Virtual Memory Page Replacement Policy

소속 기계IT대학 컴퓨터공학과

학번 22112043, 22213482

이름 김민재, 박대형

[요 약]

오늘날 컴퓨터 사용 시 메모리를 얼마나 효율적으로 관리하느냐는 전체적인 시스템의 성능뿐만이 아니라, 사용자의 체감 속도에도 직접적인 영향을 미친다. 특히, 제한된 메모리 프레임 내에서 어떤 페이지를 유지하고 어떤 페이지를 교체할지를 결정하는 페이지 교체 정책(Page Replacement Policy)의 선택은 운영체제 설계에서 핵심적인 과제 중 하나로 꼽힌다.

본 프로젝트는 운영체제 수업 시간에 다룬 가상 메모리 구조와 페이지 교체 알고리즘에 대한 이해를 기반으로, 주요 정책들을 직접 구현하고 그 동작 과정을 시각적으로 비교 및 분석할 수 있는 시뮬레이터를 개발하였다. 사용자는 GUI를 통해 참조 문자열과 프레임 수를 입력할 수 있으며, 시뮬레이터는 페이지 접근이 진행될 때마다 상태를 실시간으로 표시한다. Hit은 초록색, Page Fault는 빨간색, Migrated는 보라색으로 표현되며, 최종 결과는 원형 차트와 함께 히트 수, 폴트 수 및 비율을 통계적으로 보여준다.

초기 단계에서는 FIFO, LRU, Clock(Second-Chance) 알고리즘을 Java로 구현하고, JavaFX를 활용해 시각화 기능을 구성하였다. 같은 입력 조건에서 비교한 결과, FIFO는 구조가 단순하지만, 교체 효율이 낮았고, LRU는 성능이 우수하나 구현이 복잡했으며, Clock은 두 방식의 균형점으로 평가되었다.

이들의 상대적 성능을 평가하기 위해 Optimal 알고리즘을 기준점으로 활용하였다. Optimal은 미래 참조를 알고 있다는 가정하에 페이지 폴트를 최소화하므로, 알고리즘 성능 비교에서 이론적 상한선을 제시한다.

하지만 기존 알고리즘은 모두 예측 기반 적응력이 부족하다는 공통된 한계를 지닌다. 이에 프로젝트 후반부에서는 참조 패턴을 기반으로 예측 가능한 교체 정책을 탐색하였고, 그 결과 LPR(Lowest Probability Replacement) 알고리즘을 설계하였다.

LPR은 과거 참조 이력을 분석해, 다음 등장 확률이 가장 낮은 페이지를 교체 대상으로 선택하는 방식이다. 앞으로 참조 가능성이 높은 페이지를 보존함으로써 전체 페이지 폴트를 줄이는 것이 목표이며, 초기에는 이력 부족으로 예측 정확도가 낮아질 수 있다는 단점도 있다.

본 프로젝트는 소개한 다섯 가지의 페이지 교체 정책의 구조적 차이와 성능을 실험적으로 비교하고, 메모리 관리 전략에 대한 통찰을 제공하는 계기가 되었다.

▸ **키워드**: Process, Memory Management, Page Replacement, Virtual Memory, Page Fault, Reference String, Frame, Algorithm Comparison, System Efficiency

목차

Ⅰ. 서론

1. 배경 및 개요

컴퓨터는 CPU나 메모리, 저장장치처럼 사용할 수 있는 자원이 한정되어 있다. 만약 이 자원들을 여러 프로그램이나 사용자가 동시에 사용하려고 접근한다면 충돌이 발생하게 되는데, 이때 운영체제가 이 상황을 잘 조율해 주는 역할을 한다. 즉, 운영체제는 한정된 자원을 알맞게 나눠주고, 사용자와 프로그램 사이에서 중간다리 역할을 하는 컴퓨터의 필수적인 소프트웨어다. 앞서 수업시간에서 이 운영체제가 수행하는 여러 역할에 대해 학습을 이어왔고, 메모리 관리라는 부분을 학습하게 되었다. 이 메모리 관리는 성능에 특히 큰 영향을 미친다. 요즘처럼 여러 프로그램이 동시에 실행되는 환경에서는, 메모리를 얼마나 효율적으로 나누어 쓰느냐에 따라 전체 시스템의 성능이 크게 좌우되기 때문이다. 또한, 이는 곧 시스템의 안정성과 직결되는 요인이기에 운영체제를 학습함에 매우 중요하게 다루어진다.

앞서 메모리의 구조부터 한정된 메모리를 효율적으로 사용하기 위해 가상 메모리(Virtual Memory)라는 핵심적인 기술을 학습했다. 가상 메모리란 실제 물리 메모리의 크기보다 더 큰 논리 메모리의 크기를 제공하여, 프로그램이 전체를 메모리에 올리지 않고도 필요한 부분만 선택적으로 불러와 실행할 수 있도록 하여 메모리의 효율을 높이는 개념이다. 이 구조에서 메모리를 페이지 단위로 나누고, 필요할 때 디스크에서 메모리로 데이터를 불러오는 과정에서 페이지 교체가 일어난다.

페이지 교체 알고리즘(Page Replacement Algorithm)은 이 가상 메모리 구조에서 발생하게 되는 페이지 교체 상황 속에서 어떤 페이지를 제거할 것이고, 또 어떤 페이지를 새로 적재할 것인가를 결정하는 핵심적인 정책이다. 이 선택은 단순히 제거와 적재를 넘어서서, 시스템 전체의 처리 속도와 자원 활용 효율성에 직접적인 영향을 미친다. 따라서 운영체제의 효율적인 설계를 위해서는 다양한 페이지 교체 알고리즘의 원리를 이해하고, 실제 상황에서 각각이 어떤 성능 차이를 보이는지를 비교해보는 것이 매우 중요하다.

본 프로젝트는 이러한 내용을 바탕으로, 5가지 대표적인 페이지 교체 알고리즘들을 직접 설계 및 구현하고, 시뮬레이션을 통한 분석으로 운영체제의 핵심 역할 중 메모리 관리의 구조와, 그 성능에 대한 이해를 심화시키고자 시작되었다.

2. 주요 목표 및 의의

본 프로젝트는 운영체제 수업에서 소개된 다양한 페이지 교체 알고리즘들의 개념을 정리하고, 각 알고리즘의 설계 방식과 구현 시 고려해야 할 요소들을 분석한 후, 이를 실제 코드로 직접 구현한다. 그 후 직접 구현한 시뮬레이션 프로그램을 통하여 같은 조건에서 각각의 성능을 수치화하여 비교하고 다시 분석한다.

이는 단순히 알고리즘을 이론적으로 이해하는 것을 넘어서서, 실제 코드를 작성하고 시뮬레이션 프로그램을 구상하고, 또 직접 구현함으로써 이론과 실습을 수행하는 것이다. 특히 각 알고리즘이 동일한 조건에서 어떤 방식으로 동작하는지를 직접 눈으로 확인하고 기준을 통해 평가하여 추상적이었던 알고리즘의 개념이 실제 시스템에서는 어떻게 적용되는가를 보다 구체적으로 체감한다.

본 프로젝트의 궁극적인 목표는 수업 시간에 학습한 알고리즘 이론을 단순히 이해하는 데에만 그치는 것이 아니다. 이를 바탕으로 실험을 설계하고, 그 결과를 시각화함으로써 운영체제의 메모리 관리를 실제 코드로 구현하여 직접 체감해보는 것이다. 나아가 기존 알고리즘들이 가진 단점이나 비효율적인 상황들을 분석하여 이를 개선할 수 있는 새로운 방식을 구상한다. 창의적인 생각을 통해 제안된 알고리즘 또한, 기존의 정책들과 동일한 실험 조건에서 테스트하고, 실제로 어떤 개선이 이루어졌는지도 검토하였다.

이를 통해 운영체제의 메모리 관리에 대하여 보다 능동적인 사고를 통해, 단순하게 이론을 공부한다를 넘어서 '문제를 정의하고 해결하는 과정을 경험하는 것에 큰 의의를 둔다.

3. 핵심 구현

본 프로젝트에서는 수업 시간에 배운 여러 페이지 교체 알고리즘들을 실제로 코드로 구현해보고, 다양한 조건에서 어떻게 작동하는지 실험해보는 데에 중점을 두었다. 단순히 알고리즘을 외우고 이해하는 것에 그치지 않고, 직접 눈으로 보고 체감할 수 있도록 구성하였다.

- 먼저, FIFO, LRU, OPT, Clock 알고리즘과 함께 새롭게 제안하는 LPR 알고리즘을 Java 코드로 구현하였다.
- JavaFX를 이용해 참조 문자열과 프레임 수를 사용자가 입력할 수 있도록 하고, 화면에 시각적으로 결과를 정리하는 기능도 구현하였다. 각 시점에서 어떤 페이지가 들어오고 나가는지를 색상 등으로 표시해 한눈에 확인할 수 있도록 했다.
- 알고리즘이 실행되면서 페이지 폴트가 몇 번 발생했는지, 어떤 경우에 더 잘 작동했는지를 수치로 확인하고 비교할 수 있도록 했다.

이를 바탕으로 같은 조건에서 알고리즘들을 비교해보며 어떤 방식이 어떤 상황에서 더 좋은 결과를 내는지를 분석해보았다. 이 과정을 통해 앞서 서술한 목표에 맞게 단순한 이론 공부를 넘어서, 실제 운영체제에서 메모리를 어떻게 다루는지를 직접 체험하고, 알고리즘의 차이점을 몸으로 느껴볼 수 있도록 하였다.

Ⅱ. 배경 지식 및 관련 기술

1. 배경 지식

본 프로젝트에서 구현하고자 하는 페이지 교체 알고리즘을 서술하기에 앞서 선행되어야 할 배경지식은 다음과 같다.

1.1. 가상 메모리 시스템의 기본 구조

운영체제는 한정된 물리 메모리 자원을 효율적으로 관리하기 위해 가상 메모리라는 개념을 사용한다. 이는 실제 메모리보다 더 넓은 논리 주소 공간을 프로그램에 제공함으로써, 프로그램이 전체 메모리를 차지하지 않고도 필요한 부분만 선택적으로 사용할 수 있게 한다.

가상 메모리 시스템에서는 메모리를 페이지 단위로 나누고, 이 페이지들이 물리 메모리의 프레임에 매핑되어 적재된다. 이때 어떤 페이지가 어떤 프레임에 있는지는 페이지 테이블(Page Table)을 참조하여 관리된다.

실행 중인 프로그램이 특정 주소를 참조했을 때 해당 페이지가 메모리에 존재하지 않으면 페이지 폴트(Page Fault)가 발생한다. 이 경우 운영체제는 디스크로부터 해당 페이지를 메모리 공간으로 불러온다. 이때 공간이 부족할 경우 부득이하게 기존의 페이지 중 하나를 선택하여 제거해야 하는데, 바로 이때 필요한 것이 페이지 교체 알고리즘이다.

1.2. 페이지 교체 정책의 필요성과 의미

페이지 교체 알고리즘은 페이지 폴트가 발생했을 때 어떤 페이지를 선택하여 제거하고, 교체할지를 결정한다. 단순히 자리를 비우는 것이 아니라, 시스템의 성능을 좌우하는 중요한 요소다. 자주 사용하는 페이지를 잘못 제거할 경우 다시 불러오는 데 더 많은 시간이 소요되고, 페이지 폴트가 빈번히 발생하게 된다. 이는 시스템 전체 성능 저하로 이어진다.

운영체제에서 사용하는 주요 페이지 교체 정책으로는 FIFO, LRU, Clock, Optimal 등이 있으며, 이들의 장단점과 성능 차이는 다양한 메모리 접근 패턴에 따라 달라질 수 있다.

1.3. Locality의 개념과 페이지 교체와의 관계

가상 메모리 시스템에서 효율적인 메모리 사용은 대부분 지역성(Locality) 개념에 기반을 둔다. 이 지역성은 크게 시간 지역성과 공간 지역성 두 가지로 분류된다. 먼저 시간 지역성 (Temporal Locality)이란, 최근 접근한 데이터는 금방 다시 접근될 가능성이 높다는 특성을, 공간 지역성 (Spatial Locality)은 현재 접근한 데이터와 인접한 위치의 데이터가 곧 접근될 가능성이 높다는 특성을 의미한다.

대부분의 페이지 교체 알고리즘은 이러한 지역성 특성을 고려해 페이지를 유지하거나 교체한다. LRU는 시간 지역성을, Clock은 그 근사치를 활용하여 성능을 높이는 것이 예시이다.

1.4. 페이지 교체 알고리즘의 평가 지표

CPU 스케줄링에서는 총 처리 시간(Turnaround Time), 대기 시간(Waiting Time), 응답 시간 (Response Time), 처리량(Throughput), 공정성(Fairness) 등 다양한 평가 지표가 사용되지만, 페이지 교체 알고리즘에서는 대부분은 페이지 폴트 수, 단 하나의 지표로 성능을 판단한다.

실제 시스템 환경에서는 교체 비용, 디스크 접근 시간, 메모리 관리 오버헤드 등 다양한 요소가 전체 성능에 영향을 미친다. 하지만 본 프로젝트에서는 이러한 복잡한 요소들을 배제하고, 여러 참조 패턴과 프레임 수 변화에 따른 페이지 폴트율(Page Fault Rate)의 변화를 중심으로 각 알고리즘의 상대적인 성능을 비교하는 데 초점을 맞추었다.

2. 관련 기술

본 프로젝트에서는 페이지 교체 알고리즘을 직접 설계하고 시뮬레이션하기 위해 여러 가지 기술 요소를 활용하였다.

(1) Java

알고리즘 구현을 위해 Java를 사용하였다. 객체 지향적인 구조를 바탕으로, Frame 클래스와 PageReplacementPolicy 인터페이스 등을 설계하여 각 알고리즘의 동작을 명확하게 구분하였고, 유지보수나 확장 또한, 용이하도록 구성하였다.

(2) JavaFX 프레임워크

알고리즘의 동작 과정을 시각적으로 확인할 수 있도록, JavaFX를 기반으로 간단한 GUI를 구현하였다. 프레임의 상태를 시간 순서대로 애니메이션처럼 보여주며, 각 상황에 따라 히트(초록), 폴트(빨강), 교체(보라) 등의 색상으로 시각화하여 이해를 돕도록 하였다.

(3) 자료구조 활용

페이지 교체 알고리즘별 특성에 맞는 자료구조를 선택하여 구현하였다.

- Queue(LinkedList)는 FIFO, LRU와 같이 순서 기반의 정책에서 사용되었다.
- ArrayList는 프레임 리스트를 저장하거나 반복문을 통한 탐색에서 활용되었다.
- Map<Character, Map<Character, Integer>>은 사용자 정의 알고리즘인 LPR의 확률 기반 학습 구조를 구성하기 위해 사용되었다. 이중 맵 구조를 통해 이전 페이지 → 다음 페이지의 등장 빈도를 누적하고 이를 기반으로 예측에 활용하였다.

III. (본론) 페이지 교체 알고리즘의 구현 및 분석

1. 주요 주제의 개요

본론에서는 운영체제 수업에서 다뤘던 4가지 기본 페이지 교체 알고리즘들을 설계 및 구현하고, 구조와 특징들을 정리한다. 더불어 새롭게 제안한 알고리즘의 구조와 작동 방식도 함께 분석하고 정리한다. 이 내용은 추후 서술할 IV. 성능 평가에서 비교 분석하는 데 활용한다.

1.1. 대상 알고리즘

(1) FIFO : 가장 먼저 들어온 페이지를 교체한다.

(2) Optimal : 앞으로 가장 나중에 다시 참조될 페이지를 교체한다.

(3) LRU : 가장 오랫동안 참조되지 않은 페이지를 교체한다.

(4) Clock : FIFO 구조에 참조 비트 개념을 도입하여 최근에 사용된 페이지는 한 번 더 기회를 부여하는 방식이다.

(5) LPR : 기존 알고리즘들의 단점을 개선하기 위해 새롭게 제안한 교체 정책이다.

각각의 알고리즘들은 서로 다른 방식으로 교체 대상을 선택하며, 장단점 또한 다르다. 그렇기에 공통된 조건 속에서 성능을 비교함으로써, 어떤 알고리즘이 어떤 상황에 더 적합한지를 파악할 수 있을 것이다.

1.2. 공통 복잡도 분석

페이지 교체 알고리즘은 참조 문자열을 차례대로 읽어가며, 현재 메모리 프레임 내에 해당 페이지가 존재하는지를 탐색하는 공통된 구조를 가진다. 이때의 성능은 다음과 같은 요소들에 의해 결정된다.

- N : 참조 문자열의 길이
- F : 메모리 프레임 수

1.3. 시뮬레이터 구조

보다 공정하고 정확한 비교 실험을 위해, 모든 알고리즘은 동일한 환경에서 동작하도록 설계했다. 이를 위해 Java 기반의 공통 구조를 먼저 만들었고, 이 구조는 크게 세 부분으로 나뉜다.

① PageReplacementPolicy 인터페이스
: 각 알고리즘이 공통으로 가져야 할 기능을 정의해둔 틀이다. 예를 들어 참조 문자열 설정, 프레임 수 설정, 알고리즘 실행, Hit 및 Fault 수 조회 같은 기능들이 여기에 포함된다. 이렇게 통일된 틀을 사용함으로써 메인 시뮬레이터에서 단순히 알고리즘을 선택하여 쉽게 비교할 수 있다.

② Frame 클래스
: 각 프레임을 객체로 생성하였다. 단순히 페이지값만 저장하는 게 아니라, Clock 알고리즘처럼 참조 비트를 쓰도록 설계하였다. 이를 통해 확장에도 쉬우며, 시각화 및 결과 분석 시에도 데이터를 일관되게 다룰 수 있다.

③ ReferenceStringGenerator
: 실험 시, 사용자가 참조 문자열을 직접 입력한다면, 특정 알고리즘에 유리한 입력을 넣게 되는 불공정한 실험 환경이 조성될 수 있다. 이를 방지하기 위해 랜덤으로 참조 문자열을 생성하는 클래스를 만들었다. 이때 랜덤으로 생성되는 참조 문자열의 형식은 아래의 조건들을 만족해야 한다.

- 길이는 6~24 사이,
- 각 문자는 A~Z 중 하나,

이를 통해 실험을 반복하면서 다양한 입력 케이스를 확보할 수 있게 된다.

이러한 시뮬레이터의 구조를 통하여 각각의 알고리즘의 동작 원리를 쉽게 시각화할 수 있으며, 동일한 조건에서 공정한 비교도 가능해졌다. 이어지는 내용에서는 각 알고리즘을 하나씩 심도 있게 소개한다. 어떤 원리로 작동하였으며, 어떻게 설계하고 구현했는지를 살펴보고 주요 특징과 함께 정리한다.

2. 알고리즘 소개 및 구현

2.1. FIFO (First-In-First-Out)

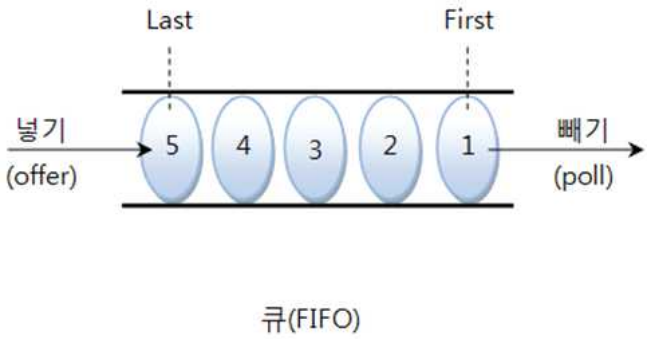
(1) 소개 및 작동 원리

앞서 운영체제 수업에 다뤘던 CPU 스케줄링 알고리즘 중에서 FCFS(First-Come, First-Served)는 말 그대로 먼저 도착한 프로세스부터 순서대로 처리하는 방식이었다. 이와 비슷하게, FIFO 페이지 교체 알고리즘 또한 "가장 먼저 들어온 것부터 처리"하는 방식이다. 여기서 말하는 "먼저 들어온 것"은 메모리에 적재된 페이지들을 의미한다. FCFS는 CPU를 어떤 순서로 나눠줄지를 결정하는 데 사용되었다면, FIFO는 메모리에 올라온 페이지 중 어떤 것을 제거할지를 결정할 때 사용된다.

예를 들어 프로그램 실행 중 새로운 페이지가 필요한 상황에서 메모리에 빈 공간이 없다면, FIFO는 그 중 가장 오래전에 올라온 페이지를 선택해서 제거한다. 이때의 핵심은 어떤 페이지가 얼마나 자주 사용되었는가는 전혀 고려되지 않고, 오로지 '먼저 들어왔기 때문'에 교체 대상이 된다는 것이다.

(2) 구현 방식 및 사용 자료구조

FIFO 페이지 교체 알고리즘의 구현은 간단하다. 이 알고리즘은 말 그대로 페이지가 들어온 순서대로 제거되는 구조로 되어 있어, 자연스럽게 큐(Queue)라는 자료구조를 떠올릴 수 있다. 큐는 선입선출(First-In-First-Out)방식으로 작동하기 때문에, 가장 먼저 메모리에 들어온 페이지는 가장 먼저 매모리에서 제거되는 구조와 일치한다.



[그림 1] 큐(Queue) 동작 방식

본 프로젝트에서는 자바의 LinkedList를 큐처럼 활용하여 구현하였다. LinkedList는 큐 인터페이스를 구현하고 있기 때문에 offer()와 poll() 메서드를 통하여 큐처럼 동작하게 할 수 있다.

(3) Pseudo Code

1. 빈 큐를 만든다.
2. 참조 문자열의 각 페이지에 대해 반복한다:
a. 큐 안에 해당 페이지가 있으면 hit
b. 없으면 fault
ㄱ. 큐가 가득 찼으면 가장 앞의 페이지를 제거
ㄴ. 새 페이지를 큐에 추가
3. 매번 현재 프레임 상태를 기록한다.

(4) 구현 시 고려사항

① 프레임이 가득 찼을 때는 poll()을 통해 가장 먼저 들어온 페이지를 제거해야 한다.
② 페이지 존재 여부는 프레임 내부의 page 필드를 탐색하여 판단한다.
- 자바의 == 연산자는 char 타입에서는 문제가 없지만, Wrapper 클래스(Character)일 경우 equals()를 사용해야 한다.
- 시각화를 위해 frameSnapshots를 저장하므로, 참조 문자열 길이가 길면 메모리를 많이 사용할 수 있다.
- 기록되는 snapshot은 각 시간에 실제 메모리에 올라간 페이지만을 표현하며, 빈 슬롯은 null 또는 생략된 값으로 표현될 수 있다.

(5) 실제 구현 코드 일부 설명

① 메모리 내 페이지 존재 여부 확인

```
for (char page : referenceString) {
    boolean hit = false;
    for (Frame frame : frames) {
        if (frame.page == page) {
            hit = true;
            break;
        }
    }
}
```

이중 반복문을 통해 현재 메모리 안에 참조 중인 페이지가 있는지 검사한다. 존재하면 hitCount++, 없으면 faultCount++가 된다.

② 페이지 폴트 처리 및 프레임 교체

```
if (!hit) {
    if (frames.size() == frameSize) frames.poll();
    frames.offer(new Frame(page));
}
```

Page Fault가 발생하면 먼저 프레임이 가득 찼는지 확인한다. 만약 현재 메모리에 프레임이 가득 찼다면 그중 가장 오래된 페이지를 제거하고 새 페이지를 삽입한다. 이후, recordSnapshot();를 통해 이 시점의 메모리 상태를 기록해 시각화에 사용한다.

(6) 복잡도 분석

① 시간 복잡도 : O(N x F)

매 참조마다 최대 F개의 프레임을 순회하며 hit 여부를 확인해야 하므로 전체적으로 N * F만큼의 시간이 소요된다.

② 공간 복잡도 : O(F)

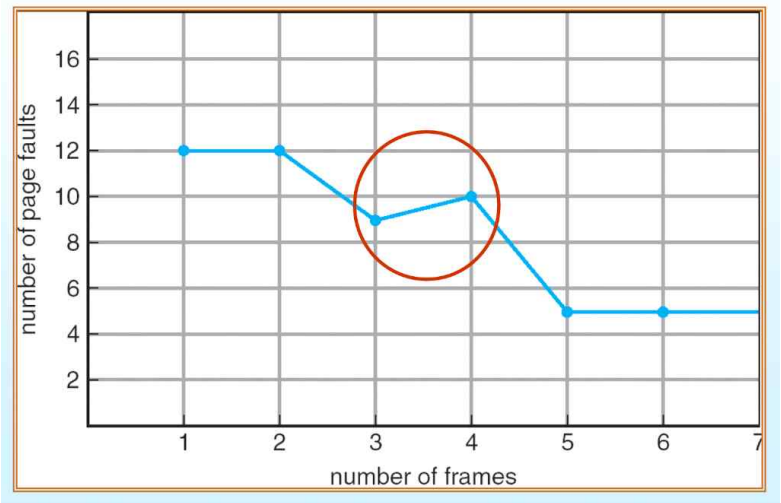
각 시간대별로 스냅 샷을 저장하고 있기 때문에 최악에는 N개의 F 길이 리스트가 생성된다. 하지만 실제 운영체제의 페이지 교체에서는 이러한 스냅 샷을 저장할 필요가 없으므로 총 프레임의 개수만큼의 공간 복잡도가 생긴다. 따라서 O(F)의 공간 복잡도로 매우 효율적인 편이다.

(7) 특징

운영체제에서 페이지 교체 알고리즘의 성능을 평가할 때, 일반적으로 메모리의 프레임 수가 많아질수록 페이지 폴트의 발생 빈도는 줄어들 것으로 예상한다. 프레임 수가 많다는 것은 곧 더 많은 페이지를 메모리에 보관할 수 있다는 것이기 때문이다.

이 개념은 우리 일상 속의 상황과도 비슷하게 연결된다. 예를 들어, 자주 입는 옷은 가까운 곳에 두고, 당분간 입지 않을 계절 지난 옷은 드레스룸이나 창고에 넣어두는 것이 일반적이다. 옷장의 크기가 넉넉하다면 자주 입는 옷을 꺼내 입기 쉬워지는 것처럼, 프레임이 수가 많을수록 필요한 페이지를 메모리 안에서 바로 사용할 가능성이 높아진다. 반대로 옷장이 좁을 때 자주 입는 옷조차 꺼내고 다시 넣는 번거로움이 반복되듯, 메모리가 부족하면 자주 사용하는 페이지도 반복적으로 교체되어야 하고, 이는 시스템 성능의 저하로 이어질 수 있다.

그런데 FIFO 알고리즘에서는 이 상식이 항상 통하지 않는다. 오히려 프레임 수를 늘렸는데도 페이지 폴트가 더 늘어나는 이상한 현상이 나타나기도 하고, 이를 Belady의 역설(Belady's Anomaly)이라고 부른다.



[그림 2] FIFO 알고리즘의 Belady's Anomaly

이는 FIFO 알고리즘의 대표적인 단점으로, 실제로 [그림 2]을 보면, 프레임 수가 3개일 때보다 4개일 때 페이지 폴트가 더 많아지는 구간이 있다는 걸 확인할 수 있다.

이러한 현상이 발생하는 이유는 생각보다 간단하다. FIFO는 단지 "들어온 순서"만을 기준으로 페이지를 제거하기 때문에, 자주 사용되는 페이지더라도 먼저 들어왔다는 이유 하나만으로 쉽게 교체될 수 있기 때문이다. 만약 그 페이지가 금방 다시 필요해진다면, 불필요한 페이지 폴트가 반복적으로 발생하게 되고, 결국 프레임을 더 늘렸음에도 시스템 성능이 더 안 좋아지는 역설적인 상황이 생기게 되는 것이다.

(8) 정리

FIFO는 페이지 교체 알고리즘 중 가장 단순한 방식이다. 큐라는 자료구조만을 사용하여 매우 쉽게 구현할 수 있으며, 메모리에 들어온 순서를 그대로 유지하면 된다는 점에서 알고리즘이 매우 직관적이다. FIFO는 이처럼 구현과 이해가 쉽다는 장점이 있다.

하지만 페이지의 '중요도'나 '최근 사용 여부'를 전혀 고려하지 않기 때문에, 자주 참조되는 페이지라도 먼저 들어왔다는 이유만으로 교체되는 경우가 발생할 수 있다. 그 결과, 불필요한 페이지 폴트가 반복되면서 전체 성능이 오히려 저하될 수 있다.

특히 일반적으로 프레임 수가 늘어나면 페이지 폴트는 줄어들어야 하지만, 오히려 늘어나는 현상인 Belady의 역설은 FIFO의 한계를 잘 보여주는 사례이다.

2.2. Optimal

(1) 소개 및 작동 원리

Optimal 페이지 교체 알고리즘은 이름처럼 가장 이상적이며, 최적의 방식이다. 작동 원리의 핵심은 새로운 메모리 적재를 위해 페이지 교체가 발생할 때, 앞으로 가장 니중에 사용될 페이지를 교체 대상으로 선택하여 제거한다는 것이다. 이를 통해 페이지 폴트 발생 빈도를 최소화하여 최적의 결과를 보장한다.

하지만 마치 당장 우리에게 내일 무슨 일이 일어날지를 알 수 없는 것처럼, 페이지를 교체할 때 미래의 메모리 접근 순서를 정확하게 아는 것은 불가능하다. 이처럼 Optimal 알고리즘은 앞으로의 참조 순서를 미리 알아야 한다는 전제가 필요하기에 실제 환경에서는 구현할 수 없다,

그럼에도 지금 다루는 Optimal 알고리즘은 이론상 최적의 알고리즘으로서 큰 의미가 있기에 다른 페이지 교체 정책들과의 성능 비교를 위한 기준으로 활용된다.

(2) 구현 방식 및 사용 자료구조

Optimal 알고리즘은 미래에 어떤 페이지가 가장 늦게 다시 사용될지를 기준으로 교체 대상을 정한다. 즉, 페이지 폴트가 발생했을 때, 현재 메모리에 올라와 있는 모든 페이지 중에서 가장 나중에 참조되거나, 아예 다시 참조되지 않을 페이지를 골라 제거한다.

구현 방식은 우선 페이지 폴트가 발생했을 때, 현재 프레임에 있는 각 페이지에 대해 앞으로의 참조 문자열에서 언제 다시 사용되는지를 확인한다. 이 중에서 가장 나중에 다시 사용될 페이지를 선택하여 교체하고, 만약 어떤 페이지는 다시 사용되지 않는다면 그 페이지가 가장 먼저 교체 대상이 된다.

이 알고리즘은 매번 전체 참조 열을 탐색하기 때문에 일종의 완전 탐색(Brute-Force) 방식에 가깝다. 모든 프레임 페이지에 대해 앞으로의 참조 시점을 일일이 확인하기에 시간 복잡도는 높지만, 그만큼 정확한 교체 판단이 가능하다는 점에서 이론적으로 큰 의미가 있다.

(3) Pseudo Code

1. 빈 프레임 리스트를 생성한다.
2. 참조 문자열의 각 페이지에 대해 반복한다:
a. 현재 프레임에 페이지가 있다면 → hit 처리
b. 없다면 → fault 처리
ㄱ. 프레임이 가득 찼다면:
- 각 프레임의 페이지에 대해 앞으로 가장 늦게 등장하는 인덱스를 조사
- 가장 나중에 사용될 페이지를 선택하여 교체
ㄴ. 프레임이 남아있다면 그냥 추가
3. 매번 현재 프레임 상태를 기록한다

(4) 구현 시 고려사항

① Optimal 알고리즘은 앞으로의 참조 순서를 미리 알아야 하므로 각 페이지 교체 시점마다 모든 이후 참조 문자열을 검사해야 한다.

② 프레임 내의 페이지 중에서 앞으로 가장 나중에 사용되거나, 다시는 사용되지 않는 페이지를 제거하는 것이 핵심이다.

③ 구현 시 해당 페이지가 다시는 등장하지 않는 경우를 처리하기 위해 Integer.MAX_VALUE를 사용하여 비교 기준으로 정했다. 이것은 무한히 먼 미래에 사용됨을 의미하며 가장 나중에 사용되는 것으로 간주한다.

④ 실제 운영체제는 미래의 참조를 알 수 없으므로 이 알고리즘을 사용할 수 없다. 다만 시뮬레이터처럼 전체 참조 문자열을 고정된 데이터로 가정하는 환경에서는 구현할 수 있다.

(5) 실제 구현 코드 일부 설명

```
if (!hit) {
    if (frames.size() == frameSize) {
        int farthestIndex = -1;
        int farthestFrameIndex = 0;

        for (int j = 0; j < frames.size(); ++j) {
            int nextUse = Integer.MAX_VALUE;
            for (int k = i + 1; k < referenceString.size(); ++k) {
                if (referenceString.get(k) == frames.get(j).page) {
                    nextUse = k;
                    break;
                }
            }
            if (nextUse > farthestIndex) {
                farthestIndex = nextUse;
                farthestFrameIndex = j;
            }
        }
        frames.set(farthestFrameIndex, new Frame(page));
    } else {
        frames.add(new Frame(page));
    }
}
```

페이지 폴트가 발생했을 때, 현재 메모리에 올라와 있는 각 페이지가 이후 참조 문자열에서 언제 다시 사용될지를 하나하나 조사한다. 먼저 프레임이 꽉 찼는지 확인한 후, 프레임이 가득 차 있다면 frames 리스트를 순회하면서 각 페이지들이 언제 참조 문자열에서 다시 나타나는지를 확인한다. 이때 참조 문자열의 현재 인덱스 이후부터 끝까지 탐색하며, 가장 먼저 그 페이지가 다시 등장하는 위치를 찾는다. 만약 해당 페이지가 이후에 전혀 등장하지 않는다면, 다시 사용되지 않는 것으로 간주하여 Integer.MAX_VALUE라는 값을 할당한다. 이렇게 페이지마다 다음 사용 시점을 계산한 후, 가장 늦게 다시 사용될 페이지를 교체 대상으로 삼는다.

이러한 방식은 미래의 참조를 안다는 전제가 있기 때문에 정확하고 페이지 폴트를 최소화할 수 있지만, 참조 문자열 전체를 반복적으로 탐색해야 하므로 매우 비효율적이다. 실제 코드에서는 farthestIndex를 이용해 가장 나중에 사용될 페이지의 인덱스를 저장하고, 그에 해당하는 farthestFrameIndex 위치에 새 페이지를 덮어쓴다. 만약 아직 프레임이 가득 차지 않았다면, 단순히 새로운 페이지를 frames 리스트에 추가만 하면 된다.

(6) 복잡도 분석

① 시간 복잡도: O(N^2 * F)

참조 문자열의 각 페이지에 대해 다음 사용 시점을 찾기 위해 참조 열의 나머지 부분을 모두 탐색해야 한다. 따라서 페이지가 N번 참조되고, 프레임 수가 F개일 때, 참조마다 최대 F개의 페이지에 대해 이후 참조 시점(N개)을 확인하므로, 최악에는 O(N × F × N), 즉 $O(N^2 \times F)$의 시간 복잡도를 가진다.

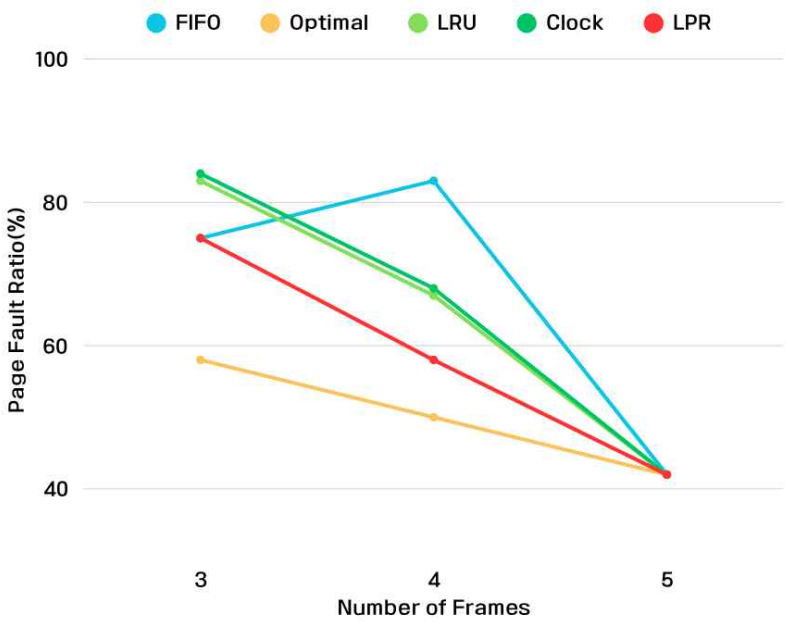
② 공간 복잡도: O(F)

시뮬레이터에서는 매시간의 프레임 상태를 시각화하기 위해 frameSnapshots를 저장하고 있으므로, 총 N번의 참조마다 F개의 프레임 정보를 기록하게 된다. 따라서 공간 복잡도는 O(N × F)이다. 하지만 실제 운영체제에서는 시각화를 위한 스냅 샷을 저장하지 않기 때문에, 순수 알고리즘 기준으로는 O(F)의 공간만 필요하다.

(7) 특징

Optimal 알고리즘은 앞으로 어떤 페이지가, 언제 다시 사용되는가를 모두 알고 있다는 전제가 필요하다. 이 조건이 충족된다면, 가장 나중에 다시 사용할 페이지를 교체 대상으로 선택할 수 있으므로, 결과적으로 가장 적은 수의 페이지 폴트를 만들어내는 것이 가능하다. 이 때문에 이론적으로는 항상 최고의 성능을 보이는 알고리즘으로 간주한다.

하지만 현실의 운영체제는 미래의 페이지 접근을 예측할 수 없으므로 실제 환경에서는 이 알고리즘을 그대로 적용하는 것이 불가능하다. 또한, 각 페이지 교체 시마다 이후의 참조 문자열을 매번 전부 확인해야 하므로, 계산량이 많고 성능상의 부담도 크다. 이런 점에서 Optimal은 실제 시스템에서 활용하기보다는, 알고리즘의 '이론적 성능 한계'를 보여주는 비교 기준으로 사용되는 경우가 많다.



[그림 3] 프레임 수 별 페이지 폴트 비율

본 프로젝트에서도 이후 서술할 IV. 성능 평가 장에서 Optimal 알고리즘을 기준선으로 삼아, 다른 알고리즘들의 성능이 얼마나 근접하는지를 비교하였다. 위의 [그림 3]가 바로 그 예시이다. Optimal 알고리즘을 노란색 그래프로 표시하여, 각 알고리즘의 페이지 폴트 수가 이론적 최적값에 얼마나 가까운지를 직관적으로 확인할 수 있도록 시각화하였다.

(8) 정리

본 프로젝트의 실험 환경에서는 참조 문자열이 고정되어 있기에, Optimal 알고리즘을 직접 구현하고 시뮬레이션해볼 수 있다. 각 프레임에 있는 페이지가 이후에 언제 다시 호출되는지를 일일이 탐색한 후, 가장 늦게 재사용될 페이지를 선택하는 방식으로 구현하였다. 이는 완전 탐색 방식에 가까워 연산량은 많지만, 비교 대상이 되는 알고리즘(FIFO, LRU 등)과의 성능 차이를 명확하게 확인할 수 있다는 장점이 있다.

2.3. LRU (Least Recently Used)

(1) 소개 및 작동 원리

LRU는 이름 그대로, 가장 오랫동안 참조되지 않은 페이지를 교체 대상으로 삼는 방식의 알고리즘이다.

운영체제는 제한된 메모리 공간 내에서 필요한 페이지를 효율적으로 유지하기 위해 시간 지역성이라는 특성을 활용한다. 즉, 최근에 참조된 페이지는 가까운 시점에 다시 참조될 가능성이 높다는 가정이다.

이러한 가정에 따라, LRU는 현재 메모리에 빈자리가 없고 새로운 페이지를 적재해야 할 경우, 가장 마지막으로 참조된 시점이 오래된 페이지부터 제거한다. 다시 말해, 사용된 지 가장 오래된 페이지는 앞으로도 당분간 사용되지 않을 것으로 판단하고 교체하는 것이다.

이 방식은 자주 사용되는 페이지를 메모리에 오래 남겨두고, 그렇지 않은 페이지는 빠르게 내보내게 하여 불필요한 페이지 폴트를 줄이고 전체 성능을 개선하려는 전략에 가깝다.

(2) 구현 방식 및 사용 자료구조

본 운영체제 수업에서는 LRU 알고리즘의 구현 방식으로 두 가지를 소개한다.

① Counter 기반 LRU

이 방식은 페이지마다 별도의 카운터 필드를 두어, 해당 페이지가 마지막으로 참조된 시점을 기록하는 방식이다. 시스템 클럭이 존재한다고 가정하고, 이 시계를 기준으로 각 페이지의 참조 시간을 저장한다. 구현 방식은 다음과 같다.

1. 시스템 전체에서 공용으로 사용하는 시계(클럭)를 가정한다.
2. 어떤 페이지가 참조되면, 그 페이지의 페이지 테이블 항목에 현재 시각 값을 기록한다.
3. 페이지 교체가 필요한 시점이 오면, 페이지 테이블을 순회하며 가장 오래된 시각을 가진 페이지(즉, 가장 오랫동안 참조되지 않은 페이지)를 찾아 교체 대상으로 선정한다.

이 방식은 이론적으로는 정확한 LRU 구현이 가능하다는 장점이 있지만, 단점 또한 명확하다. 모든 페이지의 카운터 값을 매번 확인해야 하기 때문에 탐색 시간이 오래 걸리고, 페이지마다 카운터 필드를 유지해야 하므로 추가적인 메모리 자원 소모 및 성능 오버헤드가 발생한다.

② Stack 기반 LRU

이 방식은 페이지가 참조될 때마다 해당 페이지를 스택의 가장 위로 옮기는 방식이다. 즉, 가장 최근에 사용된 페이지는 항상 스택의 맨 위에 있고, 스택의 맨 아래에 있는 페이지가 가장 오래 사용되지 않은 페이지가 되므로, 교체 대상은 항상 맨 아래에 있는 것으로 결정된다.

Stack 기반의 방식은 이론적으로 LRU를 정확하게 구현할 수 있고, 교체 대상 선정 시 추가 탐색이 필요 없다는 장점이 있다. 하지만 실제 구현에서는 페이지 참조가 발생할 때마다 스택 내부의 구조를 수정하거나 포인터를 조정해야 하므로 오버헤드가 발생할 수 있다. 또한, 이 방식은 스택 형태의 자료구조를 별도로 유지해야 하므로 Counter 방식과 마찬가지로 추가적인 메모리 자원이 필요하다.

본 구현에서는 앞선 방식들보다 간단한 방식인, LinkedList를 이용한 구조를 채택하였다.

이는 명시적인 시간 정보나 별도의 스택 구조 없이, 리스트의 순서를 통해 참조 순서를 간접적으로 추적하는 방식이다.

- frames는 현재 메모리에 올라와 있는 페이지들을 저장하는 LinkedList 자료구조이다.
- 리스트의 앞쪽에 있는 요소일수록 오래전에 사용된 페이지이며,
- 리스트의 뒤쪽에 있는 페이지가 최근에 참조된 페이지를 의미한다.

페이지가 이미 메모리에 존재한다면, 해당 페이지를 리스트에서 제거한 뒤 다시 맨 뒤에 추가해 최근에 참조되었음을 반영한다. 반면, 참조한 페이지가 메모리에 없고 프레임이 가득 찼다면, 리스트의 맨 앞에 있는 가장 오랫동안 참조되지 않은 페이지를 제거하고, 새로운 페이지를 뒤쪽에 추가하는 방식으로 처리한다.

이러한 방식은 상대적으로 간단하면서도 LRU의 핵심 개념을 비교적 정확하게 반영할 수 있기 때문에, 이번 시뮬레이터에서는 가장 적합한 구현 방식으로 선택되었다.

(3) Pseudo Code

1. 빈 큐(프레임 리스트)를 만든다.
2. 참조 문자열의 각 페이지에 대해 반복한다:
- 큐 안에 해당 페이지가 있으면 (== Page Hit)
- 기존 페이지를 큐에서 제거
- 해당 페이지를 큐의 맨 뒤에 추가 (최근 사용된 것으로 간주)
- 큐 안에 해당 페이지가 없으면 (== Page Fault)
- 큐가 가득 찼다면
- 가장 앞의 페이지를 제거 (가장 오래된 페이지)
- 새 페이지를 큐의 맨 뒤에 추가
3. 각 단계에서 현재 프레임 상태를 기록한다.

여기서 사용된 Queue는 실제로는 LinkedList로 구현되며, 페이지의 참조 순서를 반영하는 데 사용된다. 페이지가 이미 존재하는 경우에는 기존 위치에서 해당 페이지를 제거한 뒤, 맨 뒤로 다시 추가함으로써 가장 최근에 사용된 페이지가 항상 뒤쪽에 있도록 정렬된다.

반대로 페이지가 존재하지 않고 프레임이 가득 찼다면, 맨 앞의 요소(가장 오래된 페이지)를 제거하고 새로운 페이지를 뒤에 추가한다. 이처럼 구조적으로는 FIFO와 비슷하지만, 참조 발생 시 위치를 갱신한다는 점에서 실제 동작 방식은 다르다.

(4) 구현 시 고려사항

구현에서는 frames 리스트를 매번 순회하여 현재 참조 중인 페이지가 이미 존재하는지를 확인해야 한다. 만약 해당 페이지가 존재한다면, 이를 리스트에서 제거한 후 맨 뒤에 다시 추가함으로써 최근에 참조되었음을 반영해야 한다.

이 과정에서 Java의 for-each 문으로 리스트를 순회하면서 remove()를 직접 호출하면 ConcurrentModificationException이 발생할 수 있으므로, Iterator를 사용하여 안전하게 요소를 제거해야 한다. 이는 리스트 순회 중 구조 변경이 발생하는 경우를 방지하기 위한 중요한 구현상의 고려사항이다.

또한 새로운 페이지를 삽입할 때는 항상 리스트의 맨 뒤에 추가해야 하며, 이를 통해 리스트의 끝에 있는 요소가 가장 최근에 참조된 페이지임을 나타내도록 설계된다.

시뮬레이터에서는 참조가 발생할 때마다 메모리 상태를 스냅 샷 형태로 저장하게 되는데, 참조 문자열이 길어질수록 이 스냅 샷 데이터가 많아져 메모리 사용량이 증가할 수 있다.

(5) 실제 구현 코드 일부 설명

```
Iterator<Frame> it = frames.iterator();
while (it.hasNext()) {
    Frame frame = it.next();
    if (frame.page == page) {
        hit = true;
        it.remove(); // 기존 위치 제거
        break;
    }
}
```

페이지가 현재 프레임에 존재하는 경우, Iterator를 이용하여 안전하게 해당 요소를 리스트에서 삭제한다.

```
if (hit) {
    frames.addLast(new Frame(page)); // 최근 사용된 것으로 갱신
} else {
    if (frames.size() == frameSize) frames.removeFirst(); // 가장 오래된 페이지 제거
    frames.addLast(new Frame(page)); // 새 페이지 추가
}
```

페이지가 존재하면, 기존 위치에서 제거한 뒤 리스트 맨 뒤에 다시 삽입하여 최근 사용됨을 반영한다. 반대로 페이지가 존재하지 않을 경우, 가장 오래된 페이지를 제거하고 새 페이지를 맨 뒤에 추가한다.

이 구조를 통해 리스트의 앞쪽은 오래된 페이지, 뒤쪽은 최근 참조된 페이지가 되도록 유지한다. 이후 recordSnapshot();을 통해 시각화에 활용한다.

(6) 복잡도 분석

① 시간 복잡도: O(N × F)

참조 문자열의 각 페이지에 대해 매번 현재 프레임 리스트를 순회하며 해당 페이지가 존재하는지 검사해야 한다. 최악에는 프레임 수만큼 비교가 이루어지므로 참조 1건당 O(F)의 시간이 소요된다. 전체 참조 수가 N이므로, 총 시간 복잡도는 O(N*F)가 된다.

② 공간 복잡도: O(F)

매 시간마다 프레임 상태를 기록한 스냅 샷이 저장되므로, 최악에는 N개의 F길이 리스트가 생성된다. 실제 운영체제에서 스냅 샷은 저장하지 않기 때문에 핵심 알고리즘만 본다면 O(F)의 공간 복잡도로 매우 효율적인 수준이다.

(7) 특징

LRU 알고리즘은 시간 지역성을 잘 반영하는 대표적인 페이지 교체 정책이다. 최근에 참조된 페이지는 가까운 미래에도 금방 다시 참조될 가능성이 높다는 전제로 동작하며, 이 특성 덕분에 불필요한 페이지 폴트를 줄이는 데 효과적이다.

다만, LRU는 구현 측면에서는 까다로운 편이다. 참조 순서를 계속해서 추적하고, 페이지가 다시 사용될 때마다 그 위치를 갱신하므로 삽입/삭제/갱신 연산이 자주 발생한다. 이를 효율적으로 구현하려면 O(1) 시간 복잡도를 가지는 자료구조가 필요한데, 단순한 배열이나 큐로는 어렵다. Java에서는 LinkedHashMap과 같은 자료구조가 도움되지만, 운영체제 커널 수준에서 이를 직접 구현하기에는 메모리나 처리 비용 면에서 부담이 크다.

또한, LRU는 하드웨어 차원에서 직접적인 지원이 없다. 즉, 어떤 페이지가 마지막으로 참조되었는지를 운영체제가 직접 추적하고 관리해야 하며, 이 때문에 소프트웨어 오버헤드가 발생한다는 점도 한계로 작용한다.

(8) 정리

LRU는 페이지 교체 알고리즘 중에서 시간 지역성을 가장 잘 반영하는 정책으로, 이론적으로도 우수한 성능을 보여준다. 참조 순서를 기준으로 가장 오랫동안 참조되지 않은 페이지를 교체 대상으로 삼기 때문에, 자주 사용하는 페이지가 메모리에 오래 유지될 수 있다.

이번 구현에서는 LinkedList를 활용해 가장 단순한 형태로 LRU를 표현하였다. 리스트의 순서를 통해 참조 시점을 간접적으로 추적하고, 페이지가 참조될 때마다 리스트의 위치를 조정하는 방식이다. 이처럼 구조가 간단하고 직관적이어서 시뮬레이터용으로 적합하다,

하지만 실제 운영체제에서는 LRU를 그대로 구현하기 어렵다. 참조 순서를 실시간으로 추적하려면 추가적인 하드웨어 지원 또는 자료구조 관리 비용이 크기 때문이다. 따라서 실제 시스템에서는 Clock 알고리즘이나 Working Set 기반 알고리즘 등, LRU의 성능을 근사하면서도 효율적인 대체 알고리즘을 사용한다.

2.4. Clock

(1) 소개 및 작동 원리

Clock 페이지 교체 알고리즘은 LRU의 근사치를 계산하는 효율적인 대체 알고리즘으로, 실제 운영체제에서도 널리 사용된다. LRU는 최근에 사용된 페이지를 정확하게 추적해야 하므로 구현 비용이 많이 드는 반면, Clock은 이 정보를 단순한 참조 비트(Reference Bit)를 통해 간접적으로 추적함으로써 구현을 간소화한다.

이 알고리즘은 이름처럼 시계 방향으로 회전하는 포인터를 사용하여, 메모리에 적재된 페이지들을 원형 구조로 관리한다. 각 페이지는 1비트 크기의 참조 비트를 갖는데, 페이지가 참조될 때 이 비트를 1로 설정한다. 새로운 페이지를 삽입해야 할 때, 현재 포인터가 가리키고 있는 페이지의 참조 비트를 검사한다.

- reference bit == 1 : 이 페이지는 최근에 참조된 것이므로 다시 사용할 가능성이 높다고 판단하고, 참조 비트를 0으로 바꾼 후 포인터를 다음 페이지로 이동시킨다.
- reference bit == 0 : 이 페이지는 한 번 기회를 줬음에도 다시 참조되지 않은 것이므로, 교체 대상으로 선택한다.

이 과정을 통해 Clock 알고리즘은 LRU처럼 자주 사용되는 페이지를 보호하면서도 LRU보다 훨씬 적은 비용으로 구현할 수 있다.

(2) 구현 방식 및 사용 자료구조

Clock 알고리즘은 최근에 참조된 페이지는 유지하고, 그렇지 않은 페이지를 교체하는 전략을 따른다. 이는 LRU의 개념을 근사하지만, 정확한 시간 정보나 리스트 재정렬 없이, 단순히 참조 비트 하나만으로 구현 가능하다는 점에서 훨씬 효율적이다. 실제 하드웨어(MMU)에서도 이와 유사한 방식이 채택될 정도로, 간단하면서도 실용적인 구조를 가진다. 다음은 사용된 자료구조 및 구현 방식이다.

① 프레임 리스트 (frames)
: 각 페이지 프레임은 List<Frame> 형태로 구성되며, 이를 원형 구조처럼 순환시켜 사용한다. 즉, 포인터가 리스트의 끝에 도달하면 다시 처음으로 돌아가게 하여, 시계방향으로 계속 회전하는 구조를 구현한다.

② 포인터 (pointer)
: 교체 대상 프레임을 가리키는 역할을 하며, 교체가 필요한 경우 현재 위치부터 시작해 조건을 만족하는 프레임을 찾을 때까지 시계방향으로 이동한다.

③ Frame 클래스
: 각 프레임은 저장된 페이지 정보(char page)와 함께, 해당 페이지가 최근 참조되었는지를 나타내는 reference flag(boolean)를 포함한다. 이 플래그는 페이지가 다시 참조될 경우 true로 설정된다.

④ Reference Bit 처리 방식
: 페이지가 이미 메모리에 존재하고 참조되면, 해당 프레임의 reference flag를 true로 설정한다. 페이지가 존재하지 않아 교체가 필요한 경우, 포인터가 가리키는 프레임부터 시작해 reference flag가 false인 프레임을 찾는다. reference flag가 true이면 해당 플래그를 false로 변경한 후 포인터를 다음 위치로 이동한다. reference flag가 false인 프레임을 찾으면, 해당 위치의 페이지를 새로운 페이지로 교체하고 reference flag를 false로 초기화한 뒤 포인터를 한 칸 전진시킨다.

(3) Pseudo Code

1. 빈 프레임 리스트(frames)를 만들고 포인터(pointer)를 0으로 초기화한다.
2. 참조 문자열(referenceString)의 각 페이지에 대해 반복한다:
- 현재 프레임 리스트에 해당 페이지가 있는지 확인한다:
- 있다면 (== hit) -> 해당 프레임의 reference flag를 true로 설정
- 없다면 (== fault)
- 프레임이 가득 차지 않았다면:
- 새 프레임을 리스트에 추가하고 포인터를 한 칸 전진
- 프레임이 가득 찼다면:
- 다음 조건을 만족할 때까지 반복:
- 포인터가 가리키는 프레임의 reference flag가 true이면:
- reference flag를 false로 설정
- 포인터를 한 칸 전진
- reference flag가 false이면:
- 해당 위치의 페이지를 교체하고
- reference flag를 false로 초기화
- 포인터를 한 칸 전진하고 루프 종료
3. 반복마다 현재 프레임 상태를 스냅 샷으로 기록한다.

(4) 구현 시 고려사항

① 원형 구조의 구현
: Clock 알고리즘은 프레임을 원형 큐처럼 사용한다. Java에는 원형 구조를 지원하는 자료형이 없어서 일반적인 List를 사용하되 포인터가 범위를 초과하면 % frameSize 연산을 통해 리스트 처음으로 돌아가도록 구현해야 한다.

② Reference Bit 처리 방식
: 페이지가 메모리에 이미 존재하는 경우(히트), 해당 프레임의 reference 값을 true로 설정해준다. 반면, 페이지 폴트가 발생하면 포인터가 가리키는 위치부터 시작하여 reference == false인 프레임을 찾는다. reference == true일 때 그 값을 false로 바꾸고 포인터를 다음으로 넘긴다. false인 프레임을 만나면 교체 대상이 된다.

③ 포인터 이동 처리 주의
: 페이지 교체가 완료된 후에도 포인터는 반드시 다음 프레임으로 한 칸 이동해야 한다. 포인터가 같은 위치에 계속 머물면, 이후 교체 과정에서 같은 프레임만 반복해서 검사하는 오류가 생긴다. 초기 삽입 시에도 포인터를 넘겨줘야 시계 방향 순환이 유지된다.

④ 초기 상태 처리

: 프레임 수가 아직 frameSize에 도달하지 않은 초기 상태에서는, 교체 루프를 돌지 않고 바로 새 페이지를 추가해야 한다. 이를 위해 별도의 조건 분기 처리가 필요하다. 초기 상태를 제대로 다루지 않으면 교체 대상이 없는 상황에서 무한 루프에 빠질 수 있다.

⑤ Frame 클래스의 구성

: Clock 알고리즘은 단순한 페이지 값 외에도 참조 비트가 필요하므로, Frame 클래스에 boolean reference 필드를 반드시 포함해야 한다. 생성자에서도 이 필드를 초기화하도록 구현해야 한다.

⑥ 무한 루프 방지

: 교체 루프는 보통 while (true)로 작성되는데, 이때 반드시 루프 종료 조건을 명확하게 걸어야 한다. 조건을 잘못 작성하면, 교체 대상이 없어 포인터가 무한히 순환하는 상황이 발생할 수 있다. 포인터가 프레임 전체를 한 바퀴 이상 돌지 않도록 제한하는 것도 필요하다.

⑦ 시각화용 Snapshot 처리

: 알고리즘 실행 과정에서 각 참조 시점의 메모리 상태를 recordSnapshot()으로 저장한다. 이 과정을 통해 동작 흐름을 시각화할 수 있다. 다만 참조 문자열이 매우 길 경우, snapshot 저장이 누적되어 메모리 사용량이 증가할 수 있다는 점을 고려해야 한다.

(5) 실제 구현 코드 일부 설명

```
for (char page : referenceString) {
    boolean hit = false;

    for (Frame frame : frames) {
        if (frame.page == page) {
            frame.reference = true;
            hit = true;
            break;
        }
    }
```

현재 참조 중인 페이지(page)가 이미 프레임에 존재하면 page hit. 이때 해당 프레임의 reference bit을 true로 설정한다. hit된 경우 break하여 이후 교체 과정을 생략한다.

```
else {
    faultCount++;
    hitHistory.add(false);

    if (frames.size() < frameSize) {
        frames.add(new Frame(page));
        pointer = (pointer + 1) % frameSize;
    }
```

Page Fault가 났을 때, 프레임에 자리가 남아있는 경우 새로운 페이지를 그대로 추가한다. 포인터도 다음 위치로 이동하여 순환 구조 유지한다.

```
else {
    while (true) {
        Frame current = frames.get(pointer);
        if (current.reference) {
            current.reference = false;
            pointer = (pointer + 1) % frameSize;
        } else {
            frames.set(pointer, new Frame(page));
            pointer = (pointer + 1) % frameSize;
            break;
        }
    }
}
```

프레임이 가득 찼을 경우, 위의 Clock 알고리즘의 핵심 로직이 실행된다.

포인터가 가리키는 프레임의 reference bit이 true이면 false로 바꾸고 다음 위치로 이동한다. reference bit이 false인 프레임이 나오면 교체 대상이며, 해당 자리에 새 페이지를 삽입한 후 break을 통해 무한 루프를 빠져나온다. 삽입이 끝난 후에도 포인터는 한 칸 이동해야 다음 반복에서도 순환이 유지된다.

(6) 복잡도 분석

① 시간 복잡도: O(N × F)

Clock 알고리즘에서 한 번의 페이지 참조마다, 포인터가 최대 프레임 수(F)만큼 회전할 수 있다. 즉, 교체 대상 페이지를 찾기 위해 최악에는 모든 프레임을 한 바퀴 순회해야 할 수 있다는 뜻이다. 따라서 전체 참조 횟수가 N 번이라면, 최악의 시간 복잡도는 O(N × F)가 된다.

물론 일반적인 상황에서는 포인터가 몇 번의 이동만으로 교체 대상을 찾는 경우가 많아 평균 수행 시간은 이보다 낮을 수 있지만, 알고리즘 분석에서는 항상 최악의 경우를 기준으로 평가한다.

② 공간 복잡도: O(F)

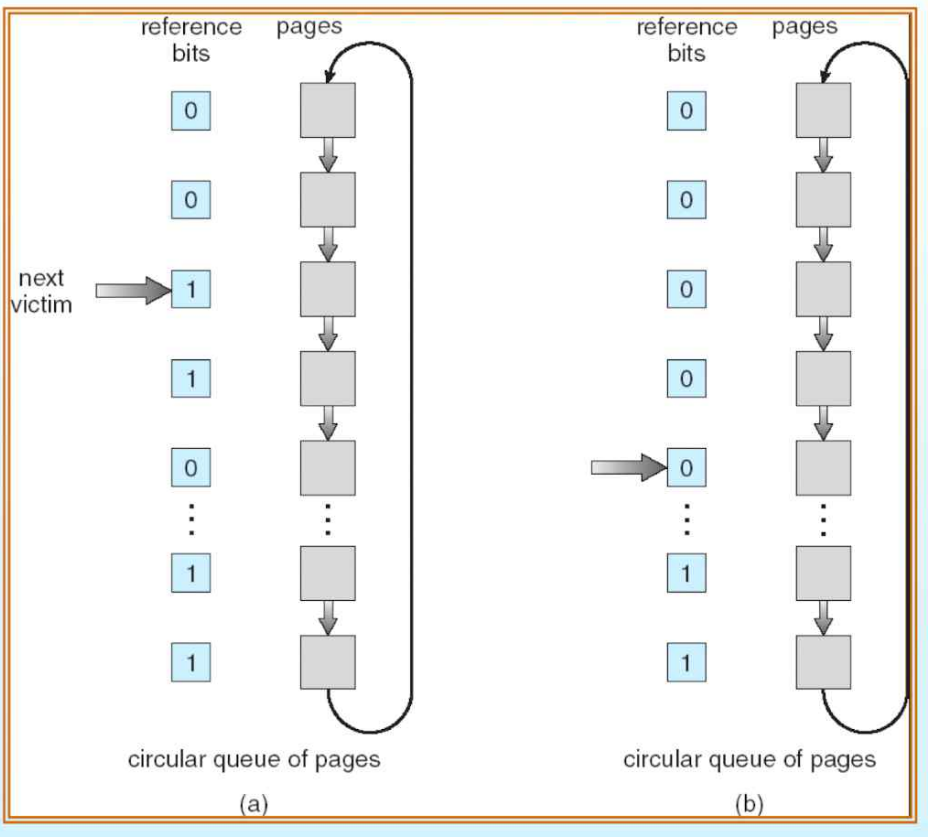
시뮬레이터에서는 각 참조 시점의 메모리 상태를 저장하기 위해 recordSnapshot()을 호출한다. 참조 문자열의 길이가 N이고, 프레임 수가 F라면 N개의 스냅 샷이 저장되고, 각 스냅 샷은 F개의 페이지 상태를 담고 있으므로 총 O(N × F)의 공간이 필요하다.

단, 이러한 스냅 샷 저장은 시각화나 분석을 위한 시뮬레이션 용도로만 사용되며, 실제 운영체제에서는 저장되지 않는다. 따라서 현실적인 환경에서의 공간 복잡도는 O(F)로 간주할 수 있다.

(7) 특징

Clock 알고리즘은 LRU의 핵심 개념을 간단한 방식으로 근사한 알고리즘이다. 최근에 참조된 페이지는 가능한 한 오래 유지하고, 오래 사용되지 않은 페이지부터 교체한다는 LRU의 아이디어를 유지하되, 참조 비트 하나로 이를 간접적으로 판단한다.

이때 구조는 메모리 프레임을 원형 큐처럼 구성하고, 하나의 포인터가 시계 방향으로 순회하면서 교체 대상을 탐색한다. 페이지가 참조된다면 해당 프레임의 reference bit은 1로 설정되고, 페이지 교체 시에는 이 reference bit 값을 바탕으로 교체 여부를 결정한다. 이때 만약 최근에 참조된 흔적이 있다면 한 번 더 기회를 주고(reference bit을 0으로 초기화하고 다음으로 넘어감), 그렇지 않은 페이지는 곧바로 교체 대상이 된다.



[그림 4] Clock 알고리즘의 교체 대상 선정 방식

이러한 동작 방식은 위의 [그림 4]에서 시각적으로 확인할 수 있다. 왼쪽 (a)는 포인터가 한 바퀴를 돌며 reference bit이 1인 페이지들을 건너뛰고 초기화하는 과정을 나타낸 것이다. 또한, 오른쪽 (b)에서는 reference bit이 0인 페이지를 찾아 교체하는 장면을 보여준다.

이러한 구조 덕분에 복잡한 시간 정보나 리스트 재정렬 없이도 비교적 효율적인 페이지 교체가 가능하다. 실제 운영체제에서도 Clock 알고리즘이 널리 사용되며, 특히 하드웨어(MMU)에서 참조 비트를 지원하는 경우 효율성이 더욱 높아진다.

다만, 참조 비트만으로는 페이지가 '정확히 언제' 참조되었는지는 판단할 수 없으므로 완전한 LRU 구현에는 미치지 못한다. 특히 모든 페이지가 한 번씩만 차례대로 참조되는 경우에는 Clock이 FIFO와 같게 동작하며, 최근 참조 여부를 반영하지 못하는 한계가 드러난다.

(8) 정리

Clock 알고리즘은 성능, 구현 난이도, 실제 적용 가능성 측면에서 균형이 잘 잡힌 페이지 교체 방식이다. 최근 참조 여부를 간단한 참조 비트로 추적하고, 포인터를 회전시켜 교체 대상을 찾는 방식은 단순하면서도 실용적이다.

이 알고리즘은 Belady의 역설이 거의 발생하지 않으며, LRU에 비해 구현 부담이 낮고, 실제 하드웨어와의 연계도 수월해 실제 운영체제 환경에서도 높은 채택률을 보인다.

2.5. LPR(Lowest Probability Replacement)

(1) 소개 및 작동 원리

LPR은 페이지 교체 시, 단순히 *"언제 사용됐는가"*나 *"앞으로 언제 등장할 것인가"* 같은 **시점을 기준으로 판단하지 않는다.** 대신, 지금 이 순간 기준으로, 앞으로 다시 등장할 가능성이 가장 낮은 페이지를 제거하자는 전략이다.

기본적인 전제는 이렇다.

> *이미 페이지 폴트가 났다면, 예측에는 한 번 실패한 것이다.*
> *그렇다면 지금이라도 최선의 선택을 해서, 앞으로의 페이지 폴트를 줄이자.*

이를 위해 LPR은 참조 문자열에서 페이지 등장 순서를 통계적으로 분석한다. 즉, 과거에 어떤 페이지 뒤에 어떤 페이지가 자주 나왔는지를 확률적으로 계산하고, 이를 바탕으로 현재 페이지 다음에 등장할 가능성이 낮은 페이지를 교체 대상으로 선택한다.

다시 말해, 메모리에 올라온 페이지 중에서 '이번 참조 이후 다시 등장할 확률이 가장 낮은 페이지'를 찾아 제거하는 방식이다. 이런 아이디어는 단순한 규칙 기반(FIFO, LRU)과는 달리, 참조 이력에 기반을 둔 데이터 중심 접근이라고 볼 수 있다.

(2) 구현 방식 및 사용 자료구조

LPR 알고리즘은 과거 참조 이력을 바탕으로 현재 들어온 페이지를 기준으로 다음에 어떤 페이지가 등장할 가능성이 높은지를 확률적으로 계산하는 방식이다. 이를 위해 다음과 같은 구조로 구현된다

① nextPageCount (Map<Character, Map<Character, Integer>>)
: 두 단계의 Map을 통해, 어떤 페이지 다음에 어떤 페이지가 얼마나 자주 등장했는지를 저장한다. 이 Map은 페이지가 참조될 때마다 갱신되며, 페이지 폴트 발생 시 다음 등장 확률 계산에 사용된다.

(ex) nextPageCount.get('A').get('B') == 3
→ 'A' 다음에 'B'가 3번 등장했다는 의미.

② frames (List)

: 현재 메모리에 적재된 프레임을 리스트 형태로 관리한다. 각 Frame 객체는 char page를 포함하며, 페이지 교체가 일어나면 교체 대상 선택에 활용된다.

③ findVictim(curr: char)

: 현재 들어온 페이지 curr를 기준으로, nextPageCount.get(curr)에 저장된 통계를 기반으로 현재 프레임 내 페이지들의 등장 확률을 계산한다. 각 페이지의 등장 확률은 count / total 로 계산되며, 이 확률이 가장 낮은 페이지가 교체 대상(victim)이 된다.

(3) Pseudo Code

1. 빈 프레임 리스트(frames)를 만든다.
2. nextPageCount 맵을 초기화한다.
→ 각 페이지 A에 대해, A 다음에 등장한 페이지들을 빈도수로 저장할 수 있도록 구성.
3. 참조 문자열을 앞에서부터 순회하면서 다음을 수행한다:
- 현재 참조된 페이지를 curr, 이전 페이지를 prev라고 하자.
- prev → curr 형태의 쌍을 nextPageCount에 기록한다.
→ nextPageCount[prev][curr] += 1

- 만약 curr가 이미 프레임에 있다면:
- hit 처리 (아무것도 하지 않음)

- 만약 curr가 프레임에 없다면 (페이지 폴트 발생):
- fault 처리
- 프레임이 가득 찼다면:
- curr를 기준으로 nextPageCount[curr]에서 통계를 가져온다.
- 현재 프레임에 있는 각 페이지의 다음 등장 확률을 계산한다.
- 가장 낮은 확률을 가진 페이지를 victim으로 선택하여 교체한다.
- 프레임에 여유가 있다면:
- curr를 프레임에 추가한다.

- 현재 프레임 상태를 기록한다.

페이지 폴트가 날 때만 예측 모델을 사용하여 가장 덜 등장할 페이지를 제거한다.
페이지 참조 패턴은 prev → curr 형태로 차례대로 학습된다.

(4) 구현 시 고려사항

① 참조 이력의 학습 구조 (nextPageCount) 구성

: Map<Character, Map<Character, Integer>> nextPageCount 구조를 통해 "이전 페이지 → 현재 페이지"의 등장 빈도를 기록한다. 이 구조는 한 페이지 다음에 어떤 페이지가 얼마나 자주 등장했는지를 통계적으로 파악할 수 있다. 구현 시 반드시 이전 참조가 존재하는 경우(i > 0)에만 prev → curr 쌍을 기록해야 하며, putIfAbsent()와 getOrDefault()를 통해 맵의 안전한 접근이 필요하다.

② 예측 확률 계산 시 주의점

: 현재 들어온 페이지(curr)를 기준으로, nextPageCount.get(curr)를 가져온다. 이 통계에 기반을 둬 현재 프레임 내 페이지 각각의 예측 확률을 계산한다.

확률 = (curr 다음에 그 페이지가 등장한 횟수) / (curr 다음 등장 전체 합계)

통계가 아예 없거나 합이 0인 경우(=예측 불가)는 기본적으로 0번째 인덱스를 victim으로 처리한다.

③ 페이지 폴트 시에만 작동

: 기존 알고리즘과 달리 hit일 때는 아무 작업도 하지 않으며, fault 시에만 예측을 통해 victim을 선택하고 교체를 수행한다. 이 때문에 "예측에 실패했지만, 지금이라도 잘하자"는 구조를 따르게 된다.

④ 자료구조 선택

- ArrayList<Frame> : 프레임을 순서대로 저장하고 교체할 수 있는 구조
- HashMap : 등장 패턴을 빠르게 조회하고 누적할 수 있는 통계 자료구조
- 효율적인 탐색과 예외 처리를 위해 Map 접근 시 null 체크가 반드시 필요하다.

⑤ 성능 오버헤드에 대한 고려

: 통계 기반으로 매 페이지 폴트 시마다 다중 맵 탐색과 확률 계산이 이루어지므로 순수한 FIFO, LRU보다 처리 비용이 더 많이 든다. 실제 운영체제에서 사용하려면 통계 계산과 업데이트가 O(1)에 가깝게 최적화되어야 한다.

⑥ 스냅샷 저장의 메모리 사용

: recordSnapshot()으로 프레임 상태를 기록하므로, 참조 문자열이 길 때 많은 메모리를 사용할 수 있다. 시각화나 학습 목적이 아니라면 이 부분은 생략할 수 있다.

(5) 실제 구현 코드 일부 설명

① 다음 페이지 등장 패턴 학습

```
if (i > 0) {
    nextPageCount.putIfAbsent(prev, new HashMap<>());
    Map<Character, Integer> pageCount = nextPageCount.get(prev);
    pageCount.put(curr, pageCount.getOrDefault(curr, 0) + 1);
}
```

참조 문자열의 이전 페이지(prev)와 현재 페이지(curr)의 쌍을 통계적으로 기록한다. nextPageCount는 이전 페이지를 키로, 그다음 등장한 페이지들의 빈도수를 저장하는 중첩 해시맵이다.

(ex) A → B, A → C, B → A 등이 얼마나 등장했는지를 누적한다.

② 페이지 폴트 처리 및 victim 선정

```
if (!hit) {
    faultCount++;
    hitHistory.add(false);

    if (frames.size() == frameSize) {
        int victimIndex = findVictim(curr);
        frames.set(victimIndex, new Frame(curr));
    } else {
        frames.add(new Frame(curr));
    }
}
```

페이지 폴트가 발생하면, 프레임이 가득 찼으면 findVictim(curr) 함수를 호출하여 교체 대상(victim)을 고른다. 아직 프레임이 가득 차지 않았다면 그대로 새 페이지를 추가한다.

③ victim 선정 함수 (findVictim)

```
Map<Character, Integer> pageCount = nextPageCount.get(curr);
// 해당 페이지 이후 등장한 통계가 없으면 기본 인덱스 0

for (int i = 0; i < frames.size(); ++i) {
    char victimPage = frames.get(i).page;
    int count = pageCount.getOrDefault(victimPage, 0);
    double prob = (double)count / total;

    if (prob < minProb) {
        minProb = prob;
        victimIndex = i;
    }
}
```

현재 페이지 curr 이후 어떤 페이지들이 얼마나 자주 등장했는지 통계를 조회하고, 그 중 등장 확률이 가장 낮은 페이지를 victim으로 결정한다.

확률 = curr 이후 victimPage가 등장한 횟수 / curr 이후 등장한 전체 횟수

확률이 가장 낮은 페이지는 앞으로 등장할 가능성이 가장 적으므로 교체 대상이 된다.

(6) 복잡도 분석

참조 문자열의 길이를 N, 프레임 수를 F, 참조 문자열에 등장하는 페이지 종류 수를 M이라고 할 때 시간 및 공간 복잡도는 다음과 같다.

① 시간 복잡도: O(N × F)

각 시점마다 최대 F개의 프레임을 순회하며 hit 여부를 확인한다. page fault 발생 시 findVictim() 메서드를 호출하는데, 이 메서드는 현재 들어오려는 페이지를 기준으로 학습된 확률 정보를 기반으로 각 프레임 페이지의 등장 확률을 계산하여 victim을 선정한다. 각 프레임에 대해 확률 계산이 필요하므로, 해당 과정 또한 O(F)에 해당한다. 전체적으로 참조마다 O(F) 작업이 2회 수행되므로, 시간 복잡도는 O(N × F) 수준이다.

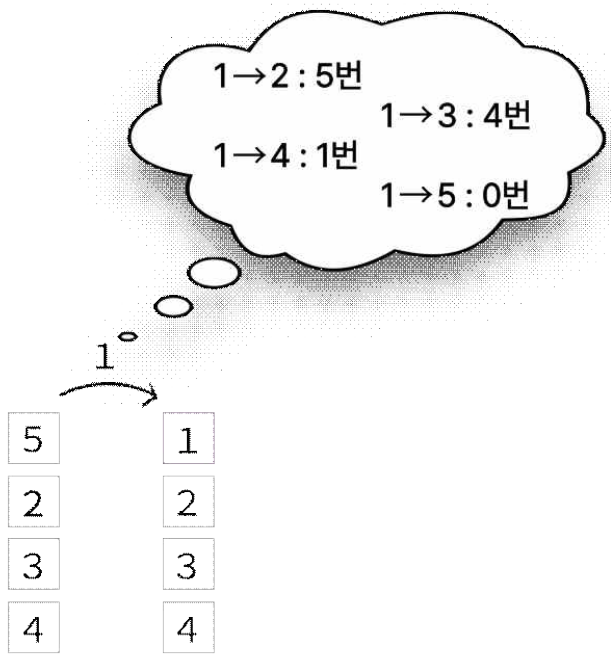
② 공간 복잡도: O(F + M²)

시간에 따라 각 시점의 프레임 상태를 기록하는 frameSnapshots 리스트는 최대 N개의 F 길이 리스트를 저장하므로, 공간 복잡도는 O(N × F)이다. 학습된 확률 정보를 저장하는 nextPageCount는 각 문자(페이지)의 다음 등장 문자 통계를 Map<Character, Map<Character, Integer>> 형태로 저장한다.

이 구조는 최악에는 모든 페이지 쌍에 대해 등장 정보를 저장하므로, O(M²)의 공간이 소유될 수 있다. 단, 이러한 스냅샷 저장은 시각화나 분석을 위한 시뮬레이션 용도로만 사용되며, 실제 운영체제에서는 저장되지 않는다. 따라서 전체 공간 복잡도는 O(F + M²)로 표현할 수 있다.

(7) 특징

본 알고리즘은 기존의 시간 기반, 순서 기반 페이지 교체 방식과는 다른 데이터 기반의 예측 전략을 도입한다는 점에서 뚜렷한 특징을 가진다. 참조 이력을 활용하여 다음에 등장할 확률이 낮은 페이지를 제거함으로써, 예측을 통해 페이지 폴트를 줄이는 새로운 방향을 제시한다. 주요 특징은 다음과 같다.



[그림 5] LPR 알고리즘의 동작 방식

① 예측 기반 페이지 교체

: 과거 참조 데이터를 학습하여, 다음에 등장할 확률이 낮은 페이지를 victim으로 선택한다. 단순한 순서 기반이 아닌 참조 관계를 분석해 교체 대상을 결정한다는 점에서 차별적이다.

② 적응형 알고리즘
: 시간이 지남에 따라 학습 데이터가 누적되며, 판단 정확도도 점차 향상된다. 반복 패턴이나 지역성이 존재할 경우 빠르게 최적화된 판단을 수행할 수 있다.

③ 확률 기반 victim 선택
: 페이지 폴트 발생 시, 지금 들어오려는 페이지를 기준으로 다음 등장 확률이 낮은 페이지를 교체 대상으로 선택한다. 이는 "예측에는 실패했지만, 지금이라도 현명한 선택을 하자"는 전략이다.

④ 학습 가능한 구조
: 반복되는 참조 패턴이나 프로그램의 루프 구조 등에 강하며, 통계적 학습을 통해 hit율 향상을 기대할 수 있다.

⑤ Belady의 Anomaly 방지 가능성
: 등장 확률을 기준으로 판단하므로, FIFO처럼 단순한 순서 기반 알고리즘보다 Belady의 Anomaly 발생 가능성이 낮다.

⑥ 기존 알고리즘과의 차별성
: OPT는 미래, LRU는 시간, Clock은 비트를 기준으로 동작하지만, LPR은 과거의 연결성을 기준으로 판단한다는 점에서 구조적으로 새로운 관점을 제시한다.

⑦ 확장 가능성
: 1-gram 기반이지만, 2-gram, 3-gram 혹은 마르코프 체인 방식으로의 확장이 가능하며, 예측 정확도를 높이기 위한 다양한 변형이 가능하다.

하지만 이러한 장점들에도 LPR은 몇 가지 구조적인 한계와 제약을 가진다. 다음과 같은 단점은 실제 구현 시 주의 깊게 고려되어야 한다.

⑧ 초기 학습 부족 문제
: 학습 기반 알고리즘 특성상, 초반에는 참조 이력이 부족하여 정확한 판단이 어렵고, 사실상 무작위에 가까운 선택이 이루어질 수 있다.

⑨ 데이터 구조의 비용
: Map<Character, Map<Character, Integer>>형태의 구조는 페이지 종류가 많아질수록 메모리 사용량과 관리 비용이 급격히 증가할 수 있다.

⑩ Optimal 대비 예측 정확도 부족
: Optimal 알고리즘은 미래 정보를 직접 활용하는 반면, LPR은 과거 데이터를 바탕으로 추정만 할 수 있기 때문에 항상 최적의 결과를 보장하지 않는다.

⑪ 최악의 경우 성능 저하
: 참조 패턴이 매우 불규칙하거나 예측이 계속 빗나가는 경우, 오히려 LRU나 FIFO보다 낮은 hit율을 기록할 수 있다. 이 경우 LPR의 전략적 선택이 오히려 불필요한 교체를 유발할 수 있다.

(8) 정리

LPR은 기존 페이지 교체 알고리즘과는 다른 관점을 제시하는 확률 기반 예측 정책이다. 과거 참조 이력을 바탕으로 다음에 등장할 가능성이 낮은 페이지를 제거함으로써, 향후의 페이지 폴트를 줄이려는 전략을 취한다. 단순히 시간 순서에 따라 교체 대상을 선정하는 LRU나 Clock과 달리, 참조 패턴을 학습하고 이를 근거로 판단을 내리는 구조는 더욱 지능적인 메모리 관리 방식이라 할 수 있다.

특히 이 알고리즘은 "이미 예측에 실패했으니, 지금이라도 최선을 다하자"는 개념에서 출발한다. 참조 문자열 속에서 "A 다음에는 보통 B가 나오더라"같은 통계적 연관성을 분석하여, 현재 들어온 페이지 기준으로 다른 페이지들의 향후 등장 확률을 추정하고, 그중 가장 가능성이 낮은 페이지를 교체 대상으로 삼는 것이 핵심 원리다.

물론, 이 알고리즘은 실제 운영체제에 도입되기엔 아직 계산 비용과 데이터 구조의 복잡성 측면에서 제약이 있다. 또한, 참조 패턴이 일정하지 않거나 예측이 어려운 환경에서는 기대한 만큼의 성능을 보장하기 어렵다. 하지만 시뮬레이터처럼 참조 열이 고정되어 있고 반복성이 존재하는 실험 환경에서는, 그 효율성을 분명히 확인할 수 있다.

나아가 이 알고리즘은 n-gram 구조 확장, 마르코프 체인 적용 등을 통해 더욱 정교한 예측 모델로 진화할 가능성이 크며, 궁극적으로는 AI 기반 메모리 관리 전략으로 발전할 수 있는 잠재력도 지닌다.

결과적으로 LPR은 단순히 새로운 교체 규칙을 제안한 데 그치지 않고, 기존 알고리즘의 맹점을 보완하고, 예측 기반 의사결정이라는 새로운 방향성을 실험적으로 구현한 사례로서, 의의가 충분한 창의적인 시도라 평가할 수 있다.

Ⅳ. 성능 평가

1. 실험 목적 및 환경

본 실험의 목적은 앞서 구현한 다양한 페이지 교체 알고리즘들이 실제로는 어떻게 성능 차이를 보이는지를 비교하고, 각 알고리즘의 특징을 더욱 명확히 분석하는 데 있다. 이론적으로는 알고리즘마다 고유한 장단점이 존재하지만, 실제 성능은 참조 문자열의 패턴이나 프레임 수 등 구체적인 입력 조건에 따라 달라질 수 있다. 따라서 본 실험을 통해 이론과 실제 간의 차이를 확인하고, 알고리즘별 특성을 구체적으로 파악할 수 있을 것이다.

앞에서 다루었던 FIFO, Optimal, LRU, Clock, LPR 알고리즘이 같은 조건에서 각각 어떤 효율을 보이는지를 확인하는 것이 핵심이다. 이를 통해 알고리즘 간 성능 차이를 시뮬레이터를 통해 수치화 및 시각화하여 특정 조건에서 어떤 방식이 더 적합한지도 파악해보고자 한다.

1.1. 실험 환경

본 실험의 시뮬레이터는 Java 언어를 기반으로 개발되었으며, 사용자 인터페이스(GUI)는 JavaFX를 이용하였다. 또한, 페이지 교체 알고리즘은 정책별로 클래스를 생성하여 구현하였다.

- 개발 언어 : Java 17
- GUI : JavaFX 17
- IDE : Eclipse IDE for Java Developers - 2024 - 12
- 실행환경
    - CPU : Intel(R) Core(TM) Ultra 7 258V
    - Memory : 32GB

각각의 알고리즘은 다음과 같은 공통된 방식으로 실험을 수행하였다.

① 먼저 같은 참조 문자열과 프레임 수를 입력값으로 사용하였다.
② 이후 각 알고리즘이 이를 처리하면서 발생하는 페이지 히트 수, 폴트 수, 교체 횟수 등의 결과를 출력하였다.
③ 그 과정을 JavaFX 기반 GUI를 통해 시각적으로 확인할 수 있도록 구성하였다.

※ 각 시점의 메모리 상태는 List<List<Character>>라는 자료구조를 통해 저장하였으며, 이를 바탕으로 페이지 히트(초록), 폴트(빨강), 교체(보라) 상황을 시각적으로 구분하여 직관적인 결과 비교가 가능하도록 구성하였다.

1.2. 실험 입력 요소 및 설계의 정당성

본 실험에서 사용된 핵심 입력 요소는 참조 문자열과 프레임 수이다. 이 두 항목은 알고리즘의 성능에 직접적인 영향을 주는 핵심 변수로, 다양한 조건에서 알고리즘의 특성과 한계를 확인할 수 있도록 설계하였다.

1. 참조 문자열(Reference String)
   : 총 5종의 문자열을 사용하여 다양한 접근 패턴을 시뮬레이션하였다. 이는 실제 시스템에서 발생할 수 있는 대표적인 메모리 접근 유형을 모사하려는 목적이 있다.

분류	예시	특징 및 목적
Belady's anomoly	1 2 3 4 1 2 5 1 2 3 4 5	FIFO의 취약점을 보여주는 대표적 예시
고정 루프	1 2 3 4 5 1 2 3 4 5 1 2 3 4 5	매우 강한 지역성을 가지며, LRU와 LPR 알고리즘에 유리
반복 + 교란 요소	1 2 3 4 1 2 3 5 1 2 3 6	반복되는 지역성 내 교란 요소가 삽입되어 핫 페이지 유지력을 평가 가능
무작위 패턴	2 5 1 4 3 2 7 2 6 1 3 6	지역성이 거의 없고 예측 불가능한 순서로 구성되어 있어 LPR 약점 노출
LRU, Clock 차이	1 2 3 4 4 3 2 1 5 1 2 3	Clock이 reference bit만으로는 LRU를 완벽히 근사하지 못함을 보여줌

2. 프레임 수(Frame Size):

- 3, 4, 5, 6 개의 프레임으로 실험을 진행하였다.
- 알고리즘이 프레임 수 증가에 어떻게 반응하는지를 비교하기 위한 조건이며,
- 특히 FIFO의 Belady's Anomaly 발생 가능성과 LPR의 Working Set 기반 교체 전략의 장점을 확인하기 위한 핵심 변수로 활용하였다.

이러한 입력 요소들은 알고리즘의 구조적 특성과 성능 차이를 일관된 조건에서 정량적으로 비교하기 위해 설계된 것으로, 단순한 코드 테스트를 넘어 알고리즘의 전략적 특성과 실제 환경에 대한 반응을 분석하는 데 중점을 두었다.

1.3. 실험 제약 사항

이번 실험은 실제 운영체제 환경이 아닌, 시뮬레이터를 기반으로 한 실험 환경이기에 몇 가지 제한사항이 있다.

① 참조 문자열 구성

실험에 사용된 참조 문자열은 알고리즘의 기본적인 특성과 성능 경향을 파악하고, 서로 다른 알고리즘 간의 성능을 비교하는 데에 목적이 있다. 이 때문에 난수로 생성하거나, 사전에 정의된 방식으로 구성되어 있기에, 실제 운영체제 시스템에서의 실제 메모리 접근 패턴을 완벽하게 반영하지는 못한다.

② Optimal 알고리즘

Optimal 알고리즘은 미래의 접근 순서를 안다는 비현실적인 가정을 전제로 한다. 따라서 본 실험에서는 이론적 기준점으로서의 비교 대상으로만 사용되었으며, 실제 적용 가능성을 고려한 것은 아니다.

③ 단일 프로세스 기반 시뮬레이션

시뮬레이터는 단일 프로세스를 대상으로 하고 있으며, 다중 프로세스 환경은 고려하지 않는다.

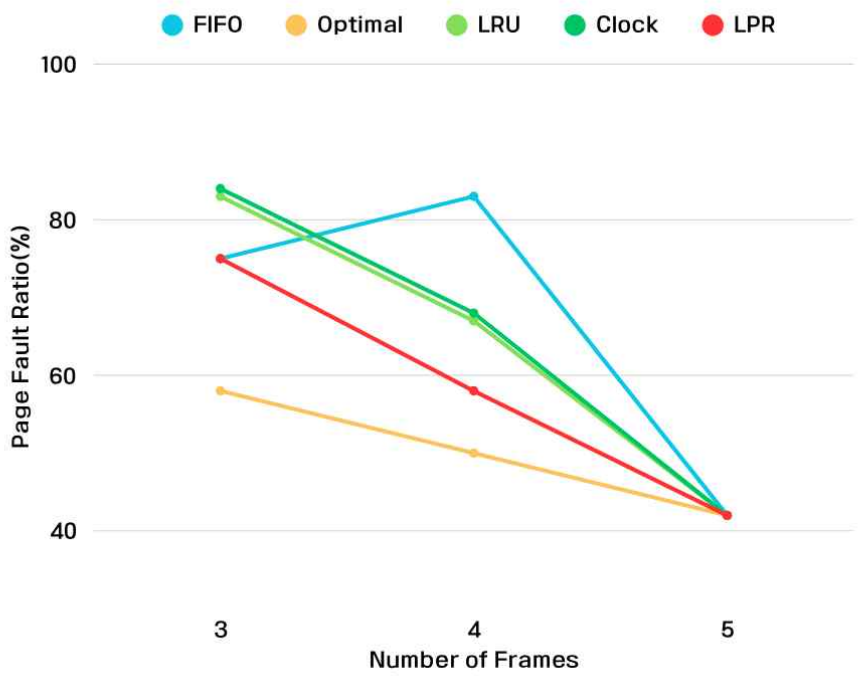
2. 실험 결과 및 분석

앞서 설정한 실험 환경과 입력 조건을 바탕으로, FIFO, LRU, Clock, Optimal, LPR 총 다섯 가지 페이지 교체 알고리즘에 대해 성능을 비교하였다. 각 알고리즘은 동일한 참조 문자열과 프레임 수 조건에서 시뮬레이션 되었으며, 실험 결과는 페이지 히트 수, 폴트 수, 교체 횟수 등을 기준으로 수집되었다.

또한, 알고리즘의 특성을 시각적으로 비교하기 위해 JavaFX GUI를 활용한 시뮬레이터 화면 캡처와 함께, 실험 결과를 바탕으로 한 꺾은선 그래프 등의 자료를 제시하였다. 이를 통해 알고리즘 간의 상대적 성능 차이와 특정 조건에서의 반응 특성을 보다 명확하게 파악할 수 있었다.

- **2-1. Reference String: 1 2 3 4 1 2 5 1 2 3 4 5**

(1) 프레임 개수 별 Page Fault 비율

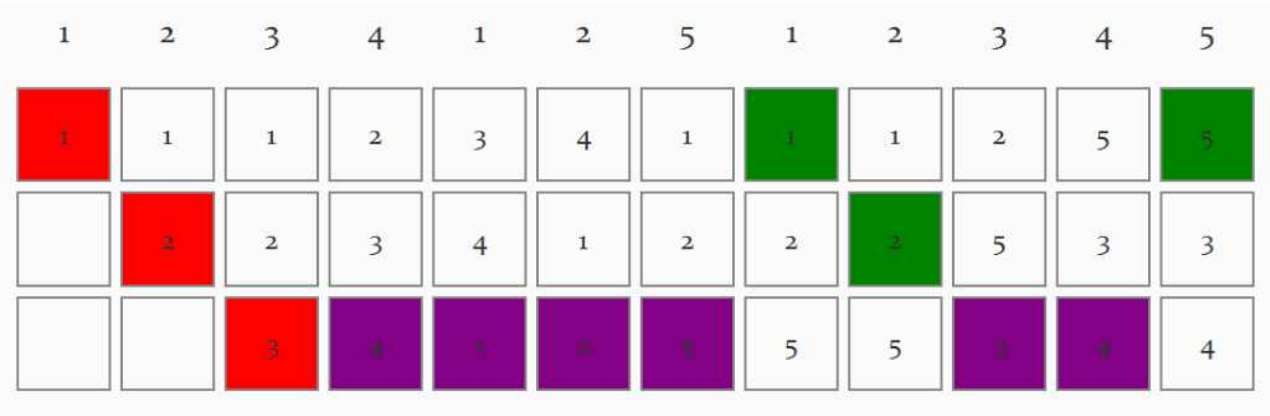


[그림 6] 프레임 수에 따른 페이지 폴트 비율

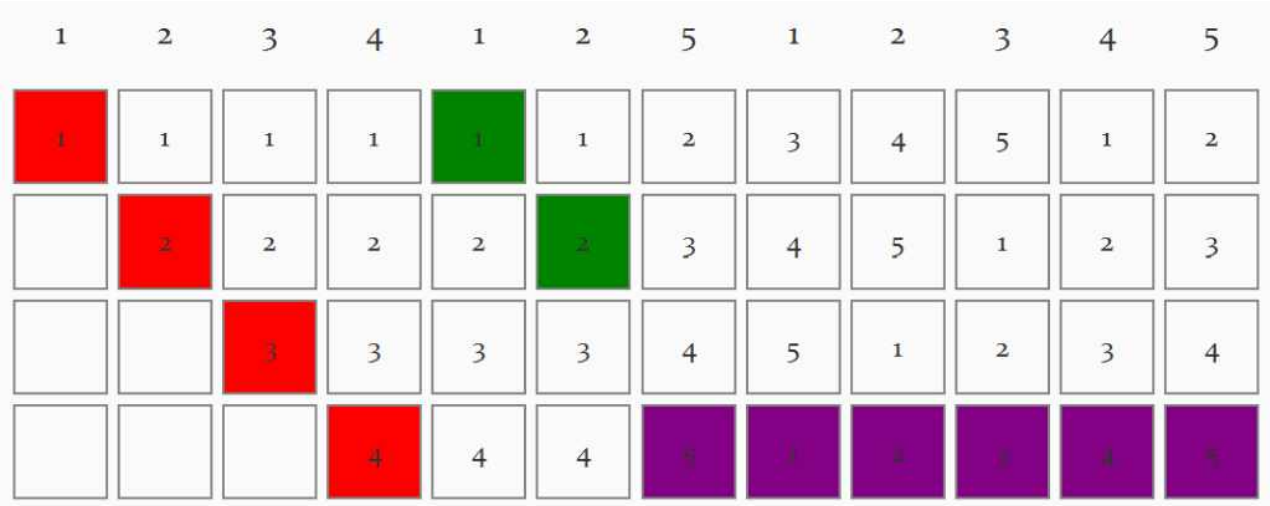
일반적으로 메모리의 프레임 수가 증가하면 Page Fault 비율은 감소하는 것이 정상적인 동작으로 간주한다. 그러나 이 참조 문자열에 대해서 FIFO 페이지 교체 알고리즘을 적용한 결과, 프레임 수가 3개일 때보다 4개일 때 오히려 Page Fault가 더 많이 발생하는 모습이 관찰되었다.

이러한 현상은 Belady's Anomaly로 알려졌으며, FIFO 알고리즘처럼 단순한 순서 기반 교체 정책에서 간헐적으로 나타날 수 있다.

(2) 시뮬레이터 결과 (FIFO)



[그림 7] FIFO, 프레임 수 = 3



[그림 8] FIFO, 프레임 수 = 4

FIFO 알고리즘은 가장 먼저 메모리에 올라온 페이지를 교체 대상으로 삼는다. 즉, 페이지들은 들어온 순서대로 큐에 저장되며, 가장 앞에 있는 페이지가 먼저 제거되고 새로운 페이지는 항상 큐의 맨 뒤에 추가된다. 이러한 동작 방식은 [그림 7]과 [그림 8]의 보라색 표시를 통해 확인할 수 있다.

[그림 7]에서는 7번째 참조 시 새로운 페이지 5가 들어오면서, 가장 오래전에 들어왔던 페이지 4가 교체된다. 이 선택은 이후 8번째와 9번째에 다시 등장하는 페이지 1과 2를 그대로 유지할 수 있어 불필요한 교체를 줄이고 Page Fault를 최소화하는 결과로 이어진다.

반면, [그림 8]에서는 프레임 수가 늘어남에 따라 4번째에 들어온 페이지 4가 이후 오랫동안 참조되지 않음에도 메모리에 계속 유지되어 있어 프레임을 비효율적으로 점유하는 상황이 발

생한다. 그 결과, 7번째 참조 시 가장 앞에 있는 1이 교체되고, 바로 다음 참조에서 1이 다시 요청되면서 추가적인 Page Fault가 발생하게 된다.

이와 같은 상황은 FIFO 알고리즘이 참조의 필요성이나 미래 사용 여부를 고려하지 않고 단순히 입력 순서만을 기준으로 교체하기 때문에 발생한다. 특히 프레임 수가 늘어난 상황에서 오히려 불필요한 페이지가 오래 유지되고 중요한 페이지가 조기에 제거되는 바람에 Page Fault가 증가하는 Belady's Anomaly의 전형적인 예라 할 수 있다.

- **2-2. Reference String: 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5**

(1) 프레임 개수 별 Page Fault 비율

[그림 9] 프레임 수에 따른 페이지 폴트 비율

[그림 9]는 참조 문자열에 대해 각 페이지 교체 알고리즘의 페이지 폴트 비율을 비교한 결과를 나타낸다. 이 참조 문자열은 반복되는 일정한 패턴을 가지고 있어, 일반적인 고정 루프 구조를 모델링한 것으로 해석할 수 있다.

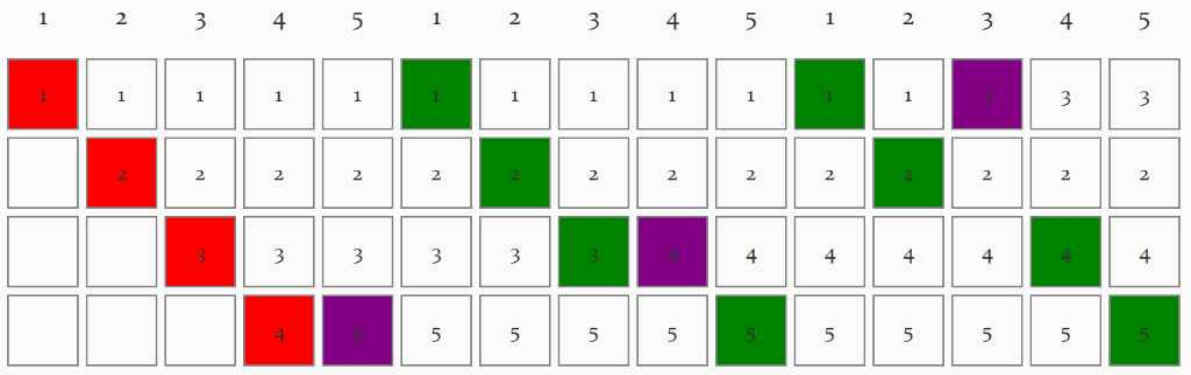
이 문자열에서 등장하는 페이지 집합은 {1, 2, 3, 4, 5}이며, 이는 해당 작업의 Working Set을 의미한다. 따라서 프레임 수가 5개 이상이면 메모리에 전체 Working Set을 모두 유지할 수 있어, 어떤 알고리즘을 적용하더라도 첫 회차를 제외하면 추가적인 Page Fault가 발생하지 않는다. 결과적으로 모든 알고리즘의 Page Fault 비율이 같게 나타난다.

반면, 프레임 수가 Working Set보다 작은 경우(예: 3개, 4개)에는 상황이 달라진다. FIFO, LRU, Clock 등의 알고리즘은 재참조 되는 페이지를 유지하지 못하고 계속해서 교체가 일어나기 때문에 100%의 Page Fault 비율을 보인다.

하지만 Optimal과 LPR 알고리즘은 상황을 다르게 처리한다. Optimal은 미래의 참조를 알고 있기 때문에 가장 나중에 다시 참조될 페이지를 교체할 수 있으며, LPR 역시 과거 참조 이력을 기반으로 일정한 순환 패턴을 학습하여 미래 등장 확률이 낮은 페이지를 정확히 예측한다.

따라서 [그림 9]를 통해, LPR이 반복성과 지역성이 강한 참조 패턴에 대해 Optimal과 유사한 수준의 Page Fault 성능을 달성할 수 있음을 확인할 수 있다.

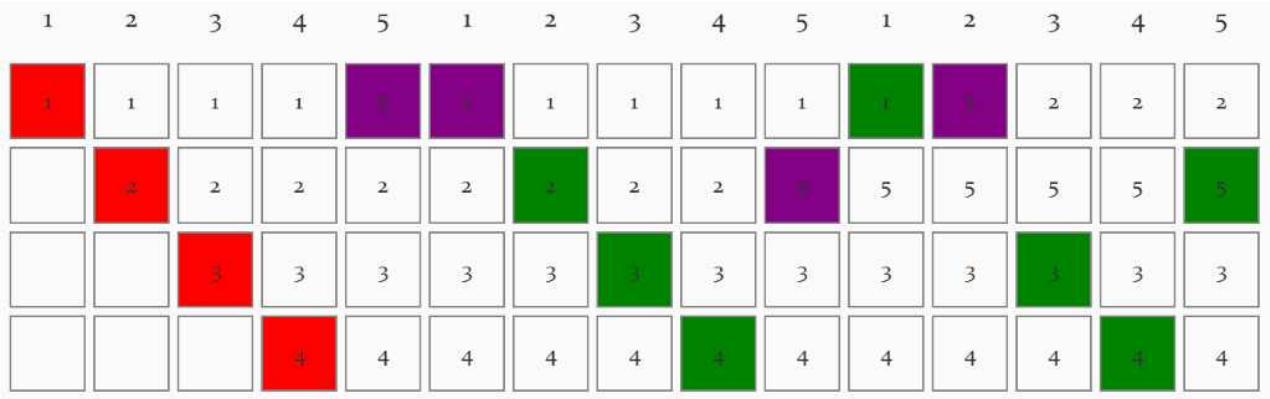
(2) 시뮬레이터 결과 (Optimal, LPR)



[그림 10] Optimal, 프레임 수 = 4개

초기 4개의 페이지는 순서대로 메모리에 적재된다. 이후 5가 등장하면 현재 프레임에 있는 1 ~ 4 중 가장 늦게 재참조 되는 페이지 4가 교체 대상이 되어 5로 대체된다. 다시 루프가 시작되며 다음 4가 참조될 때는 프레임에 존재하지 않으므로 Page Fault가 발생한다. 이때 현재 프레임은 1, 2, 3, 5이고 이후 가장 나중에 참조되는 페이지는 3이므로, 3이 교체되고 4가 적재된다.

다시 루프가 시작되고 13번째 참조인 페이지 3이 들어왔을 때 Page Fault가 발생한다. 이후 참조 순서를 보면 남은 참조가 4, 5뿐이므로 앞으로 등장하지 않을 1 또는 2중 하나를 선택하여 제거하여야 한다. 구현상 가장 먼저 등장한 페이지를 우선 교체하도록 하므로, 1이 교체 대상이 된다.



[그림 11] LPR, 프레임 수 = 4개

LPR 알고리즘은 과거의 참조 이력을 기반으로 다음에 등장할 페이지를 예측하며, 가장 등장 확률이 낮은 페이지를 교체 대상으로 선택한다. 이 방식은 반복 구조와 같이 지역성이 강한 참조 패턴에 대해 높은 예측 정확도를 보이는 특징이 있다.

LPR은 참조 연관성 예측을 위해 일정 수준의 과거 데이터가 필요하다. 하지만 페이지 5는 참조 문자열의 초반에 처음 등장하며, 그 시점까지는 충분한 통계적 학습이 이루어지지 않은 상태이다.

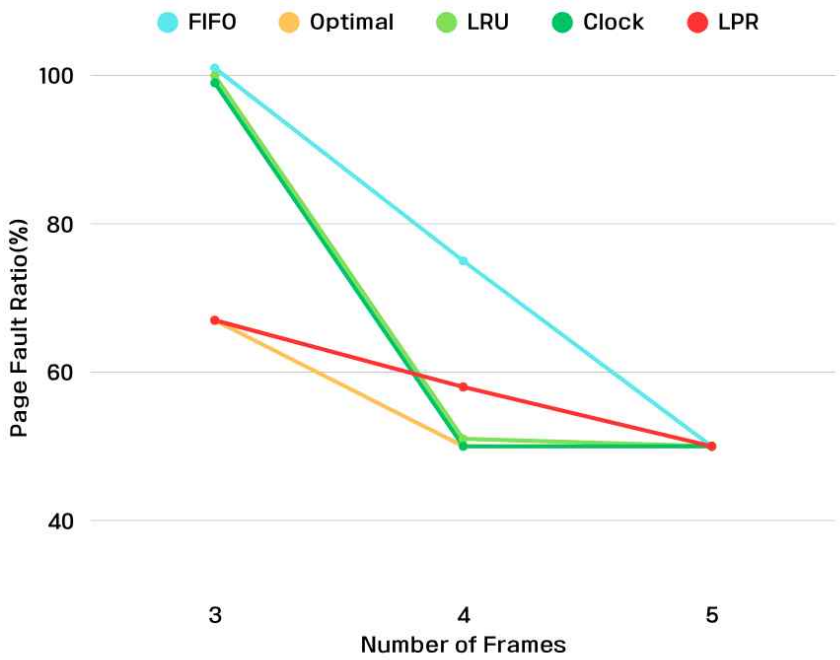
[그림 11]에서 볼 수 있듯이, 5가 처음 등장하는 순간 이후 참조될 페이지에 대한 정확한 확률 분포를 판단할 수 없어, 리스트 가장 앞에 있는 프레임 1을 교체하게 된다. 그러나 직후에 다시 페이지 1이 재참조 되며, 결국 Page Fault가 발생한다. 이 현상은 초기 예측 실패로 인한 성능 저하를 보여주는 대표적인 사례이다.

참조가 반복되면서 LPR은 점차 "1 → 2", "2 → 3", "3 → 4", "4 → 5", "5 → 1"과 같은 순환 구조를 학습하게 되고, 이를 기반으로 페이지 재등장 확률을 더 정확하게 예측할 수 있게 된다.

이 때문에 알고리즘의 교체 결정은 점차 안정적으로 바뀌며, 중후반부에서는 Optimal에 가까운 성능을 보이게 된다. 그럼에도 초기 예측 실패로 인한 Page Fault 수 증가는 최종 성능 지표에 영향을 주게 된다.

- **2-3. Reference String: 1 2 3 4 1 2 3 5 1 2 3 6**

(1) 프레임 개수 별 Page Fault 비율



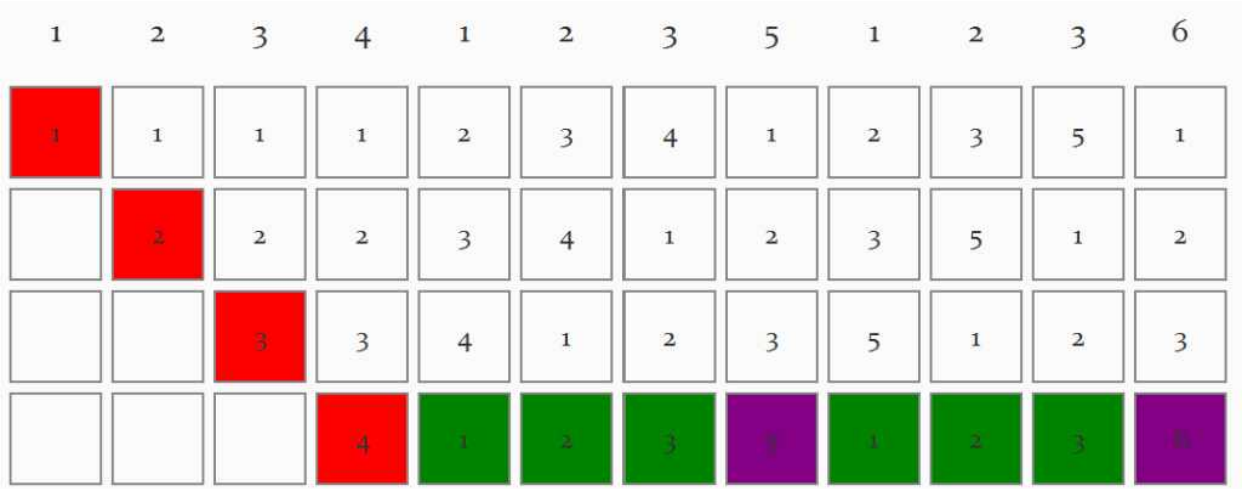
[그림 12] 프레임 수에 따른 페이지 폴트 비율

[그림 12]는 참조 문자열에 대해 다양한 페이지 교체 알고리즘의 성능을 비교한 결과를 시각화한 것이다. 이 참조 문자열은 1, 2, 3 세 페이지의 강한 지역성을 평가하기 위해 설계되었으며, 4, 5, 6과 같은 일회성 페이지를 중간에 한 번씩 삽입하는 구조로 되어 있다.

LPR은 과거의 참조 이력을 기반으로 "1 → 2", "2 → 3"과 같은 패턴을 학습함으로써, Optimal 알고리즘과 유사한 수준의 예측 성능을 달성하는 것을 [그림 12]를 통해 확인할 수 있다. 이는 LPR이 지역성이 강한 참조 패턴에 대해 효과적으로 작동함을 의미한다.

그러나 프레임 수가 4개일 때, LPR은 LRU나 Clock 알고리즘보다 상대적으로 더 높은 Page Fault 비율을 보인다. 이는 참조 초반에 등장한 페이지 5에 대해 이전 데이터가 없어, LPR이 학습 기반 예측을 수행하지 못하고 큐의 가장 앞에 있는 페이지를 교체하게 되기 때문이다. 해당 페이지가 이후 곧바로 재참조 되면서 예측 실패에 따른 추가적인 Fault가 발생하는 현상이 나타난다.

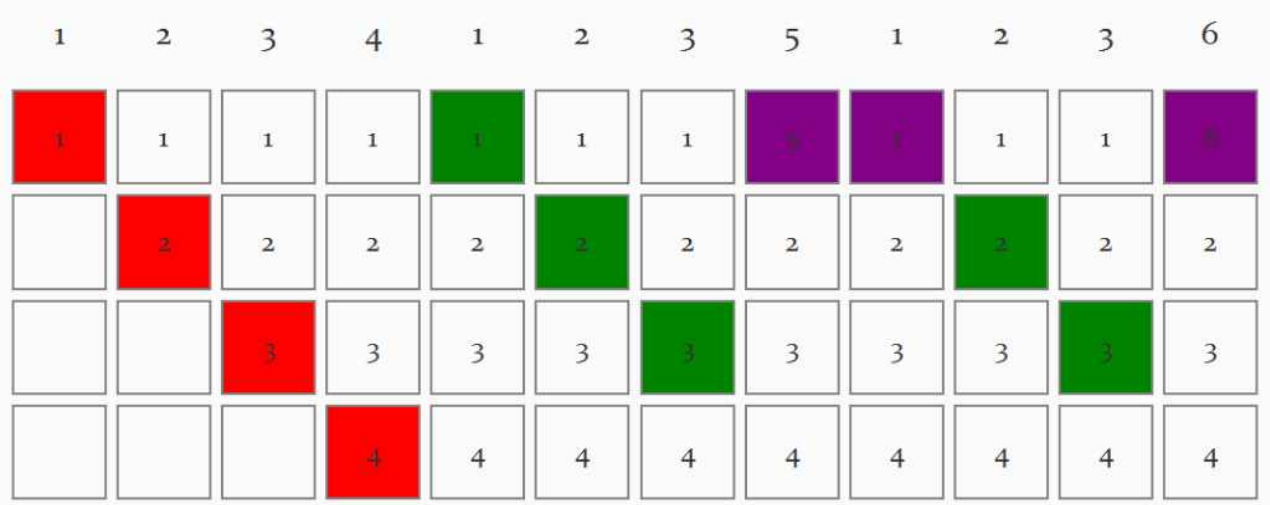
(2) 시뮬레이터 결과 (LRU, LPR)



[그림 13] LRU, 프레임 수 = 4일 때

LRU 알고리즘은 가장 오랫동안 참조되지 않은 페이지를 교체 대상으로 선택한다. 따라서 이 알고리즘은 각 페이지의 참조 시점을 기준으로 정렬된 큐를 유지하며, 페이지가 Hit 되더라도 해당 페이지를 큐의 맨 뒤로 이동시킨다. 반대로 Page Fault가 발생하면 큐의 앞에 있는 가장 오랫동안 참조되지 않은 페이지를 제거하고, 새로운 페이지를 큐의 끝에 추가한다.

[그림 13]을 통해 확인할 수 있듯이, Hit 된 페이지는 초록색 Fault가 발생하여 교체된 페이지는 보라색으로 표시되며, 이 두 색 모두 항상 큐의 맨 끝에 있게 된다. 특히, 프레임 수가 4개인 경우 1, 2, 3과 같은 핫 페이지들은 지속해서 참조되기 때문에 항상 큐의 뒷부분에 존재하게 되며, 이 때문에 새로 등장하는 일회성 페이지(4, 5, 6)들끼리 교체되는 현상이 나타난다. 이는 LRU가 지역성을 강하게 반영하는 알고리즘임을 보여주는 사례로, 기존의 중요 페이지를 유지하며 불필요한 교체를 줄이는 데 효과적이다.



[그림 14] LPR, 프레임 수 = 4개

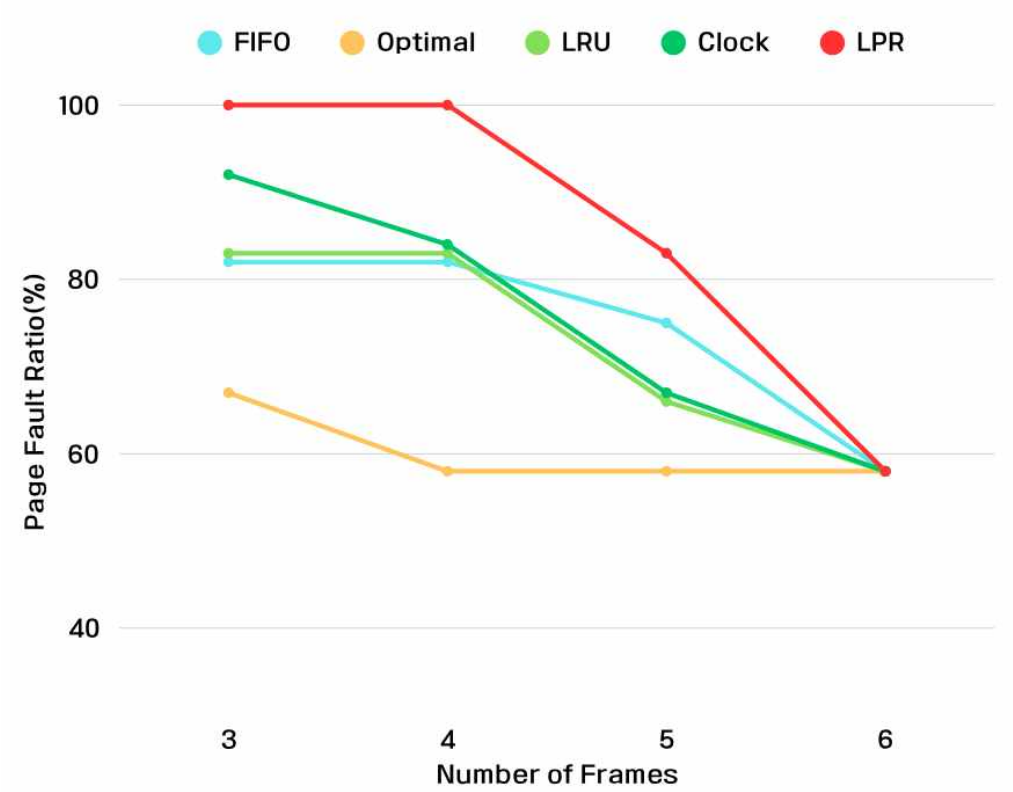
LPR 알고리즘은 과거 참조 이력을 기반으로 다음에 등장할 페이지를 예측하여, 가장 등장 확률이 낮은 페이지를 교체 대상으로 선정한다. [그림 14]에서는 프레임 수가 4개인 상황에서 참조 패턴 "1 → 2", "2 → 3", "3 → 4"와 같은 연관성을 학습하여 높은 Hit율을 유지하는 모습을 확인할 수 있다.

하지만 8번째 참조인 페이지 5가 등장하면서 예측 데이터의 한계를 드러낸다. 5는 이전에 등장한 적이 없어, 이후 등장할 페이지에 대한 학습된 정보가 존재하지 않는다. 이로 인해 LPR은 확률 기반 판단을 하지 못하고, 리스트 맨 앞에 있는 페이지 1을 교체한다. 문제는 그 직후인 9번째 참조에서 다시 1이 등장한다는 점이다. 결과적으로 LPR은 이 시점에서 예측에 실패하고 Page Fault가 발생하면서 전체 Fault 비율이 증가하게 된다.

이 사례는 LPR 알고리즘이 충분한 과거 참조 이력이 확보되지 않았을 때 일반적인 FIFO와 유사한 방식으로 동작하게 되며, 이로 인해 예외적인 성능 저하가 발생할 수 있음을 보여준다.

- **2-4. Reference String: 2 5 1 4 3 2 7 2 6 1 3 6**

(1) 프레임 개수 별 Page Fault 비율



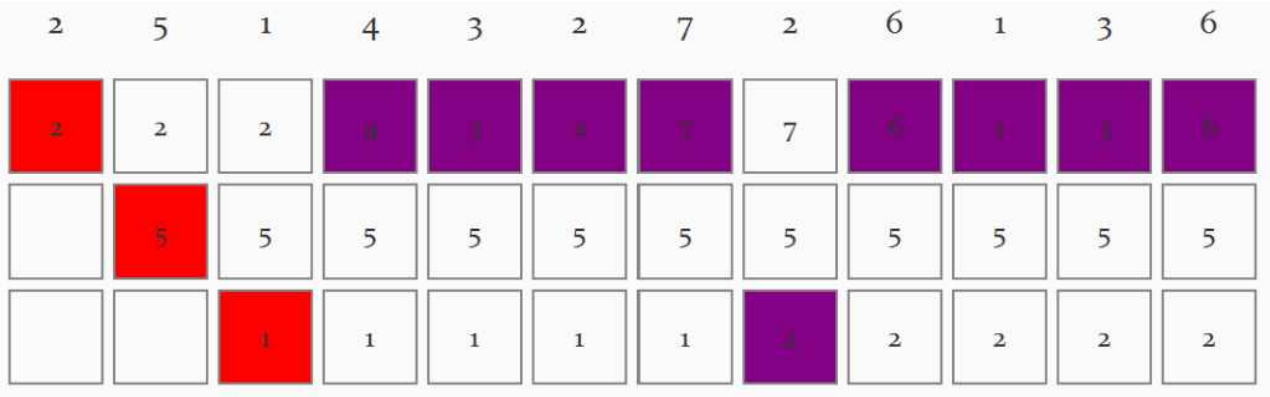
[그림 15] 프레임 수에 따른 페이지 폴트 비율

[그림 15]은 무작위로 구성된 참조 문자열에 대해 여러 페이지 교체 알고리즘의 성능을 비교한 결과이다. 해당 문자열은 명확한 반복 패턴이나 지역성이 존재하지 않는 비순차적이고 불규칙한 참조 배열로 구성되어 있으며, 이는 알고리즘의 예측 성능을 실제로 검증하기에 적합한 테스트 케이스이다.

Optimal은 미래 참조를 알고 있기 때문에 본 실험에서도 예외 없이 모든 알고리즘 중 가장 우수한 성능을 보인다. FIFO는 단순 삽입 순서에 따라 가장 먼저 들어온 페이지를 교체하기 때문에, 불규칙한 문자열에서도 준수한 수준의 Page Fault 수치를 유지한다.

LRU와 Clock 알고리즘은 무작위 문자열에서도 일정 수준의 지역성을 포착하여 FIFO와 유사한 성능을 나타낸다. LPR은 과거 참조 이력 기반으로 등장 확률이 낮은 페이지를 제거하는 방식이나 해당 문자열은 참조의 일관성이 부족하여 신뢰할 수 있는 통계가 누적되지 못한다. 이로 인해 오히려 FIFO보다 더 많은 Page Fault를 유발하며, 예측 기반 알고리즘의 한계를 드러낸다.

(2) 시뮬레이터 결과 (LPR)



[그림 16] LPR, 프레임 수 = 3개

[그림 16]은 무작위 참조 문자열에 대해 LPR 알고리즘이 어떤 방식으로 페이지 교체 대상을 선택하는지를 시각화한 것이다. 이 알고리즘은 과거의 참조 이력 데이터를 기반으로 다음에 등장할 확률이 가장 낮은 페이지를 교체하는 방식으로 동작한다.

LPR은 일반적으로 충분한 참조 이력이 존재할 경우, 다음에 등장할 가능성이 낮은 페이지를 정확히 예측하여 교체 효율을 높인다. 하지만 이번 실험에 사용된 참조 문자열은 불규칙한 무작위 배열로 구성되어 있어, 예측에 필요한 연속된 참조 패턴이 부족하다.

그 결과, 대부분 시점에서는 과거 데이터를 기반으로 한 확률 계산이 불가능했으며, 이럴 때 LPR은 기본적으로 리스트에서 가장 앞에 있는 페이지를 교체 대상으로 선택하게 된다.

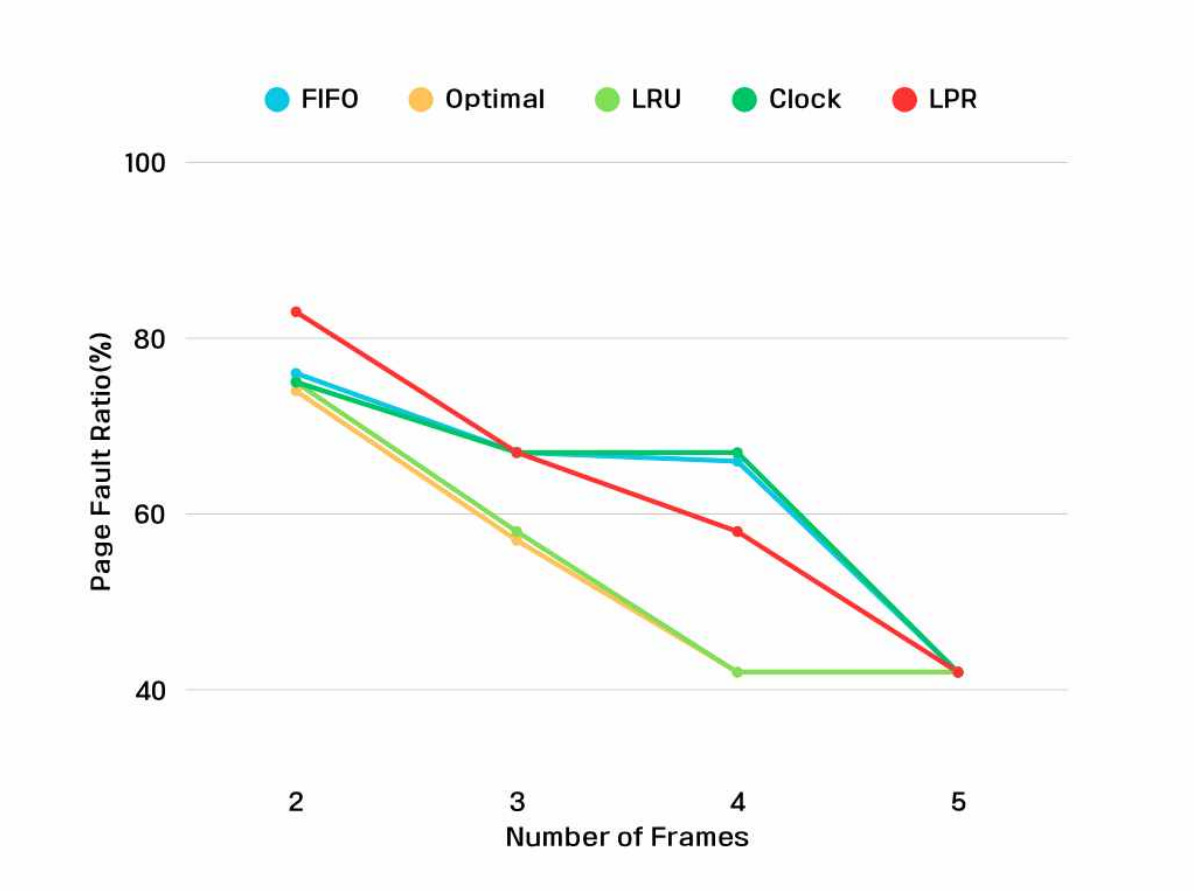
8번째 참조인 페이지 2에서 과거 이력에 2 → 5, 2 → 7이라는 연관성이 각각 1회씩 존재하므로, 프레임 내에서 다음 등장 확률이 가장 낮은 페이지인 1이 교체 대상이 된다. 실제로 [그림 16]에서는 이 시점에서 세 번째 프레임이 보라색으로 표시되어 있어, 교체가 발생했음을 확인할 수 있다.

4번째, 5번째, 7번째, 9번째, 10번째, 12번째 참조 시점의 해당 시점에서는 과거 데이터가 전혀 존재하지 않아 기본 정책에 따라 리스트의 가장 앞에 있는 페이지가 교체 대상으로 선택되었다.

6번째 참조인 페이지 2가 들어왔을 때 과거에 2 → 5가 1회 존재했으므로, 페이지 5를 제외한 나머지 3과 1중에서 교체 대상을 선택해야 한다. 구현에서는 이 경우 프레임 리스트에서 가장 앞에 있는 3을 교체 대상으로 선택하였다. 마찬가지로 11번째 참조인 페이지 3이 들어왔을 때 과거에 3 → 2가 1회 있었기 때문에, 프레임 내에서 2가 아닌 페이지 1 또는 5중 하나를 교체해야 했다. 구현 기준에 따라 가장 앞에 있는 프레임인 1이 교체되었다.

- **2-5. Reference String: 1 2 3 4 4 3 2 1 5 1 2 3**

(1) 프레임 개수 별 Page Fault 비율



[그림 17] 프레임 수에 따른 페이지 폴트 비율

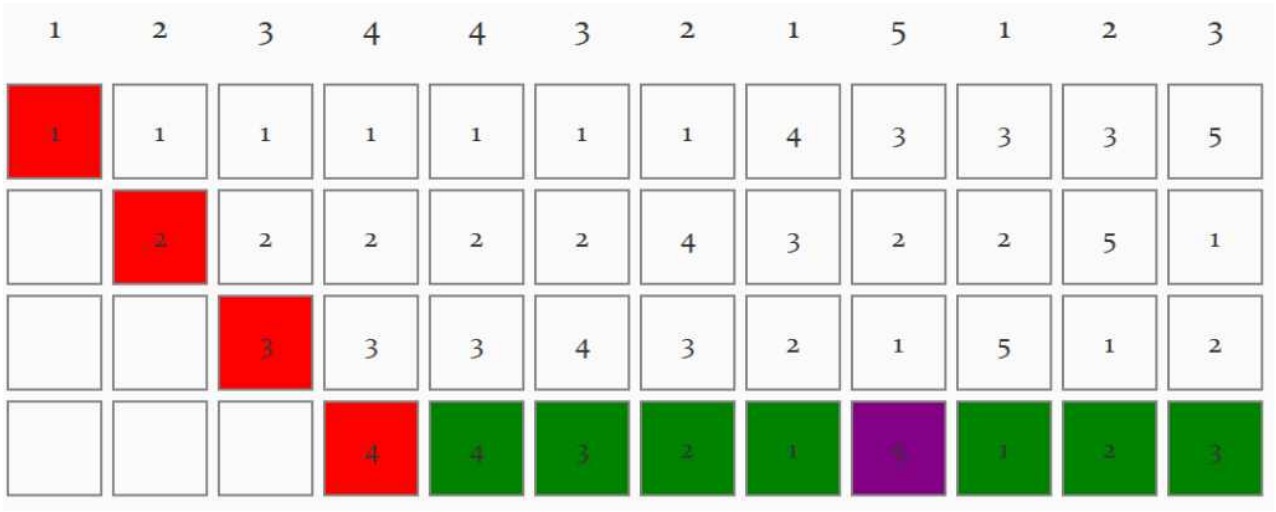
LRU 알고리즘과 Clock 알고리즘 간의 차이를 명확히 드러내기 위해 설계된 테스트 케이스이다. 두 알고리즘 모두 "가장 최근에 사용되지 않은 페이지"를 교체 대상으로 삼는다는 기본 원칙은 같지만, 구현 방식의 차이 때문에 실제 동작 결과에서 차이가 발생한다. 이번 실험은 그 대표적인 사례를 보여준다.

LRU는 참조된 페이지를 항상 가장 최근 위치로 이동시켜 정확한 참조 순서를 유지한다. 참조가 발생할 때마다 해당 페이지를 리스트의 맨 뒤로 옮김으로써 정확한 LRU 정책이 구현된다.

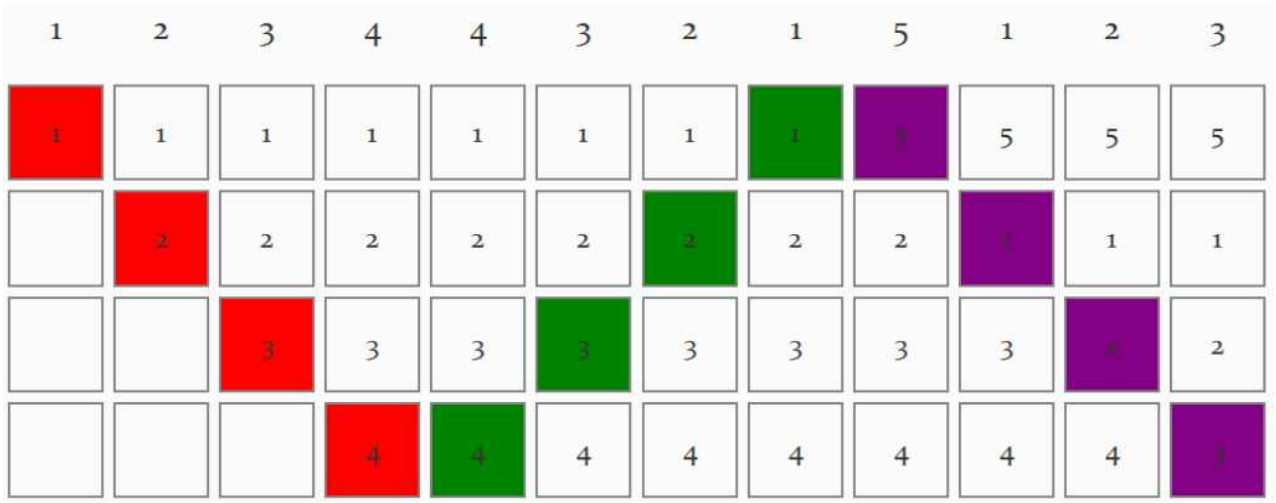
Clock 알고리즘은 각 페이지에 reference flag를 설정하여 최근에 사용된 적이 있는지만 기록한다. 교체 시에는 시계 방향으로 포인터를 돌며 reference bit이 0인 페이지를 찾는다. 이 방식은 참조 순서를 정확히 반영하지 않고, 단지 최근 사용 여부만 간접적으로 판단하기 때문에 근사적 LRU로 분류된다.

[그림 17]을 통해 프레임 수가 3개일 때는 두 알고리즘의 동작 결과에 큰 차이가 없었지만, 프레임 수가 4개로 증가한 경우에는 명확한 성능 차이가 발생함을 확인할 수 있다.

(2) 시뮬레이터 결과 (LRU, Clock)



[그림 18] LRU, 프레임 수 = 4개



[그림 19] Clock, 프레임 수 = 4개

LRU 알고리즘은 참조가 발생할 때마다 해당 프레임을 리스트에서 제거한 후, 다시 리스트의 맨 뒤로 삽입함으로써 가장 최근에 사용된 페이지 순서를 유지한다. [그림 18]을 통해 확인할 수 있듯이, 참조가 발생한 페이지(초록색)는 항상 리스트의 맨 뒤에 있게 된다. 이러한 구조는 완전한 참조 순서를 반영하며, 가장 오래 사용되지 않은 페이지를 교체하는 데 정확하게 작동한다.

Clock 알고리즘은 각 프레임에 대해 reference flag를 설정하고, pointer가 이를 순회하면서 교체 대상을 찾는다. 참조가 발생하면 해당 프레임의 reference flag는 true로 설정되며, 교체 시에는 flag가 false인 프레임이 선택된다.

5번째부터 8번째 참조까지 모두 Hit가 발생함에 따라, 리스트 내 모든 프레임의 reference flag는 true 상태가 된다. 이 기간에 실제로 페이지 교체가 발생하지 않았기 때문에, pointer는 여전히 프레임 1번을 가리키고 있다.

9번째 참조로 새로운 페이지인 5가 들어오면서 페이지 교체가 필요해진다. Clock 알고리즘은 pointer가 가리키는 프레임부터 시작하여, reference flag가 true일 때 flag를 false로 설정한 후 다음 프레임으로 이동한다. 이 과정을 통해 모든 프레임의 flag가 false가 되고, pointer는 다시 처음 위치인 프레임 1로 돌아온다. 이 시점에서 reference flag가 false인 프레임 1이 교체 대상이 되어, 페이지 5로 교체된다.

이후 10번째, 11번째, 12번째 참조에서도 새로운 페이지가 연속적으로 등장하게 되고, 모두 Page Fault가 발생한다. 이 과정에서 pointer는 한 칸씩 이동하며 차례대로 페이지를 교체하게 되는데, 교체되는 페이지들이 모두 곧바로 다시 참조되는 페이지이기 때문에, Page Fault 비율이 급격히 상승하게 된다.

Clock 알고리즘은 간단하고 효율적인 구현 방식으로 LRU를 근사하지만, 모든 reference flag가 true인 상태에서 일괄 초기화 후 교체가 일어나는 구조 때문에, 특정 참조 패턴에서는 LRU보다 더 많은 Page Fault가 발생할 수 있다. 특히, 교체 직전에 pointer가 연속으로 참조될 페이지들을 제거하는 경우, 실제 재사용 가능한 페이지를 잃게 되어 성능 저하가 발생한다.

V. 결론

1. 전체 요약

본 프로젝트에서는 운영체제 수업에서 다룬 대표적인 페이지 교체 알고리즘인 FIFO, Optimal, LRU, Clock, 그리고 새롭게 제안한 LPR 알고리즘을 직접 설계하고 구현하였다. 모든 알고리즘은 동일한 시뮬레이션 환경에서 동작하도록 구성하여, 공정한 비교와 평가를 할 수 있도록 하였다.

각 알고리즘은 교체 대상을 선택하는 기준, 사용되는 자료구조, 구현 난이도, 시간 및 공간 복잡도, 그리고 실제 환경에서의 적용 가능성 등에서 뚜렷한 차이를 보인다. 알고리즘별 작동 원리와 구현 과정을 분석하고, 시뮬레이션 결과를 바탕으로 성능 차이를 비교함으로써 각각의 장단점과 특성을 종합적으로 이해할 수 있었다.

2. 시사점

페이지 교체 알고리즘은 단순히 이론적 성능만으로 판단하기 어려운 측면이 있다. 예를 들어 Optimal 알고리즘은 가장 이상적인 기준을 제시하지만, 실제 시스템에서는 구현할 수 없다는 분명한 한계를 가진다. 또한, FIFO는 구현은 단순하지만 Belady의 Anomaly와 같은 비효율적인 예외 현상도 관측할 수 있었다.

LRU와 Clock 알고리즘은 시간 지역성을 반영하면서도 현실적인 구현이 가능하다는 점에서, 실제 운영체제에서 널리 사용되는 이유를 설명해준다. 특히 Clock 알고리즘은 적절한 타협점을 제공하면서도 안정적인 성능을 유지한다는 점에서 높게 평가할 수 있었다.

본 프로젝트에서 새롭게 제안한 LPR 알고리즘은 과거 데이터를 바탕으로 앞으로의 참조 확률을 추정하여 교체 대상을 선택하는 새로운 방식으로, 기존 방식들과는 다른 '데이터 기반 예측'이라는 관점을 시도하였다. 시뮬레이션에서는 일부 경우에서 LRU, Clock에 준하는 성능을 보여주었으며, 반복 패턴이 뚜렷한 경우에는 오히려 더 나은 결과를 내기도 했다.

이러한 비교 분석을 통해, 페이지 교체 정책은 단순한 규칙 이상의 의미가 있다는 점을 확인할 수 있었다. 사용 환경, 데이터 패턴 등에 따라 가장 적합한 알고리즘은 달라질 수 있으며, 상황에 따라 유연한 선택과 설계가 필요하다는 점을 시사한다.

3. 향후 확장 가능성 및 제안

본 프로젝트는 제한된 조건 내에서의 시뮬레이션을 기반으로 알고리즘의 성능을 비교하였다. 공통된 참조 문자열을 바탕으로, 알고리즘 간 비교는 페이지 폴트 수를 중심으로 수행되었다. 하지만 실제 운영체제 환경은 이보다 훨씬 더 복잡하고 다양한 제약을 가진다. 따라서 몇 가지 확장과 보완이 필요하다.

첫째, 참조 문자열의 다양화가 필요하다. 현재는 특정 패턴이 반영된 문자열을 기반으로 실험을 수행했지만, 실제 프로그램 실행 중 발생하는 페이지 접근 패턴은 반복성과 지역성이 훨씬 더 복잡하다. 멀티태스킹 환경 등 실제 사용자 시나리오에서 추출한 패턴을 참조 문자열로 적용한다면 알고리즘을 보다 현실적인 사용자 측면에서 더욱 정확한 평가를 할 수 있다.

둘째, LPR 알고리즘의 정밀도 향상이 가능하다. 현재는 단순한 1-gram 통계 기반 구조만을 사용했지만, 2-gram 또는 3-gram 분석, 마르코프 체인 기반 확률 모델 등으로 발전시킨다면 예측 정확성의 향상을 더욱 기대할 수 있다. 특히, 반복되는 참조 패턴이 많은 프로그램 환경에서는 LPR이 더 정교한 판단을 할 수 있을 것으로 기대된다.

셋째, 성능 평가 기준의 확장도 고려되어야 한다. 본 프로젝트는 알고리즘 간의 비교를 단순화하기 위해 페이지 폴트 수만을 성능 기준으로 삼았지만, 실제로 알고리즘이 동작하면서 발생하는 연산 비용, 알고리즘 유지에 필요한 오버헤드 등이 크게 반영되지 못하였다. 실제 운영체제 환경에서는 이보다 훨씬 다양한 요소들이 존재하기에 이러한 요소들 역시 종합적으로 고려되어야 한다.

결론적으로, 본 프로젝트는 다양한 알고리즘을 하나의 공통 구조 내에서 구현하고, 비교 가능한 실험 환경을 구성함으로써 기본적인 성능 분석을 수행하였다. 앞으로는 더 복잡한 환경과 예측 기법을 도입하여, 페이지 교체 정책에 대한 보다 정교한 연구로 확장될 수 있을 것이다.

4. 프로젝트를 통한 학습 및 소감

본 프로젝트를 통해 단순히 알고리즘을 외우는 것을 넘어서, 직접 하나하나 구현하는 과정을 통해 많은 것을 배울 수 있었다. 이론 수업 시간에서는 FIFO나 LRU, Clock 같은 알고리즘을 그저 원리만 짚고 넘어갔는지만, 실제로 알고리즘을 설계하는 과정에서 많은 고려사항이 존재하였고, 실제 구현 작업 시 작동방식 등에 충분한 고려가 필요하였다는 점에서 스스로 역량이 성장했음을 느꼈다.

특히 조건문 하나 때문에 흐름이 크게 바뀌는 일도 있어서 디버깅하면서 자연스럽게 알고리즘을 더 깊이 이해하게 됐다. 또 LPR처럼 처음부터 직접 설계한 알고리즘은, 예상보다 구현에서 많은 시행착오가 있었지만, 그만큼 구조나 방식에 대해 많이 고민할 기회가 되었다.

또한, 알고리즘별로 장단점을 비교하고, 시뮬레이션 결과를 통해 성능 차이를 분석하는 과정에서는 단순히 '무엇이 더 좋다'는 식의 판단보다는, 상황과 목적에 따라 어떤 알고리즘이 더 적합할 수 있는지를 고민해보게 되었고, 이것이 운영체제 설계의 중요한 관점이라는 점도 느낄 수 있었다.

결과적으로 이번 프로젝트는 단순한 구현 과제를 넘어서, 이론과 실습을 연결해보는 경험이 되었으며, 운영체제의 메모리 관리라는 핵심 주제를 더욱 깊이 있게 이해하는 계기가 되었다.

Ⅵ. 참고 문헌

[1] Operating System Concepts (10th Edition) - Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Wiley, 2018.

[2] 「An Experimental Evaluation of Demand Paging Algorithms」 - James Bennett, Communications of the ACM, Vol. 13, No. 6, 1970.

[3] 『JavaFX Documentation』 - OpenJFX, https://openjfx.io

[4] 『혼자 공부하는 운영체제』 - 우재남, 한빛미디어, 2021.

[5] 『이것이 자바다』 - 신용권, 임경균 저자, 한빛미디어, 2024