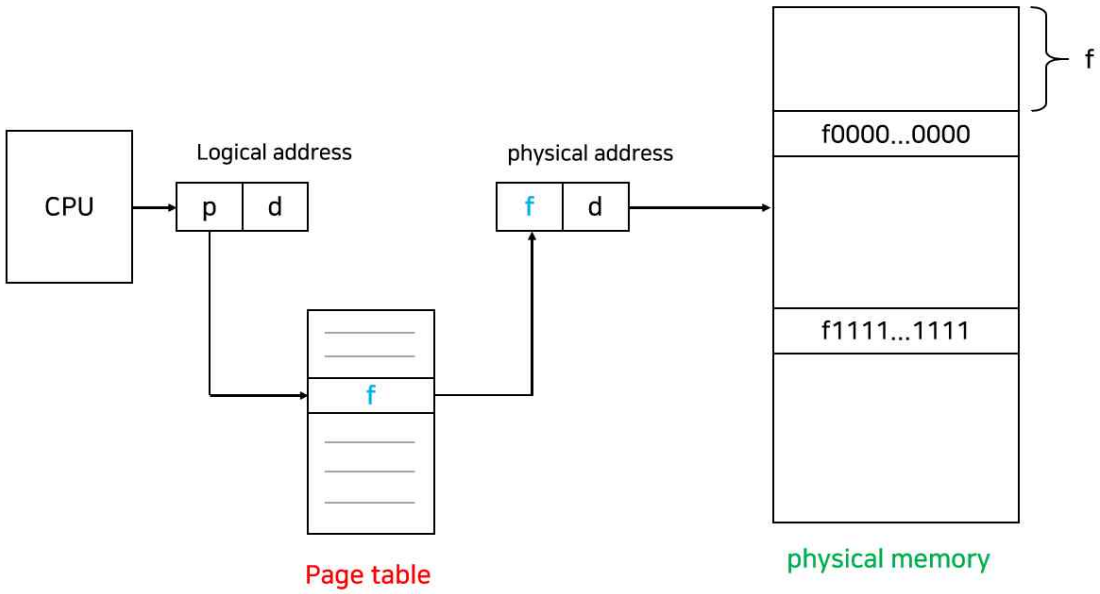


그림 2 페이징 기본 구조

- **페이징 기본 구조** : 물리적 메모리는 고정 크기의 프레임으로 나뉘고, 논리적 메모리는 동일한 크기의 페이지로 나뉜다.
- **주소 변환** : CPU가 생성하는 각 주소는 페이지 번호와 페이지 오프셋으로 나뉘며, 페이지 번호를 기반으로 페이지 테이블을 참조해 물리적 메모리의 프레임 번호를 확인한다.
- **단편화** : 외부 단편화는 발생하지 않으나, 페이지 크기와 메모리 요구량이 맞지 않으면 내부 단편화가 발생할 수 있다.

페이징 시스템에서는 논리적 주소는 페이지 번호와 페이지 내 오프셋으로 구성되며, 논리적 주소를 물리적 주소로 변환하기 위해 페이지 테이블이 사용된다. 페이지 테이블은 논리적 페이지 번호와 물리적 프레임 번호 간의 매핑 정보를 저장하며, 변환 과정에서 CPU가 생성한 논리적 주소는 페이지 테이블을 참조하여 물리적 주소로 변환된다. 이러한 과정은 운영체제에서 프로세스마다 독립적으로 관리되며, 각 프로세스는 자신만의 페이지 테이블을

갖는다. 페이지 테이블의 관리를 위해 **페이지 테이블 레지스터(Page Table Register, PTR)**가 사용되며, 자주 참조되는 매핑 정보를 저장하고 주소 변환 속도를 높이기 위해 **TLB(Translation Look-Aside Buffer)**와 같은 하드웨어 캐시가 활용된다.

페이징은 다수의 프로세스가 동일한 코드를 공유할 수 있게 하며, 메모리 공간의 중복 사용을 줄여줍니다. 예를 들어, 동적 라이브러리는 모든 프로세스가 동일한 물리적 복사본을 참조하도록 구현된다.

3.1.3. Translation Look-Aside Buffer (TLB)

TLB(Translation Look-Aside Buffer)는 메모리 접근 속도를 향상시키기 위해 사용되는 하드웨어 기반의 고속 조회 캐시로, 페이징 기법에서 발생하는 메모리 접근 지연 문제를 해결하기 위해 설계되었다. 페이징 시스템에서는 논리적 주소를 물리적 주소로 변환하기 위해 페이지 테이블을 참조해야 하며, 이 과정에서 두 번의 메모리 접근이 필요하다. 첫 번째는 페이지 테이블을 탐색하여 프레임 번호를 얻는 작업이고, 두 번째는 실제 데이터에 접근하는 작업이다. 이처럼 메모리 접근 시간이 증가하면 전체 시스템의 성능이 저하될 수 있다.

TLB는 이러한 문제를 해결하기 위해 고안된 Associative Memory의 일종으로, 페이지 테이블의 일부 매핑 정보를 캐싱하여 논리적 주소에서 물리적 주소로의 변환 속도를 대폭 향상시킨다. TLB는 각 항목이 키(Key)와 값(Value)로 구성된 소형 고속 캐시로, 키에 해당하는 페이지 번호가 주어지면 이를 TLB의 모든 항목과 동시에 비교하여 일치하는 항목의 값을 반환한다. 이와 같은 연관 검색 방식은 매우 빠르게 이루어지며, 주소 변환 작업에서 TLB를 통해 페이지 접근 과정을 생략할 수 있다.

TLB는 제한된 크기를 가지므로, 새로운 항목을 추가할 때 기존 항목을 교체해야하는 상황이 발생할 수 있다. 이를 위해 다양한 교체 정책이 사용되는데, 대표적으로 LRU, RR, 랜덤 방식이 있다.

TLB는 하드웨어 기능으로 운영체제가 직접적으로 제어하지 않지만, 운영체제는 TLB의 구조와 작동 방식을 이해하고 이를 고려한 페이징 시스템을 설계해야 한다. 예를 들어, 특정 하드웨어 아키텍처에서 TLB의 동작 방식이 변경될 경우, 운영체제의 페이징 구현에도 영향을 미칠 수 있다. 따라서 TLB 설계와 운영체제의 메모리 관리 기법은 상호 밀접하게 연관되어 있다.

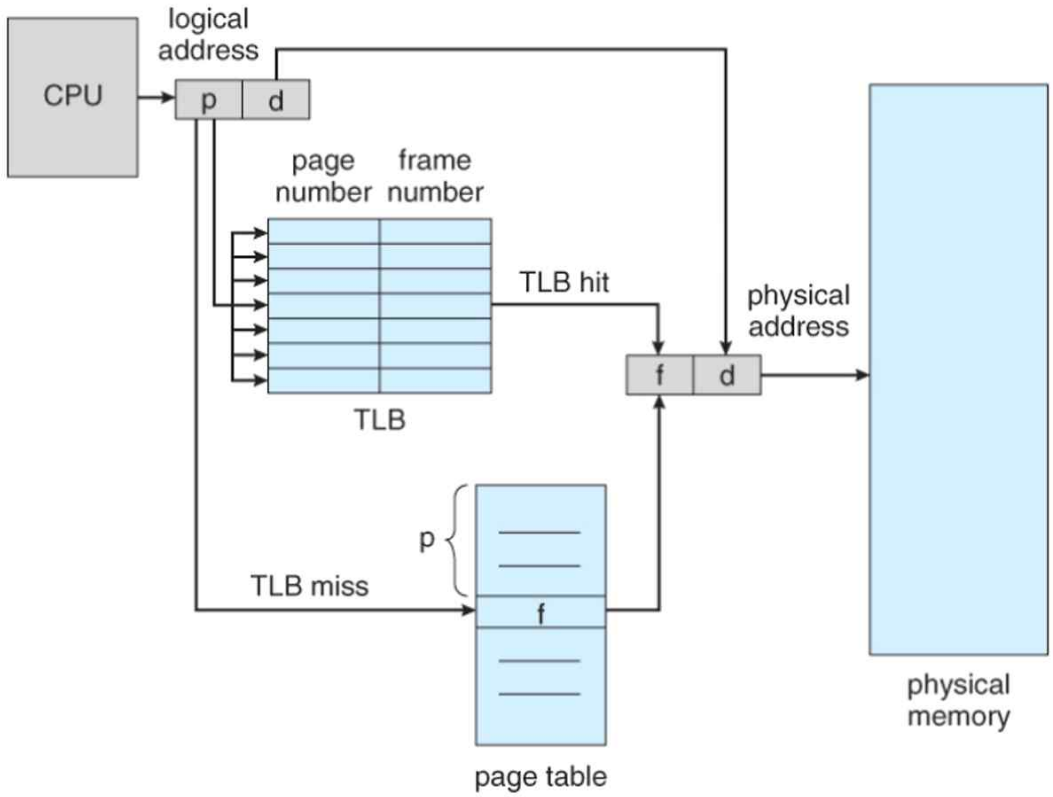


그림 3 TLB를 이용한 페이징

TLB는 현대 컴퓨터 시스템에서 필수적인 구성 요소로, 메모리 접근 성능을 크게 개선하는 역할을 한다. 특히 다중 작업 환경에서 프로세스 간 컨텍스트 스위칭이 빈번히 발생하는 경우, TLB를 효과적으로 활용함으로써 메모리 접근 지연을 최소화할 수 있다. 이러한 이유로 TLB는 고성능 컴퓨터 아키텍처에서 중요한 역할을 하며, 운영체제와 하드웨어 간의 효율적인 협력을 통해 시스템의 전반적인 성능을 최적화하는 데 기여한다.

3.1.4. 계층적 페이징

계층적 페이징(Hierarchical Paging)은 페이지 테이블의 크기가 너무 커

질 때 이를 효율적으로 관리하기 위해 도입된 기법으로, 단일 레벨의 페이지 테이블 대신 여러 계층의 테이블을 사용하는 방식이다. 현대 운영체제에서 사용하는 논리적 주소 공간은 매우 크며, 이에 따라 페이지 테이블 역시 크기가 커질 수 있다. 이 경우, 페이지 테이블 전체를 메모리에 상주시킬 수 없거나 효율적인 관리를 위해 계층적 접근 방식을 활용해야 한다.

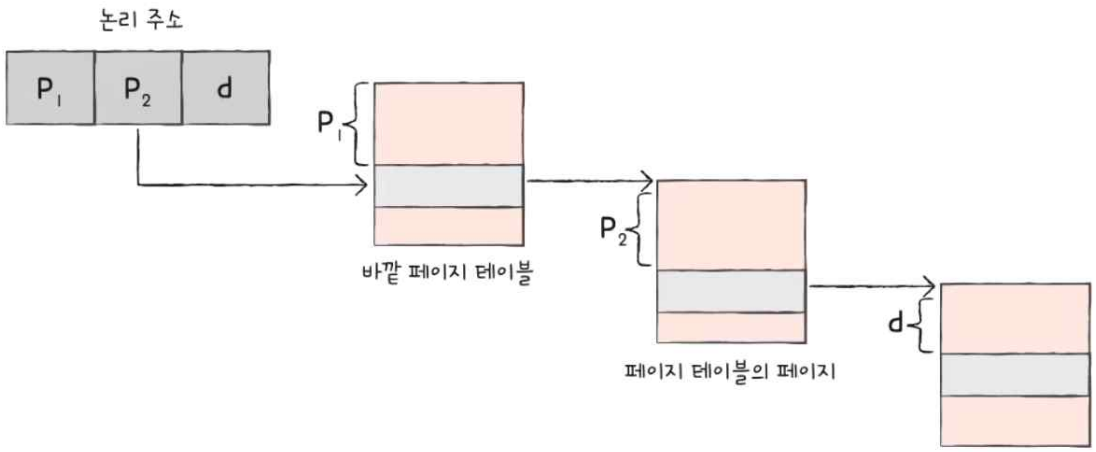


그림 4 계층적 페이지 구조

논리적 구조는 페이지 번호와 페이지 오프셋으로 나뉘며, 계층적 페이징에서는 페이지 번호를 추가적으로 더 세분화하여 여러 레벨의 페이지 번호로 구성한다. 예를 들어, 32비트 논리 주소를 사용하는 시스템에서 다음과 같이 나눌 수 있다.

- **상위 페이지 번호** : 최상위 레벨의 페이지 테이블을 탐색하는 데 사용된다.
- **하위 페이지 번호** : 하위 레벨의 페이지 테이블을 탐색하는 데 사용된다.
- **페이지 오프셋** : 물리적 메모리 내의 특정 위치를 지정한다.

계층적 페이지 테이블의 구조는 트리와 유사하다. 루트 노드에 해당하는 최상위 페이지 테이블이 있고, 각각의 엔트리가 하위 페이지 테이블로 연결된다. 최종적으로 물리적 프레임 번호가 저장된 하위 페이지 테이블을 찾아 논리적 주소를 물리적 주소로 변환한다.

3.1.5. 스와핑(Swapping)

스와핑은 운영체제에서 사용되는 메모리 관리 기법 중 하나로, 프로세스의 메모리 공간을 효율적으로 관리하기 위해 사용된다. 이 기법은 프로세스 전체 또는 일부를 디스크와 물리적 메모리 사이에서 전환하는 과정을 통해 물리적 메모리 자원을 효율적으로 활용할 수 있도록 설계되었다.

컴퓨터 시스템에서 실행 중인 프로세스는 물리적 메모리에 로드되어야 하지만, 물리적 메모리의 용량은 제한적이다. 스와핑은 실행되지 않거나 우선순위가 낮은 프로세스를 임시적으로 디스크의 백킹 스토어(Backing Store)로 이동시켜 물리적 메모리 공간을 확보하고, 필요시 이를 다시 물리적 메모리로 불러오는 방식으로 작동한다. 이를 통해 운영체제는 멀티프로그래밍을 지원하고, 시스템의 자원 활용도를 높일 수 있다.

스와핑의 동작 원리는 다음과 같다.

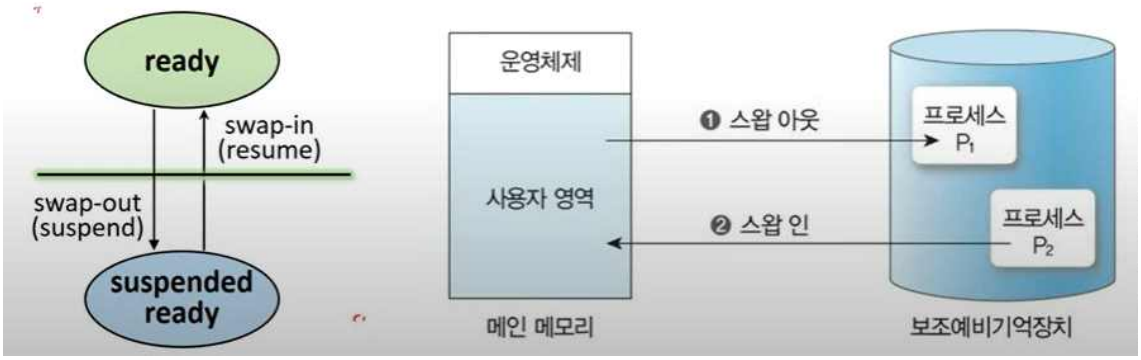


그림 5 프로세스 스와핑

- 프로세스 메모리 로드 : 실행할 프로세스는 물리적 메모리에 로드되어야 한다. 실행 중인 프로세스가 많아 물리적 메모리가 부족할 경우, 스와핑이 시작된다.
- 백킹 스토어로 전환 : 우선 순위가 낮은 프로세스나 잠시 사용되지 않는 프로세스의 데이터를 백킹 스토어로 이동한다. 이 과정은 페이지 단위 또는 프로세스 전체 단위로 이루어진다.
- 프로세스 복귀 : 백킹 스토어로 이동한 프로세스는 실행이 필요할 때 다시 물리적 메모리로 로드된다. 이 과정은 스와핑 오버헤드(Swapping Overhead)를 초래할 수 있으므로, 효율적인 관리가 중요하다.

3-2. 가상 메모리 (Virtual Memory)

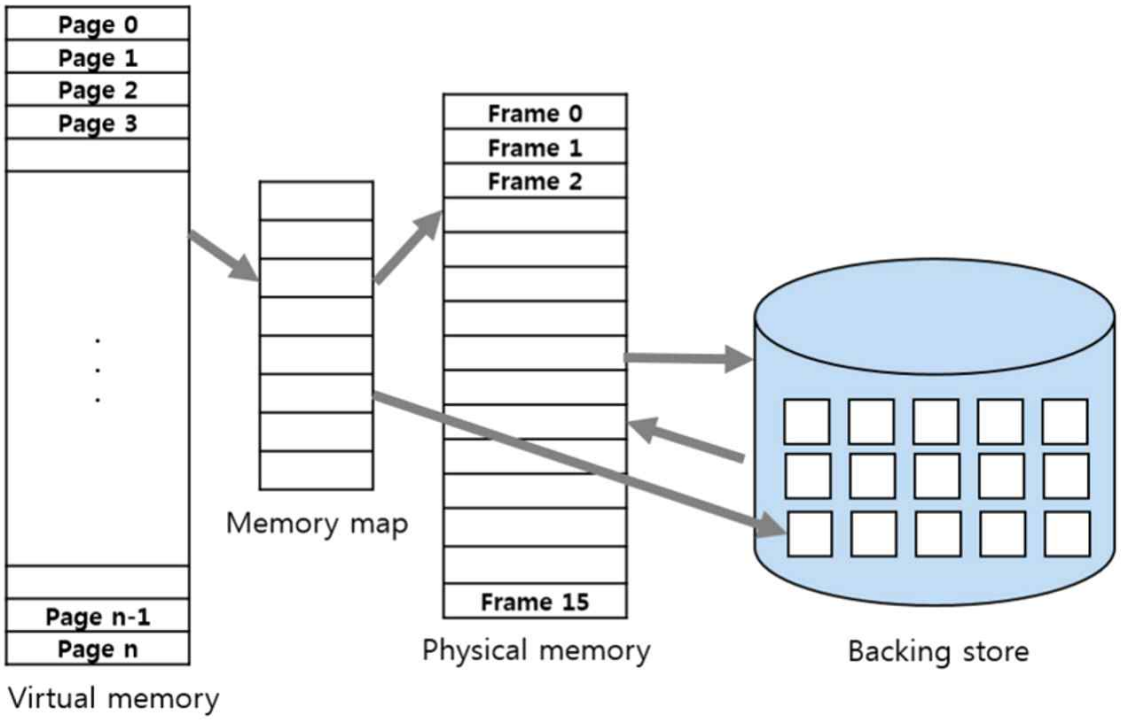


그림 6 가상 메모리 구조

가상 메모리는 물리적 메모리와 논리적 메모리를 분리하는 개념이다. 프로그래머는 실제 사용할 수 있는 물리적 메모리의 크기에 구애받지 않고, 매우 큰 가상 메모리를 사용하는 것처럼 코드를 작성할 수 있다. 이에 따라 프로그래밍이 훨씬 간단해진다.

가상 메모리는 프로세스가 메모리에 저장된 방식의 논리적 관점을 가상 주소 공간으로 제공한다. 가상 메모리 시스템에서는 프로세스가 사용하는 메모리 페이지들이 물리적으로 연속적일 필요가 없다. 메모리 관리 유닛(MMU)는 논리적 페이지를 물리적 페이지 프레임에 매핑하는 역할을 한다.

3.2.1. 요구 페이징 (Demand Paging)

요구 페이징은 필요한 페이지만 메모리에 적재하는 방식으로, 가상 메모리 시스템에서 주로 사용된다. 프로그램 실행 시 처음부터 전체 프로그램을 메모리에 적재하는 대신, 실제로 필요한 순간에만 해당 페이지를 가져온다. 이를 통해 불필요한 메모리 낭비를 방지할 수 있다. 요구 페이징의 핵심 개

넘은 페이지 테이블에 있는 유효-무효 비트를 통해 메모리에 적재된 페이지와 그렇지 않은 페이지를 구분하는 것이다. 프로세스가 메모리에 없는 페이지를 참조하려 하면 페이지 폴트가 발생하고, 운영체제는 이를 처리하여 필요한 페이지를 보조 저장 장치에서 메모리로 가져온다.

Page Fault 시 처리는 다음과 같은 과정으로 동작한다.

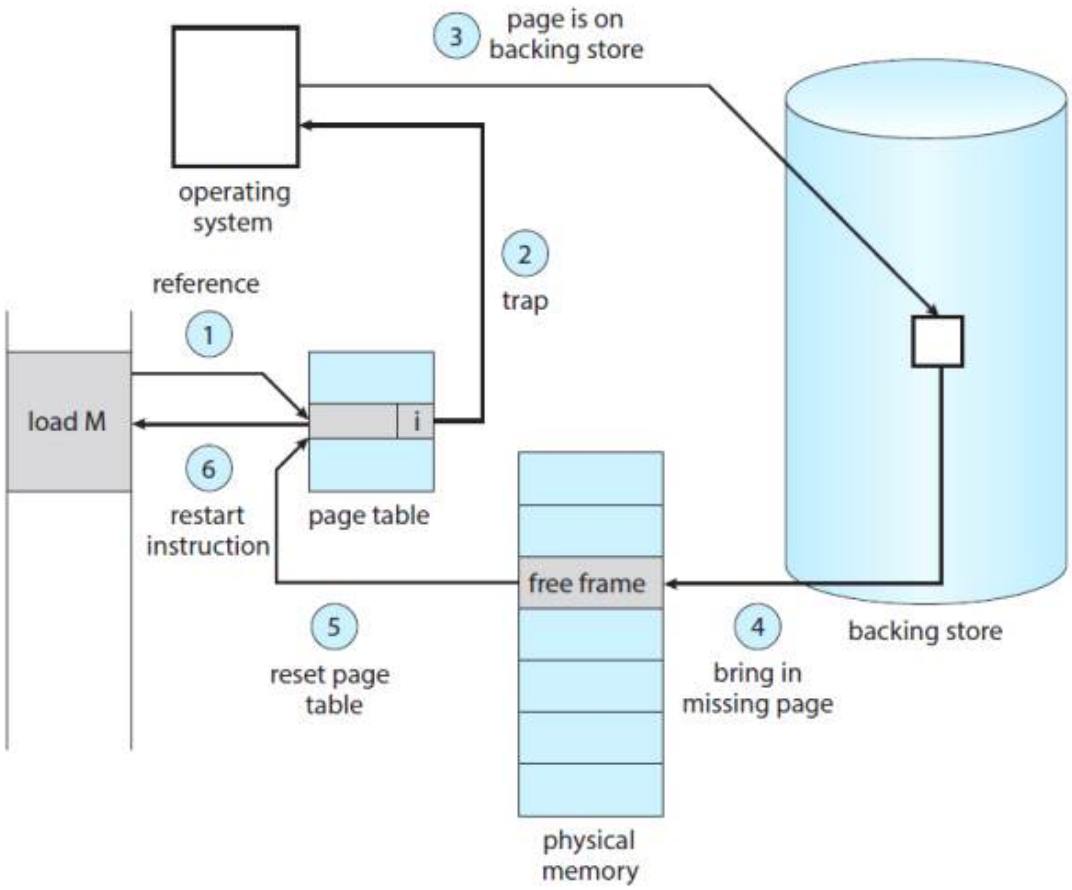


그림 7 Page Fault 시 처리 방법

- **유효-무효 비트 (Valid-Invalid Bit)** : 페이지 테이블 항목에는 유효-무효 비트가 있으며 유효 비트가 활성화되었다는 것은 페이지가 실제 메모리에 존재한다는 뜻이며, 무효 비트가 활성화되었다는 것은 페이지가 메모리에 없거나 해당 프로세스가 접근 불가능한 페이지라는 뜻이다.

- **페이지 폴트 (Page Fault)** : 프로세스가 메모리에 없는 페이지를 접근하려 할 때, 페이지 폴트가 발생하며 CPU는 페이지 테이블을 확인하다가

무효 비트를 발견하면 운영체제에 트랩을 발생시킨다.

- **페이지 폴트 처리 과정**
  - **참조가 유효한지 확인** : 유효하지 않으면 프로세스를 종료하고, 유효하지만 아직 메모리에 없는 경우, 페이지를 적재한다.
  - **사용 가능한 프레임 확인** : 여유 프레임이 있으면 바로 사용하며, 여유 프레임이 없으면 페이지를 적재한다.
  - 필요한 페이지를 보조 저장 장치에서 메모리로 읽어온다.
  - 페이지 테이블을 갱신하여 새 페이지가 메모리에 있음을 표시한다.
  - 중단된 명령을 재실행한다.

요구 페이징의 장점은 메모리를 효율적으로 사용할 수 있다는 점이다. 프로그램 전체를 메모리에 올리지 않으므로 실제로 필요 없는 페이지는 로드되지 않아 메모리 낭비가 줄어든다. 또한 이를 통해 물리적 메모리보다 훨씬 큰 가상 메모리를 사용할 수 있으며, 동시에 실행되는 프로세스의 수를 늘릴 수 있다. 하지만 단점으로는 페이지 폴트가 자주 발생하면 성능 저하가 발생할 수 있으며, 페이지를 보조 저장 장치에서 읽어오는 과정에서 디스크 I/O 비용이 증가한다는 점이 있다. 또한 페이지 교체 알고리즘과 폴트 처리 과정이 추가되면서 운영체제 구현이 복잡해질 수 있다.

3.2.2. Copy-on-Write

Copy-on-Write는 프로세스가 fork() 호출을 통해 부모 프로세스를 복제할 때, 초기 단계에서는 부모와 자식 프로세스가 동일한 메모리 페이지를 공유하도록 한다. 하지만 프로세스가 해당 페이지를 수정하려고 하면, 그제야 복사가 이루어지는 방식이다.

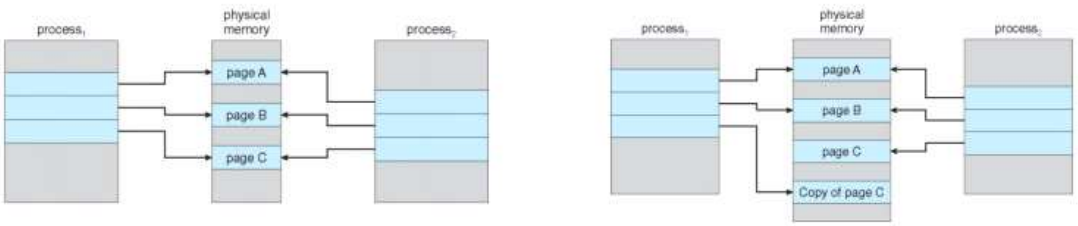


그림 8 Copy-on-Write 동작 과정

- 초기 상태에서는 부모와 자식이 동일한 페이지를 공유한다.
- 페이지 수정 시, 해당 페이지가 복사되어 자식 프로세스에서만 수정된 내용이 반영된다.

3.2.3. 페이지 교체 (Page Replacement)

페이지 교체는 제한된 물리 메모리에서 다수의 프로세스를 효율적으로 실행하기 위해 사용되는 기술이다. 요구 페이징 환경에서 필요한 페이지가 메모리에 없으면, 운영체제는 페이지 폴트를 처리하여 보조 저장 장치에 있는 페이지를 메모리에 적재한다. 이 과정에서 메모리가 가득 차 있으면 페이지 교체 알고리즘을 통해 교체할 페이지를 선택하고, 해당 페이지를 내보낸 후 새로운 페이지를 메모리에 적재한다. 페이지 교체는 시스템 성능에 직접적인 영향을 미치므로 효율적인 알고리즘 선택이 중요하다.

페이지 교체 과정에서 불필요한 데이터 전송을 줄이기 위해 운영체제는 수정 비트(Dirty Bit)를 사용한다. 수정 비트는 페이지가 메모리에서 보조 저장 장치로 다시 저장하지 않고 바로 제거할 수 있어 I/O 작업을 크게 줄일 수 있다. 수정 비트의 활용은 페이지 폴트 처리 속도를 높이고 전체 시스템 성능을 향상하는 데 이바지한다. 주요 페이지 교체 알고리즘은 아래와 같다.

- FIFO (First-In-First-Out)

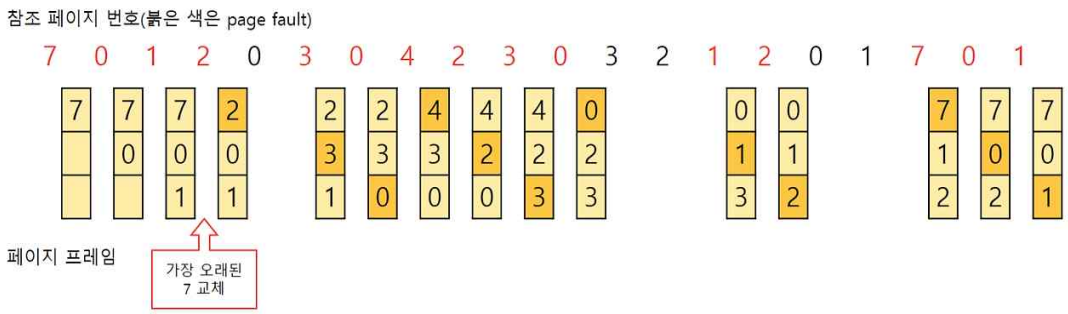


그림 9 FIFO 알고리즘

FIFO 알고리즘은 가장 먼저 메모리에 적재된 페이지를 우선적으로 교체하는 방식이다. 페이지가 적재된 순서를 유지하기 위해 큐(queue)를 사용하며, 구현이 간단하다는 장점이 있다. 그러나 오래된 페이지가 여전히

자주 사용될 가능성이 있는 경우에도 교체 대상이 되므로 1)벨라디의 역설이 발생할 수 있다. 이는 페이지 프레임 수가 증가해도 페이지 폴트율이 감소하지 않는 비효율성을 초래한다.

- OPT (Optimal Algorithm)

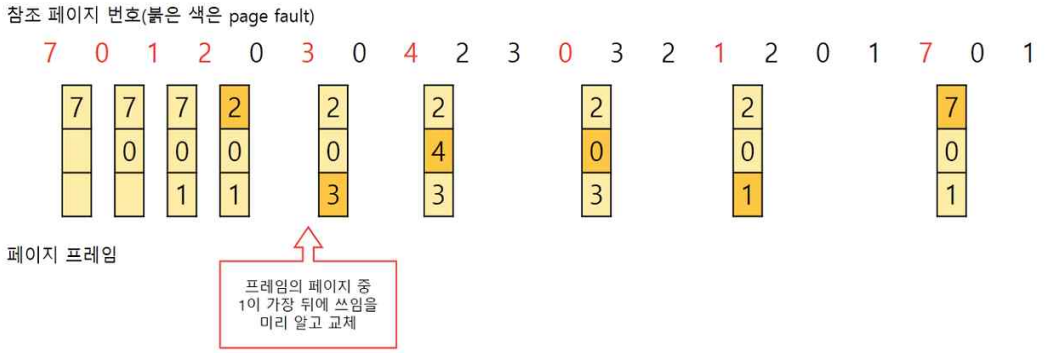


그림 10 OPT 알고리즘

OPT 알고리즘은 이론적으로 가장 효율적인 방식으로, 미래에 가장 오랫동안 상용되지 않을 페이지를 교체 대상으로 선정한다. 페이지 폴트율이 최소화되는 이상적인 알고리즘이지만, 실제로 미래를 예측할 수 없으므로 현실적인 구현은 불가능하다. 대산, 다른 알고리즘의 성능을 평가하기 위한 기준으로 활용된다.

- LRU (Least Recently Used)

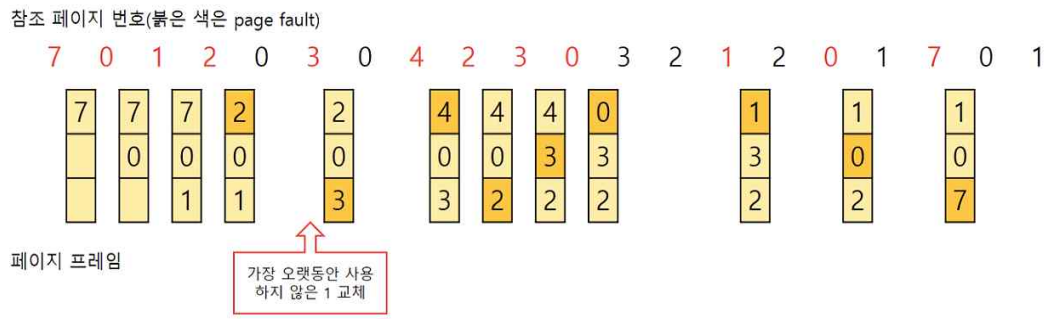


그림 11 LRU 알고리즘

1) 페이지 프레임의 수가 증가함에도 불가하고 페이지 폴트 발생 횟수가 줄어들지 않고 오히려 증가하는 현상

LRU 알고리즘은 가장 오랫동안 사용되지 않은 페이지를 교체한다. 최근 사용된 페이지는 자주 사용될 가능성이 크다는 가정에 기반한다. 구현 방식으로는 두 가지가 있다. 첫 번째로는 모든 페이지를 스택 형태로 관리하며, 가장 최근에 사용된 페이지를 스택의 맨 위로 이동시키는 방식이다. 두 번째 방식은 각 페이지에 타임스탬프를 저장하고, 가장 오래된 타임스탬프를 가진 페이지를 교체하는 것이다. LRU는 효과적인 성능을 제공하지만, 스택 유지 또는 타임스탬프 관리와 같은 구현 비용이 높을 수 있다.

- Clock Algorithm

  Clock 알고리즘은 LRU의 변형으로, 원형 큐와 참조 비트(reference bit)를 사용한다. 참조 비트는 페이지가 최근에 사용되었는지를 나타내며, 초기에는 모두 0으로 설정된다. 페이지가 참조되면 해당 비트는 1로 설정된다. 교체 대상 선택 시, 참조 비트가 0인 페이지를 희생 페이지로 선정하며, 비트가 1인 경우 0으로 리셋하고 다음 페이지로 이동한다. 이 방식은 LRU보다 구현이 간단한고 메모리 접근 비용이 낮아 현실적으로 많이 사용한다.

- LFU (Least Frequently Used)

  LFU 알고리즘은 가장 적게 참조된 페이지를 교체한다. 각 페이지의 참조 횟수를 기록하며, 참조 횟수가 가장 적은 페이지를 희생 페이지로 선정한다. 그러나 오래된 페이지가 참조 횟수를 누적하여 교체되지 않는 문제점이 있을 수 있다. 이를 해결하기 위해 참조 횟수를 주기적으로 초기화하거나 가중치를 부여하는 변형된 방법이 활용된다.

- MFU (Most Frequently Used)

  MFU 알고리즘은 LFU의 반대 개념으로, 참조 횟수가 가장 많은 페이지를 교체한다. 이 알고리즘은 최근 참조된 페이지는 다시 사용될 가능성이 크다는 LRU의 가정과 상반되는 논리를 따른다. MFU는 특정 상황에서 효율적일 수 있으나 일반적으로 자주 사용되지는 않는다.

효율적인 페이지 교체는 요구 페이징 환경에서 시스템 성능을 극대화하

는 핵심 요소이다. 각 알고리즘은 특정 워크로드에 적합한 특성을 가지며, 적절한 알고리즘 선택은 페이지 폴트율을 최소화하고 메모리 자원의 활용도를 높인다. 현대 운영체제는 하나의 알고리즘에 의존하지 않고, 다양한 접근법을 조합하여 시스템 전반의 성능을 최적화한다.

4. Implementation

해당 과제의 목적은 메모리 관리와 운영체제의 핵심 기능을 이해하고 이를 구현하는 데 있다. 특히 페이지 교체, 스와핑, Copy-on-Write 등 고급 메모리 관리 기법을 통해 메모리 제한 환경에서의 효율적인 자원 활용을 시뮬레이션하고자 한다. 과제는 단계별로 세분화되어 있으며, 각 단계는 메모리 관리 시스템의 주요 기능을 구현하는 것을 목표로 한다. 먼저 물리 메모리와 페이지 프레임을 안정적으로 관리하는 메모리 관리 기능을 시작으로, 프로세스 관리, 페이지 테이블 관리, 스와핑, Copy-on-Write와 같은 고급 기법을 순차적으로 구현하였다. 해당 부분에서는 각 단계의 주요 알고리즘과 데이터 구조의 동작 원리를 기술할 예정이다.

4-1. 함수 및 변수 목록

본 과제에서 메모리 관리 시스템 구현을 위해 여러 파일에 걸쳐 다양한 함수와 변수를 정의했다. 각각의 파일은 특정 역할을 수행한다. 해당 장에서는 구현된 함수의 역할과 변수의 용도를 설명할 예정이다.

함수 종합 테이블

파일 이름	함수 이름	파라미터	기능
ipc_manager.c	createMessageQueue	-	메시지 큐 생성
	sendMessage	int msgid, Message* msg	메시지 큐에 메시지 전송
	receiveMessage	int msgid, Message* msg, int type	특정 타입의 메시지를 메시지 큐에서 수신

파일 이름	함수 이름	파라미터	기능
memory_manager.c	init_memory	-	물리 메모리와 디스크 초기화
	free_memory	-	메모리 및 디스크 메모리 해제
	read_memory	uint32_t virtual_address, uint32_t* data	가상 주소에서 4바이트 읽기
	write_memory	uint32_t virtual_address, uint32_t data	가상 주소에 4바이트 쓰기
	handle_page_fault	uint32_t virtual_address	페이지 폴트 처리
page_replacement.c	init_replacement_manager	PageReplacementManager* manager, ReplacementPolicy policy	페이지 교체 매니저 초기화
	replace_page	PageReplacementManager* manager, uint32_t virtual_address, uint32_t frame_number, FirstPageTable* page_table, uint8_t* physical_memory, uint8_t* disk_storage	페이지 교체
	free_replacement_manager	PageReplacementManager* manager	교체 매니저 메모리 해제
swap_manager.c	init_swap_manager	-	디스크 및 디스크 블록 초기화
	free_swap_manager	-	디스크 및 디스크 블록 메모리 해제
	swap_in_page	uint8_t* disk, uint8_t* memory, uint32_t frame_number, uint32_t disk_index	디스크 데이터를 물리 메모리에 로드
	swap_out_page	uint8_t* disk, uint8_t* memory, uint32_t frame_number	물리 메모리 데이터를 디스크로 저장
page_table.c	init_page	uint32_t first_level_size, uint32_t second_level_size	페이지 테이블 초기화
	alloc_page	FirstPageTable* pt, uint32_t virtual_address, uint32_t frame_number	가상 주소에 페이지 할당

파일 이름	함수 이름	파라미터	기능
page_table.c	free_page	FirstPageTable* pt	페이지 테이블 메모리 해제
	is_page_full	SecondPageTable* second_table	페이지 테이블이 가득 찼는지 확인
	print_page	FirstPageTable* pt	페이지 테이블 내용을 출력
tlb.c	init_tlb	TLB* tlb, int size	TLB 초기화
	search_tlb	TLB* tlb, uint32_t page_number, uint32_t* frame_number	TLB에서 페이지를 검색하고 프레임 번호 반환
	update_tlb	TLB* tlb, uint32_t page_number, uint32_t frame_number	TLB 갱신 및 LRU 교체
	free_tlb	TLB* tlb	TLB 메모리 해제
	print_tlb	TLB* tlb	TLB 내용 출력
read_tar.c	load_tar_to_memory	const char* tar_file_path, uint8_t** memory, size_t* memory_size	TAR 파일 데이터를 메모리로 로드

변수 종합 테이블

파일 이름	함수 이름	파라미터	기능
ipc_manager.c	Message	구조체	메시지 정보 (타입, PID, 실행 시간, 대기 시간 등)
memory_manager.c	physical_memory	uint8_t* (포인터)	물리 메모리를 나타냄
	disk_storage	uint8_t* (포인터)	디스크 공간 (백업 저장소)
page_replacement.c	ReplacementPolicy	열거형(enum)	페이지 교체 정책 (FIFO, LRU 등)
	PageReplacementManager	구조체	교체 매니저: 정책, 사용 중인 프레임, 최대 프레임 수, 시계 핸들 포함
swap_manager.c	disk	static uint8_t*	디스크 공간

파일 이름	함수 이름	파라미터	기능
swap_manager.c	free_disk_blocks	static int*	디스크 블록 사용 상태 (0: 사용 가능, 1: 사용 중)
	disk_block_count	static int	디스크 블록의 총 개수
page_table.c	PageTableEntry	구조체	페이지 테이블 엔트리: 프레임 번호, 유효 비트, 수정 비트 포함
	SecondPageTable	구조체	2단계 페이지 테이블: 엔트리 배열 및 테이블 크기 포함
	FirstPageTable	구조체	1단계 페이지 테이블: 2단계 테이블 포인터 배열 및 크기 포함
tlb.c	TLBEntry	구조체	TLB 엔트리: 가상 페이지 번호, 물리 프레임 번호, 유효 비트, 마지막 사용 시간 포함
	TLB	구조체	TLB 구조체: 엔트리 배열, 크기, 현재 시간 포함
read_tar.c	memory	uint8_t*	TAR 파일 데이터를 저장할 메모리 공간
	memory_size	size_t	TAR 파일의 크기

4-2. 메모리 관리 (memory_manager)

해당 부분은 가상 메모리 관리 시스템을 구축하는 것을 중점으로 하며, 메모리 초기화, TLB 관리, 페이지 교체, 스와핑 등의 핵심 기능을 구현한다. 초기화 단계에서 메모리 공간과 페이지 테이블, TLB를 설정하고, 메모리에 접근 시 TLB에서 페이지를 검색한다. TLB 미스 발생 시 페이지 테이블을 통해 필요한 페이지를 불러오고, 메모리가 부족하면 FIFO 알고리즘으로 교체할 페이지를 선택해 디스크로 내보낸다. 스와핑 작업으로 페이지를 로드하며 폴트를 처리해 시스템 성능을 최적화한다.

- TLB 관리

TLB는 주서 변환 속도를 높이기 위한 캐시 구조이다. search_tlb와 update_tlb 함수는 TLB 히트/미스 처리와 캐시 정책 구현을 담당한다.

```c
int search_tlb(uint32_t va) {
    // TLB 검색을 위해 가상 주소를 분할
    uint32_t page_number = va >> offset_bits;

    // TLB에서 검색
    for (int i = 0; i < tlb->size; i++) {
        if (tlb->entries[i].valid && tlb->entries[i].page_number == page_number) {
            // TLB 히트 처리
            //printf("TLB Hit: VA=0x%X -> Index=%d\n", va, i);
            tlb->entries[i].last_used = tlb->current_time++;
            return i; // 히트된 TLB 인덱스 반환
        }
    }

    // TLB 미스 처리
    //printf("TLB Miss: VA=0x%X\n", va);
    return -1; // 미스 시 -1 반환
}
```

그림 12 search_tlb 부분

이 함수는 TLB 검색을 통해 주어진 가상 주소에 해당하는 페이지 번호가 이미 메모리의 TLB에 존재하는지 확인한다. TLB 히트 시 빠르게 인덱스를 반환하고, 미스 시 -1을 반환하여 페이지 테이블을 조회할 필요성을 알린다. TLB의 효율적인 사용과 LRU 기반 시간 추적을 통해 메모리 관리 기능을 최적화하는 데 중요한 역할을 한다.

```c
void update_tlb(uint32_t va, uint32_t frame_number, uint8_t read_only) {
    // 가상 주소에서 page_number 계산
    uint32_t page_number = va >> offset_bits; // va에서 offset_bits만큼 시프트하여 page_number 추출

    int replace_index = -1;

    // TLB에서 비어 있는 엔트리 확인 (full인지 체크)
    for (int i = 0; i < tlb->size; i++) {
        if (tlb->entries[i].valid == 0) {
            replace_index = i; // 비어 있는 엔트리 발견
            break;
        }
    }

    // TLB가 가득 찼다면 교체 정책 적용
    if (replace_index == -1) {
        if (tlb_cache == 0) { // RANDOM
            replace_index = rand() % tlb->size;
        } else if (tlb_cache == 1) { // FIFO
            replace_index = tlb->fifo_index;
            tlb->fifo_index = (tlb->fifo_index + 1) % tlb->size; // 순환 인덱스 갱신
        } else if (tlb_cache == 2) { // LRU
            uint32_t min_time = UINT32_MAX;
            for (int j = 0; j < tlb->size; j++) {
                if (tlb->entries[j].last_used < min_time) {
                    min_time = tlb->entries[j].last_used;
                    replace_index = j;
                }
            }
        } else if (tlb_cache == 3) { // LFU
            uint32_t min_count = UINT32_MAX;
            for (int j = 0; j < tlb->size; j++) {
                if (tlb->entries[j].access_count < min_count) {
                    min_count = tlb->entries[j].access_count;
                    replace_index = j;
                }
            }
        } else {
            //printf("Invalid TLB cache policy.\n");
            return;
        }

        // 교체할 엔트리 초기화
        init_tlb_entry(&tlb->entries[replace_index]);
    }

    // TLB 엔트리 갱신
    tlb->entries[replace_index].page_number = page_number;
    tlb->entries[replace_index].frame_number = frame_number;
    tlb->entries[replace_index].valid = 1;
    tlb->entries[replace_index].last_used = tlb->current_time++;
    tlb->entries[replace_index].access_count = 1; // 초기화
    tlb->entries[replace_index].read_only = read_only; // 읽기 전용 플래그 설정

    //printf("TLB Updated: Policy=%d, Index=%d, Page=0x%X, Frame=%u\n",tlb_cache, replace_index, page_number, frame_number);
}
```

그림 13 update_tlb 부분

이 함수는 TLB의 엔트리를 업데이트하고, 가상 주소(va)에 해당하는 페이지 번호를 저장하거나, TLB가 가득 찬 경우 교체 정책을 통해 기존 엔트리를 교체한다. TLB는 제한된 공간에서 자주 사용하는 페이지를 빠르게 검색하기 위해 사용되며, 공간이 꽉 찼을 때는 적합한 교체 알고리즘이 필요하다. 이 코드에서는 랜덤 교체, FIFO, LRU, LFU를 지원하며, 워크로드에 따라 적합한 정책을 선택할 수 있다.

엔트리 교체 시 기존 엔트리를 초기화한 후 새로운 페이지 정보를 저장하여, 갱신된 상태로 TLB를 준비한다. 이를 통해 페이지 정보가 효율적으로 관리되고 시스템 성능을 최적화한다.

- **페이지 교체 정책**

```c
int select_victim() {
    int victim_index = -1;

    if (page_replace == 0) { // FIFO
        // FIFO 정책: MemoryMap에서 가장 낮은 arrival_time을 가진 프레임 선택
        uint32_t min_time = UINT32_MAX;
        for (int i = 0; i < frame_count; i++) {
            if (memory_map[i].is_valid && memory_map[i].arrival_time < min_time) {
                min_time = memory_map[i].arrival_time;
                victim_index = i;
            }
        }
    }
    else if (page_replace == 1) { // LRU
        uint32_t min_time = UINT32_MAX;
        for (int i = 0; i < frame_count; i++) {
            if (memory_map[i].is_valid && memory_map[i].last_used < min_time) {
                min_time = memory_map[i].last_used;
                victim_index = i;
            }
        }
    }
    else if (page_replace == 2) { // LFU
        uint32_t min_count = UINT32_MAX;
        for (int i = 0; i < frame_count; i++) {
            if (memory_map[i].is_valid && memory_map[i].access_count < min_count) {
                min_count = memory_map[i].access_count;
                victim_index = i;
            }
        }
    }
    else if (page_replace == 3) { // SECOND_CHANCE
        static int clock_hand = 0; // CLOCK 알고리즘 포인터
        while (1) {
            if (memory_map[clock_hand].is_valid) {
                if (memory_map[clock_hand].reference_bit == 0) {
                    victim_index = clock_hand;
                    clock_hand = (clock_hand + 1) % frame_count;
                    break;
                } else {
                    memory_map[clock_hand].reference_bit = 0; // 참조 비트 초기화
                }
            }
            clock_hand = (clock_hand + 1) % frame_count;
        }
    } else {
        //printf("Invalid page replacement policy.\n");
    }

    return victim_index;
}

int find_disk_block() {
    for (int i = 0; i < MAX_DISK_SIZE/page_size; i++) {
        if (!disk_map[i].is_valid) {
            return i; // 빈 블록의 인덱스를 반환
        }
    }
    return -1; // 빈 블록을 찾을 수 없음
}
```

그림 14 select_victim 부분

select_victim 함수는 메모리가 가득 찬 상황에서 교체할 페이지를 선택하기 위해 페이지 교체 정책을 기반으로 적절한 페이지를 결정하는 함수이다. 각 정책은 메모리 사용 패턴을 분석하여 효율적으로 희생 페이지를 선택하는데 초점이 맞춰져 있다. 함수는 page_replace 값을 통해 사용할 교체 알고리즘을 결정하며, 주요 로직은 다음과 같다.

FIFO : memory_map의 arrival_time 값을 기준으로, 가장 낮은 값을 가진 페이지를 희생 페이지로 선택

LRU : memory_map의 last_used 값을 기준으로, 가장 작은 값을 가진 페이지를 희생 페이지로 선택함

LFU : memory_map의 access_count 값을 기준으로, 가장 작은 값을 가진 페이지를 희생 페이지로 선택

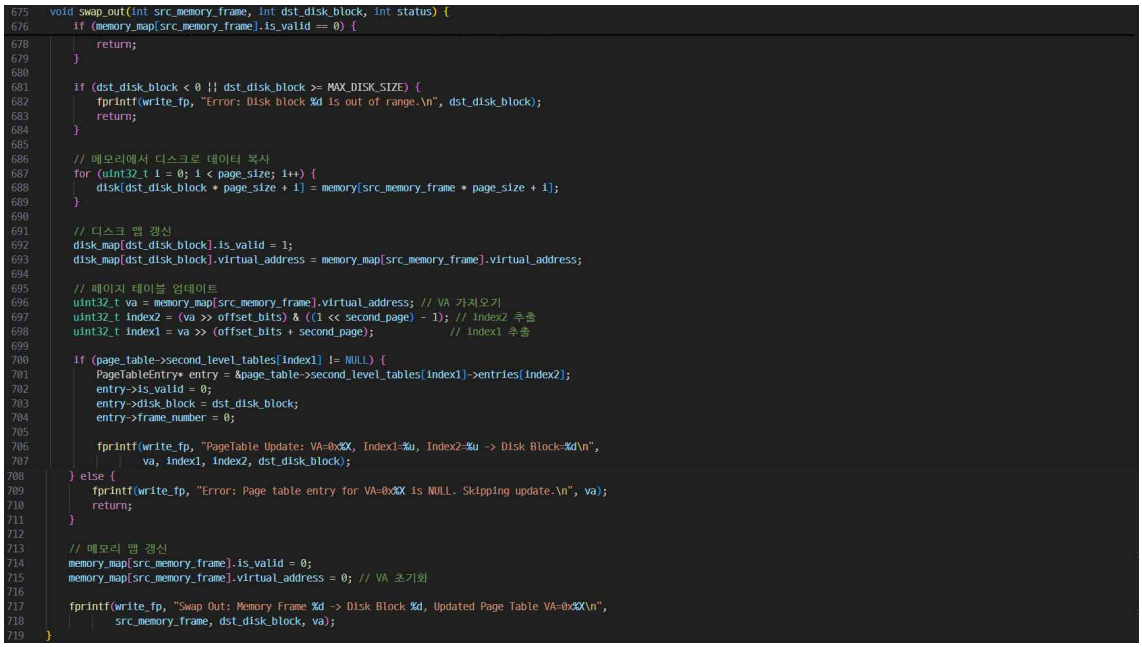
Second-Chance (Clock 알고리즘) : 참조 비트를 활용하여 교체할 페이지를 선택하는 알고리즘으로 clock_hand 변수를 사용하여 구조를 순회하며, 참조 비트가 0인 페이지를 희생 페이지로 선택한다. 참조비트가 1인 경우 해당 비트를 0으로 초기화하고 다음 페이지를 확인한다.

- **스와핑**

```
212 void swap_in(int src_disk_block, int dst_memory_frame) {
213     if (!disk_map[src_disk_block].is_valid) {
214         //printf("Error: Disk block %d is not valid.\n", src_disk_block);
215         return;
216     }
217 
218     // 디스크에서 메모리로 데이터 복사
219     memcpy(&memory[dst_memory_frame * page_size], &disk[src_disk_block * page_size], page_size);
220 
221     // 메모리 맵 갱신
222     memory_map[dst_memory_frame].is_valid = 1; // 해당 프레임 유효
223     memory_map[dst_memory_frame].virtual_address = disk_map[src_disk_block].virtual_address; // 매핑된 VA
224 
225     // 정책에 따른 필드 초기화
226     if (page_replace == 0) { // FIFO
227         memory_map[dst_memory_frame].arrival_time = access_counter; // 현재 접근 카운터로 설정
228     } else if (page_replace == 1) { // LRU
229         memory_map[dst_memory_frame].last_used = access_counter; // 최근 사용 시간 업데이트
230     } else if (page_replace == 2) { // LFU
231         memory_map[dst_memory_frame].access_count = 0; // 접근 횟수 초기화
232     } else if (page_replace == 3) { // SECOND_CHANCE
233         memory_map[dst_memory_frame].reference_bit = 0; // 참조 비트 초기화
234     }
235 
236     // 디스크 맵 갱신
237     disk_map[src_disk_block].is_valid = 0; // 디스크 블록 비우기
238 
239 
240     // 페이지 테이블 업데이트
241     uint32_t va = disk_map[src_disk_block].virtual_address; // 디스크에서 가져온 VA
242     uint32_t index2 = (va >> offset_bits) & ((1 << second_page) - 1); // index2 추출
243     uint32_t index1 = va >> (offset_bits + second_page);            // index1 추출
244 
245     if (page_table->second_level_tables[index1] != NULL) {
246         page_table->second_level_tables[index1]->entries[index2].is_valid = 1;  // 페이지 테이블 유효성 활성화
247         page_table->second_level_tables[index1]->entries[index2].frame_number = dst_memory_frame; // 메모리 프레임 번호 갱신
248     } else {
249         //printf("Error: Page table entry for VA=0x%X is NULL.\n", va);
250     }
251 
252     //printf("Swap In: Disk Block %d -> Memory Frame %d\n", src_disk_block, dst_memory_frame);
253 }
```

그림 15 swap_in 부분

swap_in 함수는 디스크에서 메모리로 페이지를 로드하여 페이지 폴트를 처리하는 스와핑 작업을 수행한다. 디스크 블록의 유효성을 확인한 후, memcpy를 사용해 데이터를 메모리로 복사하고, 메모리 맵을 갱신하여 해당 프레임의 유효성을 활성화하고 가상 주소와 매핑 정보를 설정한다. 페이지 교체 정책에 따라 관련 필드를 초기화하고, 디스크 맵은 복사된 블록의 유효성을 비활성화한다. 또한, 2단계 페이지 테이블 구조를 사용해 페이지 테이블을 갱신하고 주소 변환을 정확히 수행한다. 이를 통해 스와핑과 데이터 동기화를 통해 메모리 관리를 최적화하고 프로세스의 연속성을 보장한다.



```
void swap_out(int src_memory_frame, int dst_disk_block, int status) {
    if (memory_map[src_memory_frame].is_valid == 0) {
        return;
    }

    if (dst_disk_block < 0 || dst_disk_block >= MAX_DISK_SIZE) {
        fprintf(write_fp, "Error: Disk block %d is out of range.\n", dst_disk_block);
        return;
    }

    // 메모리에서 디스크로 데이터 복사
    for (uint32_t i = 0; i < page_size; i++) {
        disk[dst_disk_block * page_size + i] = memory[src_memory_frame * page_size + i];
    }

    // 디스크 맵 갱신
    disk_map[dst_disk_block].is_valid = 1;
    disk_map[dst_disk_block].virtual_address = memory_map[src_memory_frame].virtual_address;

    // 페이지 테이블 업데이트
    uint32_t va = memory_map[src_memory_frame].virtual_address; // VA 가져오기
    uint32_t index2 = (va >> offset_bits) & ((1 << second_page) - 1); // index2 추출
    uint32_t index1 = va >> (offset_bits + second_page);              // index1 추출

    if (page_table->second_level_tables[index1] != NULL) {
        PageTableEntry* entry = &page_table->second_level_tables[index1]->entries[index2];
        entry->is_valid = 0;
        entry->disk_block = dst_disk_block;
        entry->frame_number = 0;

        fprintf(write_fp, "PageTable Update: VA=0x%X, Index1=%u, Index2=%u -> Disk Block=%d\n",
                va, index1, index2, dst_disk_block);
    } else {
        fprintf(write_fp, "Error: Page table entry for VA=0x%X is NULL. Skipping update.\n", va);
        return;
    }

    // 메모리 맵 갱신
    memory_map[src_memory_frame].is_valid = 0;
    memory_map[src_memory_frame].virtual_address = 0; // VA 초기화

    fprintf(write_fp, "Swap Out: Memory Frame %d -> Disk Block %d, Updated Page Table VA=0x%X\n",
            src_memory_frame, dst_disk_block, va);
}
```

그림 16 swap_out 부분

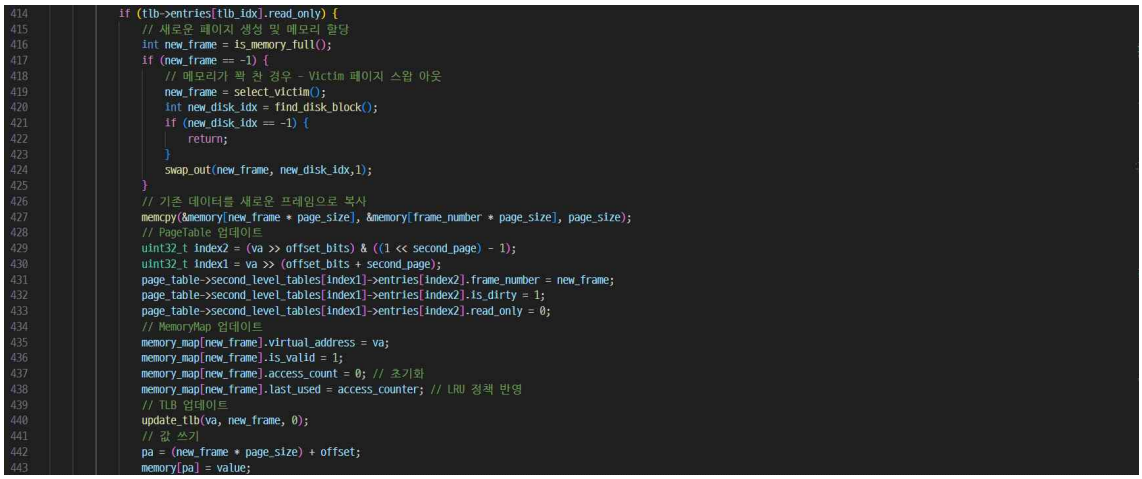
swap_out 함수는 메모리에서 디스크로 페이지를 저장하는 작업을 수행한다. 먼저, 메모리 프레임과 디스크 블록의 유효성을 확인하고, 메모리에서 디스크로 데이터를 복사한다. 그 후, 디스크 맵과 페이지 테이블을 갱신하여 페이지가 메모리에서 디스크로 이동했음을 반영한다. 메모리 맵에서 해당 프레임의 유효성을 비활성화하고, 로그 파일에 작업 결과를 기록한다. 이 함수는 페이지 교체와 데이터 동기화를 통해 가상 메모리 시스템에서 중요한 역할을 하며, 시스템의 안정성과 효율성을 보장한다.

- Copy-on-Write (CoW)

write_va 함수는 가상 주소(va)에 주어진 값을 기록하는 작업을 처리하는 함수이다. 이 함수는 먼저 TLB를 검색하여 가상 주소에 해당하는 물리 주소를 찾고 TLB에 히트가 발생하면 바로 해당 페이지에 값을 쓰고 페이지 테이블 및 메모리 맵을 갱신한다. 만약 TLB 미스가 발생하면, 페이지 테이블을 검색하여 필요한 페이지가 메모리에 있는지 확인하고, 없다면 스와핑 작업을 통해 페이지를 메모리로 로드한 후 값을 기록한다.

핵심 기술은 TLB 검색 및 히트 처리, 메모리 정책, CoW가 사용되었다. TLB 검색 및 히트 처리, 메모리 정책은 앞서 설명한 내용이 있으니 CoW 기능 구현 부분만 설명하도록 하겠다.

write_va 함수 내에서 CoW는 read_only 플래그가 설정된 페이지에 쓰기 작업을 시도할 때 발생한다. 페이지가 read_only로 설정된 상태에서 가상 주소로 값을 쓰려 할 때, 새로운 메모리 프레임을 할당하고 기존 페이지의 데이터를 새로운 프레임으로 복사한 후, read_only 플래그를 해제하여 새로운 페이지에서 값을 기록할 수 있도록 한다. 또한, page_table과 memory_map을 갱신하여 새로운 페이지가 정상적으로 매핑되고, 데이터를 기록할 수 있게 한다.



```
414             if (tlb->entries[tlb_idx].read_only) {
415                 // 새로운 페이지 생성 및 메모리 할당
416                 int new_frame = is_memory_full();
417                 if (new_frame == -1) {
418                     // 메모리가 꽉 찬 경우 - Victim 페이지 스왑 아웃
419                     new_frame = select_victim();
420                     int new_disk_idx = find_disk_block();
421                     if (new_disk_idx == -1) {
422                         return;
423                     }
424                     swap_out(new_frame, new_disk_idx,1);
425                 }
426                 // 기존 데이터를 새로운 프레임으로 복사
427                 memcpy(&memory[new_frame * page_size], &memory[frame_number * page_size], page_size);
428                 // PageTable 업데이트
429                 uint32_t index2 = (va >> offset_bits) & ((1 << second_page) - 1);
430                 uint32_t index1 = va >> (offset_bits + second_page);
431                 page_table->second_level_tables[index1]->entries[index2].frame_number = new_frame;
432                 page_table->second_level_tables[index1]->entries[index2].is_dirty = 1;
433                 page_table->second_level_tables[index1]->entries[index2].read_only = 0;
434                 // MemoryMap 업데이트
435                 memory_map[new_frame].virtual_address = va;
436                 memory_map[new_frame].is_valid = 1;
437                 memory_map[new_frame].access_count = 0; // 초기화
438                 memory_map[new_frame].last_used = access_counter; // LRU 정책 반영
439                 // TLB 업데이트
440                 update_tlb(va, new_frame, 0);
441                 // 값 쓰기
442                 pa = (new_frame * page_size) + offset;
443                 memory[pa] = value;
```

그림 17 write_va 함수 내 CoW 구현 부분

이 부분에서 read_only 상태의 페이지에 쓰기 작업이 발생하면, new_frame을 할당하여 새로운 메모리 공간을 확보하고, 기존 데이터를 새로운 페이지로 복사한 후 read_only를 해제하고 데이터를 기록한다. 이에

따라, 기존 페이지는 수정되지 않고, 새로운 페이지에 값이 기록된다.

- **read_va와 write_va 함수**

read_va와 write_va 함수는 운영체제에서 가상 메모리 시스템을 구현하는 데 중요한 역할을 하며, 이들 함수는 가상 주소를 통해 실제 메모리에 접근하는 작업을 수행한다. 이 함수들은 페이지 테이블, TLB, 페이지 교체 정책 등을 고려하여 메모리에서 데이터를 읽고 쓰는 작업을 처리한다. 이를 통해 가상 메모리 공간과 물리 메모리 공간을 효율적으로 매핑하고 관리할 수 있다. 해당 함수에서 사용되는 핵심적인 기술들은 위에서 설명했으므로 간단한 처리 흐름에 관해서만 설명하도록 하겠다.

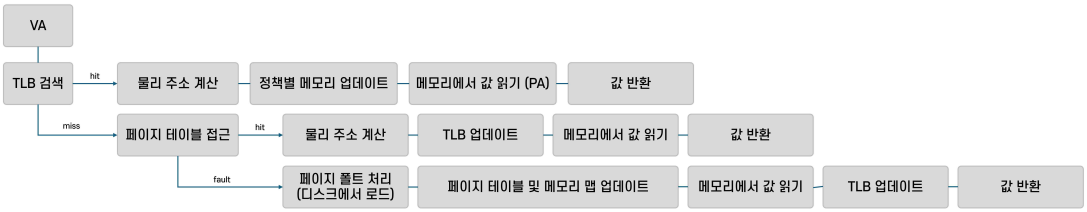


그림 18 read_va 함수의 플로우 차트

read_va 함수는 가상 주소를 입력받아 해당 주소에서 데이터를 읽는 함수이다. 이 함수의 동작은 위의 이미지처럼 요약할 수 있고 자세한 함수의 동작을 다음과 같이 단계별로 분석할 수 있다.

TLB 접근 : search_tlb 함수를 사용하여 가상 주소가 TLB에 존재하는지 확인하고, 존재하면 TLB 히트를 처리하여 물리 주소를 빠르게 계산한다.
페이지 테이블 접근 : TLB 미스가 발생하면, 페이지 테이블을 검색하여 가상 주소에 대응하는 물리 주소를 찾는다. 페이지 테이블에서 해당 주소가 유효하다면 물리 주소를 계산하여 데이터를 읽는다.
정책에 따른 메모리 업데이트 : 메모리 접근 후 페이지 교체 정책에 따라 메모리 맵을 업데이트한다.
TLB 업데이트 : 페이지 테이블을 통해 물리 주소를 계산한 후, TLB를 갱신하여 후속 접근 시 더 빠르게 데이터를 찾을 수 있게 한다.

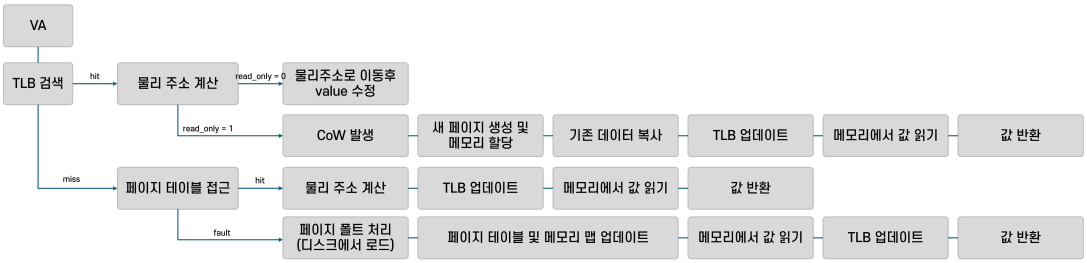


그림 19 wirte_va 함수의 플로우 차트

wirte_va 함수는 read_va와 유사하게 가상 주소에 데이터를 쓰는 작업을 수행하지만, 쓰기 작업에서는 추가적인 고려 사항이 존재한다. 특히, 읽기 전용 페이지에 쓰기 작업이 발생할 때 CoW 기법이 사용된다. 지금부터 read_va 함수와 다른 점을 중점으로 wirte_va의 흐름에 관해 설명하도록 하겠다.

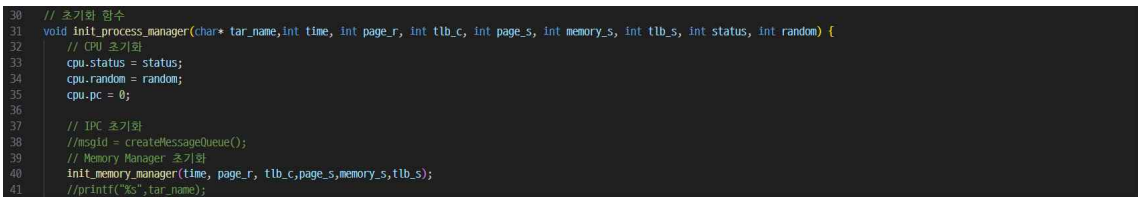
TLB 접근 : TLB 히트가 발생하면, 해당 페이지가 읽기 전용이라면 CoW가 발생하여 새로운 페이지를 할당하고 기존 데이터를 복사한다.
CoW : 읽기 전용 페이지를 찾았을 경우, 새로운 페이지를 할당하고 기존 데이터를 복사한 후, 페이지 테이블과 메모리 맵을 갱신한다.
읽기 전용이 아닌 페이지 : 이에 대해서는 데이터를 직접 수정할 수 있다. 이 경우, 페이지 테이블을 갱신하여 해당 페이지가 수정되었음을 나타낸다.

read_va와 wirte_va 함수는 가상 메모리 시스템에서 중요한 역할을 하며, TLB와 페이지 테이블을 통해 효율적인 메모리 접근을 관리한다. TLB는 빠른 주소 변환을 위한 캐시 역할을 하며, 페이지 테이블은 가상 주소와 물리 주소를 매핑한다. 또한, 페이지 폴트가 발생할 경우에는 디스크와의 스와핑을 통해 필요한 데이터를 메모리로 불러오고, 이를 통해 운영체제는 실제 메모리 용량을 넘는 가상 메모리 공간을 효율적으로 관리할 수 있다.

4-3. 프로세스 관리 (process_manager)

해당 부분에서 설명할 함수는 init_process_manager이다. 이 함수는 운영 체제에서 프로세스를 관리하기 위한 초기 설정을 담당하는 중요한 함수이다. 이 함수는 프로세스 실행을 위한 메모리 초기화, 페이지 테이블 설정,

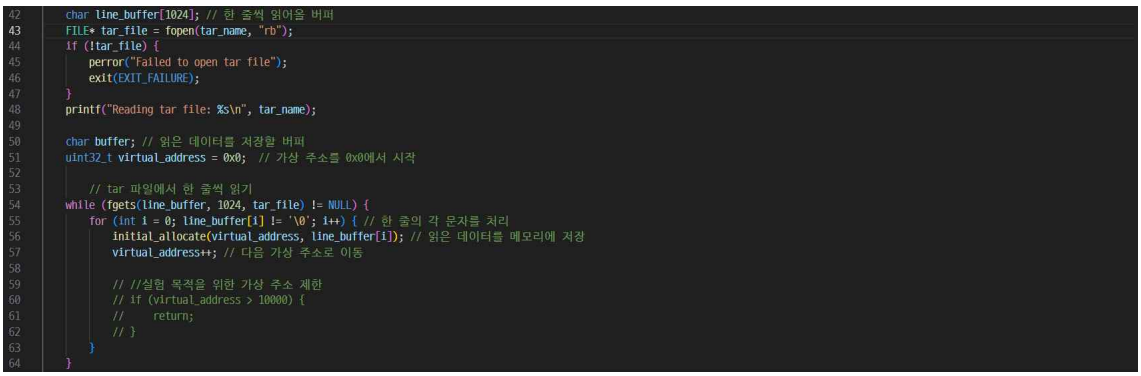
IPC 초기화, 그리고 tar 파일에서 데이터를 읽어 메모리에 할당하는 기능을 수행한다. 이 과정에서 메모리 관리 시스템과 페이지 테이블을 초기화하며, 이후 프로세스가 메모리 공간을 효과적으로 사용할 수 있도록 기반을 마련한다.



```
// 초기화 함수
void init_process_manager(char* tar_name,int time, int page_r, int tlb_c, int page_s, int memory_s, int tlb_s, int status, int random) {
    // CPU 초기화
    cpu.status = status;
    cpu.random = random;
    cpu.pc = 0;

    // IPC 초기화
    //msgid = createMessageQueue();
    // Memory Manager 초기화
    init_memory_manager(time, page_r, tlb_c,page_s,memory_s,tlb_s);
    //printf("%s",tar_name);
```

그림 20 메모리 및 페이지 테이블 초기화

init_process_manager 함수에서 config.txt 파일의 설정값을 기반으로 메모리와 디스크를 초기화하고, 가상 주소 공간을 설정한다. example.tar 파일 데이터를 읽어 initial_allocate 함수를 통해 메모리와 페이지 테이블에 로드한다.



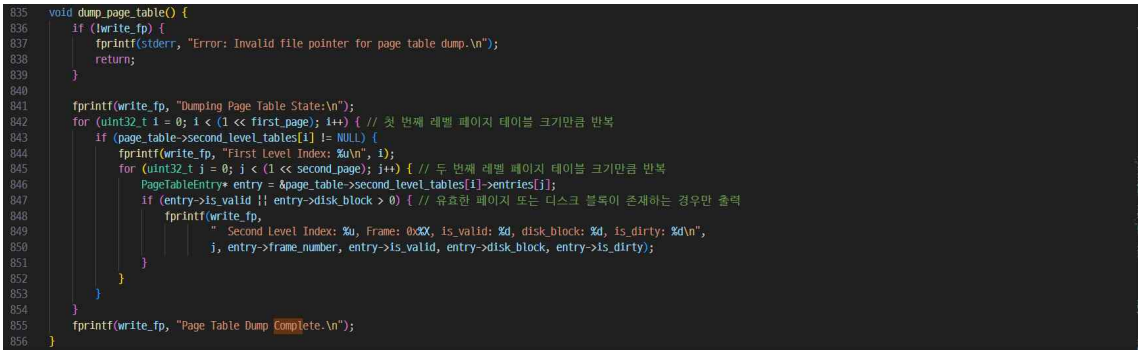
```
    char line_buffer[1024]; // 한 줄씩 읽어올 버퍼
    FILE* tar_file = fopen(tar_name, "rb");
    if (!tar_file) {
        perror("Failed to open tar file");
        exit(EXIT_FAILURE);
    }
    printf("Reading tar file: %s\n", tar_name);

    char buffer; // 읽은 데이터를 저장할 버퍼
    uint32_t virtual_address = 0x0;  // 가상 주소를 0x0에서 시작

        // tar 파일에서 한 줄씩 읽기
    while (fgets(line_buffer, 1024, tar_file) != NULL) {
        for (int i = 0; line_buffer[i] != '\0'; i++) { // 한 줄의 각 문자를 처리
            initial_allocate(virtual_address, line_buffer[i]); // 읽은 데이터를 메모리에 저장
            virtual_address++; // 다음 가상 주소로 이동

            // //실험 목적을 위한 가상 주소 제한
            // if (virtual_address > 10000) {
            //     return;
            // }
        }
    }
```

그림 21 파일 읽기 및 가상 주소 할당

example.tar 파일은 한 줄씩 읽어 각 문자를 가상 주소 공간에 저장한다. virtual_address는 0x0에서 시작해 데이터를 순차적으로 저장하며, 가상 주소 공간의 연속성을 보장한다. 이를 통해 프로세스 실행 중 메모리에 데이터를 효율적으로 저장할 수 있다.

```
66      fclose(tar_file);
67
68      printf("Tar file loaded to memory.\n");
69
70      uint32_t finish_idx = virtual_address;
71
72      dump_page_table();
73
74      printf("Page Table dumped to page_table_dump.txt\n");
```

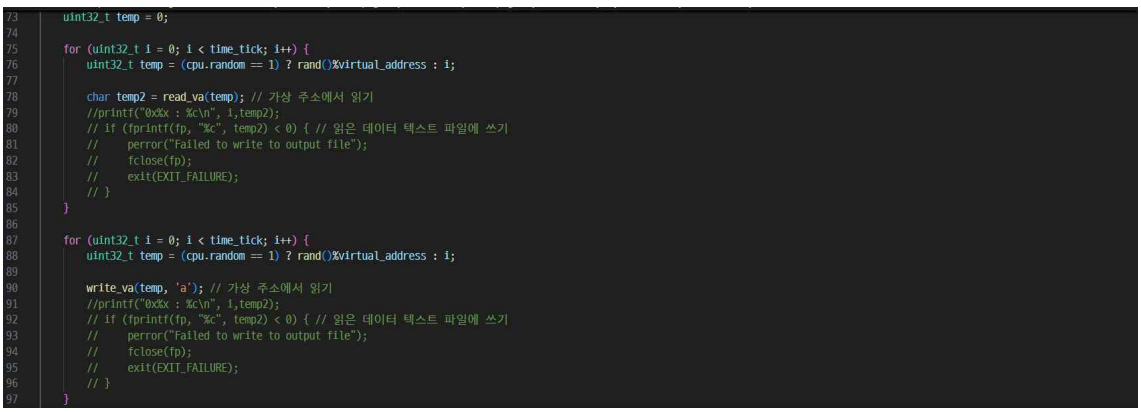
그림 22 페이지 테이블 덤프 부분 및 메모리 관리 해제 부분



```
835 void dump_page_table() {
836     if (!write_fp) {
837         fprintf(stderr, "Error: Invalid file pointer for page table dump.\n");
838         return;
839     }
840
841     fprintf(write_fp, "Dumping Page Table State:\n");
842     for (uint32_t i = 0; i < (1 << first_page); i++) { // 첫 번째 레벨 페이지 테이블 크기만큼 반복
843         if (page_table->second_level_tables[i] != NULL) {
844             fprintf(write_fp, "First Level Index: %u\n", i);
845             for (uint32_t j = 0; j < (1 << second_page); j++) { // 두 번째 레벨 페이지 테이블 크기만큼 반복
846                 PageTableEntry* entry = &page_table->second_level_tables[i]->entries[j];
847                 if (entry->is_valid || entry->disk_block > 0) { // 유효한 페이지 또는 디스크 블록이 존재하는 경우만 출력
848                     fprintf(write_fp,
849                             "  Second Level Index: %u, Frame: 0x%X, is_valid: %d, disk_block: %d, is_dirty: %d\n",
850                             j, entry->frame_number, entry->is_valid, entry->disk_block, entry->is_dirty);
851                 }
852             }
853         }
854     }
855     fprintf(write_fp, "Page Table Dump Complete.\n");
856 }
```

그림 23 memory_manager.c 내에 있는 dump_page_table 함수

dump_page_table 함수는 페이지 테이블의 상태를 확인하고 디버깅하기 위해 현재 메모리 및 디스크 메핑 정보를 파일에 출력한다. 2단계 페이지 테이블 구조를 기반으로 동작하며, 각 레벨의 페이지 테이블 엔트리 정보를 상세히 기록하여 시스템의 메모리 상태를 시각화한다. 해당 코드에서는 페이지 테이블의 상태를 page_table_dump.txt로 출력하여 디버깅에 활용된다.



```
73      uint32_t temp = 0;
74
75      for (uint32_t i = 0; i < time_tick; i++) {
76          uint32_t temp = (cpu.random == 1) ? rand()%virtual_address : i;
77
78          char temp2 = read_va(temp); // 가상 주소에서 읽기
79          //printf("0x%x : %c\n", i,temp2);
80          // if (fprintf(fp, "%c", temp2) < 0) { // 읽은 데이터 텍스트 파일에 쓰기
81          //     perror("Failed to write to output file");
82          //     fclose(fp);
83          //     exit(EXIT_FAILURE);
84          // }
85      }
86
87      for (uint32_t i = 0; i < time_tick; i++) {
88          uint32_t temp = (cpu.random == 1) ? rand()%virtual_address : i;
89
90          write_va(temp, 'a'); // 가상 주소에서 읽기
91          //printf("0x%x : %c\n", i,temp2);
92          // if (fprintf(fp, "%c", temp2) < 0) { // 읽은 데이터 텍스트 파일에 쓰기
93          //     perror("Failed to write to output file");
94          //     fclose(fp);
95          //     exit(EXIT_FAILURE);
96          // }
97      }
```

그림 24 메모리 접근 시뮬레이션

해당 부분은 config.txt에 정의된 매개 변수들을 기반으로 가상 메모리 시스템을 시뮬레이션한다. 주요 작업은 read_va와 write_va 함수를 사용해 가상 주소의 데이터를 읽고 쓰는 것으로, 이 과정에서 TLB와 페이지 교체 알고리즘의 성능을 실험할 수 있다. cpu.random 값은 접근 방식에 영향을

주는데, random이 1일 경우 난수 기반의 비순차적 메모리 접근을 시뮬레이션하고, random이 0일 경우 순차적으로 메모리 주소를 접근한다.

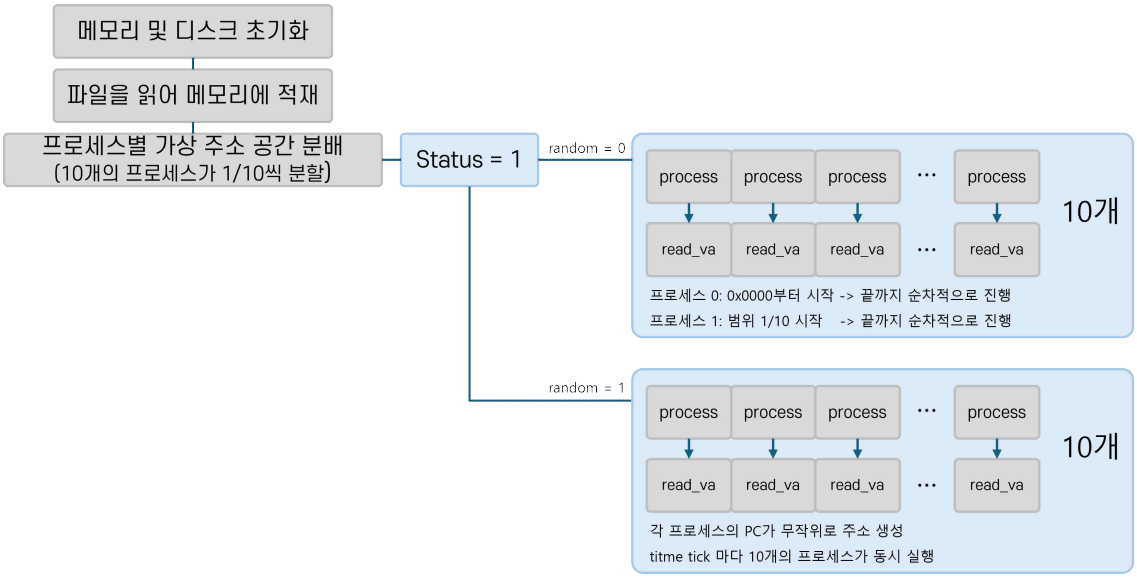


그림 25 상태가 1일 때 플로우 차트 (각각의 프로세스가 read_va 호출)

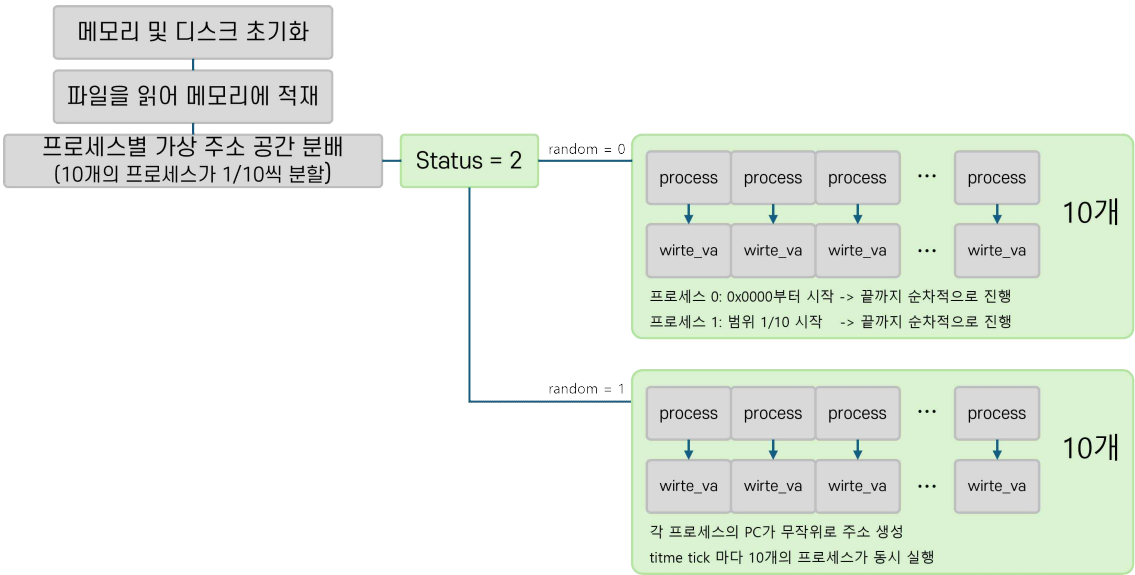


그림 26 상태가 2일 때 플로우 차트 (각각의 프로세스가 wirte_va 호출)

status 값에 따라 읽기와 쓰기 작업이 달라지며, 1일 때는 읽기, 2일 때는 쓰기, 3일 때는 읽기와 쓰기를 모두 수행한다. 4일 경우는 파이썬 스크립트를 통해 생성된 data.txt 파일에 정의된 메모리 접근 패턴을 따르게 된다. 이 접근 패턴은 실제 시스템에서 발생할 수 있는 클러스터화된 메모리 참조를 시뮬레이션한다. 반복 횟수는 time_tick으로 설정되어, 해당 값만큼 루프

가 반복되며 TLB와 페이지 교체 알고리즘의 성능을 평가한다.

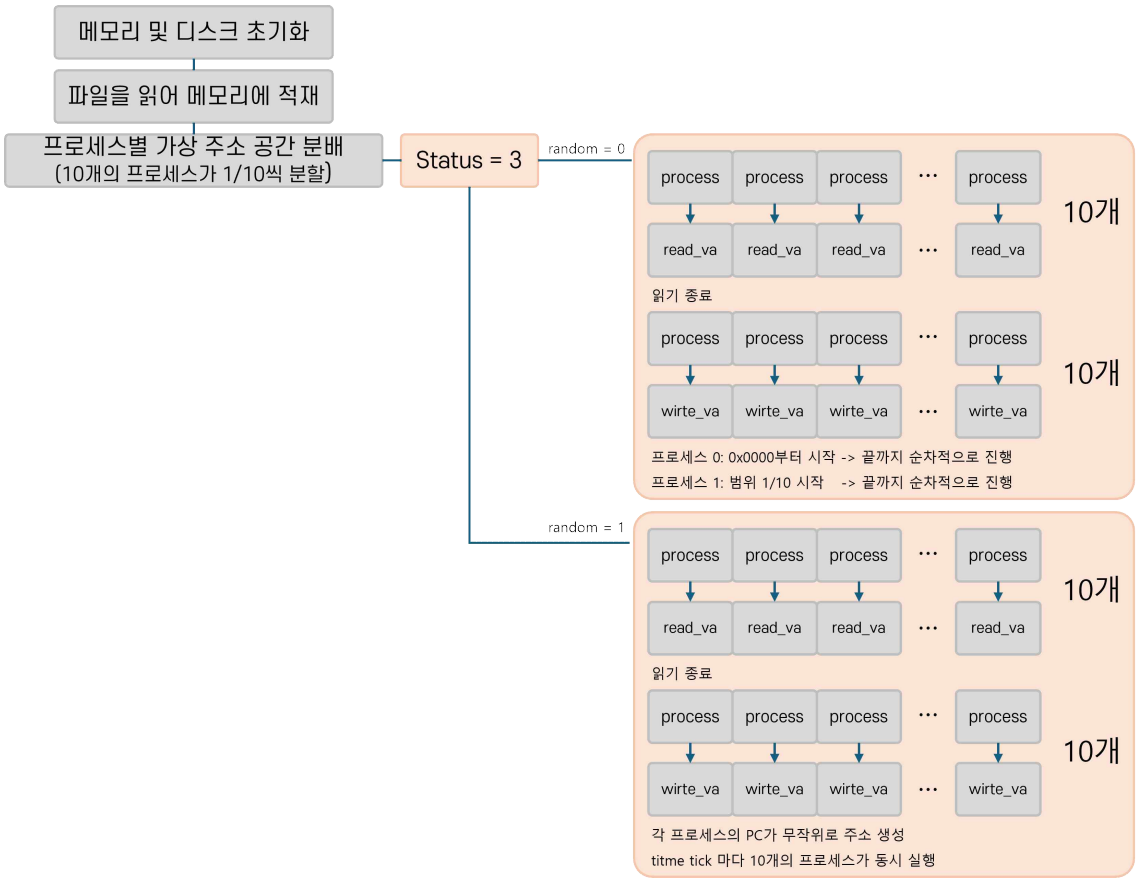


그림 27 상태가 3일 때 플로우 차트 (각각의 프로세스가 read_va, write_va 호출)

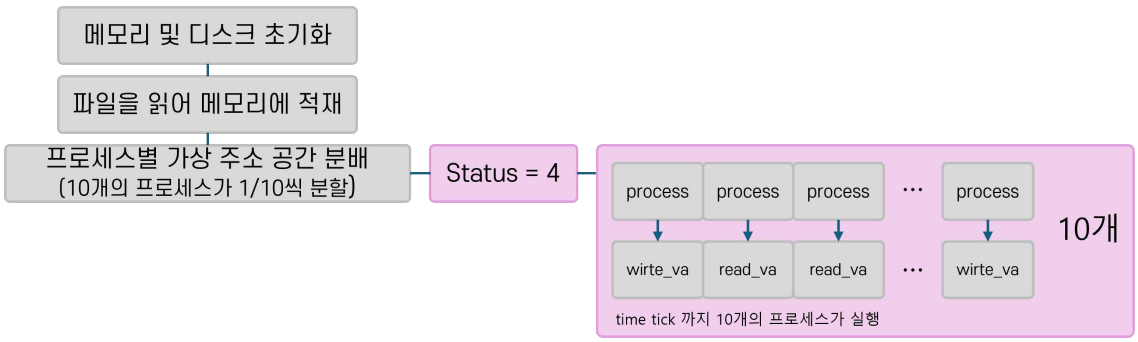


그림 28 상태가 4일 때 플로우 차트 (각각의 프로세스가 각자 필요한 함수 호출)

파이썬 스크립트를 통해 생성된 data.txt 파일은 가상 주소와 읽기/쓰기 작업 유형 및 쓰기 데이터를 정의하며, 메모리 접근 패턴을 시뮬레이션하는 데 사용된다. 이 데이터는 클러스터화된 접근 방식으로 지역성을 모사하거나, 비순차적이고 분산된 랜덤 접근 방식을 포함해 다양한 접근 시나리오를 제공한다.

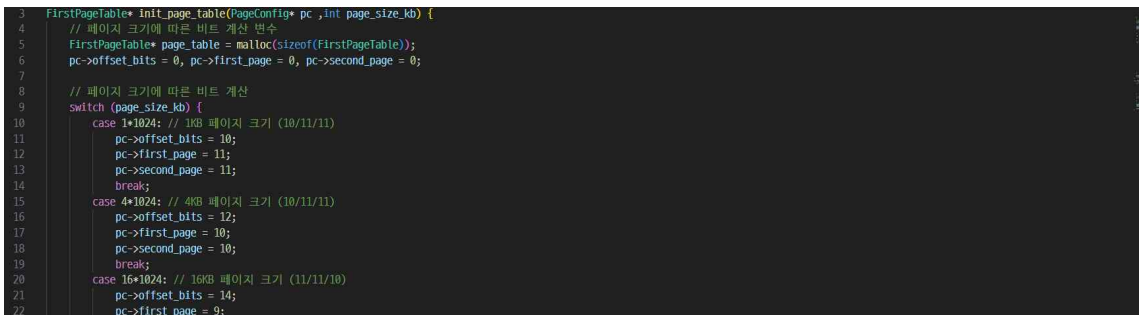
generate_memory_access_data_with_log 함수는 메모리 접근의 중심이 되는 클러스터를 설정하고, 지역성 기반 접근과 랜덤 접근을 결합하여 데이터를 생성한다. 이후, 각 접근에 대해 읽기 또는 쓰기 작업을 무작위로 결정하고, 결과 데이터를 data.txt 파일에 기록한다.

생성된 데이터는 각 프로세스가 이를 참조하여 read_va와 write_va 함수를 호출하는 방식으로 시뮬레이션된다. 이를 통해 실제 시스템의 메모리 접근 패턴을 모사하여 TLB와 페이지 교체 알고리즘의 성능을 분석할 수 있다.

read_va와 write_va는 TLB 캐시를 확인한 후 페이지 테이블을 조회하여 데이터를 읽거나 쓰며, 이 과정에서 발생하는 페이지 폴트와 TLB 미스를 처리한다.

4-4. 페이지 테이블 관리 (page_table)

페이지 테이블은 가상 메모리 시스템에서 중요한 역할을 담당하는 구조체이다. 운영체제가 프로세스의 가상 주소를 물리 주소로 변환하는 데 사용되는 페이지 테이블을 효율적인 메모리 관리와 주소 변환을 가능하게 한다.



```
FirstPageTable* init_page_table(PageConfig* pc ,int page_size_kb) {
    // 페이지 크기에 따른 비트 계산 변수
    FirstPageTable* page_table = malloc(sizeof(FirstPageTable));
    pc->offset_bits = 0, pc->first_page = 0, pc->second_page = 0;

    // 페이지 크기에 따른 비트 계산
    switch (page_size_kb) {
        case 1*1024: // 1KB 페이지 크기 (10/11/11)
            pc->offset_bits = 10;
            pc->first_page = 11;
            pc->second_page = 11;
            break;
        case 4*1024: // 4KB 페이지 크기 (10/11/11)
            pc->offset_bits = 12;
            pc->first_page = 10;
            pc->second_page = 10;
            break;
        case 16*1024: // 16KB 페이지 크기 (11/11/10)
            pc->offset_bits = 14;
            pc->first_page = 9;
```

그림 29 페이지 크기에 따라 페이지 테이블의 비트 구성을 달리하기 위한 코드

page_table.c 파일의 init_page_table 함수는 페이지 테이블을 초기화하는 역할을 하며, 페이지 크기에 따라 계산된 비트를 사용해 1단계와 2단계 페이지 테이블을 동적으로 할당하고 각 엔트리를 초기화한다. 페이지 크기에 맞춰 비트 구성을 달리하고, 2단계 페이지 테이블은 첫 번째 레벨 테이블의

엔트리들을 포인터로 연결해 가상 메모리 주소 공간을 효율적으로 관리한다.

그림 30 1단계 테이블 초기화 부분

그림 31 2단계 테이블 및 엔트리 초기화 부분

해당 함수는 PageConfig 구조체를 기반으로 페이지 크기를 계산하고, 이에 맞는 비트 정보를 offset_bits, first_page, second_page에 저장한다. 페이지 크기에 따라 첫 번째 및 두 번째 레벨의 페이지 테이블 크기를 계산하고, 이를 동적으로 할당한다. 각 페이지 테이블의 엔트리는 PageTable Entry 구조체로 초기화되어, 프레임 번호, 유효 비트, 수정 비트, 디스크 블록 번호 및 읽기 전용 여부를 관리한다.

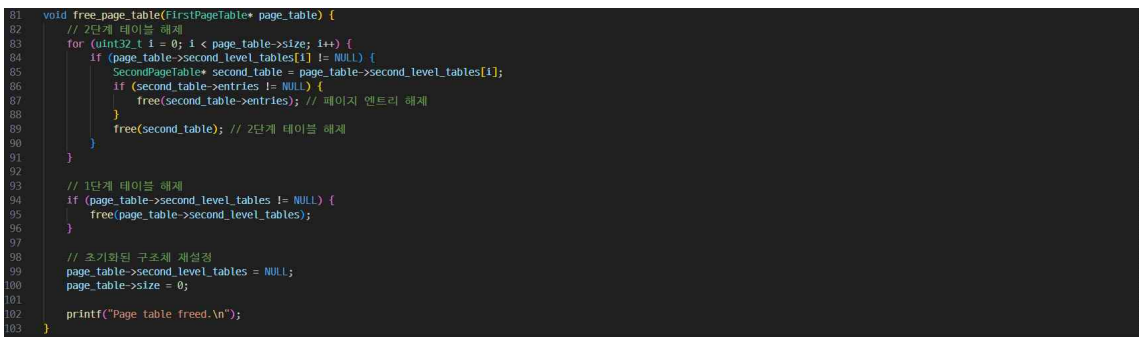


그림 32 페이지 테이블 해제 함수

페이지 테이블을 해제할 때는 먼저 2단계 테이블에서 할당된 메모리와 엔트리를 해제한다. 그런 다음 1단계 테이블을 해제하고, 마지막으로 page_table 구조체를 정리한다.

4-5. TLB (Translation Lookaside Buffer)

TLB는 가상 메모리 시스템에서 중요한 역할을 하는 하드웨어 캐시이다. 이번 구현에서는 TLB의 초기화, 해제, 그리고 TLB 엔트리 관리에 관련된 기능들을 제공한다. TLB는 TLBEntry 구조체와 TLB 구조체로 구성된다. TLBEntry 구조체는 각 TLB 엔트리의 정보를 담고 있으며, TLB 구조체는 전체 TLB를 관리한다. TLB의 크기는 size 변수로 정의되며, entries 배열은 각 TLB 엔트리의 정보를 저장한다. 또한, current_time 변수는 시간 틱을 관리하며, FIFO 정책을 위한 fifo_index도 포함되어 있다.

init_tlb 함수는 TLB를 초기화하는 함수로, 주어진 크기만큼 TLB 엔트리 배열을 할당하고, 각 엔트리의 상태를 초기화한다. free_tlb 함수는 TLB가 더 이상 필요 없을 때 할당된 메모리를 해제하는 함수이다. 이와 같은 구조를 통해 TLB의 크기, 엔트리, 그리고 정책들을 효율적으로 관리할 수 있다.

```
TLB* init_tlb(int size) {
    // TLB 메모리 할당
    TLB* tlb = malloc(sizeof(TLB));
    if (tlb == NULL) {
        fprintf(stderr, "Error: Memory allocation failed for TLB structure.\n");
        exit(EXIT_FAILURE);
    }

    // TLB 엔트리 배열 할당
    tlb->entries = malloc(sizeof(TLBEntry) * size);
    if (tlb->entries == NULL) {
        fprintf(stderr, "Error: Memory allocation failed for TLB entries.\n");
        free(tlb); // 할당된 TLB 구조체 메모리 해제
        exit(EXIT_FAILURE);
    }

    // 엔트리 초기화
    for (int i = 0; i < size; i++) {
        tlb->entries[i].page_number = 0;
        tlb->entries[i].frame_number = 0;
        tlb->entries[i].valid = 0; // 유효하지 않은 상태로 초기화
        tlb->entries[i].last_used = 0;
        tlb->entries[i].access_count = 0;
        tlb->entries[i].read_only = 0;
    }

    tlb->size = size;
    tlb->current_time = 0; // 현재 시간 초기화

    return tlb;
}
```

그림 33 TLB 초기화 부분

init_tlb 함수는 TLB의 메모리를 동적으로 할당하고, 각 엔트리를 초기화한다. 각 엔트리는 page_number, frame_number, valid, last_used,

access_count, read_only 등의 정보를 포함한다. 이러한 초기화 작업을 통해 TLB가 효율적으로 작동할 수 있도록 준비한다. vaild 비트는 TLB 엔트리가 유효한지 여부를 나타내며, last_used와 access_count는 LRU와 LFU 같은 교체 알고리즘을 구현하는데 사용된다. read_only 플래그는 페이지가 읽기 전용인지 여부를 나타내며, 이는 CoW 구현 시 중요한 역할을 한다.

```
void free_tlb(TLB* tlb) {
    if (tlb != NULL) {
        // 엔트리 배열 해제
        if (tlb->entries != NULL) {
            free(tlb->entries);
        }

        // TLB 구조체 해제
        free(tlb);
    }
}
```

그림 34 TLB 해제 부분

free_tlb 함수는 TLB와 그에 속한 엔트리 배열의 메모리를 해제한다.

5. Result & Code Compile

가상 메모리 관리 시스템의 핵심 기능을 위와 같이 구현하였으며, 해당 파트에서는 이를 실제로 동작시키고 결과를 분석하기 위한 시뮬레이션을 진행할 예정이다. 시뮬레이션은 메모리 요청, 페이지 폴트, 페이지 교체 등의 주요 이벤트를 기록하는 로그 시스템을 기반으로 진행되며, 1,000에서 10,000틱까지 실행된다. 각 틱마다 발생한 이벤트는 로그 파일에 기록된다.

해당 프로그램은 실행 결과로 두 가지 텍스트 파일 형태의 로그를 생성한다. 첫 번째 로그 파일인 process.txt는 각 프로세스의 실행 과정에서 발생한 주요 작업 흐름을 기록한다. 이 파일에는 시간 단위를 나타내는 타임 틱(Time Tick)과 해당 시간 동안 실행된 프로세스의 읽기 또는 쓰기 작업에 대한 정보가 포함된다. 또한, 각 프로세스의 ID, 수행한 작업 유형(Read/Write), 작업이 수행된 가상 주소(Virtual Address), 쓰기 작업 시 사용된 데이터 등이 상세히 기록된다. 이를 통해 전체 프로세스의 실행 흐름과 각 작업의 상태를 명확히 파악할 수 있다.

두 번째 로그 파일인 detail_log.txt는 초기 메모리 설정과 프로세스 실행 중 발생하는 메모리 관리 동작에 대한 세부 내용을 기록한다. 이 로그는 크게 두 가지로 나뉜다. 첫 번째는 초기 할당(Initial Allocation) 과정에서 작성된 로그는 페이지 테이블의 생성 및 구성 과정과 디스크 및 메모리 위치, 스와핑(Swapping) 관련 내용을 담고 있다. 두 번째로, 프로세스 실행 중 발생한 세부 로그는 TLB 접근, 페이지 폴트(Page Fault), 메모리 읽기 및 쓰기 작업, 보조 저장 장치(Secondary Storage) 접근 등의 동작을 상세히 기록한다. 해당 프로그램은 초기 실행 시 페이지 테이블을 작성하고 가상 주소 공간과 물리 메모리의 매핑을 설정하는 초기 할당 과정을 거친다. 이 과정에서 필요한 경우 스와핑이 발생하며, 이에 대한 세부 내역이 detail_log.txt에 기록된다. 이후, 각 프로세스는 라운드 로빈 방식으로 스케줄링되며, 각 타임 틱마다 지정된 읽기 또는 쓰기 작업을 수행한다. 프로세스의 실행 상태와 작업의 진행 상황은 process.txt에 기록되며, 작업 수행 중 발생하는 TLB 및 페이지 관리 관련 세부 정보는 detail_log.txt에 기록된다.

이와 같은 로그 시스템은 프로그램 동작을 명확히 드러내고, 메모리 관리와 프로세스 스케줄링의 효율성을 분석하는 데 중요한 역할을 한다. process.txt는 전체적인 흐름을 파악하기에 적합하며, detail_log.txt는 메모리 관리의 세부 내용을 분석하고 성능 최적화 및 디버깅에 활용할 수 있는 핵심 자료를 제공한다. 아래는 두 파일의 예시이다.

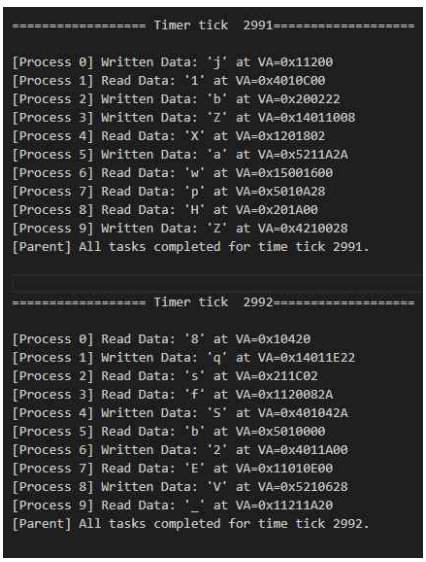


그림 36 Process.txt

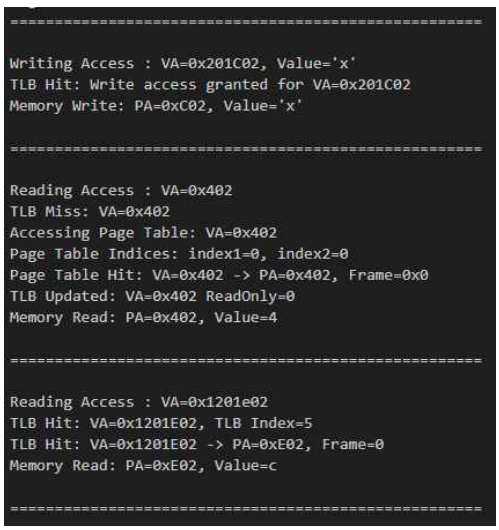


그림 37 detail_log.txt

본 프로젝트는 다양한 설정값에 따라 어떤 성능을 가지는지 확인 할 수 있으며, 실제 환경에서의 최적화된 메모리 관리 전략을 도출하는 것이 목표이다. 실험 과정에서 각 설정값을 변경하고, 프로세스의 실제 환경과 비슷한 데이터를 제공하기 위해 클러스터링을 활용한 데이터셋을 이용한다.

본 실험은 총 8가지로 아래와 같다.

1. 순차적 접근 패턴에 따른 페이징 성능 분석
2. 난수 접근 패턴에 따른 페이징 성능 분석
3. 클러스터링 지역성 데이터를 통한 페이징 및 TLB 성능 분석
4. 페이지 크기가 페이징 및 TLB 효율성에 미치는 영향 분석
5. 물리적 메모리 크기가 페이징 및 TLB 성능에 미치는 영향 분석
6. TLB 캐시 크기가 시스템 성능에 미치는 영향 분석
7. 페이지 교체 정책에 따른 페이징 알고리즘 성능 평가
8. TLB 캐시 교체 정책의 성능 평가

위 8가지 실험은 네가지 카테고리를 가진다.

첫 3가지 실험은 기본 설정에 대한 실험으로, 1번 실험에서는 접근하는 가상주소를 순차적으로 늘리는 순차적 접근 패턴을 설정하여 페이징과 TLB 성능을 비교 분석 한다. 2번 실험에서는 접근하는 가상주소의 패턴을 난수화 하여 난수 가상 주소 데이터에 대해서는 어떠한 성능을 가지는지 확인 할 수 있다. 세 번째 실험의 경우 실제 OS 환경과 유사하게 클러스터링 된 지역성을 가지는 데이터를 기반으로 기본적인 성능 평가를 한다.

다음으로, 크기 변화에 따른 분석 실험이며, 4번 실험에서는 페이지 크기를 변화시켜 페이징과 TLB 효율성에 미치는 영향을 분석한다. 5번 실험은 물리적 메모리 크기를 조정하여 페이징과 TLB 성능에 미치는 변화를 측정하며, 6번 실험에서는 TLB 캐시 크기를 조정하여 시스템 전반의 성능 변화를 비교 분석한다. 이 카테고리는 시스템 구성 요소의 크기 변화가 성능에 미치는 영향을 평가하기 위해 설계되었다.

다음으로, 교체 정책에 대한 설정을 평가하는 실험이 진행된다. 7번 실험에서는 다양한 페이지 교체 정책을 적용하여 페이징 알고리즘의 성능을 비교한다. 8번 실험은 TLB 캐시의 교체 정책을 다르게 설정하여 TLB 캐시 관리 방식이 성능에 미치는 영향을 분석하며, 9번 실험에서는 페이지 교체 정책과 TLB 캐시 교체 정책을 결합하여 두 정책 간의 상호작용이 시스템 효율성에 어떤 변화를 유발하는지 확인한다.

위 8가지 실험은 체계적인 접근을 통해 페이징 및 TLB의 성능에 영향을 미치는 주요 요인들을 분석 하고 전반적인 성능 평가를 목표로 한다.

위 실험에서의 종속 변수인 보조 메모리(디스크)의 크기는 약 300MB 이며, 보조 메모리 크기는 Tar file을 모두 담기 위해서 1GB로 설정 하였다. 이에 따라, 성능을 평가 하기 위한 적절한 물리 메모리 크기는 16MB와 32MB를 중간값으로 설정하였으며, 해당 물리 메모리 크기에 적절한 page size는 1, 4, 16, 64kb에서 수행한다. TLB 캐시 사이즈의 경우 중간 값으로 2MB의 페이지 크기를 담을 수 있게 설정하였다.

실험에서 사용 할 성능 지표는 총 4가지이다.

1. TLB Hit Rate
2. Page Hit Rate
3. EMAT(Effective Memory Access Latency)

위 지표 중 1번과 2번을 통해 4번을 구할 수 있으며, 4번 지표는 컴퓨터 시스템에서의 메모리를 접근하는데 걸리는 소요시간을 의미한다. 해당 지표는 다음과 같은 공식으로 나타 낼 수 있다.

[TLB o]

$$EMAT$$
$$= (H \cdot (TLB Access\ Time + Memory\ Access\ Time)) +$$
$$(1-H) \cdot [(P \cdot (TLB Access\ Time + Page\ Table Access\ Time + Memory\ Access\ Time)$$
$$+ (1-P) \cdot (TLB Access\ Time + Page\ Table Access\ Time + Disk Access\ Time + 2 \cdot Memory\ Access\ Time)]$$

[TLB x]

$$EMAT = (P \cdot (Page\ Table Access\ Time + Memory\ Access\ Time)) + ((1-P) \cdot (Disk Access\ Time + 2 \cdot Memory\ Access\ Time))$$

위 공식에서 P는 페이지 폴트 확률이며, TLB Access Time Page Table Access Time 과 Memory Access Time은 다음과 같은 값을 가진다고 가정하고, EMAT를 구할 수 있다.

- TLB Access Time : 10ns
- Page Table Access Time : 100ns
- Memory Access Time : 100ns
- Disk Access Time : 10ms = 10000000ns

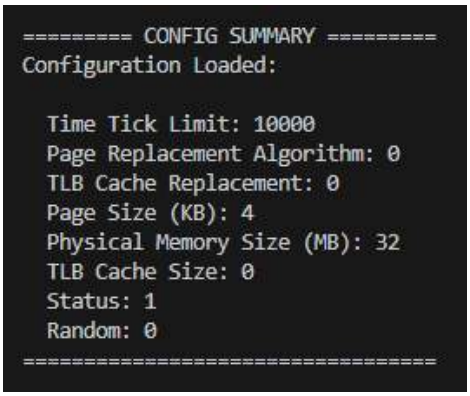
```
========= LOG SUMMARY =========
TLB Hit: 62407 (62.407%)
TLB Miss: 37593 (37.593%)
Page Hit: 35141 (93.478%)
Page Fault: 2452 (6.522%)
Swap In: 2452
Swap Out: 2452
Processes Managed: 100000
==============================
```

위 공식을 옆의 예시에 대입을 하게 된다면 최종 EMAT는 0.2453 ms 가 나오게 된다. 마지막으로 본 실험에서 배제한 것 중 하나는 메모리와 캐시 사이즈별 Access Time은 모두 같게 하고 계산하였다.

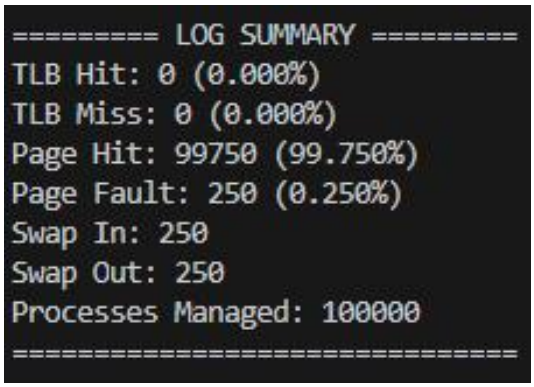
1) 순차적 접근 패턴에 따른 페이징 성능 분석

순차적 접근 패턴은 메모리 접근이 연속적으로 이루어지는 이상적인 상황을 가정하며, 실제 시스템에서도 드물게 발생한다. 현실적인 메모리 접근은 다양한 실행 흐름과 동적 데이터 구조로 인해 불규칙성을 가지는 경우가 많다. 따라서 이 실험은 메모리 관리 기법의 최적 조건에서의 성능을 평가하기 위한 기초 데이터로 활용된다. 이를 통해 페이징과 TLB의 이론적 성능 한계를 확인하고, 다른 실험과의 비교 기준을 제공할 수 있다.

[결과 1-1] EMAT = 0.0252 ms

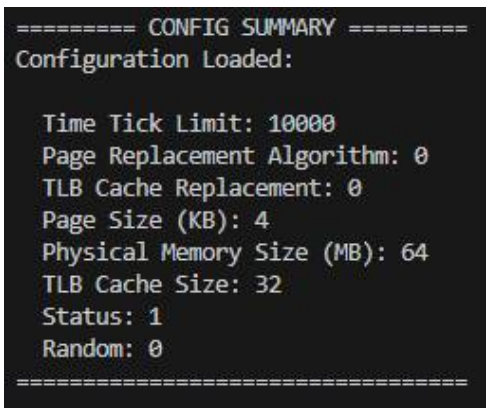


[그림 39. 실험1-1 Config]

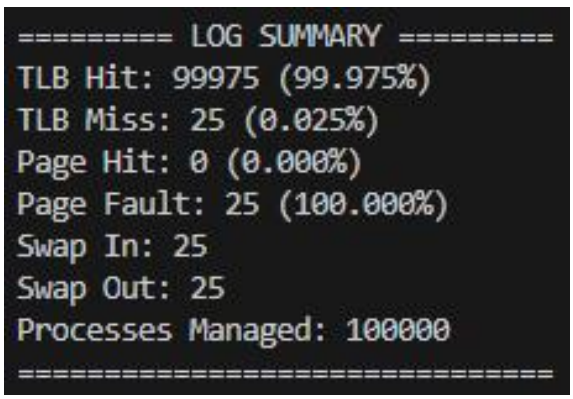


[그림 40. 실험1-1 Result]

[결과 1-2] EMAT = 0.00081 ms



[그림 41. 실험1-2 Config]



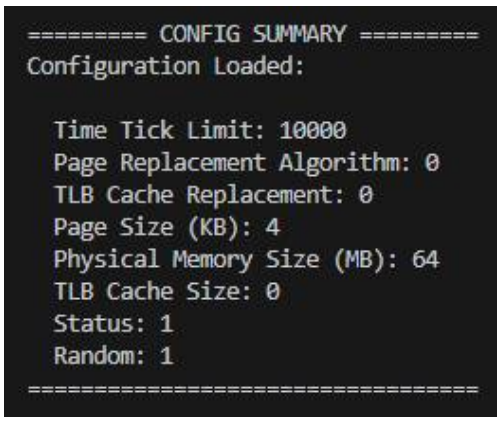
[그림 42. 실험1-2 Result]

실험 1에서는 순차적 접근 패턴을 기반으로 메모리 접근 성능을 평가하였다. 결과적으로, TLB와 페이지 히트율이 이상적으로 높은 값을 기록했으나, 이는 현실적으로 드물게 발생하는 상황을 가정한 실험이라는 점에서 큰 의미를 가지지 않는다. 순차적 접근은 메모리 접근의 비현실적인 단순화를 반영하므로, 실험 결과가 실제 시스템의 동작을 대변하지 못한다. 이러한 한계를 보완하고 보다 현실적인 환경에서의 성능을 확인하기 위해, 다음 실험에서는 난수 접근 패턴을 통해 메모리 관리 기법의 유연성과 효율성을 평가하고자 한다.

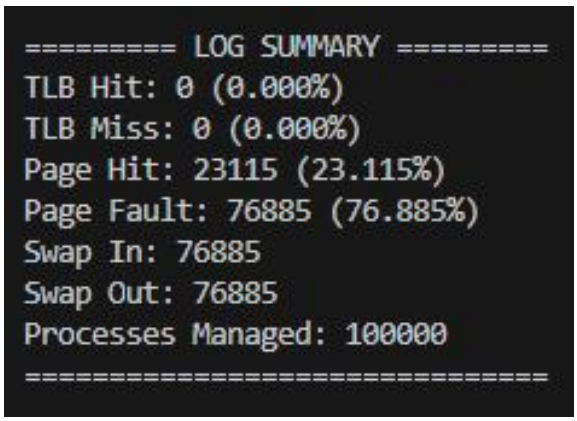
2) 난수 접근 패턴에 따른 페이징 성능 분석

난수 접근 패턴을 기반으로 한 실험은 메모리 접근이 비정형적이고 불규칙적인 환경에서의 페이징 성능을 평가하기 위한 것이다. 이러한 패턴에서는 메모리 접근의 지역성이 감소하므로 TLB 캐시와 페이지 테이블의 성능이 크게 영향을 받지 않을 것으로 예측된다. 이는 메모리 접근이 특정 페이지나 캐시에 집중되지 않고 전반적으로 분산되기 때문이다. 따라서, 난수 접근 패턴 실험은 메모리 관리 기법의 극단적 상황에서의 한계를 분석하는 데 사용할 수 있다.

[결과 2-1] EMAT = 7.6887 ms

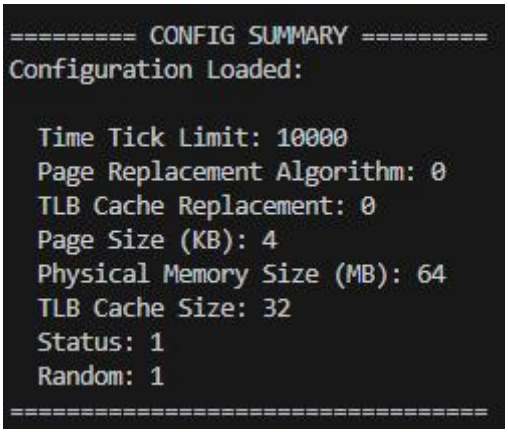


[그림 43. 실험 2-1 Config]

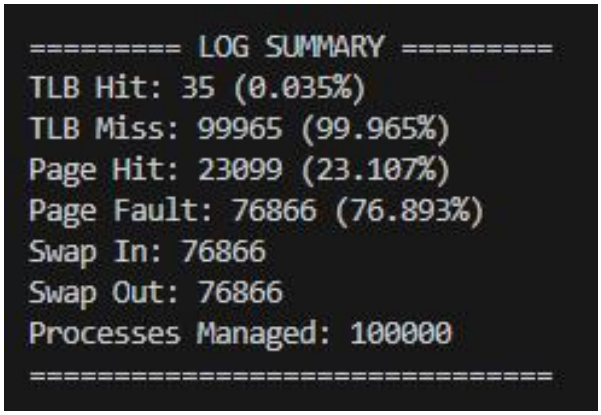


[그림 44. 실험2-1 Result]

[결과 2-2] EMAT = 7.6887 ms



[그림 45. 실험 2-2 Config]



[그림 46. 실험2-2 Result]

[결과 2-3] EMAT = 5.8760 ms

```
========= CONFIG SUMMARY =========
Configuration Loaded:

  Time Tick Limit: 10000
  Page Replacement Algorithm: 0
  TLB Cache Replacement: 0
  Page Size (KB): 4
  Physical Memory Size (MB): 128
  TLB Cache Size: 128
  Status: 1
  Random: 1
==================================
```

[그림 47. 실험 2-3 Config]

```
========= LOG SUMMARY =========
TLB Hit: 158 (0.158%)
TLB Miss: 99842 (99.842%)
Page Hit: 41085 (41.150%)
Page Fault: 58757 (58.850%)
Swap In: 58757
Swap Out: 58757
Processes Managed: 100000
===============================
```

[그림 48. 실험2-3 Result]

난수 접근 패턴은 메모리 접근의 극한 상황을 시뮬레이션하기 위한 실험으로, TLB와 페이징 간의 효율성을 평가하기 위한 중요한 기준이 된다. 실험 결과, TLB 캐시 크기를 증가시켰을 때 TLB Hit 비율이 매우 소폭 증가했지만, 난수화 된 접근 패턴에서는 TLB 크기 증가는 큰 효과를 기대하기 어려웠다. 반면, 물리 메모리 크기를 늘림으로써 Page Fault 비율을 76%에서 58%로 유의미하게 감소시킬 수 있었다. 이는 충분한 메모리 자원이 페이징 성능에 결정적인 영향을 미친다는 사실을 입증한다. 다만, 현실적인 시나리오를 고려할 때 이러한 극단적인 난수화 접근이 자주 발생하지 않음을 감안해야 한다. 64MB에서 128MB로 메모리 자원을 늘리는 것은 물리 메모리 접근 속도 관점에서는 거의 비슷하지만, 페이지 테이블 조회 비용면에서 약간의 시간과 자원이 더 소모되지만, 디스크에 접근 하는것에 비하면 월등히 낮으므로 해당 EMAT 값은 유효하다고 볼 수 있다.

3) 난수 접근 패턴에 따른 페이징 성능 분석

위 실험 들에서 다룬 순차적 접근 패턴, 난수 접근 패턴은 현실적으로 일반적인 메모리 접근 방식과 거리가 멀다. 이러한 극단적인 접근 방식은 극한의 시나리오를 테스트하는 데 유용하지만, 실제 시스템에서 자주 나타나는 접근 패턴을 반영하지 못한다. 따라서, 보다 현실적인 데이터 패턴을 기반으로 페이징 및 TLB 성능을 평가하기 위해 클러스터링 된 지역성 데이터를 활용한 추가 실험을 준비하였다.

이 실험은 특정 메모리 영역이 자주 액세스되는 실제 프로그램의 특성을 시뮬레이션하기 위해 설계되었으며, 데이터를 생성하는 과정에서 클러스터를 중심으로 한 공간적, 시간적 지역성을 강조하였다. 지역성을 기반으로 한 데이터는 메모리 효율성을 높이기 위해 TLB와 페이징 시스템이 어떻게 작동하는지 평가하기에 적합하다.

본 실험은 클러스터링 지역성 데이터의 특성을 최대한 활용하여 TLB Hit 비율과 Page Fault 비율이 개선되는지를 확인하고자 하며, 이를 통해 현실적인 데이터 패턴에서 시스템의 효율성을 측정하고자 한다. 이를 통해, 단순한 난수 패턴의 한계를 넘어 보다 실질적인 환경에서 성능을 분석할 수 있을 것으로 기대된다.

[결과 3-1] EMAT = 0.5802 ms

```
========= CONFIG SUMMARY =========
Configuration Loaded:

  Time Tick Limit: 10000
  Page Replacement Algorithm: 0
  TLB Cache Replacement: 0
  Page Size (KB): 4
  Physical Memory Size (MB): 8
  TLB Cache Size: 0
  Status: 4
  Random: 0
==================================
```

[그림 49. 실험 3-1 Config]

```
========= LOG SUMMARY =========
TLB Hit: 0 (0.000%)
TLB Miss: 49791 (100.000%)
Page Hit: 94200 (94.200%)
Page Fault: 5800 (5.800%)
Swap In: 5800
Swap Out: 5800
Processes Managed: 100000
==============================
```

[그림 50. 실험 3-1 Result]

[결과 3-2] EMAT = 0.5447 ms

```
========= CONFIG SUMMARY =========
Configuration Loaded:

  Time Tick Limit: 10000
  Page Replacement Algorithm: 0
  TLB Cache Replacement: 0
  Page Size (KB): 4
  Physical Memory Size (MB): 32
  TLB Cache Size: 0
  Status: 4
  Random: 0
==================================
```

[그림 51. 실험 3-2 Config]

```
========= LOG SUMMARY =========
TLB Hit: 0 (0.000%)
TLB Miss: 49791 (100.000%)
Page Hit: 94555 (94.555%)
Page Fault: 5445 (5.445%)
Swap In: 5445
Swap Out: 5445
Processes Managed: 100000
==============================
```

[그림 52. 실험 3-2 Result]

[결과 3-3] EMAT = 0.2722 ms

```
========= CONFIG SUMMARY =========
Configuration Loaded:

  Time Tick Limit: 10000
  Page Replacement Algorithm: 0
  TLB Cache Replacement: 0
  Page Size (KB): 4
  Physical Memory Size (MB): 8
  TLB Cache Size: 32
  Status: 4
  Random: 0
==================================
```

[그림 51. 실험 3-4 Config]

```
========= LOG SUMMARY =========
TLB Hit: 62424 (62.424%)
TLB Miss: 37576 (37.576%)
Page Hit: 34855 (92.759%)
Page Fault: 2721 (7.241%)
Swap In: 2721
Swap Out: 2721
Processes Managed: 100000
==============================
```

[그림 52. 실험 3-4 Result]

[결과 3-4] EMAT = 0.00551 ms

```
========= CONFIG SUMMARY =========
Configuration Loaded:

  Time Tick Limit: 10000
  Page Replacement Algorithm: 0
  TLB Cache Replacement: 0
  Page Size (KB): 4
  Physical Memory Size (MB): 8
  TLB Cache Size: 64
  Status: 4
  Random: 0
==================================
```

[그림 51. 실험 3-4 Config]

```
========= LOG SUMMARY =========
TLB Hit: 99931 (99.931%)
TLB Miss: 69 (0.069%)
Page Hit: 15 (21.739%)
Page Fault: 54 (78.261%)
Swap In: 54
Swap Out: 54
Processes Managed: 100000
==============================
```

[그림 52. 실험 3-4 Result]

클러스터링 지역성 데이터를 기반으로 한 실험 결과는 위와 같다. 실험 3-1과 3-2는 TLB 캐시를 사용하지 않고 메모리 크기만을 변경하여 진행하였으며, 메모리 크기를 증가시킴에 따라 페이지 폴트가 개선되는 경향을 볼 수 있다. 그러나 두 실험 간의 차이는 크지 않았으며, 페이지 히트율의 변화는 미미하다. 반면, 실험 3-3과 3-4는 TLB 캐시를 사용하는 조건에서 진행되었고, 특히 TLB 캐시 크기를 증가시키면서 TLB 히트율이 크게 증가하는 결과를 확인할 수 있다.

이 실험 결과는 현실적인 데이터 접근 패턴을 반영하고 있으며, 실제 데이터와 유사한 접근 패턴을 가진 클러스터링 데이터의 효과를 입증하였다. EMAT(Effective Memory Access Time)를 분석한 결과, 실험 1과 2와 같

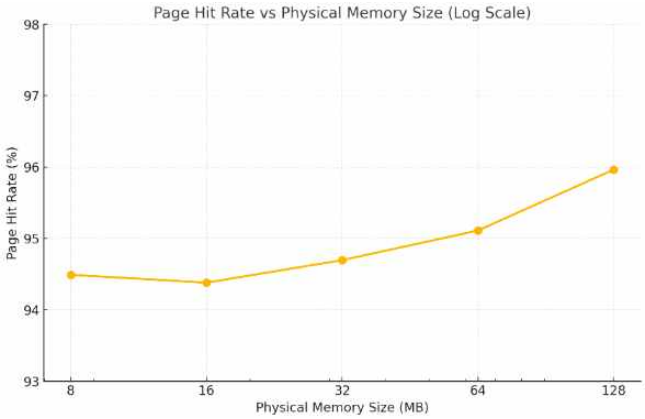
은 극단적인 조건에서는 비정상적으로 높고 낮은 EMAT 값을 보였지만, 클러스터링 지역성을 반영한 실험에서는 상대적으로 정상적인 EMAT를 확인할 수 있었다. 특히, 실험 3-4에서는 TLB 캐시를 적극 활용하며 가장 낮은 EMAT를 기록하였으며, 이는 데이터 접근의 효율성이 상당히 향상되었음을 나타낸다. 이와 같은 결과는 클러스터링 지역성을 활용한 데이터 접근 방식이 메모리 관리 전략 선택에 있어 시뮬레이션의 기여를 할 수 있음을 시사한다.

4) 물리적 메모리 크기가 페이징 및 TLB 성능에 미치는 영향 분석

해당 실험은 물리적 메모리 크기가 페이징과 TLB 성능에 어떤 영향을 미치는지 분석하기 위해 설계되었다. 물리적 메모리 크기가 증가하면 더 많은 페이지를 메모리에 유지할 수 있어 페이지 폴트를 줄이고, 결과적으로 페이징 성능이 향상될 것으로 예상된다. 또한, 충분한 메모리 크기는 TLB 히트율에도 영향을 줄 수 있어 전반적인 메모리 접근 효율성을 높일 수 있다. 이를 통해 메모리 크기에 따른 성능 변화와 최적의 메모리 할당 전략을 확인하는 데 목적을 두었다. 이 실험은 총 2가지 프로세스로 진행이 된다. 첫 번째 실험은 tlb cache가 없을 때, 다른 변수들은 동일하게 한후, 물리 메모리를 늘리면서 Page hit를 측정했다. 아래는 측정한 Config중 8MB에 해당하는 부분과 실험의 결과이다.

```
========= CONFIG SUMMARY =========
Configuration Loaded:

  Time Tick Limit: 10000
  Page Replacement Algorithm: 0
  TLB Cache Replacement: 0
  Page Size (KB): 4
  Physical Memory Size (MB): 8
  TLB Cache Size: 0
  Status: 4
  Random: 0
==================================
```

[그림 53. 실험 4-1 Config]

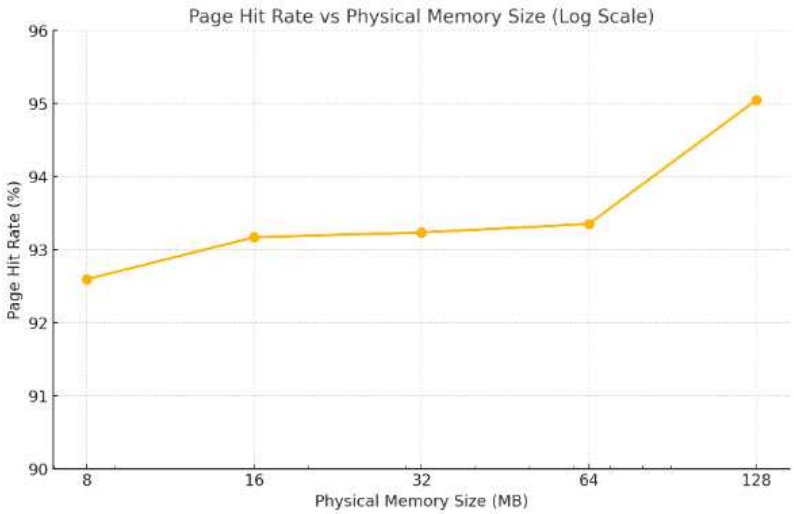


[그림 54. 실험 4-1 Result]

실험 결과, 물리적 메모리 크기가 증가함에 따라 페이지 히트율이 약간 씩 개선되는 경향을 확인할 수 있었다. 이는 더 큰 메모리가 더 많은 페이지를

메모리에 유지할 수 있어 페이지 폴트를 줄이는 데 기여함을 보여준다.

두 번째 실험에서는 실험 4-1의 config에서 tlb cache size를 32인 상태에서 진행 하였다.



[그림 55. 실험 4-2 Page Hit]

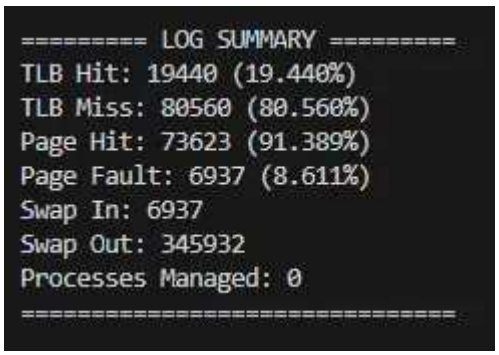
[그림 56. 실험 4-2 TLB Hit]

실험 4-2에서는 물리적 메모리 크기가 증가함에 따라 페이지 히트율(Page Hit Rate)이 개선되는 경향이 나타났지만, TLB 히트율(TLB Hit Rate)에는 큰 변화가 없었다. 이는 물리적 메모리 크기가 페이지 폴트를 줄이는 데는 효과적이지만, TLB와는 직접적인 상관관계가 없음을 보여준다.

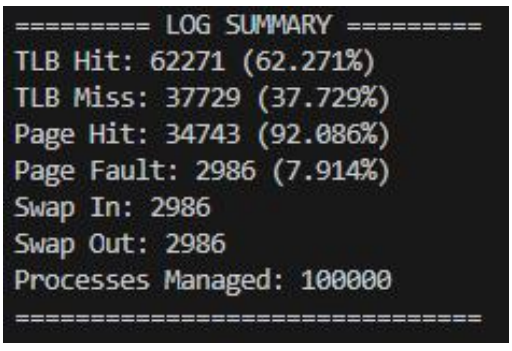
5) 페이지 크기가 페이징 및 TLB 효율성에 미치는 영향 분석

해당 실험은 페이지 크기가 페이징 및 TLB의 효율성에 미치는 영향을 분석하기 위해 설계되었다. 페이지 크기는 메모리 관리의 기본 단위로, 페이지 크기가 작아지면 페이지 테이블 항목의 수가 증가하여 관리 비용이 상승하지만, 메모리의 낭비가 줄어드는 장점이 있다. 반대로, 페이지 크기가 커지면 TLB 히트율이 증가할 가능성이 있지만, 페이지 내부 단편화로 인해 메모리 낭비가 발생할 수 있다. 이 실험은 이러한 트레이드 오프를 고려하여, 다양한 페이지 크기에 따라 페이징과 TLB 성능이 어떻게 변화하는지를 평가하는 데 초점을 맞추었다. 해당 실험은 2가지로 진행된다. 첫 번째 실험은 페이지 크기를 1kb와 4kb를 비교하는 실험을 진행하고, 두 번 째 실험에서는 4kb 16kb 64kb의 각각 성능 평가를 진행한다.

[실험 5-1] 1kb : EMAT = 0.6939 ms, 4kb : EMAT = 0.2987 ms



[그림 56. 실험 5-1 1kb 결과]

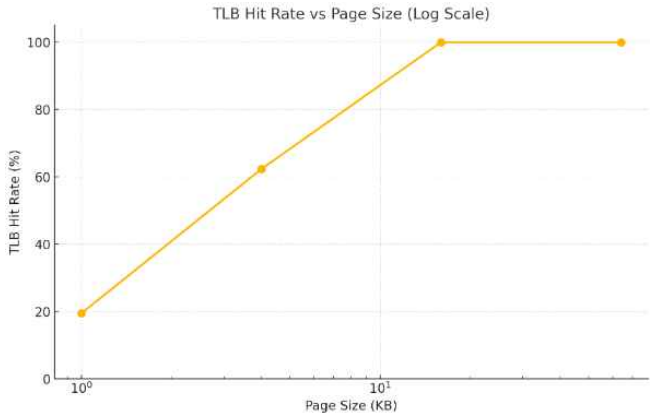


[그림 57. 실험 5-1 4kb 결과]

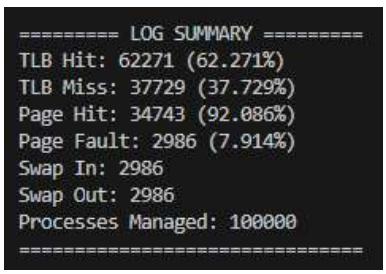
실험 5-1에서는 페이지 크기를 1KB에서 4KB로 증가시켰을 때, EMAT가 0.6939ms에서 0.2987ms로 크게 감소하며 페이징 성능이 개선되었다. 이와

함께, TLB Hit는 19.440%에서 62.271%로 크게 증가했고, Page Fault는 8.611%에서 7.914%로 소폭 감소하여 페이지 크기 증가가 TLB 성능과 페이지 폴트율 모두에 긍정적인 영향을 미쳤음을 알 수 있다.

[실험 5-2] 페이지 크기에 따른 성능

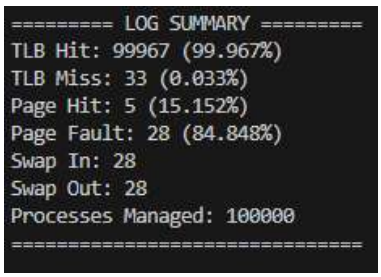


[그림 58. 실험 5-2 페이지 크기에 따른 결과]



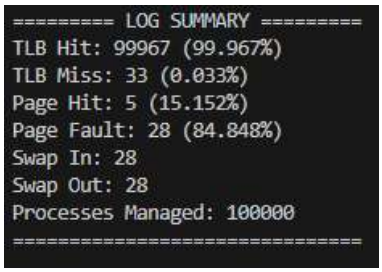
[그림 59. 실험 5-2 4kb 결과]

0.2987 ms



[그림 60. 실험 5-1 16kb 결과]

0.00291 ms



[그림 61. 실험 5-1 64kb 결과]

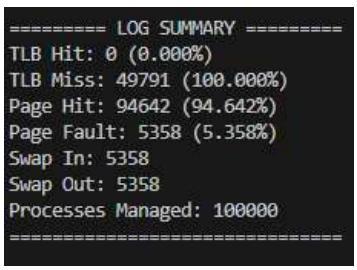
0.00291 ms

실험 5-2에서는 페이지 크기를 4KB에서 16KB, 64KB로 증가시켰을 때 TLB Hit Rate가 전체적으로 62.271%에서 99.967%로 크게 향상되었음을 확인할 수 있었다. 이는 페이지 크기 증가로 인해 TLB 엔트리의 커버 범위가 확장되었기 때문이며, Page Fault는 7.914%에서 0.033%로 감소하여 전반적인 페이징 효율이 개선되는 것을 확인 할 수 있다.

6) TLB 캐시 크기가 시스템 성능에 미치는 영향 분석

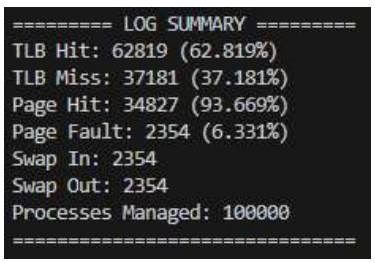
TLB 캐시 크기가 시스템 성능에 미치는 영향은 TLB Hit Rate와 Page Fault Rate를 통해 평가할 수 있다. TLB 캐시 크기를 증가시키면 더 많은 페이지 엔트리를 캐시에 저장할 수 있어 TLB Hit Rate가 높아지고, 이는 곧 메모리 접근의 효율성을 높이는 결과를 가져온다. 특히, TLB 캐시 크기가 작을 경우에는 Miss가 빈번히 발생해 페이징 성능이 저하될 수 있으나, 캐시 크기를 증가시키면 이러한 문제를 완화할 수 있다. 하지만, 캐시 크기를 무작정 크게 하는 것은 하드웨어 비용 및 설계상의 제약이 있으므로 적정 크기를 설정하는 것이 중요하다. 이를 통해 시스템의 메모리 접근 성능 최적화를 할 수 있다. 아래는 해당 부분의 실험이다.

[실험 6] TLB Cache Size 크기에 따른 성능



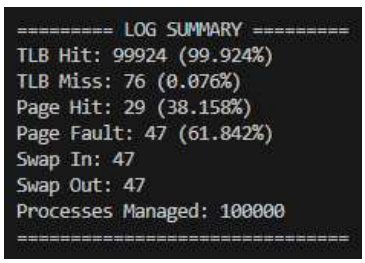
[그림 62. 실험 6 tlb x 결과]

0.536 ms



[그림 63. 실험 6 tlb 32 결과]

0.2355 ms



[그림 64. 실험 6 tlb 64 결과]

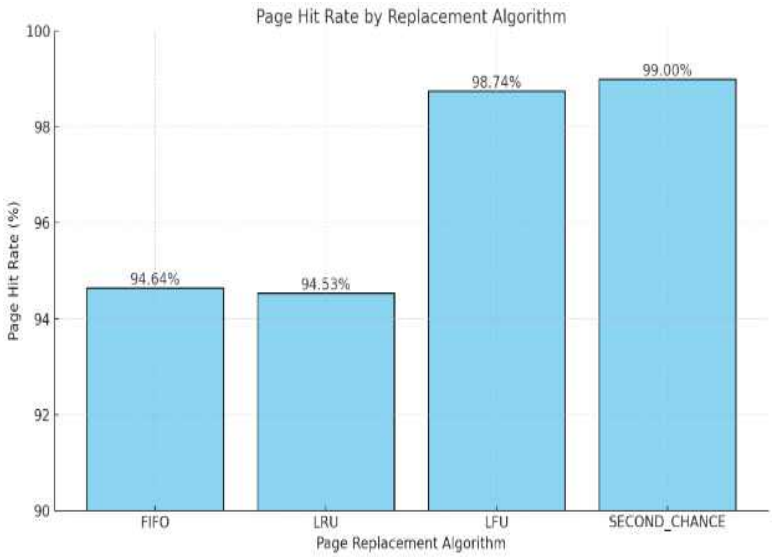
0.00481 ms

TLB Cache Size를 늘림에 따라 TLB Hit Rate가 크게 개선되었으며, 이에 따라 EMAT 값도 점진적으로 감소하는 결과를 보였다. 특히, TLB Cache가 없는 경우와 비교하여 64 엔트리로 확장했을 때, TLB Hit Rate는 99.924%로 상승하였고, EMAT는 0.00481ms로 대폭 감소하였다. 이는 TLB Cache 크기를 증가시키는 것이 메모리 접근 효율성 향상에 매우 효과적임을 보여준다.

7) 페이지 교체 정책에 따른 페이징 알고리즘 성능 평가

페이지 교체 정책에 따른 페이징 알고리즘의 성능 평가에서는 FIFO, LRU, LFU 그리고 Second Chance 기법을 비교하여 각 정책이 메모리 관리에 어떤지 분석할 수 있다. 이러한 정책들은 메모리 부족 상황에서 페이지를 교체하는 방식이 다르기 때문에, Page Fault 발생률과 EMAT 값에 큰 차이를 보일 수 있다. 실험의 주요 목적은 다양한 접근 패턴과 시스템 환경에서 각 교체 정책이 페이지 폴트를 얼마나 효율적으로 줄이는지를 확인하고, 특정 정책이 특정 상황에서 더 적합한지 평가하는 것이다. 이를 통해 페이지 교체 알고리즘 선택이 시스템 성능에 미치는 영향을 정량적으로 분석할 수 있다. 이번 실험에서 TLB는 사용되지 않는다.

[실험 7] 페이지 교체 정책에 따른 성능



[그림 63. 페이지 교체 정책에 따른 결과]

EMAT 값 :

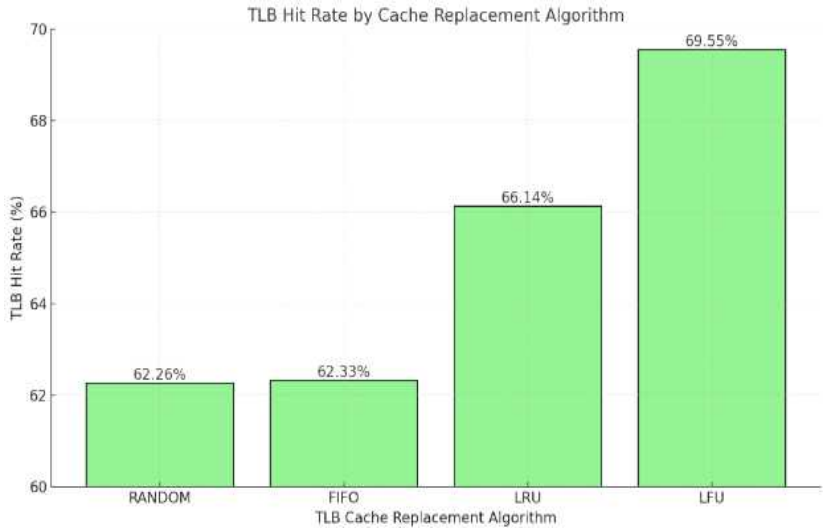
FIFO	:	0.5360 ms
LRU	:	0.5472 ms
LFU	:	0.1262 ms
SECOND_CHANCE	:	0.1005 ms

페이지 교체 정책에 따른 성능 비교 결과, LFU와 SECOND_CHANCE 알고리즘이 가장 높은 Page Hit Rate를 기록하며 각각 98.74%와 99.00%를 달성했다. 이에 반해 FIFO와 LRU는 상대적으로 낮은 Page Hit Rate를 보였으며, EMAT 값 또한 LFU와 SECOND_CHANCE가 더 낮게 측정되어 효율적임을 알 수 있다. 이러한 결과는 LFU와 SECOND_CHANCE가 페이지 교체 시 더 적합한 후보를 선택하는 데 강점을 가진다는 것을 알 수 있다. 정책 선택은 워크로드 특성에 따라 성능에 큰 영향을 미칠 수 있음을 확인할 수 있다.

8) TLB 캐시 교체 정책의 성능 평가

TLB 캐시 교체 정책의 성능 평가는 TLB 적중률과 EMAT 값을 통해 다양한 교체 알고리즘의 효율성을 분석하는 실험이다. LRU, RANDOM, FIFO, LFU와 같은 대표적인 교체 알고리즘을 비교하여 각 정책의 성능 차이를 확인한다. 특히, TLB 캐시는 시스템 성능에 직접적인 영향을 미치므로, 이 실험은 어떤 교체 정책이 메모리 접근 효율성을 최적화할 수 있는지 평가하는 데 초점이 맞춰져 있다. 이를 통해 최적의 TLB 캐시 교체 알고리즘을 제안하는 근거를 확인 할 수 있다.

[실험 8] TLB 교체 정책에 따른 성능



[그림 64. TLB cache 교체 정책에 따른 결과]

EMAT 값 :

RANDOM	:	0.0939 ms
FIFO	:	0.0911 ms
LRU	:	0.0489 ms
LFU	:	0.0461 ms

이 실험에서는 다양한 TLB 캐시 교체 정책(RANDOM, FIFO, LRU, LFU)이 TLB 성능에 미치는 영향을 분석했다. 각 정책의 TLB 히트율과 EMAT 값을 비교하여, 정책의 효율성과 메모리 접근 시간의 영향을 평가한다. 특히, LRU와 LFU와 같은 적응형 알고리즘은 캐시 공간을 더 효과적으로 활용할 가능성이 높다.

본 프로젝트는 총 8개의 실험을 통해 메모리 관리 기법의 성능에 영향을 미치는 다양한 요인을 평가하였다. 순차적 접근 패턴은 이상적인 상황을 가정한 실험으로, 페이징과 TLB 성능이 매우 높았으나 현실적인 상황과는 거리가 있었다. 이에 반해, 난수 접근 패턴은 극단적인 환경에서의 최악 성능을 평가하여 지역성이 부족할 때의 문제를 잘 보여주었다. 클러스터링 지역성을 반영한 접근에서는 현실적인 데이터 패턴을 기반으로 페이징 및 TLB 성능이 크게 개선되었음을 확인할 수 있었다. 페이지 크기와 물리적 메모리 크기의 증가는 페이지 폴트를 줄이고 EMAT 값을 개선하며 시스템 성능 최적화에 중요한 요소로 나타났다. 특히 TLB 캐시 크기의 증가와 적절한 교체 정책(LRU, LFU)은 TLB 히트율을 크게 높이고 EMAT 값을 낮추는 데 효과적이었다. 페이지와 TLB 교체 정책의 비교에서는 LFU와 Second-Chance 정책이 우수한 성능을 기록하며 효율적인 메모리 관리의 중요성을 입증하였다. 종합적으로, 본 연구는 메모리 관리 기법의 핵심 변수들이 페이징 및 TLB 성능에 미치는 영향을 실험적으로 검증하며, 현실적인 데이터 접근 패턴과 최적화된 설계가 시스템 성능에 결정적인 영향을 미친다는 점을 강조하였다.

6. Build Environment

해당 프로젝트를 실행하기 위해서는 Gcc 컴파일러가 필요하며, 해당 프로그램은 Makefile을 사용하여 빌드되기 때문에 프로그램이 위치한 디렉토리 내에서 다음과 같은 명령어를 사용하면 프로그램이 실행된다.

명령어는 다음과 같다.

1. make
2. ./project

다음은 config.txt로 config.txt에 따라서 여러 정책과 크기가 결정된다.

```
# 설정 관련
time_tick_limit=10000            # time_tick
page_replacement_algorithm=3    # FIFO, LRU, LFU, SECOND_CHANCE (0, 1, 2, 3)
tlb_cache_replacement=3        # RANDOM, FIFO, LRU, LFU (0,1,2,3)

# 크기 관련, 아래 해당 하는 값으로만 설정해야함.
page_size_kb=4                    # 1, 4, 16, 64
physical_memory_size_mb=32        # 8, 16, 32, 64, 128
tlb_cache_size=32                # 0, 32, 64, 128, 256, 512, 1024

status=4
random=0

# status, random
# 1: read 만 하게 함 read는 pc값 순서대로 random 값이 1이면 PC값을 난수를 받아  진행.
# 2: write만 하게 함 write은 pc값 순서대로 random 값이 1이면 PC값을 난수를 받아  진행.
# 3: read time_tick 만큼 진행, write 또한 time_tick 만큼 진행, random 값이 1이면 PC값을 난수를 받아  진행.
# 4: data.txt의 내용대로 수행함. random 값 무관.
```

해당 부분에서 숫자로 된 부분에 수정을 해주면 원하는 설정으로 돌릴 수 있다. config.txt의 내용이 아닌 값은 오류가 날 가능성이 높다.

7. Lesson

이 프로젝트는 가상 메모리 관리 시스템의 핵심적인 기능들을 구현하고 최적화하는 작업을 포함하였다. 주어진 목표에 맞추어 메모리 관리, 페이지 테이블 설정, TLB 관리, 페이지 교체 및 스와핑, CoW 등의 기능을 구현했

으며, 이를 통해 가상 메모리 시스템의 동작 방식을 이해하고 실습할 수 있었다.

특히, 페이지 테이블과 메모리 관리 시스템의 초기화, 페이지 교체 알고리즘, TLB와 관련된 로직을 직접 구현하면서 가상 메모리의 핵심적인 동작 원리를 체험할 수 있었다. 이 과정에서 메모리 할당, 주소 변환, 페이지 교체와 같은 개념들이 어떻게 실체화되는지를 이해할 수 있었고, 각 단계별로 필요한 메모리 관리 기법들이 어떻게 상호작용하는지를 실습으로 배울 수 있었다.

이번 프로젝트를 통해 가상 메모리 시스템에 대한 깊은 이해를 얻었으며, 실제 운영체제에서의 메모리 관리 기술이 얼마나 중요한지 실감할 수 있었다.