지역성에 따른 페이지 교체 정책의 성능 분석 및 새로운 정책 제안

컴퓨터공학과

22311947

김서현

[요 약]

본 프로젝트는 프로그램의 지역성과 페이지 교체 정책의 성능 간 연관성을 분석하고 이를 바탕으로 새로운 페이지 교체 정책인 PFU(Probabilistic Frequently User)를 제안하는 것을 목표로 한다. 실험 결과, 프로그램의 지역성에 따라 FIFO, LFU, MFU 등 기존 정책들이 성능의 변화를 보이는 것을 확인했다. 특히 제안하고자 하는 PFU는 지역성의 변화가 심한 환경에서도 뛰어난 성능을 발휘한다는 것을 입증했다. 본 프로젝트는 페이지 교체 정책에서 지역성의 중요도를 강조하며 PFU 알고리즘이 새로운 페이지 교체 정책으로서 차별화된 가치를 가지고 있음을 시사한다.

▸키워드 : operating system, program, memory, page, page hit, page fault, page migration, fault rate, locality, reference count, page replacement policy, algorithms, FIFO, count-based, LFU, MFU, PFU, frame, reference string, simulator, efficiency, probabilistic

I. 서론

운영체제는 시스템의 자원을 관리하기 위해 존재한다. 어떠한 시스템에서 가장 핵심이 되는 자원은 CPU, 메모리, 디스크로 이 자원이라는 건 다양한 정책을 통해 가장 효율적인 방법을 찾아 관리된다. 본 프로젝트는 운영체제의 메모리 관리에 대해 이야기된다.

현재 우리가 사용하는 많은 시스템에는 아주 다양한 종류의 메모리 정책이 존재한다. 컴퓨터 시스템에서 '절대적으로 유용한 정책'은 존재하지 않는다. 우리가 특정한 정책의 성능을 판단하고자 할 때는 시스템이 사용될 하드웨어의 성능, 시스템의 사용 목적, 돌아가는 프로그램의 사용 패턴 등 많은 것을 고려하여야 한다. 특정 프로그램에서는 최악의 성능을 내는 정책이 또 다른 프로그램에서는 최고의 성능을 내게 될 수도 있다. 그리고 이러한 점에서 흥미를 느껴 각 메모리 정책 사용되는 프로그램 환경과 효율성의 관계를 파악하기 위해 본 프로젝트를 시작하게 되었다.

본 프로젝트에서는 프로그램이 실행되는 참조 패턴과 다양한 메모리 정책 간의 관계에 집중해 메모리 정책 효율성을 분석한다. 실제 시스템에서 있을 법한 다양한 상황들을 가정해 문자열을 입력하고, 이에 따른 fault rate를 비교해 가며 어떤 상황에서 어떤 정책이 높은 효율성을 가지게 되는지를 판단한다. 그리고 최종적으로는 이번 프로젝트에서 구현한 기존의 정책들(FIFO, LFU, MFU)이 포용하지 못했던 상황에서 이를 보완할 수 있는 나만의 정책을 만들고자 한다.

결국 이번 프로젝트의 두 가지 목표는 참조 문자열에 따른 기존 정책들의 효율성 분석과 기존 정책에서 효율적이지 못했던 부분을 개선하는 새로운 정책을 만드는 것이다. 이를 위해 Java 코드로 작성된 페이지 교체 시뮬레이터를 사용하였으며 시뮬레이션 결과를 다방면에서 분석하여 최종적인 결과를 산출한다.

II. 배경지식 및 관련 기술

1. 배경지식

본 프로젝트를 진행하기 위해서는 몇 가지 사전 지식이 필수적이다. 우선 메모리 교체 정책에 대해 알기 위해서는 메모리가 무엇인지를 알아야만 한다. 메모리는 CPU의 저장소와 같은 역할을 한다. 프로그램이 실행되지 않을 때, 해당 프로그램의 코드는 하드 디스크의 안에 존재한다. 그러나 CPU가 해당 프로그램을 실행하고자 할 때, 실행 중인 프로그램의 코드는 반드시 메모리 위에 올라가야 한다.

메모리에 프로그램을 올리는 방법은 크게 두 가지로 존재하는데 실행 중인 프로그램의 내용을 통째로 올리는 물리 메모리와 프로그램의 일부만을 메모리에 올리는 가상 메모리이다. 이번 프로젝트의 모든 시스템은 가상 메모리로 동작하는 것을 가정하고 있다.

가상 메모리는 프로그램의 일부만을 메모리에 올리게 되고 이때 해당 프로그램은 Page라는 단위로 쪼개진다. 프로그램의 원활한 실행을 위해 메모리 위에 CPU가 필요로 하는 적절한 페이지를 올려야만 하고, 제한된 메모리 위에 가장 필요한 페이지만을 올리는 페이지 교체 정책이 무엇보다 중요하다.

페이지 교체 정책에서 페이지의 상태는 크게 Hit, Fault, Migration 세 가지로 나뉜다. 이미 메모리에 올라가 있는 페이지가 다시 한 번 사용되고자 할 경우, 페이지 Hit가 발생한다. 이 경우 별도의 추가 과정 없이 해당 페이지를 바로 사용할 수 있다. 메모리 위에 찾는 페이지가 존재하지 않는 경우, 페이지 Fault 혹은 페이지 Migration이 발생한다. 만약 메모리 위에 여분의 공간이 존재한다면 페이지 교체 없이 바로 디스크의 페이지를 메모리에 카피할 수 있다. 그러나 메모리에 여분의 공간이 존재하지 않을 경우, 메모리 위에 올라가 있는 기존 페이지 하나를 삭제하고 사용하고자 하는 페이지를 가져오는 페이지 교체 작업이 필요하다.

페이지 교체 정책에서 가장 중요한 것은 메모리에 공간을 만들기 위해 기존의 페이지들 중 어떤 페이지를 삭제하느냐이다. 같은 메모리 공간에서 같은 프로그램을 실행하더라도 페이지 교체 정책을 어떻게 선택하느냐에 따라 시스템의 효율성이 크게 달라질 수 있다. 만약 페이지 교체 정책을 잘못 선택해, 다음에 사용하려 한 될 페이지가 메모리에서 삭제된다면 페이지 Fault가 발생해 프로그램의 성능을 크게 떨어트릴 수도 있다.

따라서 모든 페이지 교체 정책의 효율성은 해당 프로그램을 실행하였을 때 페이지 Fault가 발생하는 빈도수로 결정된다. 같은 메모리 같은 프로그램이라면 적절한 교체 정책을 사용해 페이지 Fault 확률을 낮추는 것이 무엇보다 중요하다. 본 프로젝트에서 역시 페이지 교체 정책의 효율성을 판단하는 척도로 페이지 Fault 확률을 사용하게 될 것이다.

마지막으로 페이지 교체 정책에 있어 또 한 가지 중요한 것은 지역성(Locality)의 개념이다. 만약 모든 프로그램이 완전히 무작위로 페이지를 사용한다면 페이지 교체 정책을 연구하는 것은 아무런 의미를 가지지 않을 것이다. 그러나 모든 프로그램에는 지역성이 존재해, 페이지가 호출되는 것에도 일정한 규칙이 존재한다. 즉, 모든 프로그램에는 얕으나 깊으나 지역성이 존재하기에 페이지 교체 정책은 메모리 관리에 있어 큰 의미를 가진다.

2. 관련 기술

현재까지 가장 보편적으로 알려진 페이지 교체 알고리즘에는 FIFO, Optimal, LRU, Far page replacement, count based replacement 등이 있다.

FIFO 알고리즘은 선입 선출 방식으로 메모리에서 가장 오래 머무른 페이지를 가장 먼저 삭제한다. 다른 알고리즘에 비해 구현이 간단하며 오버헤드가 최소화된다는 장점이 존재한다. 그러나 중요한 페이지나 사용 중인 페이지가 교체될 수 있으며 낮은 성능으로 단독으로 사용되기엔 무리가 있다. 또한 Frame의 크기를 키웠음에도 페이지 Fault가 증가하는 벨라디의 이상 현상(Belady's Anomaly)이 발생하기도 한다.

Optimal 알고리즘은 최적 알고리즘으로 최초의 실행이 나와 제일 먼 페이지, 즉 앞으로 가장 오랫동안 사용되지 않을 페이지를 교체한다. 최적 알고리즘이라는 이름에 걸맞게 모든 알고리즘의 통틀어 최적의 성능을 보장한다. 그러나 현실에서 미래에 쓰일 페이지를 완전히 예측하는 것은 불가능하다. 따라서 최적 알고리즘은 이론적으로 존재하며 다른 알고리즘들의 성능을 측정하기 위해 활용된다.

LRU 알고리즘은 미래의 페이지를 기준으로 삼는 Optimal 알고리즘과 반대로 가장 오랫동안 사용되지 않은 페이지를 교체하는 알고리즘이다. LRU는 지금까지의 실행 패턴을 어딘가에 기록해 두었다가 현재로부터 가장 마지막에 사용되었던 페이지를 삭제한다. 해당 알고리즘은 몹시 높은 효율성을 가진다. 그러나 과거의 패턴을 분석하는 알고리즘은 구현 복잡도가 높고 오버헤드를 발생시킨다. 때문에 성능과 현실성에서 적당히 타협을 본 근사화 과정이 필요하다.

Far page replacement 알고리즘은 Optimal 알고리즘의 근사화 버전으로도 생각될 수 있다. 해당 알고리즘은 실행하려는 프로그램의 접근 그래프를 작성해 해당 그래프를 토대로 현재 위치에서 가장 먼 곳에 존재하는 페이지를 교체한다. 접근 그래프는 컴파일 시간에 미리 정적으로 생성되거나 프로그램 실행 중 실시간으로 동적 생성될 수 있다. 그러나 Far page replacement 알고리즘 역시 현재는 이론적으로만 존재하며 실제 구현이 어렵다는 단점이 존재한다. 또한 그래프를 생성하고 분석하는 것에 높은 오버헤드가 존재하며, 생성된 접근 그래프와는 별개로 특정한 경로로 프로그램이 실행될 가능성이 있다.

Count based replacement 알고리즘은 페이지의 참조 횟수를 기준으로 삼는다. 해당 알고리즘에서 참조 횟수가 적은 페이지를 우선적으로 교체할 수도 있고 참조 횟수가 많은 페이지를 우선적으

로 삭제할 수도 있다. 참조 횟수가 적은 페이지를 우선적으로 삭제하는 LFU 알고리즘은 지금까지 많이 참조된 페이지가 앞으로도 많이 참조될 확률이 높다는 것을 가정한 알고리즘이다. 해당 알고리즘은 방금 메모리 안에 들어온 페이지를 패치할 가능성이 있다. 참조 횟수가 많은 페이지를 우선적으로 삭제하는 MFU 알고리즘은 참조 횟수가 많은 페이지는 이미 충분히 사용되었을 것을 가정한 알고리즘이다. 두 알고리즘은 정의에서부터 확연한 차이를 보인다.

실제 시스템에서는 이러한 알고리즘이 단독으로 사용되는 경우보다 서로의 장점을 살려 혼합되어 사용되는 경우가 더 많다. 각 알고리즘은 절대적으로 좋은 성능을 가지지 않는다. 시스템이 사용되는 목적과 환경에 따라 각 알고리즘 정책의 성능이 크게 달라질 수 있다.

예를 들어 특정 임베디드 시스템이나 제한된 메모리 환경에서는 낮은 오버헤드를 가지는 FIFO를 사용하는 것이 유리하다. 웹 브라우저의 캐시나 대부분의 현대 운영체제의 페이징 시스템에서는 LRU 기반의 알고리즘들을 사용하고, 빅데이터 처리 시스템이나 머신러닝의 메모리 관리에는 Count based replacement 알고리즘을 사용하는 것이 유리할 수 있다.

따라서 페이지 교체 정책을 선택할 때는 앞서 언급한 여러 요소들을 모두 고려하여야 할 것이다. 아래로는 위에서 설명한 페이지 교체 정책들이 실제 환경에서는 어떻게 적용되고 발전해 왔는지에 대한 서술이다.

마이크로소프트의 Windows의 경우, LRU를 변형한 Working Set Model을 사용한다. 해당 페이지 교체 정책은 LRU와 Working Set 알고리즘을 변형한 하이브리드 알고리즘이다.[1] 애플의 macOS의 경우, 가상 메모리를 기반으로 한 Mach VM을 사용한다. 해당 정책은 LRU 기반의 Clock 알고리즘을 수정하여 사용한다. 또한 참조 비트와 함께 페이지의 수정을 의미하는 더티 비트를 사용한다.[2] Linux의 경우, 다중 큐 페이지 교체 알고리즘을 사용한다. 해당 알고리즘은 LRU 기반의 하이브리드 알고리즘이다.[3]

특히 최근에는 머신러닝을 활용한 페이지 교체 정책을 개발 중에 있으며 예측 모델링이 활발하게 연구 중에 있다. 실제 교체 정책들은 다양한 상황에서 활용 가능한 유연한 알고리즘을 채택하며 이러한 연구들은 클라우드 컴퓨팅, DB, 운영체제 등 다양한 분야에서 메모리 관리 효율을 높인다.

III. (본론) 주요 주제명

1. 주요 주제의 개요

본 프로젝트는 FIFO를 포함한 4가지 페이지 교체 정책을 구현하고 다양한 참조 문자열에서의 성능을 테스트하는 것을 목적으로 한다. 프로젝트 내에서 구현되는 페이지 교체 정책은 FIFO, LFU, MFU 이렇게 기존 정책 3가지와 기존의 단점을 보완해 작성한 나만의 페이지 교체 정책이다.

알고리즘의 구현에서는 각 알고리즘의 개념을 충실하게 반영하는 방향으로 구현을 진행하였다. FIFO의 경우, 각 페이지들은 메모리에 들어온 순서대로 List에 저장되며 가장 앞에서 삭제되고 가장 뒤에서 추가되는 형식을 취하였다. LFU와 MFU의 경우, 각 페이지들은 자신의 참조 횟수를 기억하며 페이지 Hit가 발생했을 때 참조 횟수를 증가시킨다. LFU는 참조 횟수가 가장 적은 페이지를 우선적으로 삭제하며 MFU의 경우 참조 횟수가 가장 큰 페이지를 우선적으로 삭제한다. 두 알고리즘 모두 페이지의 추가는 메모리 List의 가장 후방에서 이루어진다.

모든 페이지 교체 정책은 다양한 참조 문자열에서 성능 테스트를 진행한다. 이번 프로젝트에서는 특히 지역성이 높고 낮을 때, 한 프로그램 안에서 지역성이 변화할 때와 같은 경우에 Count based replacement 알고리즘인 LFU와 MFU의 장단점을 중점적으로 분석하고, 분석한 결과를 바탕으로 두 알고리즘의 장점을 살린 새로운 알고리즘을 설계한다.

LFU는 프로그램 전반적으로 지역성이 높은 상황에서 효율적으로 사용되며 MFU의 경우에는 지역성이 낮고 환경이 자주 바뀌는 프로그램에서 적합하다. 그러나 실제 프로그램이 운영되는 환경에서는 프로그램 전체가 하나의 지역성을 가지기보다 프로그램 내에서도 지역성이 높은 부분과 낮은 부분, 그리고 일부 페이지가 중점적으로 사용되는 부분과 사용되지 않는 부분이 나뉘게 된다.

이러한 환경에서 LFU 알고리즘을 사용할 경우, 한때 중요하게 사용되던 페이지가 지역성이 이동함에 따라 메모리 위에 불필요하게 남게 되고, 원활한 페이지 교체를 방해할 수 있다. 반대로 MFU 알고리즘의 경우, 프로그램 전반적으로 중요한 페이지가 너무 빨리 제거되어 프로그램의 성능이 저하될 수 있다.

이에 따라 본 프로젝트에서는 기존 Count based replacement 알고리즘의 한계를 보완하고 다양한 환경에서 유연성을 높이는 새로운 페이지 교체 정책 PFU(Probabilistic Frequently User)를 제안한다.

PFU는 LFU, MFU와 같이 Count based replacement 알고리즘의 한 가지 구현 방식이다. 해당 알고리즘에서는 메모리 위 페이지 중 참조 횟수가 가장 높은 페이지와 참조 횟수가 가장 낮은 페이지 중 하나를 일정 확률에 따라 랜덤하게 선택한다. 이후 선택된 페이지는 메모리에서 삭제되고 추후 페이지 교체가 일어날 경우 위의 과정을 반복한다. 또한 사용하고자 하는 프로그램의 성질에 따

라 참조 횟수가 높은 페이지와 낮은 페이지 중 어느 쪽의 삭제 확률을 높일 것인지를 사용자가 직접 결정할 수도 있다.

정리하자면 PFU 정책은 참조 횟수가 가장 높은 페이지와 가장 낮은 페이지 중 하나를 선택하여 삭제하는 방식으로, 프로그램의 패턴이 다이나믹하게 변화하는 환경에서도 안정적인 성능을 유지할 수 있다. 예를 들어 프로그램 실행 초반에는 A와 B 페이지가 집중적으로 사용되다가 중반에는 C 페이지를 자주 사용하고 후반에는 D, E, F 페이지를 주로 사용하는 복잡한 상황에서도 다른 Count based replacement 알고리즘들에 비해 효과적인 대응이 가능하다.

따라서 본 프로젝트에서는 FIFO, LFU, MFU, PFU의 네 가지 정책에 대한 성능 평가를 실시하, 각각의 장단점과 환경별 적합성을 분석한다. 특히 PFU가 다양한 실행 패턴에서 얼마나 안정적으로 성능을 발휘하는지를 검증하는 것이 핵심적인 목표가 될 것이다. 최종적으로는 PFU가 기존의 정책을 보완하고 새로운 페이지 교체 정책으로서 실질적인 가치를 갖는지에 대해 종합적인 결론을 내린다.

2. 주요 주제의 핵심 알고리즘 및 기능

아래로는 구현하고자 한 페이지 교체 시스템들의 핵심적인 알고리즘과 제공하는 기능에 대한 설명이 서술된다.

페이지를 구현한 Page 클래스의 경우, 각 페이지의 구별하기 위한 pid, 페이지의 위치를 명시하기 위한 loc, 각 페이지의 내용을 담고 있는 data, 페이지의 상태를 표시하는 status, 페이지의 참조 횟수를 저장하는 count 변수로 나뉜다. status 변수의 값은 HIT, PAGEFAULT, MIGRATION 세 가지이다. 생성자에서 정해지는 각 변수의 초기값은 아래와 같다.

```
this.pid = 0;
this.loc = 0;
this.data = '\0';
this.status = STATUS.PAGEFAULT;
this.count = 0; //해당 페이지의 참조 횟수를 저장하는 변수
```

페이지 교체 정책의 실질적인 구현은 모두 Core 클래스에서 이루어진다. Core 클래스에서 핵심적으로 사용되는 List는 아래 세 가지이다.

```
public List<Page> frame_window;
public List<Page> pageHistory;
public List<String> History;
```

frame_window는 현재의 메모리 상태를 표현한다. frame_window는 사용자가 입력한 frame size만큼의 페이지 변수를 저장한다. pageHistory는 지금까지 입력된 모든 페이지들의 입력 순서를 저장한다. History는 페이지가 입력되는 매 순간 frame_window의 상태를 String의 형태로 저장하한다. History는 사용자 인터페이스를 구현하는 GUI 클래스를 통해 사용자에게 페이지 교체 정책의 과정을 보여주기 위해 사용된다.

기존의 상태에서 새로운 페이지가 입력되면 Core 클래스의 operate() 함수가 실행된다. 해당 함수는 char 타입의 data 하나가 들어왔을 때 해당 데이터를 처리하는 역할을 맡는다. 이때 메모리 위에 존재하는 page의 data에 따라 페이지 Hit, Fault, Migration이 발생한다.

페이지 Hit와 페이지 Fault의 경우, 교체 정책이 무엇인지에 상관 없이 항상 동일한 코드가 실행된다. 페이지 hit가 발생하면 Core 클래스의 hit 횟수를 추가하고 frame_window에서 Hit가 발생한 페이지를 찾아 count의 횟수를 늘린다.

```
for (; i < frame_window.size(); i++) { //hit가 발생한 페이지 찾기
    if (frame_window.get(i).data == data) break;
}

if (frame_window.get(i).count <= MAX_COUNT) { //현재 MAX_COUNT = 10으로 정의
    frame_window.get(i).count++;
}
```

이때 count가 MAX_COUNT의 상수값을 넘지 않도록 count 변수의 최댓값을 조절한다. 이러한 처리는 LFU 알고리즘에서 특정 페이지의 참조 횟수가 과도하게 높아져 메모리에서 계속해서 살아남는 현상(메모리 고착화 현상)을 방지한다.

페이지 Fault가 발생한 경우, frame_window에 입력된 data를 담은 새 페이지를 추가하고 해당 페이지의 count 변수값을 0에서 1로 증가시킨다. 또한 Core 클래스의 fault 횟수를 1 증가시킨다.

페이지 Migration이 발생한 경우, 메모리 안의 페이지 하나를 삭제하고 새로운 페이지를 삽입하는 과정이 일어난다. 이때 사용자가 선택한 메모리 교체 정책에 따라 어떤 페이지를 삭제할 것인가가 결정된다.

```
if (this.memoryPolicy == 'F') { //FIFO 정책
    frame_window.remove(0);
    newPage.loc = cursor;
}
```

만약 선택된 교체 정책이 FIFO인 경우 frame_window의 가장 앞 메모리를 삭제하고

List.add() 함수를 통해 리스트의 가장 뒤에 새 페이지를 추가한다.

```
else if (this.memoryPolicy == 'L') { //LFU 정책
    int minIndex = 0;
    for (int i = 1; i < frame_window.size(); i++) {
        if (frame_window.get(i).count < frame_window.get(minIndex).count) {
            minIndex = i;
        }
    }
    frame_window.remove(minIndex);
    newPage.loc = minIndex + 1;
}
```

선택된 교체 정책이 LFU인 경우 frame_window의 모든 페이지에서 가장 낮은 count 변수값을 갖는 페이지의 위치인 minIndex를 찾는다. 이후 해당 페이지를 삭제하고 List.add() 함수를 통해 리스트의 가장 뒤에 새 페이지를 추가한다.

선택된 교체 정책이 MFU라면 minIndex 대신 maxIndex를 찾아 해당 페이지를 삭제하고 새 페이지를 추가한다.

선택된 교체 정책이 PFU(Probabilistic Frequently User)인 경우 minIndex와 maxIndex를 모두 구하고 두 인덱스 중 하나의 인덱스를 삭제 인덱스로 선택한다.

```
// 삭제 인덱스 선택
Random rand = new Random();
double randProb = rand.nextDouble(); //0.0 ~ 1.0 사이의 난수 생성
int chooseIndex;
if (randProb < PFU_PROBABILITY) { //PFU_PROBABILITY = 0.5로 정의
    chooseIndex = minIndex;
} else {
    chooseIndex = maxIndex;
}
```

PFU의 확률을 정하는 PFU_PROBABILITY 상수값은 필요에 따라 바뀔 수 있으나 해당 코드에서는 프로젝트의 원활한 진행을 위해 0.5의 값으로 임의 설정하였다.

페이지 교체 정책을 거쳐 메모리에 빈 공간이 마련되면 메모리 List에 새 페이지를 추가하고 migration, fault의 값을 각각 1씩 증가시킨다.

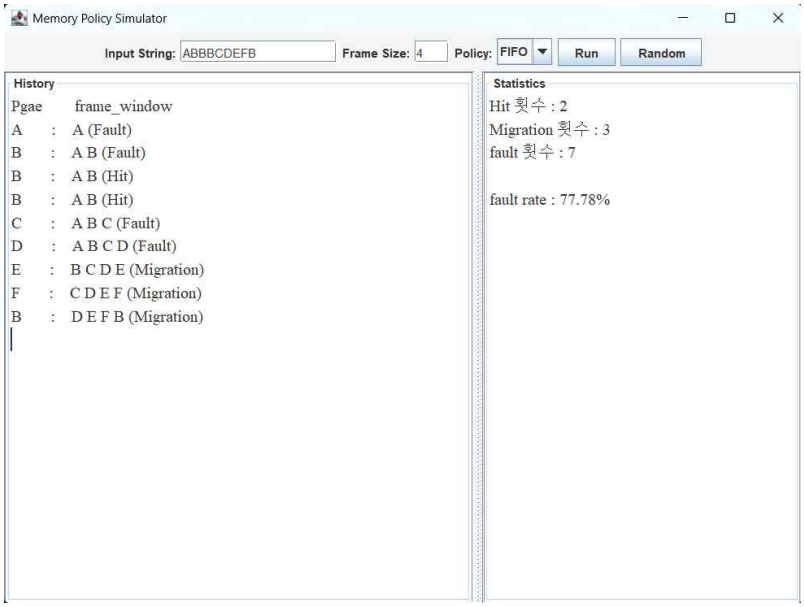
operate() 함수에서 새 페이지를 추가하는 과정이 모두 끝나면, 입력 페이지의 순서를 저장하는 pageHistory와 frame_window의 상태를 저장하는 History에 현재의 교체 결과를 백업한다. History의 경우 현재 입력된 페이지의 data, frame_window의 data들, 현재 입력된 페이지의 status를 “inputData : window_Data1 win_dowData2, win_dowData3... (status)” 포멧에 맞춰 String으로 저장한다.

Core 클래스에는 operate() 외에 List<String>을 반환하는 getInfo() 함수가 존재한다. 해당 함수는 함수가 호출되는 타이밍의 hit, fault, migration 값을 String 형태로 List에 저장하고 fault / (hit + fault)의 수식을 통해 fault rate 값을 계산한다. fault rate는 List에 저장될 시 소수점 둘째 자리에서 반올림된다.

해당 시뮬레이션 프로그램을 사용하기 위한 GUI 클래스에서는 Input Sring, Frame size, memory Policy를 입력받고 frame_window의 변화를 기록해 둔 History와 getInfo()에서 반환된 정보를 사용자에게 보여준다.

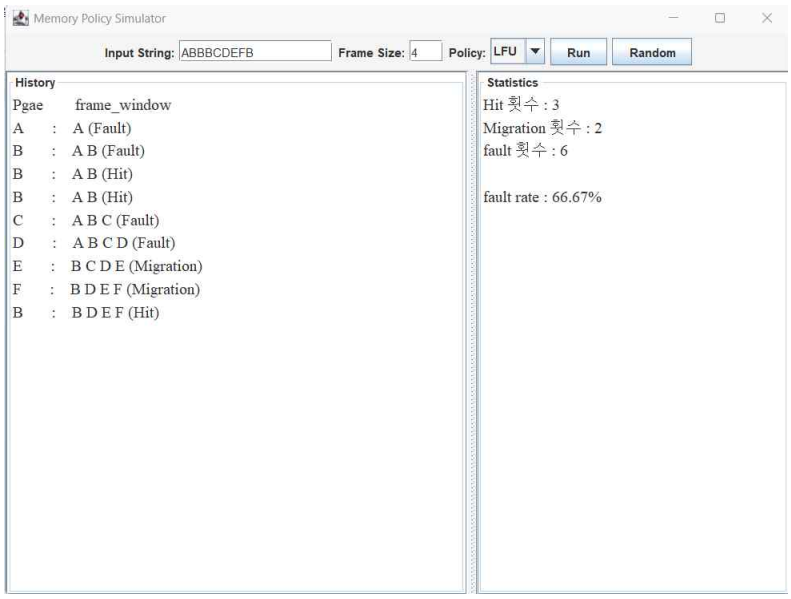
3. 주요 주제의 알고리즘 동작 사례

구현된 알고리즘들의 구체적인 동작 과정을 알아보기 위해 Frame을 4로 설정하고 Input String에 “ABBBCDEFB” 문자열을 입력한 뒤, 각 정책에 따른 결과를 확인해 보았다. 아래 사진은 선택된 정책이 FIFO인 경우의 결과를 나타낸다.



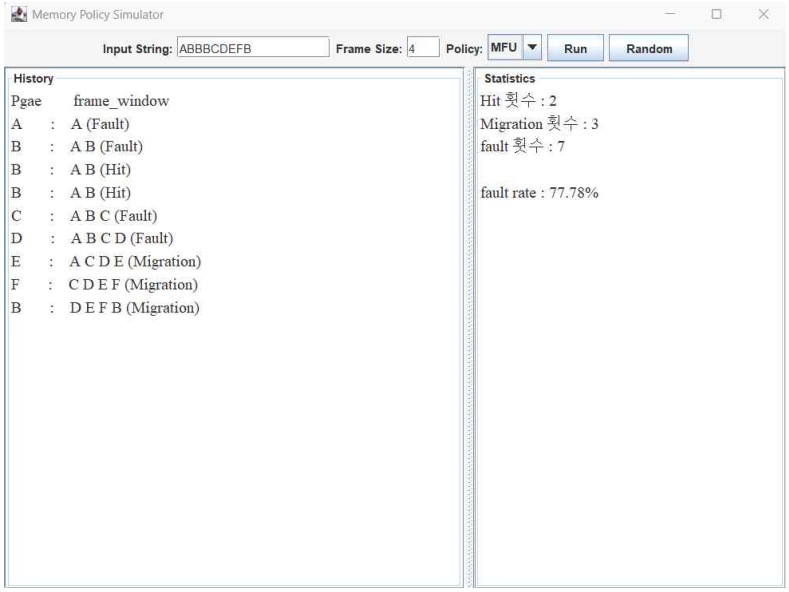
“ABBBCDEFB” 문자열은 B 페이지가 여러 번의 참조 횟수를 가지고 나머지 페이지들은 모두 한 번의 참조 횟수를 가진다. FIFO는 메모리에 들어온 순서대로 페이지를 삭제한다. 위의 실행 화면 속 E, F에서 페이지 교체가 일어날 때 참조 횟수와 관련 없이 메모리에 가장 먼저 들어왔던 A, B가 순서대로 삭제되는 것을 볼 수 있다.

아래 사진은 선택된 정책이 LFU인 경우의 결과이다.



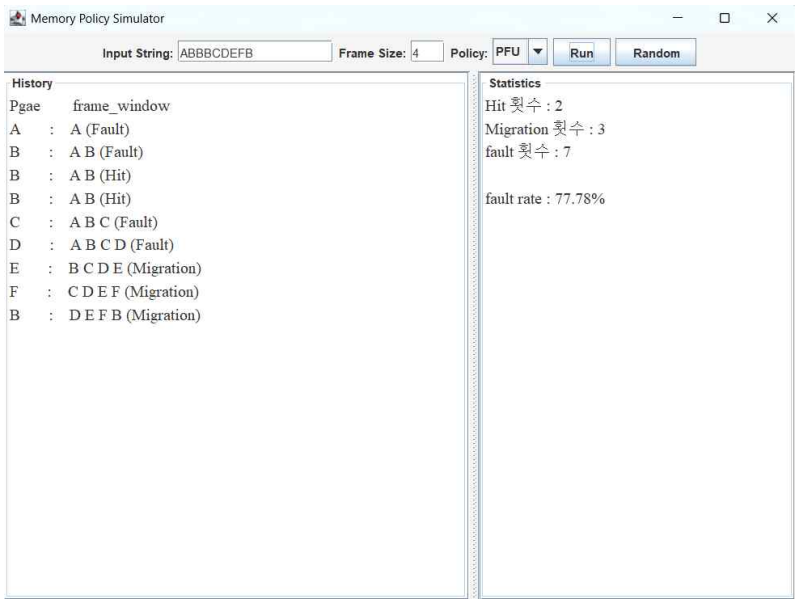
LFU는 참조 횟수가 가장 적은 페이지를 우선적으로 삭제한다. E, F 페이지에서 페이지 교체가 일어날 때 참조 횟수가 3회인 B 페이지는 메모리 위에 남고 각각 참조 횟수가 1회인 A, C 페이지가 삭제되는 것을 확인할 수 있다.

아래 사진은 선택된 정책이 MFU인 경우의 결과이다.



MFU는 참조 횟수가 가장 많은 페이지를 우선적으로 삭제하는 알고리즘이다. 위의 사진 속 실행 화면의 E 페이지에서 페이지 교체가 일어날 때 참조 횟수가 3회인 B가 가장 먼저 삭제되는 것을 확인할 수 있다.

아래 사진은 선택된 정책이 PFU인 경우의 결과이다.

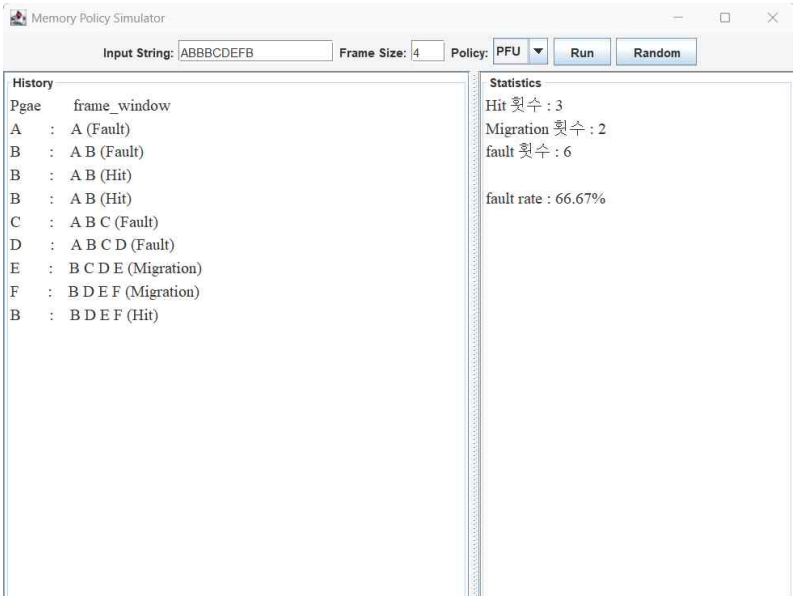


PFU는 참조 횟수가 가장 많은 페이지와 참조 횟수가 가장 적은 페이지 중 하나의 페이지가 확률적으로 삭제되는 알고리즘이다. 위의 사진 속 실행 화면의 E 페이지에서 페이지 교체가 일어날 때

참조 횟수가 가장 많은 B 페이지와 참조 횟수가 가장 적은 A 페이지 중 A 페이지가 메모리에서 삭제된 것을 확인할 수 있다.

또한 F 페이지에서 페이지 교체가 일어날 때 참조 횟수가 가장 많은 B 페이지와 참조 횟수가 가장 적은 C 페이지 중 B 페이지가 메모리에서 삭제된 것을 확인할 수 있다. 때문에 바로 다음의 B 페이지에서 페이지 교체가 발생해 총 7번의 fault가 발생하였음을 확인할 수 있다.

PFU 알고리즘은 앞서 소개한 다른 알고리즘들과 달리 확률형 알고리즘이므로 같은 정책, 같은 문자열을 두고도 메모리에 남은 페이지와 페이지 fault 확률이 매번 다르게 나타나게 된다.



지금 사진의 실행 결과는 직전 실행에서와 같은 문자열, 같은 Frame 사이즈를 사용하고 있지만 페이지 fault 확률이 77.78%에서 66.67%로 감소한 것을 확인할 수 있다. 이는 저번 실행과 달리 이번 실행에서는 페이지 E, F의 페이지 교체 과정에서 가장 높은 참조 횟수를 가진 B가 연속해서 살아남았기 때문이다.

IV. 성능 평가

1. 실험 환경

본 프로젝트의 페이지 교체 시뮬레이터는 Java를 기반으로 작성되었다. 프로그램은 크게 세 가지 주요 클래스로 구성되어 있다. 각 페이지의 구현을 담당하는 Page 클래스, 페이지 교체 정책의 핵심 코드를 담당하는 Core 클래스, 사용자 인터페이스를 제공하는 GUI 클래스가 그것이다. 모든 실험은 통합 개발 환경(IDE)에서 수행되었다.

성능 평가에서는 메모리 관리의 가장 핵심적 요소인 프로그램의 지역성에 초점을 맞춰, 임의의 프로그램의 지역성이 낮을 때부터 높을 때까지, 프로그램의 지역성이 이동하는 상황을를= 가정하여 실험을 진행한다.

이때 페이지 교제 정책의 효율성을 판단하는 가장 중요한 척도는 프로그램 실행 중 발생하는 페이지 fault의 횟수이다. 따라서 이번 성능 분석 실험의 결과는 페이지 fault 횟수와 페이지 hit 횟수의 비율을 확인할 수 있는 fault rate로 결정한다.

공정성을 보장하기 위해 모든 실험은 동일한 문자열과 동일한 frame 크기를 두고 4가지 정책을 순서대로 실행하는 방식으로 진행될 것이다. 자세한 실험 계획은 아래와 같다.

1) 프로그램의 지역성이 낮은 경우
2) 프로그램의 지역성이 보통인 경우
3) 프로그램의 지역성이 높은 경우
4) 프로그램의 지역성이 변화하는 경우

1, 2, 3번 실험의 경우 frame의 크기는 4로 고정한다. 또한 프로그램의 지역성이 낮은 경우라 해도 특정 문자열에서 좋은 결과를 보일 수 있으므로 지역성이 낮은 임의의 문자열을 5개 만들어 5번의 시뮬레이션을 진행한다. 실험을 위한 문자열은 A~Z까지 50개의 문자로 이루어진다.

4번 실험의 경우 지역성이 변화하는 것을 문자열에 표현하기 위해 하나의 문자열에 A~Z까지 100개의 문자를 사용한다. 또한 1, 2, 3번 실험과 마찬가지로 특정 문자열에서의 유리한 조건을 없애기 위해 한 정책 당 5번의 시뮬레이션을 진행한다.

실험의 결과를 투명하게 밝히기 위해 사용된 모든 문자열과 시뮬레이션 결과는 보고서 내에 빠짐없이 기록한다. 모든 결과 분석은 5번 시뮬레이션의 평균값을 이용하며, 그래프를 이용해 시각적으로 나타낸다.

2. 실험 결과 및 분석

① 프로그램의 지역성이 낮은 경우

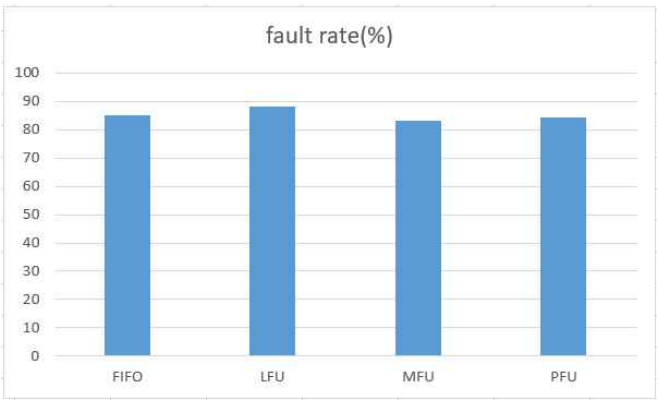
프로그램의 지역성이 낮은 경우를 실험하기 위해 준비된 문자열은 아래 5개와 같다.

회차	문자열
1	AGSNQHNXSSPVTKOJFQMTBVGYEECLVFUIXUYPBOLPQYJRACUCRZ
2	VQEMRWGDEIZYANKUOVYRXHOGCSWMTCNTUFDWRJVPKOABZXUATR
3	DWJGHUCQRKKXXVPNWQSTATWFWKBHQPDYWLLDCAHQPEDYJVAZBS
4	HVJRYTXUZXHGGSMAUCIIREEEURZMXCQZBZWVDODPWISPKFFNSC
5	GMTAAGJHWTVYZYPUBYIBRZHAYEHTYCIOOTATUMMTPRDQCKMKZZ

해당 문자열들은 GUI 클래스의 generateRandomString() 함수를 사용하여 생성되었다. 위 5개의 문자열들은 완전히 무작위의 랜덤한 값을 가지며, 특정 한 페이지가 자주 참조되거나 같은 문자들이 가까운 거리에서 참조되지 않는다. 따라서 위의 문자열들은 지역성이 매우 낮은 문자열이라는 것을 확인할 수 있다.

아래는 각 회차별 실험에서 4개의 페이지 교체 정책의 fault rate를 기록한 표이다.

회차	FIFO	LFU	MFU	PFU
1	86	94	84	84
2	98	96	98	98
3	88	90	88	88
4	78	86	72	80
5	76	74	74	72
평균	85.2	88	83.2	84.4



프로그램의 낮은 지역성으로 인해 네 가지 페이지 교체 정책의 전반에 걸쳐 80%가 넘는 높은 fault rate가 기록되었다. 문자열에서 각 문자들이 서로를 거의 참조하지 않으므로 참조 횟수가 가장 높은 페이지를 우선적으로 삭제하는 MFU 알고리즘의 성능이 가장 뛰어나고, 참조 횟수가 가장 낮은 페이지를 우선적으로 삭제하는 LFU 알고리즘이 가장 낮은 성능을 보였다.

또한 PFU 알고리즘은 설계 시 예측했던 대로 MFU와 LFU의 중간 정도 되는 84.4%의 fault rate 측정값을 가졌다.

② 프로그램의 지역성이 보통인 경우

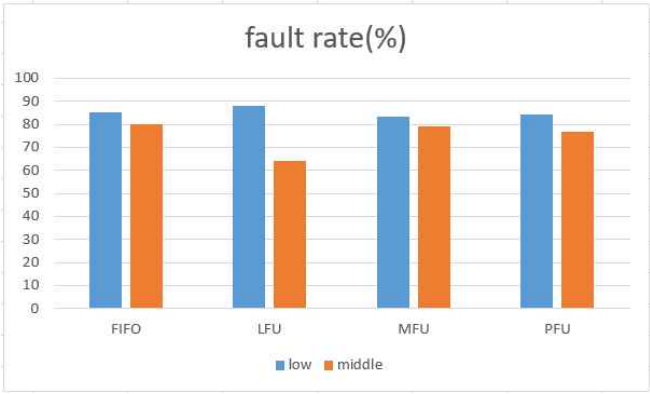
프로그램의 지역성이 보통인 경우를 실험하기 위해 준비된 문자열은 아래 5개와 같다.

회차	문자열
1	ABSNCBAXSSBCAKBCAQMBFCGABECLCAUIBCYABOCPQBJRACUCAB
2	VQAMCBCDABCYANACBVYRXBOCASCMBCATCFABRJVCKBAGZCBATC
3	CBJGABCQRAKXBVCAWQACABWAWBBHCPDYBCLDCABQPACBJVACBS
4	HVARYCBAZBHGGSBAUCBIRABEURBAXCQBAZCVBODBAISCKAFBSC
5	GCTAACJBWCVYCAPCFBIBACHABEATBCIAOBACUMCBPCBQCBMKZZ

해당 문자열들은 실험 ①의 랜덤 문자열에서 A, B, C가 일정 구간에 한 번씩 등장하도록 조정을 거친 문자열들이다. 완전히 연속되진 않지만 일정 간격으로 A, B, C가 반복되고 나머지 문자열들은 A, B, C에 비해 드물게 분포된다. 따라서 특정 문자들이 일정 간격으로 반복되면서도 군집은 존재하지 않는, 지역성이 보통 정도인 문자열이라고 볼 수 있다.

아래는 각 회차별 실험에서 4개의 페이지 교체 정책의 fault rate를 기록한 표이다.

회차	FIFO	LFU	MFU	PFU
1	80	70	80	76
2	82	52	76	78
3	72	68	76	74
4	90	80	90	82
5	76	50	74	74
평균	80	64	79.2	76.8



프로그램에서 일부 지역성이 나타남에 따라 모든 정책들의 fault rate가 실험 ①에서보다 감소하였음을 확인할 수 있다. 특히 참조 횟수가 가장 낮은 페이지를 우선적으로 삭제하는 LFU 알고리즘의 fault rate가 88%에서 64%로 가장 크게 감소한 것을 확인할 수 있다.

또한 참조 횟수를 보지 않는 FIFO와 참조 횟수가 가장 많은 페이지를 우선적으로 삭제하는 MFU에서 거의 80% 가까이 되는 높은 fault rate가 나타났다. 지역성이 낮은 경우에 비해 거의 fault rate가 줄어들지 않았음을 확인할 수 있다. PFU의 경우 LFU에 비교해서는 높지만 FIFO와 MFU보다는 약간 낮은 76.8%의 fault rate를 기록했다. 지역성이 낮은 경우에 비해 현재 실험 결과에서 fault rate가 7.6% 감소한 것을 확인할 수 있다.

③ 프로그램의 지역성이 높은 경우

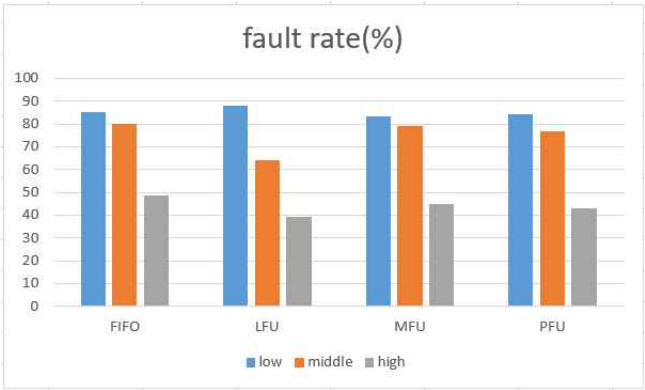
프로그램의 지역성이 높은 경우를 실험하기 위해 준비된 문자열은 아래 5개와 같다.

회차	문자열
1	AAAABBIBCCCCTDDDEEEJFFFFGVGGHHHHAAAAEBBBCCCCDDDD
2	ABCDEFAHABCDAFGHABCDAFGAABCDEAGAABCDEFAAABCDEAGHAA
3	AAAABBBBCCCCDDHHEEEEAAAABBFFCCCCDDDDEEEEGGAABBBBCC
4	ABABCDIDABABCICDABABCDCDJBABCDCDJBABCVCDABABIDCDAB
5	AABBCCDDEEDDGGEEAABBEEDDEEFFGGEEAABBCCDDEEFFGGHHAA

해당 문자열들은 A~H까지 8개의 알파벳을 반복적으로 구성한 뒤 일부 문자열에 나머지 알파벳들을 군데군데 섞어 생성되었다.

아래는 각 회차별 실험에서 4개의 페이지 교체 정책의 fault rate를 기록한 표이다.

회차	FIFO	LFU	MFU	PFU
1	35.42	35.42	35.42	31.25
2	80	70	80	80
3	32	24	30	26
4	54	32	44	42
5	42	34	36	36
평균	48.68	39.08	45.08	43.05



좌측의 fault rate 그래프에서 프로그램의 지역성이 높아짐에 따라, 이전에 비해 fault rate가 크게 감소한 것을 확인할 수 있다. 네 가지 페이지 교체 정책 중에서 LFU의 fault rate가 39.08%로 가장 낮고 다음으로 PFU 알고리즘의 fault rate가 43.05%로 낮다.

특히 이번 실험에서는 실험 ①, ②에서 관찰할 수 없었던 특이한 현상이 발생했는데, 바로 해당 실험의 1회차에서 PFU의 fault rate가 LFU의 fault rate보다도 더 낮게 나타났다는 점이다.

이전의 모든 실험에서 PFU의 값은 항상 LFU보다는 낮고, MFU보다는 높은 두 정책의 중간값을 유지했다. 그러나 실험 ③의 1회차에서는 35.42%의 값을 가지는 나머지 정책들과 달리 유일하게 31.25%라는 낮은 fault rate를 기록하였다.

이 현상의 원인은, 고정된 결과만을 내보내는 FIFO, LFU, MFU와 달리 PFU는 매 실행마다 결과가 달라지는 확률적인 특성을 가지고 있어, 일부 경우에 나머지 정책에서는 기대하기 어려운 더 나은 결과를 만들어 낸 것이라고 추측된다.

④ 프로그램의 지역성이 변화하는 경우

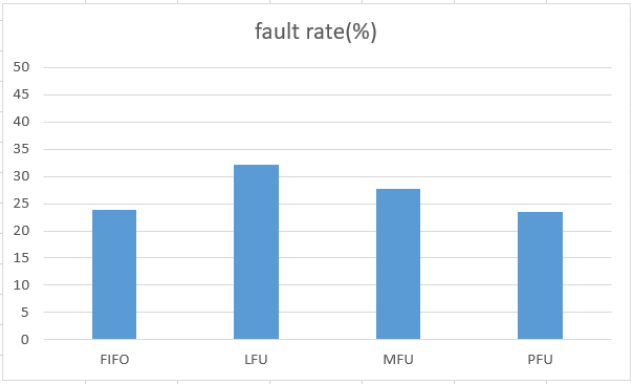
프로그램의 지역성이 변화하는 경우를 실험하기 위해 준비된 문자열은 아래 5개와 같다.

회차	문자열
1	AABBIABAJBAABKABABACCCICCCJCCGCCCCCCCCHKCCCCJCCCCCCCDDEJDFFGDDDEEFJHFFFDDEEDDFFFDIFFFDEEDDEEFF
2	ABAAIABBABBKAABABAIBABBABBPBABAAABBABBJBAABBABBABAACKCCCCCCCCCCCCCCCKCCCCCCCDDDDEEFFFIDEJEEFFF
3	AABABAPBABKABAAABBIABABABAABPCCICCCCCCCCCCCCCPCCCCCCCCCCCCCCCCCCDDEEDDFFEEIDFFHFFDKEEEDFFFDDIEED
4	AABABKABAAJBBAIABPABBABBABABABBKBABJABCCGCPCCCCCCKCCCCCJCCCCCCCCCKCCDDEEDPFFJDDEKDFFFDDFFDDDEE
5	ABABABAPBJABAAKAIBBAJAABPCCCCCCCCCKCCCCCCJCCCCCCCCPDEEDKFFDDFFJFFFIFFDDEEEPFFEJDFFFDDEEKDFFDDFF

해당 문자열들은 초반에는 A와 B를 위주로, 중반에는 C를 위주로, 후반에는 D, E, F를 위주로 흘러가도록 구성된 문자열이다. 회차에 따라 세 구간의 비율을 20:40:40, 50:25:25, 30:30:40 등으로 조절하였다. 또한 구성된 문자열에서 나머지 알파벳들을 군데군데 추가하였다.

아래는 각 회차별 실험에서 4개의 페이지 교체 정책의 fault rate를 기록한 표이다.

회차	FIFO	LFU	MFU	PFU
1	27.17	39.13	33.7	27.17
2	15.96	20.21	20.21	17.02
3	19.35	30.11	24.73	19.35
4	28.72	30.85	28.72	24.47
5	27.96	40.86	31.18	29.03
평균	23.83	32.23	27.71	23.41



프로그램의 지역성이 부분별로 높은 해당 경우, FIFO와 PFU 알고리즘의 fault rate가 각각 23.83%와 23.41%로 가장 낮게 나타났다. 또한 앞서 언급했던 대로 LFU 알고리즘은 지역성이 변화하는 환경에서 다른 교체 정책들에 비해 높은 fault rate를 기록했다. 이는 과거에 중요하게 참조되던 페이지가 지역성의 이동으로 인해 메모리 위에 불필요하게 남아 페이지 교체를 방해했기 때문으로 보인다.

4번의 실험 결과를 모두 종합해 보면, PFU는 언제나 최적의 성능을 보이지는 않지만 지역성이 낮은 경우, 보통인 경우, 높은 경우, 변화하는 경우 등 대부분의 환경에서 평균 이상의 성능을 유지하였다.

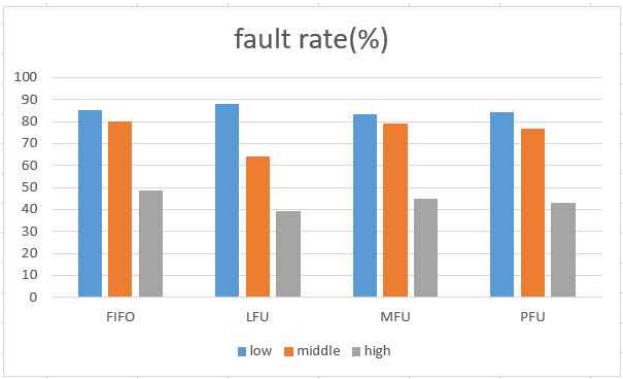
특히 프로그램의 지역성이 높아질수록 뛰어난 성능을 보였던 LFU는 지역성이 변화하는 실험 ④에서 다른 알고리즘들에 비해 성능이 크게 떨어지는 것을 확인할 수 있었다. 이를 통해 본 프로젝트에서 만들어 낸 PFU 알고리즘이 LFU와 MFU의 단점을 보완하며, 변화하는 환경에서도 안정적인

성능을 유지하고 특히 지역성의 변화가 큰 경우에 뛰어난 성능을 발휘한다는 것을 확인할 수 있다.

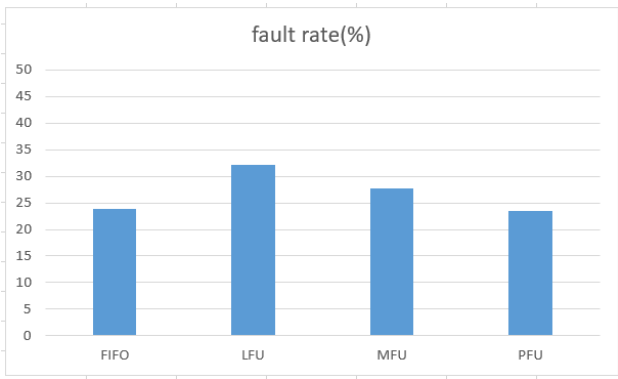
V. 결론

본 프로젝트는 프로그램의 지역성이 페이지 교체 정책에 끼치는 영향을 분석하고 기존 정책을 보완한 새로운 페이지 교체 정책을 제안하기 위해 수행되었다. 이를 위해 FIFO, LFU, MFU를 구현한 페이지 교체 시뮬레이터를 제작하고, 새로운 정책인 PFU(Probabilistic Frequently User)를 제안하였다.

제작한 시뮬레이터는 네 가지 정책의 성능을 평가하기 위해 사전에 수립한 계획에 따라 실험을 진행하였다. 각 실험은 프로그램의 지역성을 독립 변수로 설정하여 지역성이 낮은 경우, 보통인 경우, 높은 경우, 변화하는 경우로 나누어 수행되었다.



[그래프1] 지역성이 변화하지 않는 경우



[그래프2] 지역성이 변화하는 경우

[그래프1] 지역성이 변화하지 않는 경우를 살펴보면, 프로그램의 지역성이 높아질수록 모든 정책에서 fault rate가 감소하는 경향을 확인할 수 있었다. 이 중 지역성이 낮을 때에는 FIFO와 MFU가 우수한 성능을 보였으며 지역성이 높아질수록 LFU의 성능이 급격하게 향상되었다. PFU는 지역성의 높고 낮음과 관계 없이 LFU와 MFU의 중간 수준의 안정적인 성능을 유지하였다.

반면 [그래프2] 지역성이 움직이는 경우에는 [그래프1]에서 우수했던 LFU의 성능이 급격히 하락하는 모습을 보였다. 이는 과거에 자주 참조되었던 페이지가 지역성의 이동으로 인해 불필요하게 메모리에 남아, 이후의 페이지 교체를 방해했기 때문으로 추정된다. 이와 반대로 FIFO는 [그래프2]에서 우수한 성능을 보였지만 [그래프1]에서는 지역성이 높아짐에 따라 성능이 급격하게 저하되는 양상을 보였다.

이러한 실험 결과를 통해, 지역성이 낮은 상황에서는 FIFO나 MFU, 지역성이 높은 상황에서는 LFU를 사용하는 것이 유리하며, 지역성이 변화하는 상황에서는 PFU가 가장 우수한 성능을 보인다는 점을 확인할 수 있었다. 특히, PFU는 지역성이 변화할 때 가장 뛰어난 성능을 보였으며, 지역성이 고정된 환경에서도 LFU와 MFU의 중간 수준의 성능을 안정적으로 유지하였다. 따라서 프로그램

의 지역성을 예측하기 어려운 경우, PFU를 사용하는 것이 범용적인 상황에서 안정적인 성능을 기대할 수 있을 것이다.

또한 PFU 알고리즘에서는 참조 횟수가 낮은 페이지와 높은 페이지가 선택될 확률을 상황에 따라 조정할 수 있다. 본 프로젝트에서는 두 확률을 0.5대 0.5로 고정하였으나, 추후에는 실행 환경에 따라 0.2, 0.4, 0.6, 0.8 등으로 세분화하여 보다 정밀한 성능 실험을 진행할 수 있을 것이다. 이를 통해 PFU의 성능을 더욱 정확하게 측정하고, 알고리즘 개선 방향 역시 구체적으로 모색할 수 있을 것이 기대된다.

결론적으로 본 프로젝트를 통해 프로그램의 지역성이 페이지 교체 정책의 성능에 큰 영향을 미친다는 사실을 확인하였으며, PFU 알고리즘이 기존의 Count based 정책이 포용하지 못하는 환경을 보완할 수 있음을 입증하였다. 이를 통해 PFU가 새로운 페이지 교체 정책으로서 실질적인 가치를 지니고 있음을 확인할 수 있었다.

참고 자료

1) Microsoft Windows Internals
2) Apple's Open Source Darwin Kernel, https://github.com/apple/darwin-xnu.git
3) Linux Kernel Documentation