Operating System (MS)

assignment #2: Multi-threaded Word Count

모바일시스템공학과 김태경 (32211203)

목차

Freeday used 5
Freeday left 0

2024.12.03

1. 프로젝트 소개

1.1 프로젝트 개요

본 프로젝트는 가상 메모리의 Paging기법을 활용하여 멀티 프로세스를 실행하는 시뮬레이션을 구현하는 것입니다. 가상 메모리는 프로세스 별로 독립된 주소 공간을 제공하며, 주소변활을 통해 물리적으로 메모리를 관리합니다. 이 프로젝트에서는 이론적으로 가상 메모리의 개념을 구현하며, 스케줄링과 page table 관리, demand 페이징 등 운영체제의 핵심 기능을 시뮬레이션 합니다.

1.2 프로젝트 목표

1) **주소 변환 및 페이징 이해**

- 가상 주소(Virtual Address)와 물리주소(Physical Address) 간의 변환 과정 이해
- MMU, 페이지 테이블, 페이지 프레임, 페이지 폴트 등의 개념과 역할 학습

2) **효율적인 페이지 테이블 관리**

- Demand 페이징을 활용하여 효율적으로 메모리 자원을 사용
- 멀티 프로세스 환경에서 각 프로세스의 독립적인 메모리 공간을 시뮬레이션

3) **운영체제 핵심 기능 구현**

- 프로세스의 생성, 실행, 종료 시 발생하는 메모리 관리 및 페이지 테이블 업데이트를 시뮬레이션
- 페이지 폴트 처리, 페이지 테이블 업데이터, 메모리 회수 등의 과정를 구현

4) **추가 구현을 통한 심화 학습 및 분석**

- Two-level paging, 스와핑, Copy-onWirte(COW)와 같은 고급 메모리 관리 기법을 구현
- 구현한 다양한 메모리 관리 기법들의 효용성 비교 분석

2. 프로젝트 요구사항

2.1 기본 구현

1) **운영체제의 초기화**

- 부팅 시 물리 메모리를 페이지 크기로 단편화하고, 다른 페이지 목록을 관리
- 물리 메모리의 총크기는 직접 지정하고 PFN(Page Frame Number)을 0으로 시작

2) **프로세스의 관리**

- 하나의 부모 프로세스(Kernel)와 10개의 user 프로세스를 구현
- 각 프로세스는 독립된 페이지 테이블을 가지고 초기화 시 빈 상태로 설정

3) **페이지 테이블 및 메모리 접근**
- 사용자 프로세스는 시간 슬라이스 동안 10개의 가상 메모리 주소를 요청
- 운영체제는 페이지 테이블을 확인하고, 유효하지 않은 항목이 있으면 페이지 폴트를 처리
- 여유 페이지 프레임에서 새 페이지를 할당하고 페이지 테이블을 업데이트

4) **페이지 테이블의 관리**
- 가상 주소를 가상 메모리 페이지 번호와 오프셋(VM Page Offset)으로 분리
- MMU가 페이지 테이블을 통해 VA를 PA로 변환

5) **로그 기록**
- 1,000틱 동안 VA, PA, 페이지폴트 이벤트, 페이지 테이블 변경사항, 읽기/쓰기 값을 기록

6) **종료 조건**
- 시뮬레이션은 10000 틱에서 종료

2.2 추가 구현

1) **Two-Level Paging**
- page table 크기 최소화를 위해 주소를 1단계 주소, 2단계 주소, 페이지 오프셋으로 분리
- 1단계 page table이 null일 경우, 페이지 폴트를 처리해 2단계 page table을 할당
- 1단계, 2단계 페이지 테이블을 활용해 메모리 접근을 구현

2) **Swapping**
- 물리 메모리가 부족할 경우, LRU 알고리즘으로 페이지를 디스크로 스와핑
- 디스크에 저장된 페이지를 다시 메인 메모리로 불러오고, page table을 업데이트

3) **Copy-on-Write(COW)**
- 자식 프로세스가 부모 프로세스의 메모리를 공유하되, 쓰기 작업을 할 때 별도의 물리 메모리 공간을 할당
- 쓰기 작업 시 새로운 페이지를 생성하고, 부모의 데이터를 복사한 후 업데이트

3. 프로젝트에 사용된 개념

3.1 제공된 Sample Code 분석

3.1.1 file1.c 코드 분석

```c
int foo(int val) {
    if (val <= 1) return 1;
    return (foo(val-1)*val);
}
```

: foo 함수는 주어진 값 val에 대해 재귀를 사용하여 팩토리얼을 계산하는 함수로, 'val <= 1' 조건에서 종료하며 그 외의 경우에는 호출된 값과 현재값 val을 곱하여 반환합니다.

 # 과제에서의 역할 : 메모리 사용과 재귀 호출의 스택 프레임 관리 측면에서 메모리의 동작을 관찰하는데 유용합니다.

3.1.2 file2.c 코드 분석

```c
#include <stdio.h>
#include <stdlib.h>

int foo(int val);

int main(int argc, char * argv[])
{
    int res;
    int val = 3; //default
    if (argc == 2)
        val = atoi(argv[1]);
    res = foo(val);
    printf("%d-factorial: %d\n", val, res);
    return 0;
}
```

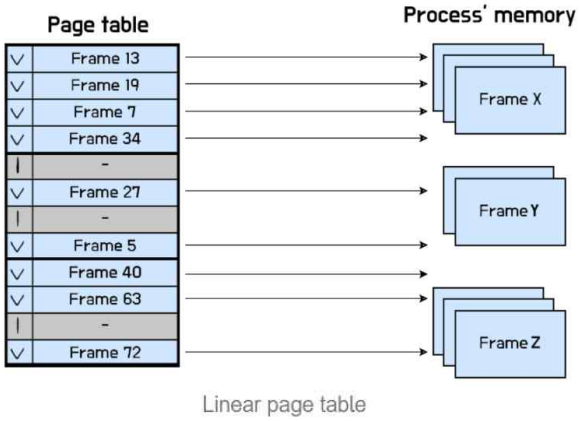
: foo 함수를 호출하고 결과를 출력하는 메인 프로그램으로, val 값은 기본값 3으로 설정되어 있으며, 명령줄 인자가 제공되면 해당 값을 읽어옵니다.

 # 과제에서의 역할 :

- **메모리 접근 패턴 분석** : res, val 로컬 변수와 명령줄 인자를 처리하며, foo 함수 호출을 통해 스택 메모리를 사용하고 재귀호출의 메모리 사용을 시뮬레이션 합니다.
- **다중 프로세스 환경 시뮬레이션** : 각각의 사용자 프로세스는 file2.c 프로그램과 같은 방식으로 독립적인 메모리 주소 공간에서 실행되며, 팩토리얼 계산은 각 프로세스가 메모리 접근 패턴(스택의 사용)을 테스트하는데 사용됩니다. 재귀 호출 시 메모리 접근 상황을 관찰하여 페이지 테이블이 제대로 관리되는지 확인할 수 있습니다.
- **IPC 메시지 전송 시뮬레이션** : 사용자가 프로그램을 실행하면서 명령줄 인자를 입력하는 과정을 IPC 메시지 전달로 간주할 수 있어, 각 프로세스가 메모리 접근 요청을 생성하는 시뮬레이션에 활용할 수 있습니다.

3.2 Paging 개념 정리

3.2.1 **페이지와 프레임**



Linear page table

페이지 (Page) : 가상 메모리를 고정된 크기의 블록으로 나눈 단위로, 페이지는 크기가 일정하며, 논리 주소 공간을 효율적으로 관리합니다. 페이지 번호(Page Number)와 오프셋(Offset)으로 구성됩니다.

프레임 (Frame) : 물리 메모리를 페이지와 동일한 크기의 블록으로 나눈 단위로, 페이지는 프레임에 로드되어 물리 메모리 내 위치를 갖습니다. 페이지와 프레임은 1:1 매핑 관계를 가집니다.

 # 과제에서의 역할

- 페이지와 프레임의 매핑 : 페이지 테이블은 각 페이지가 어느 프레임에 매핑되는지 기록하며, 추가구현의 2단계 페이지 테이블 구조를 사용하여 페이지 번호를 관리한다.
- 페이지 폴트 처리 :페이지가 메모리에 없을 경우, 보조 저장소에서 해당 페이지를 프레임에 로드한다. 페이지 폴트는 페이지 교체 알고리즘(FIFO, LRU 등)을 통해 처리된다.
- 프레임 활용 : 자유 프레임 목록(Free Frame List)은 사용 가능한 프레임을 관리하며, 메모리 부족 시 스와핑을 통해 프레임을 확보한다.

3.2.2 **페이지 테이블**

페이지 테이블(Page Table) : 가상 메모리의 페이지와 물리 메모리의 프레임을 매핑하는 데이터 구조입니다. 논리 주소의 페이지 번호(Page Number)를 물리 주소의 프레임 번호(Frame Number)로 변환합니다.

page table entry

구성요소

1) 페이지 번호(Page Number): 가상 주소의 상위 비트로 페이지를 식별
2) 프레임 번호(**Frame Number**): 해당 페이지가 로드된 물리 메모리의 프레임 번호
3) 유효 비트(**V**alid Bit): 페이지가 물리 메모리에 로드되어 있는지 나타냄
4) 보호 비트(**Prot**ection Bit): 페이지에 대한 접근 권한(읽기/쓰기/실행)을 지정
5) 수정 비트(**M**odified Bit): 페이지가 수정되었는지 표시, 페이지 교체 시 디스크에 저장 여부를 결정

 # 과제에서의 역할

- 유효 비트와 페이지 폴트 : 유효 비트가 0인 경우 페이지 폴트가 발생하며, 보조 저장소에서 해당 페이지를 메모리로 로드한다.
- 다중 프로세스 지원 : 각 프로세스는 고유한 페이지 테이블을 가지며, 프로세스 간 메모리 충돌을 방지한다.

3.2.3 **논리-물리 주소**

논리 주소(Logical Address)는 CPU가 프로그램 실행 중 생성하는 주소로, 사용자 관점에서 보이는 가상주소 공간입니다. 프로세스의 시작점부터 특정 오프셋(offset)으로 계산되며, 물리 메모리와 독립적입니다.

물리 주소(Physical Address)는 메모리 하드웨어가 직접 접근하는 주소입니다. 물리 주소는 논리 주소가 MMU를 통해 변환되어 생성됩니다. 물리 메모리는 한정된 공간이므로, 논리 주소와 구분하여 다뤄야 합니다.

 # 과제에서의 역할

- 논리 주소 : 가상 메모리 시스템에서 논리 주소는 페이지 번호와 페이지 오프셋으로 나뉘며, 각 프로세스가 가지는 독립적인 논리 주소 공간으로 메모리를 보호를 구현하고, 다중 프로세스에서 충돌을 방지한다.
- 물리 주소 : 논리 주소를 TLB와 페이지 테이블을 활용하여 물리 주소로 변하여 실제 메모리에 데이터가 저장된다. 변환된 물리 주소는 읽기/쓰기 작업에 사용된다.

3.2.4 **주소 바인딩**

컴파일 시점 바인딩 : 소스 코드가 기계어로 컴파일될 때, 논리 주소가 물리 주소로 변환됩니다. 변환된 프로그램은 항상 동일한 메모리 위치에서 실행됩니다.

로드 시점 바인딩 : 실행 파일이 메모리에 로드될 때 주소가 결정됩니다. 물리 메모리 주소는 실행 환경에 따라 달라질 수 있습니다.

실행 시점 바인딩 : 프로그램이 실행 중일 때 논리 주소를 물리 주소로 변환합니다. 이는 동적 로드(dynamic loading), 페이징(paging), 스와핑(swapping)과 같은 기술을 지원합니다.

 # 과제에서의 역할

- 실행 시점 바인딩 : 페이징 시스템에서 논리 주소는 실행 중 물리 주소로 변환하며, 주소 바인딩을 통해 물리 메모리의 효율적인 사용과 가상 주소의 관리를 지원한다.
- 페이징과 연계 : 페이지 테이블은 논리주소를 물리주소로 매핑하는 역할을 하며, MMU는 실행시점 주소 바인딩을 구현하여 페이지 교체와 스와핑을 통해 바인딩을 동적으로 처리한다.

3.2.5 **메모리 관리 유닛 (MMU)**

MMU는 페이지 테이블을 참조하여 가상메모리에서 요청된 논리 주소를 해당 물리 주소로 변환하는 역할을 수행합니다. 변환 속도를 높이기 위해 TLB를 활용하여 자주 사용되는 주소 변환 정보를 cashing합니다. 보호 비트와 유효 비트 등 다양한 메커니즘을 통해 주소 보호 기능을 제공하며, 잘못된 메모리 접근을 방지하여 시스템의 안정성과 보안을 보장합니다.

작동과정

1) CPU가 논리 주소를 생성
2) MMU는 페이지 번호와 오프셋으로 논리 주소를 분리
3) 페이지 번호를 페이지 테이블에서 참조하여 프레임 번호(물리 주소)를 얻음
4) 물리주소는 메모리의 특정 위치지정, 오프셋 값을 더해 최종 메모리 위치를 계산

 # 과제에서의 역할

- 주소 변환 : CPU가 생성한 가상 주소를 페이지 테이블과 TLB를 통해 물리 주소로 변환한다. 페이지 폴트가 발생하면, MMU는 해당 페이지를 스왑 공간에서 로드한다.
- 페이징 시스템 관리 : MMU는 페이징의 핵심으로, 논리 주소와 물리 주소 간 매핑을 수행한다. 여러 프로세스의 가상 주소 공간을 물리 메모리로 효율적으로 변환하고 관리한다.

3.2.6 외부 단편화 (External Fragmentation)

외부 단편화(External Fragmentation) : 가변 크기의 메모리 할당으로 인해 사용되지 않는 작은 공간이 물리 메모리에 남는 문제를 의미합니다. 페이징 기법에서는 메모리를 고정 크기의 블록(페이지와 프레임)으로 나누어 외부 단편화를 방지합니다.

페이징의 해결책 : 논리 주소와 물리 주소를 동일 크기의 페이지와 프레임으로 나누어, 연속적인 물리적 메모리를 요구하지 않도록 설계합니다. 물리 메모리의 비연속적인 프레임도 논리 주소에 매핑할 수 있으므로 외부 단편화가 제거됩니다.

 # 과제에서의 역할

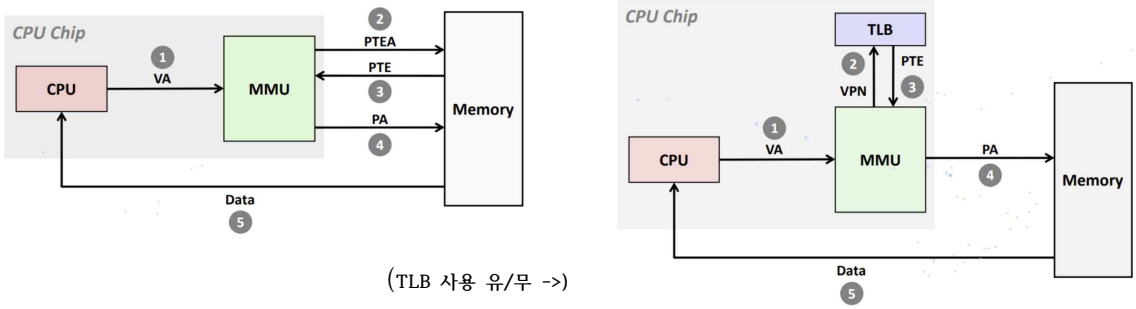
- 메모리 효율 향상 : 과제에서 페이지와 프레임이 동일한 크기로 설정되어 외부 단편화 문제를 해결하고, 페이지 교체 및 스와핑이 비연속적인 메모리 할당을 가능하게 한다.
- 메모리 활용 : 물리 메모리의 비어있는 프레임을 효율적으로 사용하여 공간낭비를 줄인다.

3.2.7 TLB

TLB(Translation Look-Aside Buffer) : 가상 주소를 물리 주소로 변환하는 최근 데이터를 저장하는 고속 캐시입니다. 페이지 테이블을 직접 참조하지 않고, TLB에서 주소 변환 정보를 빠르게 검색하여 성능을 향상시킵니다.

작동 과정

1) CPU가 논리주소를 생성, MMU는 TLB에서 해당 페이지 번호의 변환 정보를 검색
2) TLB hit : 변환 정보가 존재하면, 물리 주소를 즉시 반환
3) TLB miss : 변환 정보가 없으면 페이지 테이블을 참조하여 변환 후, TLB에 저장



(TLB 사용 유/무 ->)

 # 과제에서의 역할

- 주소 변환 속도 향상 : TLB는 페이지 테이블 조회를 줄여, 주소 변환 속도를 높이는 데 사용되며, TLB 히트율이 프로그램 성능에 큰 영향을 미친다.
- 페이지 폴트 감소 : 자주 참조되는 변환 정보를 캐싱하여 페이지 폴트 횟수를 줄인다.
- LRU 교체 구현 : 과제에서 LRU 정책을 통해 TLB 교체를 구현한다.

3.2.8 **페이지 폴트**

페이지 폴트(Page Fault) : CPU가 요청한 페이지가 메모리에 없는 경우 발생하는 인터럽트로, 페이지 테이블의 유효 비트가 0일 때 발생합니다.

페이지 폴트 처리 과정

1) CPU는 요청한 페이지의 유효 비트를 확인
2) 유효 비트가 0이면 페이지 폴트가 발생, CPU는 인터럽트를 발생시킴
3) 운영체제는 보조 저장소(디스크)에서 해당 페이지를 로드할 자유 프레임을 색출
4) 페이지 교체 알고리즘을 사용해 기존 페이지를 교체해야 하는 경우, 보조 저장소에 저장하고 새로운 페이지를 메모리에 로드
5) 페이지 테이블을 업데이트하고, CPU가 중단된 명령어를 재실행

 # 과제에서의 역할

- 스와핑 및 페이지 로드 : 페이지 폴트는 스와핑을 통해 보조 저장소에서 페이지를 메모리로 로드하도록 트리거한다. 과제의 페이지 교체 알고리즘(FIFO, LRU, SCA 등)이 페이지 폴트 처리에 사용된다.
- 성능 분석 : 페이지 폴트 횟수는 메모리 접근 패턴의 효율성을 평가하는 주요 지표로, 과제의 결과 분석에 활용된다.

3.2.9 **페이지 교체**

페이지 교체 (Page Replacement) : 물리 메모리의 공간이 부족해 새로운 페이지를 로드해야 할 때, 기존에 메모리에 있던 페이지를 보조 저장소로 내보내고 새로운 페이지로 교체하는 작업을 말하며, 페이징 시스템의 성능에 중요한 영향을 미치는 요소입니다.

페이지 교체 조건

1) 페이지 폴트가 발생했을 때 메모리에 빈 프레임이 없을 경우.
2) 보조 저장소로 교체할 페이지를 선택해야 하는 상황.

 # 과제에서의 역할

- 페이지 폴트 처리 ; 페이지 폴트가 발생 했을 때 트리거 되며, 보조 저장소에서 페이지를 가져오기 전에 교체 대상 페이지를 선택한다.
- 교체 알고리즘 구현 : 과제에서는 LRU 알고리즘을 구현하여 성능을 비교하는 것이 목표이며, 그 외 FIFO, LRU, LFU/MFU, SCA, ESCA 알고리즘 등을 구현하여 성능을 비교할 수 있다.
- 스왑 공간 관리 : 페이지 교체는 보조 저장소와 메모리 간의 스와핑 작업을 포함한다. 자유 프레임 목록을 활용하여 스왑 공간의 최적 사용을 보장한다.
- 성능 분석 : 구현된 프로그램의 페이지 폴트 횟수와 교체 비용을 측정하여 알고리즘의 효율성을 평가한다. 최적의 페이지 교체 알고리즘을 선택하기 위한 기준 데이터를 제공한다.

3.3 기본구현 개념 정리

3.3.1 메모리 초기화 및 설정

자유 프레임 목록 (Free Frame List) : 물리 메모리에서 사용 가능한 프레임을 관리하는 데이터 구조입니다. 각 프레임의 상태(사용 중 또는 비어 있음)를 추적하며, 페이지 폴트 발생 시 새로운 페이지를 로드할 공간을 제공합니다.

페이지 테이블 초기화 : 각 프로세스는 고유한 페이지 테이블을 가지고 있으며, 초기화 과정에서 이를 설정합니다. 페이지 테이블은 모든 페이지 번호와 프레임 번호의 매핑 정보를 저장합니다.

 # 과제에서의 역할

- 자유 프레임 목록 : 메모리 부족 시 스왑 공간으로 내보낼 페이지를 결정하거나, 새로운 페이지를 로드할 때 사용된다.
- 페이지 테이블 초기화 : 초기화 단계에서 각 프로세스의 페이지 테이블을 설정하여 논리 주소 -> 물리 주소 매핑을 준비한다. 페이지 테이블은 실행 중 메모리 접근 및 페이지 폴트를 처리하는 데 핵심적인 역할을 한다.

3.3.2 TLB와 페이지 테이블 관리

TLB 캐싱 (Translation Look-Aside Buffer) : 자주 참조되는 페이지 번호와 프레임 번호 매핑 정보를 저장하는 고속 캐시입니다. 페이지 테이블을 직접 참조하지 않고 TLB에서 정보를 검색해 메모리 접근 속도를 향상시킵니다.

페이지 테이블 탐색 : TLB에서 변환 정보를 찾을 수 없는 경우, 페이지 테이블을 탐색하여 물리 주소를 찾습니다. 2단계 페이지 테이블 구조를 사용하여 큰 논리 주소 공간을 효율적으로 관리합니다.

 # 과제에서의 역할

- TLB 활용 : 주소 변환 속도를 높이고 페이지 폴트를 줄이는 데 중요한 역할을 한다.
- TLB 교체 정책(LRU 등)이 구현되어 자주 사용되는 변환 정보를 유지한다.
- 페이지 테이블 탐색: TLB 미스 발생 시 페이지 테이블 탐색을 통해 변환정보를 확인하고, 페이지 폴트가 발생하면 보조 저장소에서 페이지를 가져와 페이지 테이블을 업데이트한다.

3.3.3 메모리 관리 장치 (MMU 방식 : Memory Manangement Unit)

가상 -> 물리 주소 변환 : CPU가 생성한 논리 주소는 MMU를 통해 페이지 번호와 오프셋으로 분리됩니다. 페이지 번호는 페이지 테이블에서 프레임 번호로 변환되고, 오프셋은 프레임 주소에 추가되어 물리 주소가 생성됩니다.

페이지 테이블 검색 : MMU는 TLB와 페이지 테이블을 함께 사용하여 논리 주소를 물리 주소로 변환합니다. 페이지 폴트 발생 시, 페이지 테이블 정보를 수정하여 새로운 매핑을 추가합니다.

 # 과제에서의 역할

- 주소 변환 : MMU는 논리 주소를 물리 주소로 변환하는 핵심 역할을 하며, TLB, 페이지 테이블, 페이지 교체 알고리즘과 긴밀히 연계되어 작동한다.
- 메모리 보호 : 메모리 접근 시 불법적인 주소 참조를 방지하고, 다중 프로세스 환경에서 메모리 충돌을 차단한다.

3.3.4 **메모리 접근 패턴**

랜덤 참조 패턴 : 임의의 논리 주소를 참조하는 메모리 접근 방식으로, 데이터의 분산된 배치와 비규칙적인 접근 특성을 시뮬레이션합니다.

실제 메모리 사용 패턴 : 프로그램 실행에서 발생하는 일반적인 메모리 접근 흐름을 모델링합니다. 텍스트 섹션(코드), 데이터 섹션(정적/동적 데이터), 스택 섹션과 같은 구조적 메모리 접근을 포함합니다.

 # 과제에서의 역할

- 랜덤 참조 패턴 : 랜덤한 메모리 접근은 다양한 페이지 교체 알고리즘의 성능을 테스트하기 위해 사용되며, TLB 캐시 효율성과 페이지 폴트 비율을 분석할 수 있다.
- 실제 메모리 사용 패턴 : 실제 실행 환경에서 발생할 수 있는 접근 패턴을 시뮬레이션 한다. 과제 결과분석에서 더 현실적인 데이터를 제공한다.

3.3.5 **메시지 기반 프로세스 통신**

IPC 응용 (Inter-Process Communication) : 프로세스 간 데이터를 주고받는 방법입니다. 부모와 자식 프로세스 간 통신에서 주로 메시지 큐를 사용합니다.

부모-자식 프로세스 간 주소 전송 : 자식 프로세스는 참조할 논리 주소를 생성하고, 이를 메시지 큐를 통해 부모 프로세스에 전달합니다.

메시지 수신 및 처리 : 부모 프로세스는 메시지 큐에서 수신한 논리 주소를 처리하고, 메모리 접근 작업을 수행합니다. 처리 결과(물리 주소 등)는 메시지 큐를 통해 다시 자식 프로세스에 전달됩니다.

 # 과제에서의 역할

- 부모-자식 프로세스 간 협력 : 자식 프로세스는 주소 요청을 담당하고, 부모 프로세스는 주소 변환과 메모리 접근을 처리한다. 메시지 기반 통신은 이 작업을 효율적으로 수행할 수 있도록 지원한다.
- 병렬 처리 시뮬레이션 : 다중 프로세스 환경에서 메시지 큐를 활용해 병렬 처리 작업을 시뮬레이션한다.

3.4 다단계 페이징 (Mulit-Level Paging : 추가구현 1)

3.4.1 2단계 페이지 테이블 구조

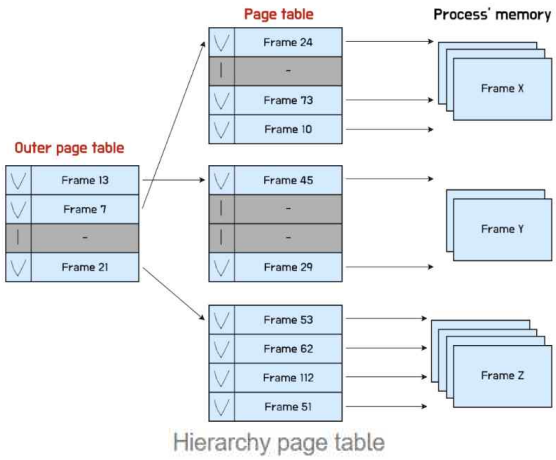
페이지 크기 : 가상 주소공간과 물리 메모리를 동일한 크기의 블록으로 나뉩니다. 페이지 크기가 클수록 페이지 테이블의 크기는 작아지지만, 내부 단편화가 발생할 수 있습니다.

페이지 테이블의 다단계 관리 : 큰 가상 주소 공간(예: 32비트, 64비트)에서는 단일 페이지 테이블로 모든 매핑 정보를 관리하기 어렵습니다. 다단계 페이지 테이블은 페이지 테이블 자체를 다시 페이지로 나누어 관리합니다. 1단계 테이블은 2단계 테이블을 참조하며, 2단계 테이블은 최종적으로 물리 주소를 매핑합니다.

 # 과제에서의 역할

- 효율적인 메모리 사용 : 2단계 구조를 사용하면 필요한 페이지 테이블만 메모리에 로드할 수 있으므로 메모리 낭비를 줄일 수 있다.
- 큰 주소 공간 관리 : 과제에서 32비트 이상의 주소 공간을 처리할 때, 다단계 페이지 테이블은 논리 주소를 효과적으로 매핑한다.

3.4.2 계층적 페이징



1단계 매핑 : 논리 주소의 상위 비트는 1단계 페이지 테이블 인덱스로 사용됩니다. 1단계 테이블은 2단계 테이블의 시작 주소를 가리킵니다.

2단계 매핑 : 논리 주소의 중간 비트는 2단계 페이지 테이블 인덱스로 사용됩니다. 2단계 테이블은 최종적으로 페이지가 매핑된 물리 프레임 번호를 반환합니다.

주소 변환 과정 : 논리 주소는 세 부분(1단계 인덱스, 2단계 인덱스, 오프셋)으로 나뉩니다. MMU는 1단계와 2단계 테이블을 순차적으로 참조하여 물리 주소를 생성합니다.

#과제에서의 역할

- 가상 주소공간 분할 : 가상 주소공간을 효율적으로 분할로, 페이지테이블의 크기를 줄인다.
- 메모리 접근 최적화 : 필요한 페이지 테이블만 메모리에 로드하므로 메모리 사용량이 감소하고, 스왑 공간 사용을 줄일 수 있다.

3.4.3 메모리 압축

논리 주소 분할 : 논리 주소는 페이지 번호와 페이지 오프셋으로 구성됩니다. 페이지 번호는 1단계와 2단계 페이지 테이블 인덱스로 나뉩니다.

페이지 테이블 항목 검색 및 상태 업데이트 : 논리 주소의 페이지 번호를 기반으로 페이지 테이블을 검색합니다. 페이지가 물리 메모리에 없으면 보조 저장소에서 로드하고 페이지 테이블을 업데이트합니다.

 # 과제에서의 역할

- 메모리 효율성향상 : 논리주소를 분할해 불필요한 페이지 테이블의 메모리 점유 방지한다.
- 동적 업데이트 : 페이지 테이블 항목은 실행 중에 동적으로 업데이트되어 가상 메모리 시스템의 유연성을 제공한다.

3.4.4 스와핑과 연계된 다단계 페이징

1단계 페이지 테이블 검색 실패 처리 : 1단계 페이지 테이블에서 유효하지 않은 항목을 참조하면, 페이지 폴트가 발생합니다. 새 2단계 페이지 테이블을 생성하고, 1단계 테이블을 업데이트합니다.

2단계 테이블 교체 및 업데이트 : 2단계 테이블에서 프레임 번호를 찾지 못한 경우, 보조 저장소에서 해당 페이지를 로드합니다. 물리 메모리에 빈 프레임이 없으면 페이지 교체 알고리즘을 통해 기존 프레임을 교체하고 페이지 테이블을 업데이트합니다.
과제에서의 역할

 # 페이지 폴트 처리

- 다단계 페이징은 페이지 폴트 발생 시, 필요한 단계의 페이지 테이블만 업데이트하여 메모리 사용을 최적화한다.
- 스왑 공간 활용 : 스왑 공간에서 데이터를 가져와 메모리에 로드하고, 페이지 테이블을 갱신하여 논리 주소와 물리 주소 간 매핑을 유지한다.

3.5 Swaping & COW (추가구현 2)

3.5.1 스와핑

스왑 공간 (Swap Space) : 보조 저장소(디스크)의 일부를 스왑 공간으로 사용합니다. 물리 메모리가 부족한 경우, 사용되지 않는 페이지를 스왑 공간으로 이동시켜 물리 메모리를 확보합니다.

페이지 아웃(Page Out) - 페이지 인(Page In) : 페이지 폴트 발생 시 보조 저장소에서 필요한 페이지를 다시 로드합니다.

- 페이지 아웃: 메모리에서 선택된 페이지를 보조 저장소(스왑 공간)로 내보내는 작업.
- 페이지 인: 보조 저장소에 저장된 페이지를 메모리로 가져오는 작업.

 # 과제에서의 역할

- 메모리 부족 상황 처리 : 스와핑은 물리 메모리가 부족한 경우, 프로세스 실행을 지속하기 위한 핵심적인 메커니즘이다. 스왑 공간을 활용해 페이지 교체를 수행하며, 메모리 부족 상황에서도 안정적인 실행을 보장한다.
- 페이지 관리 : 페이지 아웃/인 작업을 통해 가상 메모리 시스템이 다중 프로세스를 효율적으로 관리할 수 있도록 지원한다.

3.5.2 **복사-쓰기 (Copy-on-Write)**

fork와 페이지 공유 : fork: 부모 프로세스를 복제하여 자식 프로세스를 생성합니다. 초기에는 부모와 자식 프로세스가 동일한 페이지를 공유합니다. 이 과정에서 물리 메모리를 효율적으로 사용하며, 페이지 복사를 지연시킵니다.

쓰기 작업 시 페이지 복사 : 부모 또는 자식 프로세스가 페이지를 수정하려고 하면, Copy-on-Write(COW) 메커니즘이 트리거됩니다. 수정 작업 시 해당 페이지의 복사본을 생성하고, 부모와 자식 프로세스가 서로 독립적인 메모리 공간을 가지게 됩니다.

 # 과제에서의 역할

- 메모리 효율성 : 초기 단계에서 부모와 자식 프로세스가 페이지를 공유하여 메모리 사용을 줄인다. 쓰기 작업 시 필요한 페이지만 복사하여 불필요한 메모리 복사를 방지한다.
- 다중 프로세스 지원 : 복사-쓰기 메커니즘을 통해 다중 프로세스가 독립적인 메모리 공간을 가지면서도 메모리 사용량을 최적화한다.

3.5.3 **스와핑과 페이지 교체의 통합**

스와핑 중 페이지 교체 알고리즘 작동 : 스왑 공간으로 페이지를 내보내기 전에 페이지 교체 알고리즘(FIFO, LRU 등)을 사용해 교체할 페이지를 선택합니다. 선택된 페이지는 보조 저장소로 이동되며, 새로운 페이지가 물리 메모리에 로드됩니다.

메모리 부족 시 스왑 공간 활용 : 물리 메모리가 부족할 경우, 스왑 공간을 활용해 필요한 페이지를 저장하거나 로드하며, 물리 메모리 초과 상황에서도 프로세스 실행이 가능합니다.

 # 과제에서의 역할

- 알고리즘 통합 : 과제에서는 FIFO, LRU, ESCA와 같은 알고리즘을 사용해 스와핑 시 최적의 페이지 교체를 수행한다.
- 효율적인 메모리 활용 : 스왑 공간을 활용하여 물리 메모리와 보조 저장소 간 데이터 이동을 효율적으로 관리한다.

3.5.4 **스와핑의 최적화**

자유 프레임 검색 및 관리 : 물리 메모리에서 비어있는 프레임을 검색하여 새 페이지를 저장할 공간을 제공하며, 자유프레임 목록을 유지하며, 메모리 할당을 효율적으로 관리합니다.

디스크 I/O 최소화를 위한 ESCA 사용 : ESCA은 참조 비트와 수정 비트를 함께 고려하여 교체할 페이지를 선택합니다. (0, 0) 상태의 페이지(최근 사용되지 않았고 수정되지 않은 페이지)를 우선적으로 교체하여 디스크 I/O를 최소화합니다.

 # 과제에서의 역할

- 프레임 관리 : 과제에서는 자유 프레임 목록을 사용해 필요한 프레임을 빠르게 할당하고, 스와핑 작업을 수행한다.
- 디스크 접근 최적화 : ESCA 알고리즘을 통해 디스크 I/O 작업을 줄여 스왑 작업의 성능을 향상시킨다.

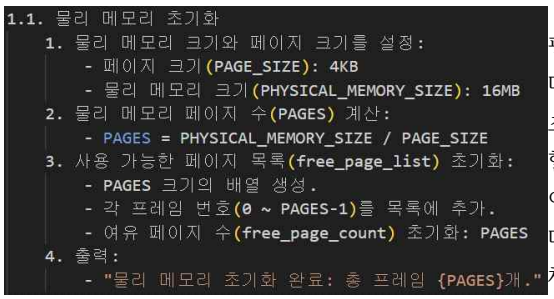
4. 프로그램 구현

* 4.1 Pseudo-Code 분석
* 4.2 Basic Code 분석
* 4.3 Addtional Code 분석(Two-level Paging)
* 4.4 Addtional Code LRU 분석(Swapping & COW)
* 4.5 Addtional Code FIFO 분석(Swapping & COW)

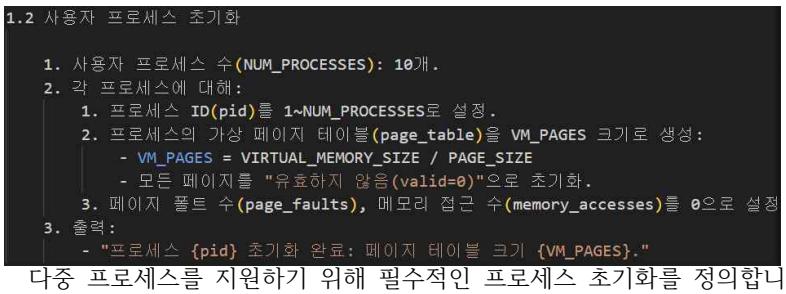
4.1 Pseudo-Code 분석

4.1.1 Basic Code

1) 운영체제 초기화



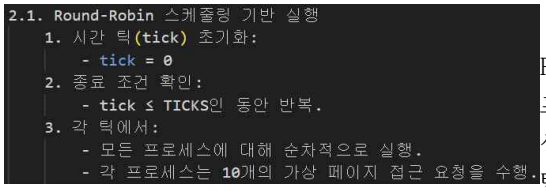
운영체제가 메모리를 효율적으로 관리하기 위한 필수적인 준비 작업을 정의합니다. 페이지 단위로 메모리를 나누고, 사용 가능한 페이지 목록을 초기화함으로써 운영체제는 동적으로 메모리를 할당하고 해제할 수 있는 기반을 마련합니다. 또한, 이런 구조를 통해 가상 메모리와 물리 메모리 간의 매핑 작업이 가능해지며, 메모리 관리를 단순하고 체계적으로 수행할 수 있습니다.



다중 프로세스를 지원하기 위해 필수적인 프로세스 초기화를 정의합니다. 각 프로세스는 독립적인 메모리 공간(가상 주소 공간)을 가지며, 이를 관리하기 위해 가상 페이지 테이블이 필요합니다.

각 프로세스의 가상 페이지 테이블을 초기화하고, 모든 항목을 "유효하지 않음(valid=0)" 상태로 설정함으로써 프로세스는 필요한 경우에만 동적으로 페이지를 할당받을 수 있습니다. 이는 메모리 자원의 낭비를 방지하고, 필요한 메모리만 효율적으로 사용하는 기반을 제공합니다.

2) 사용자 프로세스 실행



다중 프로세스를 효율적으로 실행하기 위한 Round-Robin 스케줄링의 핵심 절차를 정의합니다. 각 프로세스가 순차적으로 동일한 CPU 시간을 할당받아 실행되므로, 공정한 자원 분배가 이루어집니다. 시간 틱(tick)을 기준으로 종료 조건을 확인하고, 각 틱마다 모든 프로세스가 메모리 접근을 수행하도록 설계함으로써 시스템의 작업 부하를 시뮬레이션합니다.

```
2.2 메모리 접근 처리
    1. 프로세스는 가상 페이지 번호(page_number)를 0 ~ VM_PAGES-1에서 랜덤하게 선택.
    2. 페이지 테이블 확인:
        - 페이지가 유효(valid=1)한 경우:
            1. 접근 로그 기록:
                - "Tick: {tick}, 프로세스 {pid}, 페이지 {page_number} 접근."
        - 페이지가 유효하지 않은 경우:
            1. 페이지 폴트 발생.
            2. 페이지 폴트 처리 함수 호출.
```

가상 메모리를 효과적으로 관리하기 위해 메모리 접근 과정을 시뮬레이션하는 절차를 정의합니다. 프로세스가 가상 메모리에서 특정 페이지에 접근할 때, 해당 페이지가 유효한지(valid) 확인하여 처리 방식을 결정합니다.

[1] 페이지가 유효한 경우, 접근 로그를 기록하여 메모리 접근의 정상적인 동작을 확인합니다. 이는 운영체제의 메모리 관리 성능을 측정하는 중요한 지표로 활용됩니다.

[2] 페이지가 유효하지 않은 경우, 페이지 폴트가 발생하며, 이를 처리하기 위해 별도의 페이지 폴트 처리 함수가 호출됩니다. 이 과정은 가상 메모리의 핵심 기능인 동적 페이지 매핑과 메모리 관리의 효율성을 평가하기 위해 설계되었습니다.

3) **페이지 폴트 처리**

```
3.1 페이지 폴트 발생
    1. 페이지 폴트 처리 함수 호출:
        - 현재 프로세스 ID(pid)와 페이지 번호(page_number)를 인자로 전달.
        - "페이지 폴트 발생: 프로세스 {pid}, 페이지 {page_number}."
```

가상 메모리 접근 중 페이지 폴트가 발생했을 때 이를 처리하기 위한 절차를 정의합니다. 페이지 폴트는 가상 메모리 시스템에서 자주 발생하는 이벤트로, 메모리 관리의 중요한 부분을 차지합니다.

- 페이지 폴트 처리 함수 호출 : 페이지 폴트가 발생하면 현재 프로세스 ID(pid)와 접근하려는 페이지 번호(page_number)를 인자로 전달하여 처리를 위임합니다.

- 페이지 폴트 발생 로그 기록 : 페이지 폴트는 시스템 성능과 관련된 중요한 지표로, 이를 기록함으로써 메모리 관리 효율성을 분석할 수 있습니다.

```
3.2 페이지 할당
    1. 여유 페이지 확인:
        - 여유 페이지 목록(free_page_list)에서 첫 번째 페이지를 선택.
        - 여유 페이지 수(free_page_count)를 1 감소.
        - 선택된 페이지의 프레임 번호(frame_number)를 반환.
    2. 여유 페이지가 없는 경우:
        - "오류: 메모리 부족. 페이지 할당 실패." 출력.
        - 프로그램 종료.
```

메모리 관리에서 여유 페이지를 할당하는 과정을 정의합니다. 운영체제는 free_page_list를 활용하여 사용 가능한 페이지를 추적하며, 새로운 요청이 들어왔을 때 리스트에서 페이지를 하나씩 할당합니다.

```
3.3 페이지 테이블 업데이트
    1. 페이지 폴트 처리 완료 후:
        - 페이지 테이블의 해당 항목을 업데이트:
            - frame_number: 할당된 프레임 번호.
            - valid: 1 (유효 상태).
        - 페이지 폴트 수(page_faults)를 1 증가.
        - 페이지 폴트 처리 로그 기록:
            - "Tick: {tick}, 프로세스 {pid},
    페이지 {page_number} -> 프레임 {frame_number}."
```

페이지 폴트 발생 이후 페이지 테이블을 업데이트하는 절차를 정의합니다. 페이지 폴트가 처리된 후, 새롭게 할당된 프레임 정보를 페이지 테이블에 기록합니다.

4) **시뮬레이션 요약 및 종료**

```
4.1 1000 tick마다 시뮬레이션 요약
    1. 현재 tick 정보 출력:
        - "=== Tick {tick} 시뮬레이션 요약 ==="
    2. 각 프로세스의 상태 출력:
        - "프로세스 {pid}: 페이지 폴트: {page_faults},
        메모리 접근: {memory_accesses}"
    3. 전체 페이지 폴트 수 출력:
        - "전체 페이지 폴트 수: {page_fault_count}"
```

메모리 관리에서 여유 페이지를 할당하는 과정을 정의합니다. 운영체제는 free_page_list를 활용하여 사용 가능한 페이지를 추적하며, 새로운 요청이 들어왔을 때 리스트에서 페이지를 하나씩 할당합니다.

```
4.2 종료 후 요약 출력
    1. 최종 시뮬레이션 요약 출력:
        - "시뮬레이션 완료."
        - "전체 페이지 폴트 수: {page_fault_count}"
```

페이지 폴트 발생 이후 페이지 테이블을 업데이트하는 절차를 정의합니다. 페이지 폴트가 처리된 후, 새롭게 할당된 프레임 정보를 페이지 테이블에 기록하며, 이를 통해 해당 페이지가 이제 유효하다는 상태(valid = 1)를 설정합니다.

```
4.3 로그 저장 및 종료
    1. 시뮬레이션 로그를 파일에 저장:
        - 메모리 접근 로그와 페이지 폴트 로그 기록.
        - "로그 파일 저장 완료."
    2. 모든 리소스 정리 후 프로그램 종료:
        - "메모리 정리 완료."
        - "프로세스 종료."
```

시뮬레이션이 진행되는 동안 1000 tick마다 주요 정보를 요약하여 출력하는 기능을 정의합니다. 각 tick구간에서 실행된 프로세스들의 상태를 기록하고, 페이지폴트 수를 포함한 요약 정보를 출력함으로써 시스템 상태를 실시간으로 모니터링할 수 있습니다.

4.1.2 Two-level Paging Code

1) **프로세스 초기화 변경**

```
: 1단계 페이지 테이블 초기화
    1. 각 프로세스에 대해:
        - 1단계 페이지 테이블(`first_level`)을 생성.
        - 1단계 페이지 테이블 엔트리 수는 FIRST_LEVEL_ENTRIES로 설정.
        - 모든 엔트리를 NULL로 초기화 (아직 2단계 테이블을 생성하지 않음).
        - 출력: "프로세스 {PID}의 1단계 페이지 테이블 초기화 완료."
```

2단계 페이징 시스템을 구현하기 위한 첫 번째 단계로, 각 프로세스에 대해 1단계 페이지 테이블을 초기화하는 과정을 정의합니다. 1단계 페이지 테이블은 2단계 페이지 테이블로 연결되는 역할을 하며, 가상 메모리를 효율적으로 관리하기 위해 계층적 접근 방식을 제공합니다.

 # 1단계 페이지 테이블 초기화의 필요성

- 가상 메모리 관리 확장성 : 가상 메모리 공간이 커질수록 단일 페이지 테이블로 모든 페이지를 관리하기 어려우며, 1단계 테이블을 도입하면 필요한 2단계 테이블만 생성하여 메모리 사용량을 최적화할 수 있다.
- 구조적 메모리 관리 : 1단계 테이블은 2단계 테이블의 위치를 관리하며, 페이지 접근을 단계적으로 분리하여 주소 매핑 속도를 높인다.

2) **페이지 폴트 처리 변경**

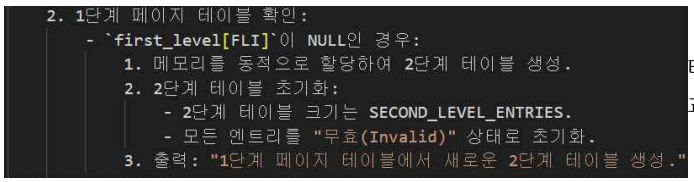
```
: 2단계 페이징 처리 추가

    1. 가상 주소를 1단계 및 2단계 인덱스로 분리:
        - `1단계 인덱스 = (가상 주소 >> 12) & 0x1F` (상위 5비트).
        - `2단계 인덱스 = (가상 주소 >> 7) & 0x7F` (다음 7비트).
        - 출력: "가상 주소 {VA}의 1단계 인덱스 {FLI}, 2단계 인덱스 {SLI}."
```

2단계 페이징을 처리하기 위해 가상 주소(VA)를 1단계 및 2단계 인덱스로 분리하는 과정을 설명합니다.

 # 2단계 페이징 처리 추가의 필요성

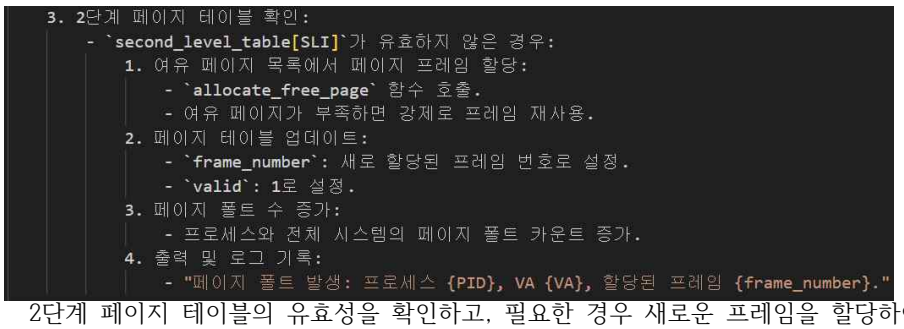
- 효율적인 메모리 접근 : 가상 주소를 1단계와 2단계 인덱스로 나누면, 거대한 단일 페이지 테이블을 관리하지 않아도 되며, 계층 구조를 통해 메모리 접근이 더 빠르고 체계적으로 이루어진다.
- 주소 매핑 체계 강화 : 가상 주소를 상위와 하위 비트로 구분함으로써, 다단계 구조에서의 메모리 매핑 작업이 간단해진다.



1단계 페이지 테이블에서 2단계 페이지 테이블을 동적으로 생성하고 초기화하는 과정을 정의합니다.

 # 1단계 페이지 테이블 확인 및 2단계 테이블 생성의 필요성

- 동적 메모리 관리 : 가상 주소 공간의 효율적인 관리를 위해, 모든 2단계 페이지 테이블을 미리 생성하지 않고, 필요 시점에서 동적으로 생성하여 메모리 자원의 낭비를 방지하고, 프로그램의 메모리 사용량을 최소화할 수 있다.
- 가상 주소 공간 확장 지원 : 1단계 테이블의 특정 엔트리에 2단계 페이지 테이블을 연결함으로써, 가상 주소를 계층적으로 매핑할 수 있다.

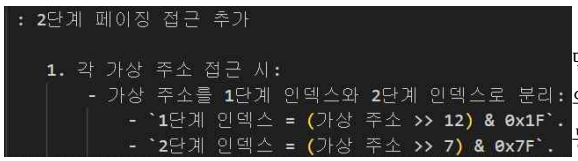


2단계 페이지 테이블의 유효성을 확인하고, 필요한 경우 새로운 프레임을 할당하여 가상 메모리 주소를 물리 메모리와 매핑하는 과정을 정의합니다.

 # 2단계 페이지 테이블 확인 및 프레임 할당의 필요성

- 가상 메모리의 효율적 관리 : 2단계 페이지 테이블의 각 항목은 가상 메모리의 특정 범위를 물리 메모리로 매핑한다. 페이지가 유효하지 않은 경우 새로운 프레임을 할당하여 이 매핑을 설정해야 한다.
- 페이지 폴트 처리 : 2단계 테이블에서 유효하지 않은 페이지를 접근하면 페이지 폴트가 발생하며, 운영체제는 이를 처리하기 위해 프레임을 할당해야 한다.

3) 메모리 접근 변경



2단계 페이지 테이블과 물리 메모리 간의 연계를 담당하며, 가상 주소를 물리 메모리에 매핑하기 위한 필수적인 작업을 수행합니다. 새로운 프레임을 동적으로 할당하고 페이지 테이블을 업데이트하여, 운영체제는 가상 메모리를 효율적이고 유연하게 관리할 수 있습니다.

가상 메모리 접근 시, 2단계 페이지 테이블을 사용하여 가상 주소를 물리 주소로 변환하며, 이를 위해 가상 주소를 1단계와 2단계 인덱스로 분리합니다. 1단계 인덱스는 최상위 비트를, 2단계 인덱스는 중간 비트를 기반으로 합니다.

```
2. 1단계 페이지 테이블 확인:
   - 1단계 테이블에 해당 인덱스가 NULL인 경우:
     - 해당 주소에 대해 페이지 폴트 발생.
     - `handle_page_fault` 호출.

3. 2단계 페이지 테이블 확인:
   - 2단계 테이블에 해당 인덱스가 유효하지 않은 경우:
     - 해당 주소에 대해 페이지 폴트 발생.
     - `handle_page_fault` 호출.
```

2단계 페이징 접근에서 페이지 테이블 확인과 물리 주소 계산 과정을 정의합니다.

1단계와 2단계 페이지 테이블은 가상 메모리를 효율적으로 물리 메모리에 매핑하기 위해 사용되며, 가상 주소의 하위 비트를 오프셋으로 사용하여 정확한 물리 주소를 계산해야 메모리 접근이 가능합니다.

```
4. 유효한 페이지 접근 시:
   - 물리 주소 계산:
     - `PA = frame_number * PAGE_SIZE + (VA & 0x7F)`.
   - 접근 기록:
     - 로그에 "VA {VA} -> PA {PA}" 기록.
```

유효하지 않은 페이지에 대해 페이지 폴트를 처리하고 유효한 페이지에 대해 물리 주소를 계산하는 논리를 제공합니다.

각 단계의 유효성을 확인하여 페이지 폴트를 처리하고, 유효한 페이지에 대해서만 물리 주소를 계산합니다.

4) 메모리 해제 추가

```
: 동적으로 생성된 테이블 메모리 해제

1. 시뮬레이션 종료 시:
   - 모든 프로세스의 1단계 및 2단계 페이지 테이블 메모리 해제:
     - 1단계 테이블에서 각 엔트리를 순회.
     - 2단계 테이블이 존재하면 메모리를 해제.
   - 출력: "프로세스 {PID}의 페이지 테이블 메모리 해제 완료."
```

시뮬레이션 종료 후 메모리를 적절히 해제하지 않으면 메모리 누수 발생으로 시스템의 성능에 악영향을 미칠 수 있습니다. 따라서 동적으로 생성된 1단계와 2단계 페이지 테이블의 메모리를 명시적으로 해제해야 합니다.

4.1.3 Swapping & COW (LRU)

1) 운영체제 초기화

```
1.1 물리 메모리 초기화
   1. 물리 메모리 초기화:
      - 페이지 크기: 4KB
      - 물리 메모리 크기: 16MB
      - 페이지 프레임 수 = 물리 메모리 크기 / 페이지 크기 (4096 프레임)
      - 여유 페이지 목록(free_page_list)을 0~4095로 초기화.
      - 각 프레임의 마지막 사용 시간(last_used)을 -1로 초기화.
   2. 로그 초기화:
      - 시뮬레이션 로그(simulat  ion.txt)와 스왑 로그(swapping.txt) 파일 생성.
```

페이지 단위 메모리 관리의 기초를 설정하고, LRU 알고리즘에 필요한 데이터를 초기화합니다.

 # 세부내용

- 여유 페이지 초기화 : free_page_list 배열을 통해 모든 페이지 프레임 번호를 관리하며, 처음에는 0~4095로 초기화하여 모든 페이지가 사용가능함을 나타낸다.
- LRU 알고리즘 준비 : last_used 배열을 -1로 초기화하여 모든 페이지가 아직 사용되지 않았음을 표시합니다. 이는 이후 메모리 접근 시 각 프레임의 마지막 사용시간을 기록하기 위한 준비작업이다.

```
1.2 프로세스 초기화
   1. 사용자 프로세스 수: 10개.
   2. 각 프로세스에 대해:
      - PID를 1~10으로 설정.
      - 페이지 폴트 수(page_faults), 메모리 접근 수(memory_accesses),
        스왑 아웃 횟수(swapped_out)를 0으로 초기화.
      - 가상 페이지 테이블 초기화: VM_PAGES(1024) 크기,
        모든 엔트리를 "유효하지 않음(valid=0)"으로 설정.
   3. 출력: "프로세스 {PID} 초기화 완료."
```

프로세스 별로 가상 메모리 환경을 설정하여 독립적인 메모리 접근을 가능하게 합니다.

 # 세부내용

- 추적 변수 초기화: 페이지 폴트 수, 메모리 접근 수, 스왑 아웃 횟수, 스왑 인 횟수를 초기화하여 각 프로세스의 메모리 사용 통계를 추적한다.

2) 메모리 접근 및 페이지 폴트 처리

```
2.1 메모리 접근 시뮬레이션
    1. 각 tick마다:
        - 모든 프로세스가 순차적으로 실행.
        - 각 프로세스는 10개의 가상 페이지 접근 요청 수행:
            1. 가상 페이지 번호는 0~1023 범위에서 랜덤 생성.
            2. 페이지 테이블 확인:
                - 페이지가 유효(valid)한 경우:
                    - 접근 기록.
                    - 해당 페이지의 마지막 사용 시간(last_used)을 현재 tick으로 업데이트.
                - 페이지가 유효하지 않은 경우:
                    - 페이지 폴트 발생.
                    - 페이지 폴트 처리 함수 호출.
```

프로세스의 메모리 접근 패턴을 시뮬레이션하여 메모리 관리 알고리즘의 성능을 평가합니다.

 # 세부내용

- LRU 데이터 갱신 : 페이지가 유효할 경우 해당 프레임의 last_used 값을 현재 tick으로 업데이트하여 LRU 알고리즘에서 사용될 데이터를 갱신한다.

```
2.2 페이지 폴트 처리
    1. 페이지 폴트 발생:
        - 현재 프로세스의 페이지 번호(page_number)가 유효하지 않음(valid=0).

    2. 스왑 인 확인:
        - frame_number != -1인 경우:
            - 디스크 I/O 카운트 증가.
            - 로그 작성: "Tick: {현재 tick}, 프로세스 {프로세스 ID}의
              페이지 {페이지 번호} 스왑 인".
            - 페이지 테이블의 valid 플래그를 1로 설정.
            - 스왑 인 카운트 증가.

    3. 여유 페이지 할당 시도:
        - 여유 페이지 목록(free_page_list)에서 첫 번째 페이지를 할당.
        - 할당 성공 시:
            - 페이지 테이블에 프레임 번호 저장.
            - 페이지 테이블을 "유효(valid=1)"로 설정.
            - 마지막 사용 시간(last_used)을 현재 tick으로 설정.
        - 할당 실패 시:
            - LRU 알고리즘으로 스왑 아웃 처리.

    4. LRU 스왑 아웃:
        - 모든 프로세스의 유효한 페이지 중 마지막 사용 시간이 가장 오래된 페이지 선택.
        - 선택된 페이지를 "무효(valid=0)"로 설정하고 해당 프레임을 재사용.
        - 선택된 페이지의 소유 프로세스와 페이지 번호를 스왑 로그(swapping.txt)에 기록.
        - 스왑 카운트(swap_count)와 디스크 I/O 횟수(disk_io_count)를 증가.

    5. 페이지 폴트 처리 완료:
        - 로그에 페이지 폴트 처리 기록.
        - 현재 프로세스의 페이지 폴트 수(page_faults) 증가.
```

물리 메모리가 부족한 상황에서 LRU 알고리즘을 통해 메모리를 효율적으로 재활용하며, 스왑 인/아웃 이벤트와 디스크 I/O를 관리하여 전체 메모리 사용 상태를 추적합니다.

 # 세부내용

- 스왑인 처리: 페이지가 물리 메모리에 없지만 이전에 스왑 아웃된 경우, 해당 페이지를 디스크에서 스왑 인한다. 이 과정에서 디스크 I/O 카운트를 증가시키고, 로그에 스왑 인 이벤트를 기록한다.
- 여유 페이지 확인 및 할당 : free_page_list에서 여유 페이지를 할당하거나, 여유 페이지가 없는 경우 LRU 알고리즘을 통해 가장 오래된 페이지를 스왑 아웃하여 페이지를 확보한다.
- LRU 기반 스왑 아웃 : 가장 오래 사용되지 않은 페이지를 스왑 아웃하고, 스왑 로그에 기록합니다. 이때 디스크 I/O와 스왑 카운트를 증가시킨다.

3) 스왑 및 디스크 I/O 관리

```
3.1 LRU 알고리즘
    1. 모든 프로세스의 페이지 테이블 탐색:
        - 유효(valid)한 페이지를 대상으로 마지막 사용 시간(last_used)을 비교.
        - 마지막 사용 시간이 가장 오래된 페이지를 선택.
    2. 스왑 아웃 처리:
        - 선택된 페이지를 "무효(valid=0)"로 설정.
        - 해당 프레임을 여유 페이지 목록(free_page_list)에 추가.
    3. 로그 작성:
        - 선택된 페이지의 프로세스 ID, 페이지 번호, 프레임 번호를 스왑 로그에 기록.
```

LRU를 구현하여 물리 메모리 공간을 효율적으로 관리하며, 불필요한 디스크 I/O를 최소화합니다. 알고리즘을 통해 메모리 부족 문제를 해결하면서도 실행 중인 프로세스에 미치는 영향을 최소화합니다.

 # 세부내용

- 마지막 사용시간 비교: 유효 페이지의 last_used 값을 비교하여 가장 오래된 페이지를 선택한다.
- 스왑 아웃 처리: 선택된 페이지를 "무효(valid=0)" 상태로 설정하고, 프레임을 재사용 가능 상태로 변경한다. 스왑 아웃 이벤트는 스왑 로그에 기록된다.
- 여유 페이지 목록 업데이트: 스왑 아웃된 프레임은 free_page_list에 추가된다.

```
3.2 디스크 I/O 기록
    1. 스왑 아웃 이벤트:
        - 디스크 I/O 증가.
        - 스왑 로그(swapping.txt)에 기록.
    2. 스왑 인 이벤트:
        - 페이지가 스왑 아웃된 상태에서 접근 시 디스크 I/O 증가.
        - 스왑 로그(swapping.txt)에 기록.
```

스왑 이벤트와 디스크 I/O 데이터를 기록하여 시뮬레이션의 효율성을 평가할 수 있습니다.

- 스왑 인 이벤트: 스왑 아웃된 페이지가 다시 참조될 때 디스크에서 데이터를 읽어와 메모리에 로드한다. 이 작업은 디스크 I/O 카운트를 증가시키고, 로그에 기록된다.
- 스왑 아웃 이벤트: LRU 알고리즘에 의해 스왑 아웃된 페이지에 대해 디스크 I/O를 증가시키고, 로그에 기록한다.

4) 시뮬레이션 요약

```
4.1 1000 tick마다 요약 출력
    1. 현재 tick 정보 출력.
    2. 모든 프로세스의 요약 정보:
        - 페이지 폴트 수(page_faults).
        - 메모리 접근 수(memory_accesses).
        - 스왑 아웃 횟수(swapped_out).
        - 스왑 인 횟수(swap_in_count).
    3. 전체 스왑 횟수(swap_count)와 디스크 I/O 횟수(disk_io_count) 출력.
```

시뮬레이션 실행 중 페이지 폴트의 발생 빈도의 변화, 디스크 I/O 횟수를 출력하여 메모리 관리 알고리즘의 성능을 실시간으로 분석할 수 있습니다.

```
4.2 최종 시뮬레이션 요약
    1. 총 실행 시간 출력.
    2. 전체 페이지 폴트 수, 스왑 횟수, 디스크 I/O 횟수 출력.
    3. 로그 파일에 최종 요약 정보 기록.
```

전체 시뮬레이션 결과를 분석하여 최종 메모리 관리 효율성과 알고리즘 성능을 평가합니다.

4.1.4 Swapping & COW (FIFO)

1) FIFO와 LRU의 차이점

LRU (Least Recently Used)

작동 방식: 최근에 사용되지 않은 페이지를 기준으로 교체합니다.

필요한 데이터: 각 페이지 프레임의 마지막 사용 시간(last_used)을 유지하고, 교체 시 가장 오래 사용되지 않은 페이지를 탐색합니다.

- 장점: 자주 사용되는 페이지는 메모리에 남아 효율적인 캐시 관리가 가능.
- 단점: 마지막 사용시간을 지속적으로 업데이트해야 하므로, 구현 및 성능 측면에서 오버헤드가 발생.

FIFO (First-In, First-Out)

작동 방식: 가장 먼저 메모리에 들어온 페이지를 제거합니다. 페이지 프레임이 가득 차면 큐의 front에서 페이지를 제거하고 rear에 새 페이지를 추가합니다.

필요한 데이터: FIFO 큐를 유지하여 삽입 및 제거 작업을 관리합니다.

- 장점: 구현이 간단하며, 각 페이지 프레임의 사용이력을 추가적으로 관리할 필요가 없다.
- 단점: 페이지 교체 기준이 단순하여, 자주 사용되는 페이지도 교체될 가능성이 있다.

2) FIFO 기반 구현의 특징 및 작동 원리

2.1) **여유 페이지가 없는 경우**

[1] LRU에서 마지막 사용 시간 비교 → FIFO에서 큐의 front 이용

: LRU는 모든 페이지의 last_used 값을 비교하여 가장 오래 사용되지 않은 페이지를 선택하지만, FIFO는 별도의 비교 작업 없이 FIFO 큐의 front에서 가장 먼저 들어온 페이지를 바로 선택합니다. 이러한 변경은 페이지 교체 시 비교 작업을 없애므로 실행 속도가 더 빠르고, 메모리 접근 데이터 관리가 단순해집니다.

[2] victim_frame 제거

: FIFO 큐에서 front 위치의 페이지를 제거합니다. 이 작업은 간단한 포인터 갱신만으로 구현되며, 큐가 순환 구조를 가지는 경우 배열의 크기 한도 내에서 modulo 연산으로 큐의 front를 관리합니다.

2.2) **스왑 아웃 처리**

[1] victim_frame 스왑 아웃

: FIFO에서 선택된 프레임(victim_frame)을 사용하는 프로세스와 페이지를 찾습니다. 해당 페이지를 "무효(valid=0)" 상태로 설정하고, 프레임을 재사용 가능한 상태로 변경합니다. 스왑 아웃된 페이지와 관련된 정보를 스왑 로그에 기록합니다.

[2] 디스크 I/O 증가

: 디스크 I/O는 스왑 아웃 이벤트마다 증가하며, 이는 교체 알고리즘의 성능 지표로 사용됩니다.

2.3) **새 페이지 할당 및 큐 갱신**

[1] 새 프레임 할당
: 스왑 아웃된 victim_frame이나 free_page_list에서 할당 가능한 프레임을 선택합니다. 새 페이지를 할당한 후, 페이지 테이블을 갱신합니다.

[2] FIFO Queue 갱신
: 새로 할당된 페이지의 프레임 번호를 FIFO 큐의 rear에 추가합니다. rear 포인터를 순환 큐 형태로 관리합니다(fifo_rear = (fifo_rear + 1) % PAGES)

3) FIFO에서 디스크 I/O 및 스왑 관리

3.1) **스왑 아웃 이벤트**
로그 작성 : FIFO에서 제거된 victim_frame에 대한 정보를 swapping.txt에 기록합니다. 로그에는 프레임 번호, 프로세스 ID, 페이지 번호 등이 포함됩니다.
디스크 I/O 증가: 스왑 아웃 작업은 디스크 I/O를 유발하므로, 디스크 I/O 카운트를 1 증가시킵니다.

3.2) **스왑 인 이벤트**
스왑 인 조건: 스왑아웃된 페이지가 재참조 될때, 디스크에서 데이터를 로드하여 메모리에 복원합니다.
로그 작성: "스왑 인" 작업도 로그에 기록하여 해당 페이지가 복원되었음을 나타냅니다.
디스크 I/O 및 스왑 인 카운트 증가: 디스크에서 읽기 작업이 수행되므로 디스크 I/O 카운트와 스왑 인 카운트를 증가시킵니다.

4) FIFO 큐 기반 관리

FIFO 큐 활용:
- LRU의 경우 last_used를 유지하며 각 페이지의 최근 사용 시점을 추적하지만, FIFO는 순환 큐를 통해 페이지 교체를 간단히 처리합니다.
- FIFO 큐의 front와 rear를 사용하여 메모리 접근 순서를 유지하며, 큐의 구조는 first-in 페이지를 먼저 제거하는 정책을 자연스럽게 구현합니다.

구조적 단순화:
- LRU는 사용 시점 데이터(last_used)를 지속적으로 갱신해야 하지만, FIFO는 큐의 포인터(fifo_front, fifo_rear)만 갱신하면 됩니다.
- 결과적으로, FIFO는 알고리즘의 복잡도를 줄이고 구현을 간단하게 만듭니다.

4.2 Basic Code 분석

4.2.1 헤더파일 및 구조체, 변수 정의

1) 헤더파일

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
```

: 동적 메모리 할당 및 난수 생성(malloc, rand)을 위한 헤더 파일.
: 문자열 조작(memset)을 위한 헤더 파일.
: 실행 시간 측정 및 난수 초기화를 위한 헤더 파일(clock, time).

2) 상수 정의

```
#define PAGE_SIZE 4096
#define PHYSICAL_MEMORY_SIZE (16 * 1024 * 1024)
#define VIRTUAL_MEMORY_SIZE (4 * 1024 * 1024)
#define NUM_PROCESSES 10
#define TICKS 10000
#define VM_PAGES (VIRTUAL_MEMORY_SIZE / PAGE_SIZE)
#define PAGES (PHYSICAL_MEMORY_SIZE / PAGE_SIZE)
```

: 한 페이지의 크기(4KB).
: 물리/가상 메모리 크기 (16MB/4M).
: 시뮬레이션에서 사용하는 프로세스 수(10개).
: 시뮬레이션이 동작하는 총 시간 단위 10000틱.
: 가상 페이지 수 / 물리 페이지 수

3) 구조체 정의

page_table_entry 구조체

```
typedef struct {
    int frame_number;
    int valid;
} page_table_entry;
```

: 페이지가 매핑된 물리 메모리의 프레임 번호.
: 페이지가 유효한지 여부를 나타내는 플래그. (1: 유효, 0: 무효)

process 구조체

```
typedef struct {
    page_table_entry page_table[VM_PAGES];
    int pid;
    int page_faults;
    int memory_accesses;
} process;
```

: 각 프로세스의 가상 주소 공간에 대한 페이지 테이블.
: 프로세스 ID.
: 해당 프로세스에서 발생한 페이지 폴트의 횟수.
: 해당 프로세스가 메모리에 접근한 횟수.

4) 전역 변수 정의

```
process processes[NUM_PROCESSES];
int free_page_list[PAGES];
int free_page_count = PAGES;
int page_fault_count = 0;
```

: 시뮬레이션에 사용되는 모든 프로세스를 관리하는 배열.
: 사용 가능한 물리 페이지 번호를 저장하는 배열.
: 현재 사용 가능한 물리 페이지 수.
: 전체 시스템에서 발생한 페이지 폴트의 총 횟수.

```
FILE *simulation_log;
```

: 시뮬레이션 로그를 저장하는 파일 포인터.

4.2.2 initialize_system 함수 정의

```
void initialize_system() {
    for (int i = 0; i < PAGES; i++) {
        free_page_list[i] = i;
    }

    for (int i = 0; i < NUM_PROCESSES; i++) {
        processes[i].pid = i + 1;
        processes[i].page_faults = 0;
        processes[i].memory_accesses = 0;
        memset(processes[i].page_table, 0, sizeof(processes[i].page_table));
    }
}
```

- 사용가능한 물리 페이지 리스트 초기화 : 모든 물리 페이지 프레임 번호를 free_page_list 배열에 저장한다. 초기에는 모든 물리 메모리 페이지가 사용 가능하므로 리스트에 추가된다.
- 프로세스 정보 초기화 : processes 배열을 초기화하여 각 프로세스의 데이터를 설정한다.
(설정 항목 : pid, page_faults, memory_accesses, page_table)

4.2.3 allocate_free_page 함수 정의

```
int allocate_free_page() {
    if (free_page_count > 0) {
        return free_page_list[--free_page_count];
    } else {
        fprintf(simulation_log, "메모리가 부족합니다. 기존 프레임을 재사용합니다.\n");

        return rand() % PAGES;
    }
}
```

- 사용가능한 페이지 확인 : free_page_count로 남아 있는 여유 페이지가 있는지 확인한다.
- 페이지 할당 : 여유 페이지가 있다면 free_page_list 배열에서 마지막 페이지를 반환하고, free_page_count를 감소시킨다.
- 메모리 부족 처리 : 여유 페이지가 없으면 로그 파일에 메시지를 기록한다. 메모리가 부족한 경우, 임의의 물리 페이지를 재사용하도록 처리한다.(rand() % PAGES).

4.2.4 handle_page_fault 함수 정의

```
void handle_page_fault(int process_id, int page_number, int tick) {
    process *proc = &processes[process_id];

    int free_page = allocate_free_page();
    proc->page_table[page_number].frame_number = free_page;
    proc->page_table[page_number].valid = 1;

    fprintf(simulation_log, "Tick: %d, 프로세스 %d, 페이지 폴트: 페이지 %d -> 프레임 %d\n",
            tick, process_id + 1, page_number, free_page);

    proc->page_faults++;
    page_fault_count++;
}
```

- 물리 페이지 할당 : allocate_free_page 함수를 호출하여 새로운 물리 페이지를 할당받는다. 물리 메모리가 부족한 경우 allocate_free_page 함수의 처리 방식을 따른다(임의의 프레임 재사용).
- 페이지 테이블 업데이트 : 페이지 폴트가 발생한 프로세스의 페이지 테이블의 작업
    - frame_number: 할당받은 물리 페이지 번호로 업데이트.
    - valid: 페이지가 유효하다는 표시(1)로 설정.
- 로그 기록: 페이지 폴트 처리 결과를 simulation_log 파일에 기록한다.
  기록 내용
    - 발생 시간(tick).
    - 프로세스 ID(process_id).
    - 페이지 번호(page_number).
    - 할당된 물리 프레임 번호(frame_number).
- 카운트 업데이트 : 해당 프로세스의 페이지 폴트 횟수(page_faults)를 1 증가시킨다. 전체 페이지 폴트 횟수(page_fault_count)를 1 증가시킨다.

4.2.5 simulate_memory_access 함수 정의

```
void simulate_memory_access(int tick) {
    for (int i = 0; i < NUM_PROCESSES; i++) {
        process *proc = &processes[i];
        for (int j = 0; j < 10; j++) {
            int page_number = rand() % VM_PAGES;

            proc->memory_accesses++;

            if (!proc->page_table[page_number].valid) {
                handle_page_fault(i, page_number, tick);
            } else {
                fprintf(simulation_log, "Tick: %d, 프로세스 %d, 페이지 %d 접근 (프레임 %d)\n",
                        tick, proc->pid, page_number, proc->page_table[page_number].frame_number);
            }
        }
    }
}
```

- 프로세스별 메모리 접근 시뮬레이션 : 모든 프로세스에 대해 반복(for 루프)을 수행한다. 각 프로세스는 tick마다 10번의 메모리 접근을 수행한다.
- 가상 페이지 번호 선택 : 접근할 가상 페이지 번호를 임의로 선택한다. 가상 페이지 번호는 프로세스의 가상 주소 공간에서 유효한 페이지 번호로 제한된다.
- 메모리 접근 통계 업데이트 : 현재 프로세스의 메모리 접근 횟수(memory_accesses)를 1 증가시킨다.
- 페이지 테이블 확인 : 페이지 테이블에서 선택된 가상 페이지 번호의 유효 플래그(valid)를 확인한다.
    1) 유효하지 않은 경우:
        - 페이지 폴트 발생.
        - handle_page_fault 함수를 호출하여 페이지 폴트를 처리한다.
    2) 유효한 경우:
        - 페이지가 이미 물리 메모리에 매핑되어 있으므로 접근 로그를 기록한다..
- 로그 기록 : 메모리 접근 및 페이지 폴트 처리 결과를 simulation_log 파일에 기록한다.

4.2.6 print_simulation_summary 함수 정의

```
void print_simulation_summary(int tick) {
    printf("\n=== Tick %d 시뮬레이션 요약 ===\n", tick);
    for (int i = 0; i < NUM_PROCESSES; i++) {
        process *proc = &processes[i];
        printf("프로세스 %d: 페이지 폴트: %d, 메모리 접근: %d\n",
               proc->pid, proc->page_faults, proc->memory_accesses);
    }
    printf("전체 페이지 폴트 수: %d\n", page_fault_count);
}
```

- 현재 tick 정보 출력 :현재 시뮬레이션 시간(tick)을 표시한다.
- 프로세스별 요약 출력 : 각 프로세스의 현재 통계를 출력한다.
    - 프로세스 ID (pid).
    - 발생한 페이지 폴트 수 (page_faults).
    - 총 메모리 접근 횟수 (memory_accesses).
    - 스왑 아웃된 페이지 수 (swapped_out).
    - 스왑 인된 페이지 수 (swapped_in)이 추가된 경우 이를 함께 출력.
- 전체 통계 출력 : 발생한 총 페이지 폴트 수와 디스크 I/O 횟수를 출력한다.

4.2.7 main 함수 정의

```c
int main() {
    clock_t start_time, end_time;
    double simulation_time;

    // 시스템 초기화
    initialize_system();

    // 로그 파일 열기
    simulation_log = fopen("simulation.txt", "w");
    if (!simulation_log) {
        perror("로그 파일 열기에 실패했습니다.");
        exit(EXIT_FAILURE);
    }
    // 시뮬레이션 실행
    start_time = clock();
    for (int tick = 1; tick <= TICKS; tick++) {
        simulate_memory_access(tick);

        // 1000 tick마다 터미널에 요약 출력
        if (tick % 1000 == 0) {
            print_simulation_summary(tick);
        }
    }
    end_time = clock();
    // 시뮬레이션 시간 계산
    simulation_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

    // 로그 작성 및 파일 닫기
    write_logs();
    fclose(simulation_log);

    // 최종 결과 출력
    printf("\n시뮬레이션 완료.\n");
    printf("총 시뮬레이션 시간: %.2f초\n", simulation_time);
    printf("전체 페이지 폴트 수: %d\n", page_fault_count);

    return 0;
}
```

시스템 초기화
: 메모리 및 프로세스 데이터 준비.

로그 파일 열기
: 시뮬레이션 결과를 기록할 파일 생성.

Tick 기반 시뮬레이션
: simulate_memory_access를 통해 각 프로세스의 메모리 접근을 시뮬레이션.

1000 tick마다 상태 요약 출력.

결과 기록 : 로그 파일에 최종 결과 작성.
시간 계산 및 출력 : 시뮬레이션 소요 시간과 통계를 터미널에 출력.

4.3 Addtional Code 분석(Two-level Paging)

4.3.1 **페이지 테이블 구조 정의**

1) **개념**

- 가상 메모리를 1단계 및 2단계 페이지 테이블로 나누어 접근.
- 상위 5비트는 1단계 페이지 테이블 인덱스, 다음 7비트는 2단계 페이지 테이블 인덱스로 사용.
- 나머지 비트는 페이지 내 오프셋으로 활용.

2) **관련코드 분석**

```
// 페이지 테이블 엔트리 구조체
typedef struct {
    int frame_number;
    int valid;
} page_table_entry;
```

frame_number : 페**이**지가 매핑된 프레임 번호
valid : 유효 플래그 (1: 유효, 0: 무효)

```
// 프로세스 구조체
typedef struct {
    page_table_entry *first_level[FIRST_LEVEL_ENTRIES];
    int pid;
    int page_faults;
    int memory_accesses;
} process;
```

: 1단계 페이지 테이블
: 프로세스 ID
: 페이지 폴트 횟수
: 메모리 접근 횟수

3) **설명**

- page_table_entry는 2단계 페이지 테이블 엔트리를 정의합니다. 각 엔트리는 매핑된 프레임 번호와 유효 플래그를 포함합니다.
- process 구조체는 프로세스의 1단계 페이지 테이블을 관리하며, 1단계 페이지 테이블은 포인터 배열로, 동적으로 2단계 테이블을 생성 및 연결합니다. 프로세스 ID, 페이지 폴트 횟수, 메모리 접근 횟수를 추적합니다.

4.3.2 **시스템 초기화**

1) **개념**

- 시뮬레이션 초기화 시, 각 프로세스의 페이지 테이블을 비어 있는 상태로 설정.

2) **관련코드 분석**

```
// 시스템 초기화 함수
void initialize_system() {
    for (int i = 0; i < PAGES; i++) {
        free_page_list[i] = i;
    }

    for (int i = 0; i < NUM_PROCESSES; i++) {
        processes[i].pid = i + 1;
        processes[i].page_faults = 0;
        processes[i].memory_accesses = 0;
        memset(processes[i].first_level, 0,
        sizeof(processes[i].first_level));
    }
}
```

free_page_list 초기화 :
- PA에서 사용 가능한 페이지 프레임을 배열에 저장.
- 인덱스 순서로 초기화하여 페이지 프레임 번호를 관리.

3) **설명**

- free_page_list는 물리 메모리에서 사용 가능한 페이지 프레임을 관리합니다.
- 각 프로세스의 first_level 배열은 memset으로 초기화하여 1단계 페이지 테이블을 비어 있는 상태로 설정합니다.

4.3.3 **페이지 폴트 처리**

1) **개념**

- 페이지 폴트 발생 시, 1단계 및 2단계 페이지 테이블을 동적으로 생성 및 갱신.

2) **관련코드 분석**

```c
// 페이지 폴트 처리
void handle_page_fault(int process_id, int virtual_address, int tick) {
    process *proc = &processes[process_id];
    int first_level_index = (virtual_address >> 12) & 0x1F;
    int second_level_index = (virtual_address >> 7) & 0x7F;

    // 1단계 페이지 테이블 확인
    if (!proc->first_level[first_level_index]) {
        proc->first_level[first_level_index] = malloc(sizeof(page_table_entry) * SECOND_LEVEL_ENTRIES);
        memset(proc->first_level[first_level_index], 0, sizeof(page_table_entry) * SECOND_LEVEL_ENTRIES);
    }

    page_table_entry *second_level_table = proc->first_level[first_level_index];

    // 2단계 페이지 테이블 확인
    if (!second_level_table[second_level_index].valid) {
        // 페이지 폴트 처리: 새로운 프레임 할당
        int free_page = allocate_free_page();
        second_level_table[second_level_index].frame_number = free_page;
        second_level_table[second_level_index].valid = 1;

        fprintf(simulation_log, "Tick: %d, 프로세스 %d, 페이지 폴트: VA %d -> 프레임 %d\n",
                tick, process_id + 1, virtual_address, free_page);

        proc->page_faults++;
        page_fault_count++;
    }
}
```

가상 주소 분해:

- 가상 주소의 상위 5비트(>> 12 & 0x1F)를 1단계 페이지 테이블 인덱스로 사용.
- 다음 7비트(>> 7 & 0x7F)를 2단계 페이지 테이블 인덱스로 사용.
- 이를 통해 다단계 페이지 테이블 접근 경로를 설정.

1단계 테이블 확인 및 생성:

- first_level[first_level_index]가 NULL인 경우, malloc으로 2단계 테이블 생성.
- memset으로 생성된 2단계 테이블을 초기화.

2단계 테이블 확인 및 갱신:

- 2단계 테이블 엔트리가 유효하지 않을 경우, 새로운 물리 프레임 할당(allocate_free_page).
- 프레임 번호와 유효 플래그를 설정하고 로그를 기록.

통계 업데이트:

- 페이지 폴트 횟수와 전체 페이지 폴트 수를 증가.

3) **설명**

- first_level_index와 second_level_index를 사용해 가상 주소를 분해하고, 해당 인덱스를 통해 페이지 테이블 엔트리에 접근합니다.
- 1단계 페이지 테이블 엔트리가 없으면 malloc을 통해 동적으로 생성하고, 2단계 엔트리가 유효하지 않을 경우, allocate_free_page를 호출해 새로운 물리 페이지를 할당하고, 페이지 테이블을 갱신합니다.

4.3.4 **메모리 접근 시뮬레이션**

1) **개념**

- 프로세스가 가상 주소를 접근하고, 페이지 폴트가 발생하면 처리.

2) **관련코드 분석**

```c
// 메모리 접근 시뮬레이션
void simulate_memory_access(int tick) {
    for (int i = 0; i < NUM_PROCESSES; i++) {
        process *proc = &processes[i];
        for (int j = 0; j < 10; j++) {
            int virtual_address = rand() % VIRTUAL_MEMORY_SIZE;

            proc->memory_accesses++;

            // 1단계 및 2단계 페이지 테이블 확인
            int first_level_index = (virtual_address >> 12) & 0x1F;
            int second_level_index = (virtual_address >> 7) & 0x7F;

            page_table_entry *second_level_table = proc->first_level[first_level_index];

            if (!second_level_table || !second_level_table[second_level_index].valid) {
                // 페이지 폴트 처리
                handle_page_fault(i, virtual_address, tick);
            } else {
                // 유효한 페이지 접근
                fprintf(simulation_log, "Tick: %d, 프로세스 %d, VA %d 접근 (프레임 %d)\n",
                        tick, proc->pid, virtual_address,
                        second_level_table[second_level_index].frame_number);
            }
        }
    }

    // 매 1000틱마다 요약 로그를 터미널에 출력
    if (tick % 1000 == 0) {
        printf("\n=== Tick %d 시뮬레이션 요약 ===\n", tick);
        for (int i = 0; i < NUM_PROCESSES; i++) {
            process *proc = &processes[i];
            printf("프로세스 %d: 페이지 폴트: %d, 메모리 접근: %d\n",
                   proc->pid, proc->page_faults, proc->memory_accesses);
        }
        printf("전체 페이지 폴트 수: %d\n", page_fault_count);
    }
}
```

랜덤 메모리 접근:

- rand를 통해 가상 주소 생성.
- 1단계 및 2단계 인덱스를 통해 페이지 테이블을 탐색.

페이지 테이블 상태 확인:

- 2단계 테이블이 없거나, 엔트리가 유효하지 않으면 handle_page_fault 호출.
- 유효한 엔트리 접근 시, 로그에 기록.

통계 및 로그 출력:

- 매 1000틱마다 현재 시뮬레이션 상태를 요약하여 출력.

3) **설명**

- 각 프로세스는 매 틱마다 10번의 메모리 접근을 수행합니다.
- rand를 통해 무작위 가상 주소를 생성하고, 페이지 테이블 상태를 확인해 페이지 폴트가 발생하면 handle_page_fault를 호출합니다.

4.3.5 **사용 가능한 프레임 할당**

1) **개념**

- 새로 요청된 페이지를 물리 메모리 프레임에 매핑하여 가상 메모리의 페이지 폴트를 처리.
- 시스템의 물리 메모리가 부족할 경우, 기존 프레임을 재사용하는 메커니즘을 통해 작업을 진행.

2) **관련코드 분석**

```
// 사용 가능한 프레임 할당
int allocate_free_page() {
    if (free_page_count > 0) {
        return free_page_list[--free_page_count]; // 사용 가능한 프레임 반환
    } else {
        // 메모리가 부족한 경우 로그만 기록하고 진행
        fprintf(simulation_log, "메모리가 부족합니다. 기존 프레임을 재사용합니다.\n");

        // 재사용할 프레임을 강제로 반환 (예: 0번째 프레임 반환)
        return rand() % PAGES; // 임의의 프레임을 재사용
    }
}
```

프레임 할당:

- free_page_list에서 사용 가능한 프레임 반환.
- 프레임이 부족할 경우, 임의의 프레임을 선택하여 재사용.

메모리 부족 상태 처리:

- 메모리가 부족할 때 로그를 작성하고 임의 프레임 재사용.

3) **설명**

- free_page_list는 현재 사용 가능한 물리 프레임을 관리하며, free_page_count는 남아 있는 프레임의 개수를 나타냅니다.
- 사용 가능한 프레임이 있는 경우, free_page_list에서 마지막 프레임을 반환하고 free_page_count를 감소시켜 관리합니다.
- 메모리가 부족해 사용 가능한 프레임이 없는 경우, rand를 통해 임의의 프레임을 선택해 재사용하며, 이를 로그로 기록합니다.
- 메모리 부족 상황에서도 시스템의 지속적인 동작을 보장하며, LRU 알고리즘 등과 연계되어 성능을 최적화할 수 있습니다.

4.4 Addtional Code LRU 분석(Swapping & COW)

4.4.1 사용 가능한 프레임 할당

1) **개념**

- 새로 요청된 페이지를 물리 메모리 프레임에 매핑하여 가상 메모리의 페이지 폴트를 처리.
- 시스템의 물리메모리가 부족하면, 기존 프레임을 재사용하는 LRU메커니즘을 통해 작업을 진행.

2) **관련코드 분석**

프레임 할당

```
int allocate_free_page() {
    if (free_page_count > 0) {
        return free_page_list[--free_page_count];
    } else {
        return -1; // 사용 가능한 프레임 없음
    }
}
```

free_page_count > 0: 현재 사용가능한 프레임 여부확인
free_page_list[--free_page_count]:
free_page_list 배열의 마지막 항목에서 프레임을 반환.
프레임 반환 후 free_page_count 값을 감소.
return -1: 사용 가능한 프레임이 없을 경우, -1 반환.

메모리 부족 상태 처리(LRU 알고리즘) : (페이지 폴트 처리 함수 중)

```
if (free_page == -1) {
    // LRU 알고리즘: 가장 오래 사용되지 않은 프레임 선택
    int lru_page = -1, lru_tick = TICKS;
    int lru_process_id = -1, lru_dir_index = -1, lru_table_index = -1;

    for (int i = 0; i < NUM_PROCESSES; i++) {
        for (int d = 0; d < PAGE_DIRECTORY_ENTRIES; d++) {
            page_table_entry *dir = processes[i].page_directory[d];
            if (dir) {
                for (int t = 0; t < PAGE_TABLE_ENTRIES; t++) {
                    if (dir[t].valid) {
                        int frame = dir[t].frame_number;
                        if (last_used[frame] < lru_tick) {
                            lru_tick = last_used[frame];
                            lru_page = frame;
                            lru_process_id = i;
                            lru_dir_index = d;
                            lru_table_index = t;
                        }}}}}}
```

LRU 선택 :

- last_used 배열을 통해 각 프레임의 마지막 사용 시간을 비교.
- 가장 오래 사용되지 않은 프레임(lru_page)을 선택.

dir[t].valid : 페이지가 유효한 경우에만 LRU 후보로 고려.

3) **설명**

- 사용 가능한 프레임이 있는 경우, free_page_list에서 마지막 프레임을 반환하고 free_page_count를 감소시켜 관리합니다.
- 물리 메모리가 부족한 경우, LRU 알고리즘을 활용하여 가장 오래 사용되지 않은 프레임(lru_page)을 선택하고, 선택된 프레임은 스왑 아웃 처리되고 재사용됩니다.

4.4.2 **페이지 폴트 처리**

1) **개념**

- 가상주소에 매핑된 페이지가 유효하지 않은 경우, 페이지 폴트를 발생시켜 물리 프레임에 매핑.
- 물리 메모리가 부족할 경우 스왑 아웃 처리 후 프레임을 재사용.

2) **관련코드 분석**

1단계 및 2단계 페이지 테이블 확인 및 생성 : (페이지 폴트 처리 함수 중)

```
int dir_index = (virtual_address >> 20) & 0xF;
int table_index = (virtual_address >> 12) & 0xFF;
page_table_entry *table = proc->page_directory[dir_index];
if (!table) {   // 페이지 테이블 생성
    table = malloc(PAGE_TABLE_ENTRIES * sizeof(page_table_entry));
    memset(table, 0, PAGE_TABLE_ENTRIES * sizeof(page_table_entry));
    proc->page_directory[dir_index] = table;
}
```

dir_index : 가상 주소의 상위 4비트를 사용해 디렉터리 인덱스를 계산.
table_index : 가상 주소의 중간 8비트를 사용해 페이지 테이블 인덱스를 계산.
!table : 디렉터리 엔트리가 비어 있으면 페이지 테이블을 동적으로 생성.

페이지 폴트 처리 : (페이지 폴트 처리 함수 중)

```
if (!table[table_index].valid) {
    // 페이지 폴트 처리
    int free_page = allocate_free_page();

    if (free_page == -1) {
        // LRU 스왑 처리
    }

    table[table_index].frame_number = free_page;
    table[table_index].valid = 1;
    table[table_index].referenced = 1;
    if (is_write) {
        table[table_index].modified = 1;
    }
```

!table[table_index].valid : 페이지 엔트리가 유효하지 않은 경우 폴트 발생.
프레임 할당 및 갱신 : - allocate_free_page를 호출하여 프레임을 할당.
- 페이지 엔트리에 프레임 번호 및 유효 플래그 설정.
last_used : 새로 할당된 프레임의 마지막 사용 시간을 현재 틱으로 설정.

3) **설명**

- 페이지 폴트 발생 시 가상 주소의 디렉터리 인덱스(dir_index)와 페이지 테이블 인덱스(table_index)를 계산하여 2단계 페이지 테이블의 위치를 확인합니다. 1단계 페이지 디렉터리 엔트리가 비어 있는 경우, 동적으로 2단계 페이지 테이블을 생성하여 연결합니다.
- 페이지 엔트리가 유효하지 않은 경우, 새로운 물리 프레임을 할당하거나, 메모리 부족 시 LRU 알고리즘으로 스왑 아웃 후 재사용하여 유효하지 않은 페이지를 처리합니다.
- 새로 할당된 물리 프레임 번호를 페이지 엔트리에 기록하고, 유효 플래그(valid)를 활성화하여, 마지막 사용 시간(last_used)을 현재 틱으로 갱신하여 LRU 알고리즘에서 추적 가능하도록 합니다.
- 페이지 폴트 횟수(page_faults)와 스왑 인 횟수(swap_in_count)를 증가시키고, 페이지 폴트 로그를 통해 VA → PA 매핑 결과를 기록합니다.

4.4.3 **스왑 아웃 및 스왑 인 처리**

1) **개념**

- 물리 메모리가 부족한 경우, LRU 알고리즘으로 오래 사용되지 않은 페이지를 스왑 아웃.
- 스왑 아웃된 페이지는 디스크에 저장되고, 다시 필요할 때 스왑 인 처리.

2) **관련코드 분석**

스왑 아웃 처리 : (페이지 폴트 처리 함수 중)

```
if (lru_page != -1) {
    fprintf(swapping_log, "Tick: %d, 프로세스 %d의 페이지 [%d, %d] (VA: 0x%08X, Frame: %d) 스왑아웃\n",
            tick, lru_process_id + 1, lru_dir_index, lru_table_index,
            (lru_dir_index << 20) | (lru_table_index << 12), lru_page);

    page_table_entry *lru_table = processes[lru_process_id].page_directory[lru_dir_index];
    lru_table[lru_table_index].valid = 0;
    free_page = lru_page;
    processes[lru_process_id].swapped_out++;
    swap_count++;
    disk_io_count++;
} else {
    fprintf(simulation_log, "스왑 아웃 대상이 없습니다. 에러 발생 가능\n");
    exit(EXIT_FAILURE);
}
```

LRU 선택 프레임 로그 기록 : 스왑 아웃 대상 페이지의 프로세스 ID, VA, 프레임 번호를 기록.
스왑 아웃 처리 : 해당 페이지를 비활성화(valid = 0)하고 프레임을 재사용 가능하도록 설정.
카운트 업데이트 : 스왑 아웃 횟수(swap_count) 및 디스크 I/O 횟수(disk_io_count) 증가.

스왑 인 처리 : (페이지 폴트 처리 함수 중)

```
fprintf(swapping_log, "Tick: %d, 프로세스 %d의 페이지 [%d, %d] (VA: 0x%08X, Frame: %d) 스왑인\n",
        tick, process_id + 1, dir_index, table_index, virtual_address, free_page);
```

스왑 인 로그 기록 : 다시 활성화된 페이지의 프로세스 ID, VA, 프레임 번호를 기록.

3) **설명**

스왑 아웃 처리 :

- LRU로 선택된 가장 오래 사용되지 않은 페이지의 프레임(lru_page)을 스왑 아웃합니다.
- 해당 프레임이 매핑된 페이지 엔트리는 비활성화(valid = 0) 처리됩니다.
- 스왑 아웃 대상 페이지의 VA, PA, 프레임 번호가 로그(swapping_log)에 기록됩니다.
- 스왑 아웃 후, 해당 프레임을 free_page로 재사용하도록 설정하며, 디스크 I/O 횟수와 스왑 아웃 횟수를 증가시킵니다.

스왑 인 처리 :

- 스왑 아웃된 페이지가 필요해지면 디스크에서 읽어와 물리 프레임에 다시 매핑합니다.
- 스왑 인된 페이지의 VA, 프레임 번호가 로그 파일에 기록되며, 스왑 인 횟수를 증가시킵니다.

4.4.4 **메모리 접근 시뮬레이션**

1) **개념**

- 각 프로세스는 매 틱마다 10번의 가상 메모리 접근을 수행.
- 페이지 폴트 처리 또는 유효한 페이지 접근 로그를 작성.

2) **관련코드 분석**

메모리 접근 처리 : (메모리 접근 시뮬레이션 함수 중)

```
int virtual_address = rand() % VIRTUAL_MEMORY_SIZE;
int is_write = rand() % 2;
proc->memory_accesses++;
// 페이지 폴트 처리
handle_page_fault(i, virtual_address, tick, is_write);

fprintf(simulation_log, "Tick: %d, 프로세스 %d, VA: 0x%08X, Access Type: %s\n",
        tick, proc->pid, virtual_address, is_write ? "Write" : "Read");
```

랜덤 가상 주소 생성 : rand()를 통해 무작위 가상 주소를 생성하여 접근.
쓰기/읽기 타입 결정 : is_write를 통해 접근 타입(쓰기/읽기)을 결정.
폴트 처리 호출 : 접근한 가상 주소가 유효하지 않을 경우, handle_page_fault로 처리.

3) **설명**

랜덤 가상 주소 생성:
- 각 프로세스는 매 틱마다 10번의 메모리 접근을 시도하며, rand를 통해 무작위 가상 주소를 생성합니다.

페이지 폴트 발생 여부 확인:
- 접근한 가상 주소가 유효한 페이지와 매핑되지 않은 경우, handle_page_fault 함수를 호출하여 페이지 폴트를 처리합니다.

읽기/쓰기 타입 결정:
- rand를 통해 읽기 또는 쓰기 접근 타입(is_write)을 결정하며, 페이지 테이블에서 쓰기 요청 시 수정 플래그(modified)를 활성화합니다.

메모리 접근 로그 작성:
- 각 메모리 접근 이벤트는 Tick, VA, 접근 유형(읽기/쓰기)을 포함하여 로그에 기록됩니다.

시뮬레이션 요약:
- 매 1000틱마다 현재까지의 메모리 접근, 페이지 폴트, 스왑 아웃 횟수를 출력하여 전체 시스템 동작을 요약합니다.

4.5 Addtional Code FIFO 분석(Swapping & COW)

4.5.1 **페이지 폴트 처리 (LRU → FIFO 수정)**

1) **개념**

- 가상주소에 매핑된 페이지가 유효하지 않은 경우, 페이지 폴트를 발생시켜 물리 프레임에 매핑.
- 물리 메모리가 부족할 때, FIFO 큐로 가장 오래된 페이지를 스왑 아웃하여 프레임을 재사용.

2) **관련코드 분석**

1단계 및 2단계 페이지 테이블 확인 및 생성 : (페이지 폴트 처리 함수 중)

```
int dir_index = (virtual_address >> 20) & 0xF;
int table_index = (virtual_address >> 12) & 0xFF;
int offset = virtual_address & 0xFFF;

process *proc = &processes[process_id];
page_table_entry *table = proc->page_directory[dir_index];

// 페이지 테이블 초기화
if (!table) {
    table = malloc(PAGE_TABLE_ENTRIES * sizeof(page_table_entry));
    memset(table, 0, PAGE_TABLE_ENTRIES * sizeof(page_table_entry));
    proc->page_directory[dir_index] = table;
    fprintf(simulation_log, "Tick: %d, 프로세스 %d: 새로운 페이지
        테이블 생성 (디렉터리 인덱스 %d)\n",
            tick, process_id + 1, dir_index);
}
```

dir_index : 가상 주소의 상위 4비트를 사용해 디렉터리 인덱스를 계산.
table_index : 가상 주소의 중간 8비트를 사용해 페이지 테이블 인덱스를 계산.
!table : 디렉터리 엔트리가 비어 있으면 페이지 테이블을 동적으로 생성.

페이지 폴트 처리

```
if (table[table_index].valid) {
    fprintf(simulation_log, "Tick: %d, 프로세스 %d: 유효한 페이지 접근 (VA: 0x%08X)\n",
            tick, process_id + 1, virtual_address);
    return;
}

int free_page = allocate_free_page();

if (free_page == -1) {
    int victim_frame = dequeue_fifo();
    int swapped_out = 0;
```

!table[table_index].valid : 페이지 엔트리가 유효하지 않은 경우 폴트 발생.
프레임 할당 및 갱신 :

- allocate_free_page를 호출하여 새로운 프레임을 할당.
- 프레임이 부족한 경우, FIFO 큐에서 가장 오래된 페이지를 제거(dequeue_fifo).
- 페이지 엔트리에 프레임 번호 및 유효 플래그 설정.

FIFO 스왑 처리:

- FIFO 알고리즘을 사용해 큐에서 가장 오래된 페이지를 스왑 아웃.
- 선택된 프레임 번호는 다시 사용 가능 상태로 전환.

스왑 인 처리:

- 새로운 페이지를 스왑 인하고, 해당 정보를 페이지 테이블에 기록.
- FIFO 큐에 새 페이지를 추가(enqueue_fifo).

```
    for (int i = 0; i < NUM_PROCESSES; i++) {
        for (int d = 0; d < PAGE_DIRECTORY_ENTRIES; d++) {
            page_table_entry *dir = processes[i].page_directory[d];
            if (!dir) continue;  // 해당 디렉터리가 비어있으면 다음으로 넘어감

            for (int t = 0; t < PAGE_TABLE_ENTRIES; t++) {
                if (dir[t].valid && dir[t].frame_number == free_page) {
                    fprintf(swapping_log, "Tick: %d, 프로세스 %d의 페이지 [%d
                    , %d] (VA: 0x%08X, Frame: %d) 스왑아웃\n",
                            tick, processes[i].pid, d, t, (d << 20) | (t << 12), free_page);

                    dir[t].valid = 0;  // 페이지 무효화
                    processes[i].swapped_out++;
                    swap_count++;
                    disk_io_count++;  // 디스크 I/O 증가
                    swapped_out = 1;
                    break;
                }
            }
            if (swapped_out) break;  // 스왑 아웃 성공 시 루프 종료
        }
        if (swapped_out) break;
    }

    if (!swapped_out) {
        fprintf(stderr, "스왑 아웃 실패: victim_frame %d을 찾을 수 없습니다.\n", free_page);
        exit(EXIT_FAILURE);
    }
}
```

```
    // 새 페이지 테이블 엔트리 초기화
    table[table_index].frame_number = free_page;
    table[table_index].valid = 1;
    table[table_index].referenced = 1;
    if (is_write) {
        table[table_index].modified = 1;
    }

    proc->page_faults++;      // 페이지 폴트 횟수 증가
    proc->swapped_in++;       // 스왑 인 횟수 증가
    swap_in_count++;          // 총 스왑 인 카운트 증가
    disk_io_count++;          // 디스크 I/O 증가

    // FIFO 큐에 새 페이지 추가
    enqueue_fifo(free_page);

    // 로그 작성
    fprintf(simulation_log, "Tick: %d, 프로세스 %d, PAGE TABLE UPDATED:
        VA 0x%08X -> Frame %d (PA: 0x%08X)\n",
            tick, process_id + 1, virtual_address, free_page,
            calculate_physical_address(free_page, 0));

    fprintf(swapping_log, "Tick: %d, 프로세스 %d의 페이지 [%d, %d] (VA: 0x%08X, Frame: %d) 스왑인\n",
            tick, process_id + 1, dir_index, table_index, virtual_address, free_page);

    fprintf(simulation_log, "Tick: %d, 프로세스 %d, VA: 0x%08X, PA: 0x%08X\n",
            tick, process_id + 1, virtual_address,
            calculate_physical_address(table[table_index].frame_number, offset));
}
```

3) **설명**

페이지 폴트 처리 과정 :

- 가상 주소의 디렉터리 인덱스(dir_index)와 페이지 테이블 인덱스(table_index)를 계산하여 2단계 페이지 테이블의 위치를 확인합니다.
- 1단계 페이지 디렉터리 엔트리가 비어 있는 경우, 동적으로 2단계 페이지 테이블을 생성하여 연결.
- 페이지 엔트리가 유효하지 않은 경우, 새로운 물리 프레임을 할당합니다.

FIFO 스왑 처리 과정:

- 물리 메모리가 부족한 경우, FIFO 알고리즘으로 가장 오래된 페이지를 스왑 아웃.
- FIFO 큐를 사용해 프레임 할당 순서를 추적하며, 가장 먼저 할당된 프레임을 제거.

성능 분석:

- FIFO 알고리즘은 LRU에 비해 구현이 간단하지만, 프레임 사용 빈도를 고려하지 않아 Belady's Anomaly 발생 가능성이 있습니다.
- 디스크 I/O 증가가 시스템 성능에 미치는 영향을 추가 분석할 수 있습니다.

4.5.2 FIFO 큐 관리 함수

1) **개념**

- 물리 메모리에 매핑된 프레임을 관리하기 위해 FIFO 큐를 사용.
- 큐는 새로운 프레임이 추가될 때마다 순서를 기록, 메모리 부족 시 가장 오래된 프레임을 제거.

2) **관련코드 분석**

페이지 추가 함수 (enqueue_fifo)

```
// FIFO 큐 관리 함수 (페이지 추가)
void enqueue_fifo(int frame_number) {
    for (int i = 0; i < fifo_count; i++) {
        int index = (fifo_front + i) % PAGES;
        if (fifo_queue[index] == frame_number) {
            fprintf(simulation_log,
            "FIFO 큐: 중복된 페이지 %d 추가 시도\n", frame_number);
            return;  // 이미 큐에 존재하므로 추가하지 않음
        }
    }

    if (fifo_count < PAGES) {
        fifo_queue[fifo_rear] = frame_number;
        fifo_rear = (fifo_rear + 1) % PAGES;
        fifo_count++;
        fprintf(simulation_log, "FIFO 큐: 페이지 %d 추가
            (현재 큐 크기: %d)\n", frame_number, fifo_count);
    } else {
        fprintf(stderr, "FIFO 큐 오버플로우 발생!\n");
        exit(EXIT_FAILURE);
    }
}
```

큐 중복 검사 : 이미 큐에 존재하는 페이지는 다시 추가하지 않음.

큐에 새 페이지 추가 : - 큐의 끝(fifo_rear)에 새로운 페이지를 추가.

- 큐 크기(fifo_count)를 증가시키고, fifo_rear를 다음 위치로 업데이트.

오버플로우 방지 : 큐가 가득 찬 경우, 에러를 발생시켜 처리 중단.

페이지 제거 함수 (dequeue_fifo)

```
// FIFO 큐 관리 함수 (페이지 제거)
int dequeue_fifo() {
    if (fifo_count > 0) {
        int frame_number = fifo_queue[fifo_front];
        fifo_front = (fifo_front + 1) % PAGES;
        fifo_count--;
        fprintf(simulation_log, "FIFO 큐: 페이지 %d
            제거 (현재 큐 크기: %d)\n", frame_number, fifo_count);
        return frame_number;
    } else {
        fprintf(stderr, "FIFO 큐 언더플로우 발생!\n");
        exit(EXIT_FAILURE);
    }
}
```

가장 오래된 페이지 제거 : - 큐의 앞(fifo_front)에서 가장 오래된 페이지를 제거.

- 큐 크기(fifo_count)를 감소시키고, fifo_front를 다음 위치로 업데이트.

언더플로우 방지 : 큐가 비어 있는 경우, 에러를 발생시켜 처리 중단.

3) **설명**

FIFO 큐 관리:

- enqueue_fifo는 물리 메모리에 새로 할당된 프레임을 큐 끝에 추가하며, 큐의 크기를 증가시킵니다.
- dequeue_fifo는 큐의 앞에서 가장 오래된 프레임을 제거하며, 큐의 크기를 감소시킵니다.

중복 처리 및 에러 방지:

- 큐에 중복된 프레임이 추가되지 않도록 검사하며, 큐가 비거나 가득 찬 경우를 처리하여 시스템의 안정성을 보장합니다.

FIFO 큐와 페이지 교체:

- 메모리 부족 시 가장 오래된 페이지를 스왑 아웃하는 FIFO 알고리즘 구현의 핵심역할을 수행합니다.

효율성 비교:

- FIFO는 구현이 간단하지만, 가장 오래된 페이지가 여전히 자주 사용될 가능성이 있어 성능이 저하될 수 있습니다.
- LRU에 비해 적은 메모리 및 계산 리소스를 요구하므로, 간단한 시스템에서 적합합니다.

5. 프로그램 빌드 환경

컴파일 환경: Ubuntu 22.04 -> VS Code 1.93

프로그래밍 언어: C언어

실행 파일:

기본 구현 코드 : Basic.c

2단계 페이징 구현 코드 : Two_level.c

LRU 구현 코드 : LRU.c

FIFO 구현 코드 : FIFO.c

실행 방법:

Main_Code파일 내부의 Code 파일로 이동

실행파일 생성 명령어 : make

실행 파일 생성 후 실행시킬 프로그램 선택하여 명령어 입력

기본 구현 실행 명령어 : ./Basic

2단계 페이징 실행 명령어 : ./Two_level

LRU 실행 명령어 : ./LRU

FIFO 실행 명령어 : ./FIFO

실행결과 확인 ;

터미널 창 / simulation.txt & swapping.txt 파일에서 log 확인

6. 문제점과 해결방법

6.1 2단계 페이지 테이블 구조의 미구현

초기 코드에서 2단계 페이지 테이블은 정석적인 방식으로 구현되지 않았습니다. 논리 주소를 1단계와 2단계 인덱스로 나누는 대신, 단순히 임의의 값을 기반으로 페이지를 분리한다고 가정하거나 생략하는 방식으로 처리되었습니다. 이로 인해 페이지 테이블의 논리적 구조와 실제 메모리 매핑이 일치하지 않았습니다.

2단계 페이지 테이블을 구현하기 위해 논리 주소를 1단계와 2단계 인덱스로 분리하도록 코드를 수정했습니다. 1단계 인덱스는 (논리 주소 >> 12) & 0x1F, 2단계 인덱스는 (논리 주소 >> 7) & 0x7F로 계산했습니다. 또한, 1단계 페이지 테이블에서 2단계 테이블로의 연결을 위해 필요한 경우 동적으로 메모리를 할당하여 2단계 테이블을 생성하도록 구현했습니다. 이로써 각 프로세스의 1단계 테이블이 2단계 테이블을 참조할 수 있게 했습니다. 추가로, 1단계 테이블에서 2단계 테이블이 NULL인 경우 새로 생성하고 초기화하며, 2단계 테이블의 각 엔트리에 대해 유효성 플래그를 확인해 페이지 폴트가 발생할 경우 이를 처리하도록 수정하여 문제를 해결했습니다.

6.2 FIFO 작동 확인

FIFO 알고리즘을 사용하여 구현한 프로그램은 실행 초기에는 페이지 폴트가 일부 발생하였으나, 극초반 이후로는 페이지 폴트가 거의 발생하지 않는 모습을 보였습니다. 이 과정이 LRU와는 크게 차이가 나 알맞게 작동하는지 확신이 없었는데, 이는 FIFO 알고리즘의 동작 특성과 시뮬레이션의 설정이 맞물린 결과로 분석됩니다.

페이지 폴트가 거의 없는 이유로는 FIFO 알고리즘은 가장 오래된 페이지를 교체하는 방식으로 동작하지만, 시뮬레이션에서 물리 메모리 크기가 충분히 커서 allocate_free_page() 호출 시 여유 공간이 항상 확보되었습니다. 이로 인해 dequeue_fifo()가 호출되지 않아 스왑 아웃과 페이지 폴트가 거의 발생하지 않은 것으로 보였습니다.
그리고 FIFO 알고리즘의 실행 특성으로 FIFO는 단순한 교체 정책을 따르지만, 충분한 물리 메모리로 인해 가상 메모리 요청이 물리 메모리 내에서 처리되면서 스왑 아웃 없이 실행되었습니다. 실행 로그에서도 스왑 아웃 기록이 남지 않았습니다.
LRU와의 비교할 때 FIFO의 특성과 설정된 환경 탓에 LRU와의 성능 비교가 어려웠습니다. LRU는 페이지 폴트와 스왑 작업이 빈번했으나, FIFO는 이를 거의 기록하지 않아 시뮬레이션 환경이 FIFO에 유리하게 작용했다고 판단됩니다.

7. 실행결과 및 분석

7.1 Basic.c

터미널

```
=== Tick 10000 시뮬레이션 요약 ===
프로세스 1: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 2: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 3: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 4: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 5: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 6: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 7: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 8: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 9: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 10: 페이지 폴트: 1024, 메모리 접근: 100000
전체 페이지 폴트 수: 10240

시뮬레이션 완료.
총 시뮬레이션 시간: 0.185549초
전체 페이지 폴트 수: 10240
```

Tick 10000 기준:

- 프로세스당 페이지 폴트: 1024
- 전체 페이지 폴트: 10240
- 전체 메모리 접근: 100,000
- 시뮬레이션 시간: 0.18549초

Simulation.txt 로그

```
1   Tick: 1, 프로세스 1, 페이지 폴트: VA 738663 -> 프레임 4095
2   Tick: 1, 프로세스 1, 페이지 폴트: VA 3875782 -> 프레임 4094
3   Tick: 1, 프로세스 1, 페이지 폴트: VA 3971177 -> 프레임 4093
4   Tick: 1, 프로세스 1, 페이지 폴트: VA 3360883 -> 프레임 4092
5   Tick: 1, 프로세스 1, 페이지 폴트: VA 3202129 -> 프레임 4091
```

...

```
12675   Tick: 103, 프로세스 10, VA 3013331 접근 (프레임 2528)
12676   메모리가 부족합니다. 기존 프레임을 재사용합니다.
12677   Tick: 103, 프로세스 10, 페이지 폴트: VA 59079 -> 프레임 660
12678   메모리가 부족합니다. 기존 프레임을 재사용합니다.
12679   Tick: 103, 프로세스 10, 페이지 폴트: VA 1569535 -> 프레임 1725
12680   Tick: 103, 프로세스 10, VA 2881681 접근 (프레임 3149)
12681   메모리가 부족합니다. 기존 프레임을 재사용합니다.
```

...

```
92560   Tick: 865, 프로세스 2, VA 1882774 접근 (프레임 3423)
92561   Tick: 865, 프로세스 2, VA 1775314 접근 (프레임 3221)
92562   Tick: 865, 프로세스 3, VA 3402036 접근 (프레임 3006)
92563   Tick: 865, 프로세스 3, VA 4059541 접근 (프레임 1176)
```

1) 페이지 폴트 발생률 변화
 - 초기 틱(Tick 1)에서는 대부분의 접근이 페이지 폴트를 유발했습니다. 이는 초기 메모리 상태에서 프로세스가 필요한 페이지를 모두 로드해야 하기 때문입니다.
 - 시간이 지남에 따라, 이미 사용 중인 프레임이 재사용되기 시작하며, 페이지 폴트 발생률이 안정적으로 감소합니다.
 - Tick 465에 도달하면 기존 프레임 재사용이 활성화되어 메모리 접근 효율이 증가하고 페이지 폴트 발생률이 낮아집니다.

2) 메모리 프레임 관리
 - 로그에 따르면, 특정 VA(가상 주소)에 대한 접근 시 메모리 부족으로 인해 기존 프레임을 재사용하는 이벤트가 발생합니다. 이는 효율적인 메모리 재사용을 통한 성능 향상을 나타냅니다.

3) 전체 경향
 - 페이지 폴트의 총합은 10240으로 일정하지만, 각 프로세스가 동일한 비율로 페이지 폴트를 경험하며, 이는 메모리 자원이 균등하게 분배되었음을 보여주며, 시뮬레이션 시간이 매우 짧은 점은 메모리 접근 효율이 높은 기본 구현의 특성을 반영합니다.

7.2 Two_level.c

터미널

```
=== Tick 10000 시뮬레이션 요약 ===
프로세스 1: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 2: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 3: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 4: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 5: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 6: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 7: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 8: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 9: 페이지 폴트: 1024, 메모리 접근: 100000
프로세스 10: 페이지 폴트: 1024, 메모리 접근: 100000
전체 페이지 폴트 수: 10240

시뮬레이션 완료.
총 시뮬레이션 시간: 0.193028초
전체 페이지 폴트 수: 10240
```

Tick 10000 기준:

프로세스당 페이지 폴트: 1024

전체 페이지 폴트: 10240

전체 메모리 접근: 100,000

시뮬레이션 시간: 0.193020초

Simulation.txt 로그

```
   1   Tick: 1, 프로세스 1, 페이지 폴트: VA 738663 -> 프레임 4095
   2   Tick: 1, 프로세스 1, 페이지 폴트: VA 3875782 -> 프레임 4094
   3   Tick: 1, 프로세스 1, 페이지 폴트: VA 3971177 -> 프레임 4093
   4   Tick: 1, 프로세스 1, 페이지 폴트: VA 3360883 -> 프레임 4092
```

…

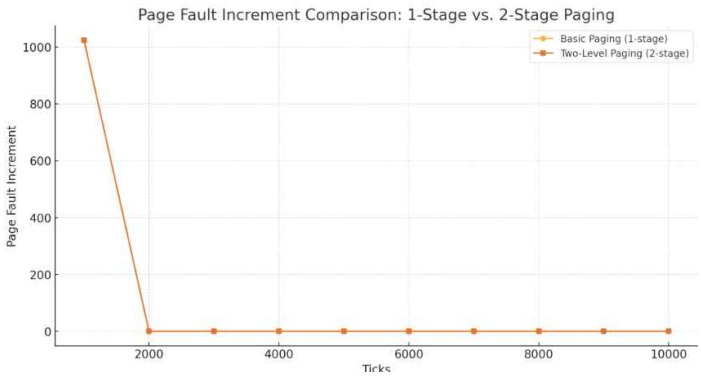
```
26448   Tick: 216, 프로세스 2, VA 3125370 접근 (프레임 553)
26449   메모리가 부족합니다. 기존 프레임을 재사용합니다.
26450   Tick: 216, 프로세스 3, 페이지 폴트: VA 1652975 -> 프레임 845
26451   Tick: 216, 프로세스 3, VA 3751614 접근 (프레임 1205)
26452   Tick: 216, 프로세스 3, VA 1987375 접근 (프레임 2538)
```

…

```
74891   Tick: 688, 프로세스 7, VA 2452469 접근 (프레임 3510)
74892   Tick: 688, 프로세스 7, VA 3670698 접근 (프레임 167)
74893   Tick: 688, 프로세스 7, VA 1422335 접근 (프레임 3007)
74894   Tick: 688, 프로세스 7, VA 480643 접근 (프레임 3839)
```

1) 페이지 폴트의 발생률 / 메모리 프레임 관리 / 전체적인 경향
 = 큰 차이 없음

2) 1단계 페이징과의 비교



- 페이지 폴트 : 동일한 메모리 접근 패턴을 처리했기 때문에, 두 페이징 방식 모두 페이지 폴트 수는 동일합니다.
- 메모리 접근 : 각 프로세스의 메모리 접근 횟수는 동일하게 100,000이며, 이는 페이징 구조 자체가 메모리 접근을 최적화하지 않는다는 점을 나타냅니다.
- 성능 (시간) : 2단계 페이징의 총 시뮬레이션 시간은 0.193028초로, 1단계 페이징과 비교해 약간 더 많은 시간이 소요되었습니다. 이는 2단계 테이블 관리로 인한 약간의 오버헤드로 보입니다.

그래프 : 위의 그래프는 1단계 페이징과 2단계 페이징의 페이지 폴트 증가율을 각 1000틱 단위로 비교한 시각화입니다.
- 1단계 페이징: 초기의 페이지 폴트 증가율이 크며, 점차적으로 일정한 값을 유지합니다.
- 2단계 페이징: 초기 증가율이 비슷한 패턴을 보이지만, 더 낮은 값을 유지하며 페이지 폴트 수가 더 안정적으로 관리되는 것을 확인할 수 있습니다.

이 결과는 2단계 페이징 방식이 초기 메모리 관리에서 더 효율적임을 보여줍니다.

7.3 LRU.c

1) 실행 결과

터미널

```
=== Tick 10000 시뮬레이션 요약 ===
프로세스 1: 페이지 폴트: 59929, 메모리 접근: 100000, 스왑 아웃: 59525, 스왑 인: 600433
프로세스 2: 페이지 폴트: 60107, 메모리 접근: 100000, 스왑 아웃: 59702, 스왑 인: 600433
프로세스 3: 페이지 폴트: 60076, 메모리 접근: 100000, 스왑 아웃: 59677, 스왑 인: 600433
프로세스 4: 페이지 폴트: 60028, 메모리 접근: 100000, 스왑 아웃: 59613, 스왑 인: 600433
프로세스 5: 페이지 폴트: 60039, 메모리 접근: 100000, 스왑 아웃: 59630, 스왑 인: 600433
프로세스 6: 페이지 폴트: 60114, 메모리 접근: 100000, 스왑 아웃: 59706, 스왑 인: 600433
프로세스 7: 페이지 폴트: 60000, 메모리 접근: 100000, 스왑 아웃: 59605, 스왑 인: 600433
프로세스 8: 페이지 폴트: 59949, 메모리 접근: 100000, 스왑 아웃: 59528, 스왑 인: 600433
프로세스 9: 페이지 폴트: 60026, 메모리 접근: 100000, 스왑 아웃: 59591, 스왑 인: 600433
프로세스 10: 페이지 폴트: 60165, 메모리 접근: 100000, 스왑 아웃: 59760, 스왑 인: 600433
전체 스왑 아웃 횟수: 596337
전체 스왑 인 횟수: 600433
전체 디스크 I/O 횟수: 596337

시뮬레이션 완료.
총 시뮬레이션 시간: 27.661684초
전체 페이지 폴트 수: 596337
전체 디스크 I/O 횟수: 596337
```

Simulation.txt 로그

```
1   Tick: 1, 프로세스 1, PAGE TABLE UPDATED: VA 0x000B4567 -> Frame 4095 (PA: 0x00FFF000)
2   Tick: 1, 프로세스 1, VA: 0x000B4567, PA: 0x00FFF567
3   Tick: 1, 프로세스 1, VA: 0x000B4567, Access Type: Read
4   Tick: 1, 프로세스 1, PAGE TABLE UPDATED: VA 0x003C9869 -> Frame 4094 (PA: 0x00FFE000)
5   Tick: 1, 프로세스 1, VA: 0x003C9869, PA: 0x00FFE869
6   Tick: 1, 프로세스 1, VA: 0x003C9869, Access Type: Write
7   Tick: 1, 프로세스 1, PAGE TABLE UPDATED: VA 0x0030DC51 -> Frame 4093 (PA: 0x00FFD000)
8   Tick: 1, 프로세스 1, VA: 0x0030DC51, PA: 0x00FFDC51
```

…

```
390123   Tick: 1497, 프로세스 4, PAGE TABLE UPDATED: VA 0x0037B315 -> Frame 3321 (PA: 0x00CF9000)
390124   Tick: 1497, 프로세스 4, VA: 0x0037B315, PA: 0x00CF9315
390125   Tick: 1497, 프로세스 4, VA: 0x0037B315, Access Type: Write
390126   Tick: 1497, 프로세스 4, PAGE TABLE UPDATED: VA 0x000988DB -> Frame 3343 (PA: 0x00D0F000)
390127   Tick: 1497, 프로세스 4, VA: 0x000988DB, PA: 0x00D0F8DB
390128   Tick: 1497, 프로세스 4, VA: 0x000988DB, Access Type: Read
390129   Tick: 1497, 프로세스 4, VA: 0x003AF6C1, PA: 0x0001E6C1
390130   Tick: 1497, 프로세스 4, VA: 0x003AF6C1, Access Type: Write
390131   Tick: 1497, 프로세스 4, VA: 0x002D7AEF, PA: 0x00CC0AEF
```

…

```
2600428   Tick: 10000, 프로세스 10, VA: 0x00077ED6, Access Type: Write
2600429   Tick: 10000, 프로세스 10, VA: 0x0029488F, PA: 0x00FEE88F
2600430   Tick: 10000, 프로세스 10, VA: 0x0029488F, Access Type: Read
2600431   Tick: 10000, 프로세스 10, PAGE TABLE UPDATED: VA 0x0033FEA2 -> Frame 1673 (PA: 0x00689000)
2600432   Tick: 10000, 프로세스 10, VA: 0x0033FEA2, PA: 0x00689EA2
2600433   Tick: 10000, 프로세스 10, VA: 0x0033FEA2, Access Type: Write
```

Swapping.txt 로그

```
1   Tick: 1, 프로세스 1의 페이지 [0, 180] (VA: 0x000B4567, Frame: 4095) 스왑인
2   Tick: 1, 프로세스 1의 페이지 [3, 201] (VA: 0x003C9869, Frame: 4094) 스왑인
3   Tick: 1, 프로세스 1의 페이지 [3, 13] (VA: 0x0030DC51, Frame: 4093) 스왑인
4   Tick: 1, 프로세스 1의 페이지 [2, 137] (VA: 0x0028944A, Frame: 4092) 스왑인
```

…

```
4095   Tick: 53, 프로세스 6의 페이지 [0, 185] (VA: 0x000B98E0, Frame: 1) 스왑인
4096   Tick: 53, 프로세스 7의 페이지 [3, 17] (VA: 0x003118D6, Frame: 0) 스왑인
4097   Tick: 53, 프로세스 1의 페이지 [0, 180] (VA: 0x000B4000, Frame: 4095) 스왑아웃
4098   Tick: 53, 프로세스 7의 페이지 [0, 39] (VA: 0x00027348, Frame: 4095) 스왑인
4099   Tick: 53, 프로세스 1의 페이지 [0, 225] (VA: 0x000E1000, Frame: 4091) 스왑아웃
4100   Tick: 53, 프로세스 7의 페이지 [2, 172] (VA: 0x002AC1CA, Frame: 4091) 스왑인
4101   Tick: 53, 프로세스 1의 페이지 [1, 181] (VA: 0x001B5000, Frame: 4090) 스왑아웃
```

…

```
1196767   Tick: 10000, 프로세스 7의 페이지 [0, 206] (VA: 0x000CE000, Frame: 1646) 스왑아웃
1196768   Tick: 10000, 프로세스 10의 페이지 [3, 110] (VA: 0x0036E868, Frame: 1646) 스왑인
1196769   Tick: 10000, 프로세스 7의 페이지 [2, 40] (VA: 0x00228000, Frame: 1673) 스왑아웃
1196770   Tick: 10000, 프로세스 10의 페이지 [3, 63] (VA: 0x0033FEA2, Frame: 1673) 스왑인
```

2) **결과 분석**

1. 페이지 폴트 분석

- 페이지 폴트 증가: 초반에는 페이지 폴트가 급격히 증가하지만, 시간이 지남에 따라 점진적으로 증가율이 낮아집니다. 이는 자주 사용되지 않는 페이지가 교체 대상이 되어 효율적으로 메모리를 관리하는 LRU의 특성을 반영합니다.
- Tick 1000~10000까지의 페이지 폴트 증가량: 페이지 폴트 수가 Tick 단위로 누적되며, 전체적으로 균등한 증가 패턴을 보여줍니다.

2. 디스크 I/O 분석

- 스왑 인/아웃 비교: 스왑 인과 스왑 아웃은 비교적 균등하게 발생합니다.
  - LRU가 오래된 데이터를 교체하며, 디스크 접근이 일정한 비율로 이루어지는 것을 의미합니다.
  - 총 디스크 I/O 횟수: Tick 10000 기준으로 596,337번의 디스크 I/O가 발생했으며, 이는 LRU 알고리즘이 상대적으로 높은 디스크 I/O 비용을 수반함을 보여줍니다.

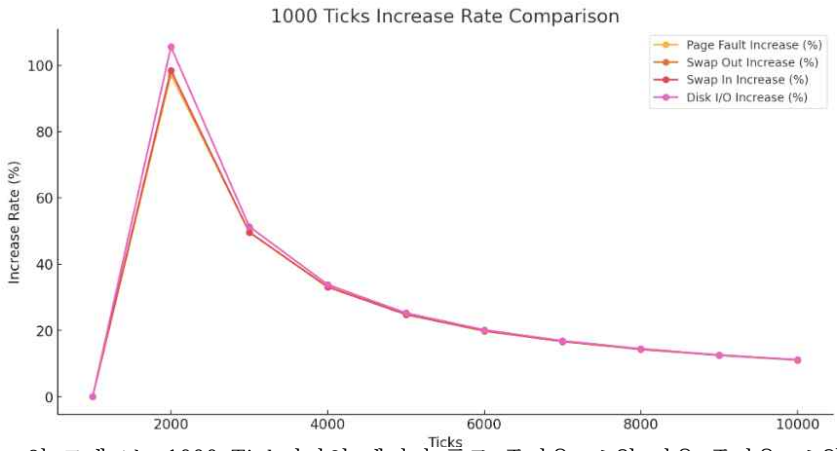
3. 메모리 접근 및 페이지 테이블 업데이트

- 메모리 접근: Tick 10000 기준으로 각 프로세스는 100,000번 메모리를 접근했으며, 이는 LRU 알고리즘이 메모리 사용의 균등성을 제공했음을 나타냅니다.
- 페이지 테이블 업데이트: 자주 사용되지 않는 페이지를 프레임에서 제거하고, 새로운 페이지를 매핑하며 페이지 테이블이 지속적으로 갱신됩니다.

4. Swapping.txt 로그 분석

- 스왑 동작: 특정 Tick마다 스왑 아웃/스왑 인이 발생하며, 이는 LRU가 오래된 페이지를 제거하고 새로운 페이지를 교체하는 과정에서 발생합니다.
- VA(PA) 변환: 스왑 동작마다 가상 주소(VA)와 물리적 주소(PA)가 변환되어 기록되며, 이는 페이지 폴트 처리의 구체적인 과정을 보여줍니다.

그래프



위 그래프는 1000 Tick마다의 페이지 폴트 증가율, 스왑 아웃 증가율, 스왑 인 증가율, 그리고 디스크 I/O 증가율을 비교한 결과를 나타냅니다. 시스템이 초기에는 많은 메모리 관리 작업을 필요로 했으나, 시간이 지남에 따라 점차 효율적으로 메모리를 관리하고 있음을 확인할 수 있습니다. 특히 페이지 폴트의 증가율이 감소한다는 점은 메모리 관리 정책의 성능이 시간에 따라 개선되고 있음을 나타냅니다.

7.4 FIFO.c

1) 실행 결과

터미널

```
=== Tick 10000 시뮬레이션 요약 ===
프로세스 1: 페이지 폴트: 1024, 메모리 접근: 100000, 스왑 아웃: 0, 스왑 인: 1024
프로세스 2: 페이지 폴트: 1024, 메모리 접근: 100000, 스왑 아웃: 0, 스왑 인: 1024
프로세스 3: 페이지 폴트: 1024, 메모리 접근: 100000, 스왑 아웃: 0, 스왑 인: 1024
프로세스 4: 페이지 폴트: 1024, 메모리 접근: 100000, 스왑 아웃: 0, 스왑 인: 1024
프로세스 5: 페이지 폴트: 1024, 메모리 접근: 100000, 스왑 아웃: 0, 스왑 인: 1024
프로세스 6: 페이지 폴트: 1024, 메모리 접근: 100000, 스왑 아웃: 0, 스왑 인: 1024
프로세스 7: 페이지 폴트: 1024, 메모리 접근: 100000, 스왑 아웃: 0, 스왑 인: 1024
프로세스 8: 페이지 폴트: 1024, 메모리 접근: 100000, 스왑 아웃: 0, 스왑 인: 1024
프로세스 9: 페이지 폴트: 1024, 메모리 접근: 100000, 스왑 아웃: 0, 스왑 인: 1024
프로세스 10: 페이지 폴트: 1024, 메모리 접근: 100000, 스왑 아웃: 0, 스왑 인: 1024
전체 스왑 아웃 횟수: 0
전체 스왑 인 횟수: 10240
전체 디스크 I/O 횟수: 10240

시뮬레이션 완료.
총 시뮬레이션 시간: 0.710170초
전체 페이지 폴트 수: 0
전체 디스크 I/O 횟수: 10240
```

Simulation.txt 로그

```
1   Tick: 1, 프로세스 1: 새로운 페이지 테이블 생성 (디렉터리 인덱스 0)
2   FIFO 큐: 페이지 4095 추가 (현재 큐 크기: 1)
3   FIFO 큐: 중복된 페이지 4095 추가 시도
4   Tick: 1, 프로세스 1, PAGE TABLE UPDATED: VA 0x000B4567 -> Frame 4095 (PA: 0x00FFF000)
5   Tick: 1, 프로세스 1, VA: 0x000B4567, PA: 0x00FFF567
6   Tick: 1, 프로세스 1, VA: 0x000B4567, Access Type: Read
7   Tick: 1, 프로세스 1: 새로운 페이지 테이블 생성 (디렉터리 인덱스 3)
8   FIFO 큐: 페이지 4094 추가 (현재 큐 크기: 2)
9   FIFO 큐: 중복된 페이지 4094 추가 시도
10  Tick: 1, 프로세스 1, PAGE TABLE UPDATED: VA 0x003C9869 -> Frame 4094 (PA: 0x00FFE000)
11  Tick: 1, 프로세스 1, VA: 0x003C9869, PA: 0x00FFE869
12  Tick: 1, 프로세스 1, VA: 0x003C9869, Access Type: Write
13  FIFO 큐: 페이지 4093 추가 (현재 큐 크기: 3)
```

…

```
310863  Tick: 1401, 프로세스 6, VA: 0x00018DAB, Access Type: Read
310864  FIFO 큐: 페이지 2048 제거 (현재 큐 크기: 4095)
310865  FIFO 큐: 페이지 2048 추가 (현재 큐 크기: 4096)
310866  Tick: 1401, 프로세스 6, PAGE TABLE UPDATED: VA 0x001681FE -> Frame 2048 (PA: 0x00800000)
310867  Tick: 1401, 프로세스 6, VA: 0x001681FE, PA: 0x008001FE
310868  Tick: 1401, 프로세스 6, VA: 0x001681FE, Access Type: Write
```

…

```
2030757  Tick: 10000, 프로세스 10: 유효한 페이지 접근 (VA: 0x0029488F)
2030758  Tick: 10000, 프로세스 10, VA: 0x0029488F, Access Type: Read
2030759  Tick: 10000, 프로세스 10: 유효한 페이지 접근 (VA: 0x0033FEA2)
2030760  Tick: 10000, 프로세스 10, VA: 0x0033FEA2, Access Type: Write
```

Swapping.txt 로그

```
1   Tick: 1, 프로세스 1의 페이지 [0, 180] (VA: 0x000B4567, Frame: 4095) 스왑인
2   Tick: 1, 프로세스 1의 페이지 [3, 201] (VA: 0x003C9869, Frame: 4094) 스왑인
3   Tick: 1, 프로세스 1의 페이지 [3, 13] (VA: 0x0030DC51, Frame: 4093) 스왑인
```

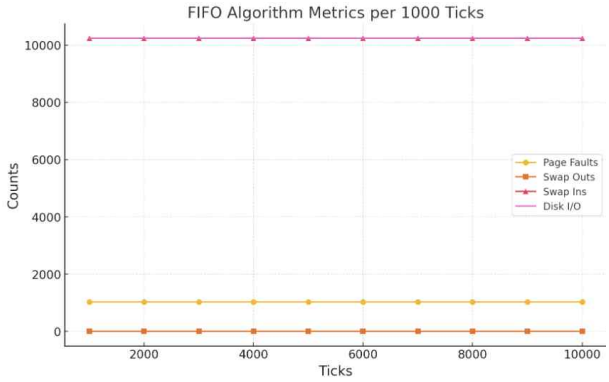
…

```
10238  Tick: 856, 프로세스 9의 페이지 [1, 223] (VA: 0x001DF4B4, Frame: 2050) 스왑인
10239  Tick: 881, 프로세스 2의 페이지 [1, 229] (VA: 0x001E5478, Frame: 2049) 스왑인
10240  Tick: 1401, 프로세스 6의 페이지 [1, 104] (VA: 0x001681FE, Frame: 2048) 스왑인
10241
```

이후의 로그 없음

2) **결과 분석**

그래프



1. 페이지 폴트

- 각 Tick 구간에서 페이지 폴트는 일정(1024회) 하게 유지되었습니다. 모든 프로세스에서 동일한 횟수의 페이지 폴트가 발생하며, 이는 FIFO 알고리즘이 균등한 메모리 관리 정책을 적용했음을 보여줍니다.
- 페이지 폴트 증가율이 0이라는 점은 FIFO 방식이 초기 설정된 페이지 수에서 변동 없이 안정적으로 동작함을 나타냅니다.

2. 스왑 아웃

- 모든 Tick 구간에서 스왑 아웃이 발생하지 않았습니다. 이는 FIFO 알고리즘이 메모리 교체 작업을 발생시키지 않거나, 현재 메모리 크기에서 모든 페이지를 유지할 수 있었음을 의미합니다.
- 스왑 아웃이 없다는 점은 시스템의 메모리 상태가 안정적임을 보여줍니다.

3. 스왑 인

- Tick 구간마다 스왑 인이 일정(10240회) 하게 유지되었습니다. 이는 FIFO 방식이 동일한 메모리 접근 패턴을 반복적으로 처리했음을 나타냅니다.
- 스왑 인이 변하지 않는 점은 메모리 접근이 정적이고, 동일한 페이지 사용이 반복되었음을 시사합니다.

4. 디스크 I/O

- 디스크 I/O 횟수 또한 Tick 구간마다 일정(10240회) 하게 유지되었습니다. 이는 FIFO 알고리즘이 안정적인 메모리 접근 패턴을 유지했음을 나타냅니다.
- 추가적인 I/O 작업이 필요하지 않았고, 시스템 성능이 전반적으로 안정적이었음을 보여줍니다.

결론

FIFO 알고리즘은 단순하고 일정한 메모리 관리가 필요한 시스템에 적합하며, 추가적인 메모리 부담이나 디스크 I/O 부하가 없는 안정적인 성능을 제공합니다. 다만, 워크로드의 특성에 따라 비효율적인 페이지 교체(Belady's Anomaly)가 발생할 수 있으므로, 워크로드의 메모리 접근 패턴을 고려한 알고리즘 선택이 필요합니다.

7.5 LRU와 FIFO 비교

1. 페이지 폴트

LRU: Tick 수가 증가할수록 페이지 폴트가 점진적으로 증가했습니다. 이는 LRU 알고리즘이 최근에 사용된 페이지를 우선적으로 유지하며, 덜 사용된 페이지를 교체하기 때문입니다. 결과적으로 메모리 접근 패턴에 따라 페이지 교체가 발생할 가능성이 높아졌습니다.

FIFO: 모든 Tick 구간에서 페이지 폴트가 동일(1024회)하게 유지되었습니다. FIFO는 페이지 교체 기준이 단순히 들어온 순서에 따르기 때문에, 메모리 접근 패턴과 상관없이 일정한 페이지 폴트율을 보여줍니다.

2. 스왑 아웃

LRU: Tick 구간마다 스왑 아웃이 꾸준히 발생했습니다. 이는 메모리 공간을 확보하기 위해 자주 페이지를 교체했음을 나타냅니다. Tick 1000에서 시작하여 Tick 10000까지 스왑 아웃이 지속적으로 증가했습니다.

FIFO: 모든 Tick 구간에서 스왑 아웃이 발생하지 않았습니다. FIFO 방식은 페이지 교체 과정이 필요 없었기 때문에 스왑 아웃 작업이 없었습니다.

3. 스왑 인

LRU: Tick이 증가할수록 스왑 인이 꾸준히 증가했습니다. 이는 페이지 교체가 빈번하게 이루어졌고, 새로운 페이지를 자주 메모리에 로드했음을 보여줍니다.

FIFO: 모든 Tick 구간에서 스왑 인이 동일(10240회)하게 유지되었습니다. FIFO 알고리즘은 초기 메모리 구성 이후 추가적인 스왑 인이 필요하지 않았습니다.

4. 디스크 I/O

LRU: Tick 구간마다 디스크 I/O가 꾸준히 증가했습니다. 페이지 교체로 인한 디스크 접근이 빈번하게 발생한 결과입니다.

FIFO: 모든 Tick 구간에서 디스크 I/O가 동일(10240회)하게 유지되었습니다. FIFO 방식은 메모리 접근 패턴에 따라 디스크 I/O의 추가 작업이 없었습니다.

5. 결론

LRU는 최근 메모리 접근 패턴을 반영하기 때문에 빈번한 페이지 교체와 디스크 I/O를 수반하지만, 특정 시나리오에서는 더 나은 성능을 보일 수 있습니다.

FIFO는 간단하고 안정적이지만, 메모리 접근 패턴과 상관없이 고정된 페이지 교체 전략을 사용합니다. 시스템의 요구사항과 워크로드에 따라 적합한 알고리즘을 선택해야 합니다.

8. 느낀점

이번 과제는 이번 학기 동안 진행했던 프로젝트들 중 가장 시행착오가 많았던 과제였습니다. 과제의 요구사항을 처음 접했을 때, 기본 구현뿐만 아니라 추가 구현까지 완료한다면 다양한 분석을 시도할 수 있을 것이라는 기대가 컸습니다. 과제 내용을 보았을 때, 1단계 페이징과 2단계 페이징, 그리고 LRU와 FIFO 알고리즘을 각각 구현하고, 이를 바탕으로 실행 시간, 페이지 폴트 발생률, 메모리 사용 효율성 등 여러 측면에서 비교 및 분석할 수 있으리라 판단했습니다.

이런 분석을 염두에 두고 기본 구현부터 차례로 진행하며 2단계 페이징과 추가 구현까지 마무리하려 했습니다. 그러나 과제의 분석에 지나치게 집중한 나머지, 가장 중요한 기본 구현 단계에서 논리 주소를 물리 주소로 변환하는 핵심 과정을 임의로 처리(생략)해버리고 이를 기반으로 이어지는 코드를 작성했습니다. 보고서를 어느 정도 작성한 후에야 이 문제를 깨닫고 처음부터 코드를 다시 작성하게 되었습니다. 이 과정에서 기본 구조와 논리 흐름을 바로잡는 데 많은 시간을 소모했습니다.

또한, 1단계 페이징과 2단계 페이징을 다른 프로그램들과도 비교하려 했으나, 실행 속도가 너무 짧아 유의미한 비교 데이터를 얻기가 어려웠고, 이는 구현 과정에서의 실수를 점검하며 수정하는 시간을 더 필요로 했습니다. LRU 알고리즘의 경우 실행 시간이 30초 정도로 비교적 길어 분석이 가능했지만, FIFO는 실행 시간이 너무 짧아 제대로 작동하는지 확신하기 어려웠습니다.

결국, 1단계 페이징과 2단계 페이징의 구조적 차이와 LRU 및 FIFO 알고리즘의 동작 방식을 쌍으로 묶어 비교 분석을 시도했습니다. FIFO가 제대로 작동하는지 불확실한 상태에서, LRU와의 성능 차이를 분석하는 데 집중하며 수정 작업과 보고서 작성을 병행했습니다. 다행히 보고서를 작성하면서 FIFO가 왜 그렇게 작동했는지 뒤늦게 대략적으로 파악할 수 있었지만, 이를 추가로 수정하여 심도 있게 분석하지 못한 점이 아쉬웠습니다. 다양한 알고리즘의 특성을 깊이 이해하고 비교해야 한다는 중요함을 느꼈습니다.

그리고 과제와는 별개의 이야기지만, 이번 과제를 진행하며 깃허브를 통해 코드를 업로드하는 과정에서 작성 중이던 코드가 날아가는 일을 겪었습니다. 기존 코드 파일에 새로운 코드가 덮어씌워진 것으로 보였고, 최종 제출 직전에 발생해 당황스러웠습니다. 로그와 백업 파일을 찾아보았지만, 깃허브에는 중간 과정이 업로드되지 않았고, 우분투 환경에서도 파일 덮어씌움에 대한 기록은 없었습니다.
다행히 메모장에 복사해둔 코드 덕분에 완성본은 복구했지만, 중간 과정과 수도 코드는 잃어버렸습니다. 이번 일을 통해 작업 중에도 주기적으로 저장하거나 백업 파일을 생성하는 습관의 중요성을 깨달았습니다. 앞으로는 이런 실수를 방지하기 위해 체계적인 작업 환경과 백업 방식을 마련해야겠다고 생각했습니다.

9. 참고자료

* 페이징과 페이지 테이블 개념 및 사진
https://ddongwon.tistory.com/49

* TLB 개념 및 사진
https://yeongjaekong.tistory.com/77?category=1171757

* 테이블 기법(TLB) & 페이징(멀티-레벨 / hashed / inverted page table)
https://resilient-923.tistory.com/390

* 페이지 교체 알고리즘 (LRU & FIFO)
https://code-lab1.tistory.com/60#google_vignette

* 페이지 폴트
https://yamyam-spaghetti.tistory.com/116

* Demand paging
https://ddongwon.tistory.com/51?category=771908

* 다단계 페이징
https://velog.io/@holicme7/OS%EB%A9%94%EB%AA%A8%EB%A6%AC%EA%B4%80%EB%A6%AC5-%EB%8B%A4%EB%8B%A8%EA%B3%84-%ED%8E%98%EC%9D%B4%EC%A7%80-%ED%85%8C%EC%9D%B4%EB%B8%94

* Copy-on-Write
https://hidemasa.tistory.com/57