
REPORT



Project#02 – Virtual Memory

Freedays: 0 (4 used)

과목명	운영체제(MS)	담당교수	유시환 교수님
학 번	32204292, 32217027	전 공	모바일시스템공학과
이 름	조민혁, 김도익	제 출 일	2024/12/04

목 차

1. Introduction	1
2. Requirements	2
3. Concepts	3
3-1. Virtual Memory	3
3-2. Paging	4
3-3. Page table	5
3-4. MMU	6
3-5. Demand Paging	6
3-6. Page Fault	8
3-7. Free Frame List	9
3-8. Swapping	10
3-8-1. FIFO	11
3-8-2. LRU	11
3-9. Two Level Paging	11
3-10. COW	12
4. Implements	14
4-1. Idea for Implementation	14
4-2. Implement Virtual Memory Program	15
4-2-1. Build Environment	15
4-2-2. virtual_memory.h	16
4-2-3. Initialize Part	17
4-2-4. Memory Management Unit	19
4-2-5. Handle Page Fault Function	20
4-2-6. Swapping Function	21
4-2-7. Replacement Function	22
4-2-8. Copy Page Function (Copy on write)	24
4-2-9. Disk Function	25
4-2-10. Demand Paging Function	28
5. Results	30
5-1. Log File	30
5-2. Disk File	31
5-3. Disk File Based Result	31
5-4. Disk Array Based Result	32
6. Evaluation	33
7. Conclusion	36

1. Introduction

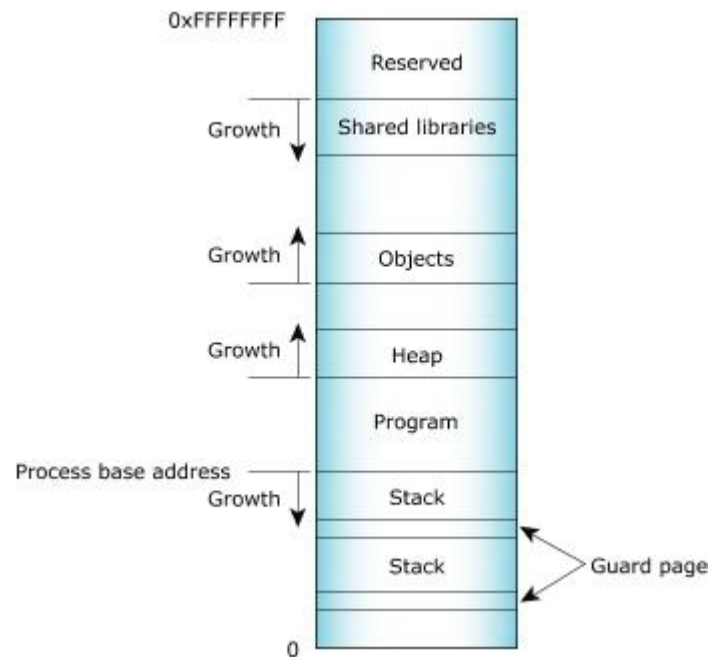


Fig 1. Virtual Memory Layout

OS에서는 다중 프로세스를 지원하기 위해 물리 메모리에 프로세스를 적재하고, CPU 스케줄링을 통해 프로세스에 CPU를 할당하여 작업을 수행한다. 이전 프로젝트에서는 프로세스 스케줄링을 구현함으로써 프로세스간 자원관리를 효율적으로 수행하였다. 그러나 물리 메모리의 크기보다 프로세스의 크기가 더 큰 경우, 해당 프로세스는 실행이 될 수 없는 문제가 발생할 수 있다.

이러한 문제를 해결하기 위해서 가상 메모리 기술이 사용된다. 가상 메모리란 컴퓨터 시스템에서 각 프로세스가 물리 메모리 전체를 소유하고 있는 것처럼 보이게 만든다. 이를 통해 프로그램은 물리 메모리의 한계를 고려하지 않고 작성될 수 있고, 프로세스간 메모리 침범을 방지해 보안을 강화한다.

본 레포트에서는 이전에 프로젝트의 스케줄링 시스템에서 페이징을 사용하여 가상 메모리 관리 기법을 추가하여 다중 프로세스 환경에서 메모리 관리 과정을 시뮬레이션 한다. 이를 통해 다중 프로세스 환경에서 메모리 관리의 중요성과 페이징 기반 가상 메모리를 이해하며, 안정적이고 효율적인 성능을 가진 가상 메모리 시스템의 설계 및 구현을 설명한다.

2. Requirements

Table 1. Requirements Table

Level	Index	Requirements
Basic	1	1개의 부모 프로세스는 커널 역할, 10개의 자식 프로세스는 프로세스 역할 수행
	2	VA->PA 변환은 OS에서 올바르게 수행
	3	OS가 런타임 중 페이지 테이블을 업데이트한다.
	4	OS 초기화 시 물리 메모리는 페이지 크기로 단편화 및 Free Page Frame List 유지한다.
	5	OS는 각 프로세스에 대해 페이지 테이블 유지한다.
	6	Tick 마다 프로세스는 10개의 메모리 주소에 접근하도록 한다.
	7	페이지 테이블의 유효성 검사하도록 한다.
	8	가상 주소는 페이지 넘버와 Offset으로 분리된다.
	9	MMU는 페이지 테이블 시작 주소를 가르킨다.
Extra Option	10	2단계 페이징을 구현한다.
	11	교체 알고리즘 정책을 통해 스와핑을 구현한다.
	12	메모리 변경 요청 발생 시 새로운 PA를 할당하는 Copy on Write를 구현한다.
Output	13	1,0000 Tick 초과 시 프로그램을 종료하며 각 Tick 마다의 로그를 작성한다. (VA, PA, 페이지 폴트, 페이지 테이블 변화, 읽기/쓰기 값 기록)

Table 1에 가상 메모리 프로그램을 구현하는 데 있어 필요한 요구사항들이 제시 되어있다.

3. Concepts

본 장에서는 해당 프로그램을 구현하는데 있어서 필요한 관련 개념들을 서술한다.

3-1. Virtual Memory

가상 메모리는 물리 메모리보다 더 많은 메모리에 액세스하는 것처럼 프로세스에 제공하는 기술이다. 이를 달성하기 위해 하드웨어의 디스크의 일부분을 가상 메모리 공간으로 사용하여 메인 메모리의 크기보다 더 큰 프로그램을 실행하거나 여러 프로세스에게 각각 독립된 가상 주소 공간을 할당하여 다중 프로세스를 효율적으로 진행시킬 수 있도록 한다. 사용자는 하나의 연속된 메모리를 사용하는 것처럼 느끼지만, 이는 실제로 가상 주소와 물리 주소간 변환으로 인해 가능한 일이다. 이를 통해서 메모리 주소 공간이 분리되므로, 프로세스 간의 메모리 보호 또한 가능하다.

그러나 가상 메모리에 단점도 존재한다. 논리적 주소와 메인 주소를 매핑하는 페이지 테이블에서 메인 메모리에 없는 페이지에 접근하게 되면 디스크에서 페이지를 가져오는 동안 지연이 발생하는데 이것을 페이지 폴트라고 한다. 이러한 지연은 성능 저하의 원인이 된다. 따라서 가상 메모리 관리 기법을 활용하여 페이지 폴트를 최소화하고, 적절히 처리하는 것이 중요하다.

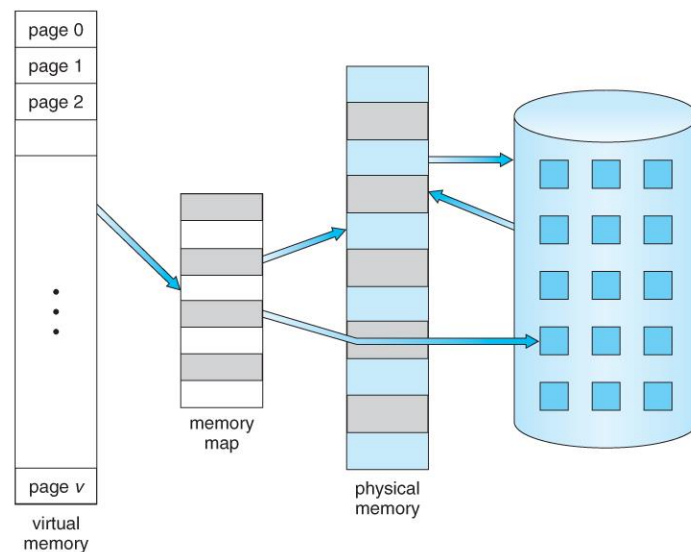


Fig 2. Virtual Memory Process

아래에서 가상 메모리의 전체적인 동작 흐름을 간략히 설명한다.

1. **가상메모리 초기 설정:** OS는 부팅과 동시에 각 프로세스마다 논리적 주소 공간과 페이지 테이블을 생성한다.
2. **프로세스 실행 및 가상 주소 생성:** CPU는 각 프로세스의 논리적 주소 공간에 접근하여 가상 주소를 생성한다. 가상 주소는 Page Number와 Offset으로 구성되어 있다.
 - A. Page Number: 가상 주소의 상위 12비트를 사용하여 페이지 테이블의 인덱스로 사용된다.

- B. Offset: 물리 메모리에서 정확한 위치를 나타낸다.
3. **주소변환**: MMU는 페이지 테이블을 참조하여 해당 항목이 유효 상태일 경우, 물리 주소를 계산하고, 유효 하지 않다면, 페이지 폴트를 발생시킨다.
 4. **페이지 폴트 처리**: 페이지 폴트가 발생했다면 OS는 페이지 폴트를 적절히 처리한다.
 5. **프로세스 종료**: 프로세스가 종료되면, OS는 해당 프로세스의 페이지 테이블과 메모리 프레임을 해제하고, 프리 페이지 리스트로 반환하여 다른 프로세스가 사용할 수 있도록 한다.

3-2. Paging

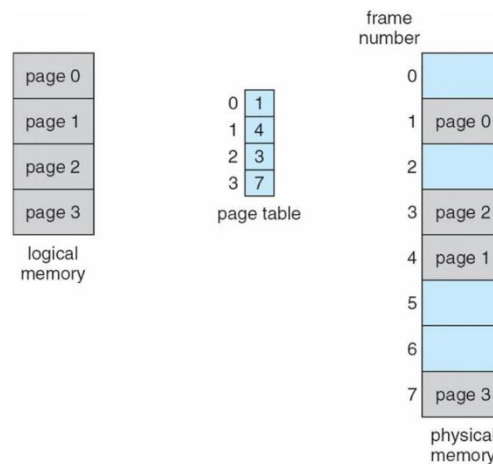


Fig 3. Logical to Physical Memory using a Page Table

가상 메모리에서 OS 는 프로세스를 생성할 때 논리 메모리라는 가상 주소 공간을 할당한다. 이 논리 메모리는 실제 물리 메모리와는 독립적으로 작동한다. 가상 메모리 시스템에서는 Fig 3 과 같이 논리 메모리를 일정한 크기 단위인 페이지로 나누고, 물리 메모리는 페이지와 동일한 크기와 동일한 단위로 프레임을 나눈다. 이렇게 고정된 크기로 메모리를 분할하여 매핑하는 방식을 페이지징이라고 한다.

가상메모리에서는 Paging 방식을 통해 메모리를 동일한 크기로 분할하고, 이곳에 가상 메모리를 매핑하여 불연속적으로 할당한다. 이를 통해 메모리 공간이 연속적이지 않아서 실행되지 못하는 외부 단편화 문제를 효과적으로 해결할 수 있다는 장점이 있다. 외부 단편화란 프로세스가 필요로 하는 메모리 공간 보다 비어 있는 메모리 공간이 크지만, 비어 있는 공간이 연속적이지 않아 프로세스를 할당할 수 없어 메모리 공간이 낭비되는 현상을 말한다.

그러나 페이지징에도 단점이 있다. 물리 메모리를 나눈 프레임 중 마지막 프레임이 메모리 크기와 정확히 일치하지 않는 경우, 페이지 크기보다 작게 사용되는 메모리 공간이 낭비되는 내부 단편화가 발생할 수 있다. 내부 단편화는 외부 단편화보다 메모리 낭비가 적지만, 여전히 성능에 영향을 미칠 수 있다. 따라서, 적절한 페이지 크기 설정과 메모리 관리 기법을 통해 이러한 단점을 최소화하는 것이 중요하다.

3-3. Page Table

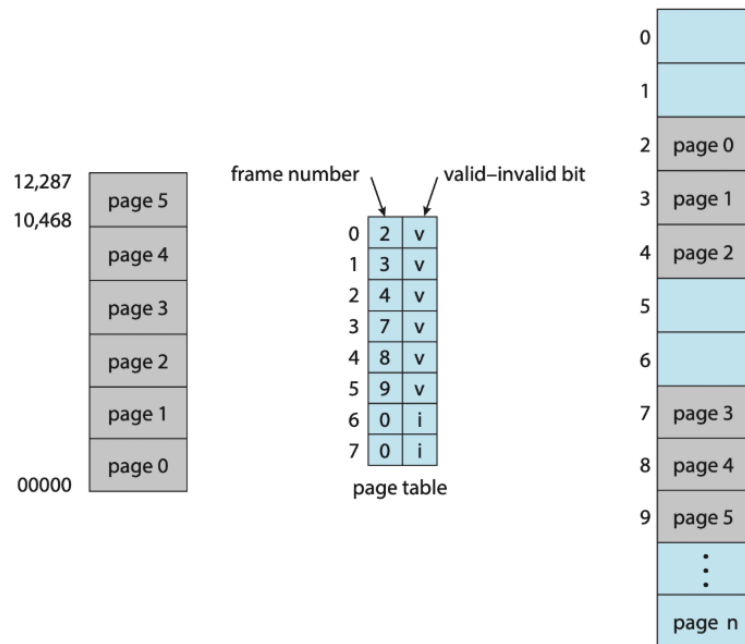


Fig 4. Page Table Process Diagram

페이징 기법은 논리 메모리와 물리 메모리를 매핑해주는 메모리 관리 기법이다. 페이지 단위로 분할된 논리 메모리와 프레임 단위로 분할된 프레임을 매핑하기 위해 필요한 방법이 바로 페이지 테이블이다. 페이지 테이블은 가상 주소와 물리주소를 연결하는 역할을 하며, OS 가 각 프로세스마다 독립적으로 생성하여 관리한다.

페이지 테이블은 논리 메모리의 페이지 번호를 물리 메모리의 프레임 번호와 매핑하는 구조를 가진다. 페이지 테이블의 인덱스는 페이지 번호를 나타내며 각 항목에는 물리 메모리의 프레임 번호와 함께 유효 비트가 저장된다. 유효 비트는 페이지 상태를 나타내는 비트이며, 해당 페이지가 물리 메모리에 존재한다면, Valid 존재하지 않는다면 Invalid 로 설정된다. 페이지 테이블은 이러한 구조를 통해 현재 물리 메모리에 어떤 페이지가 적재되었는지 확인할 수 있다.

이처럼 페이지 테이블은 논리 주소를 물리 주소로 변환하는 데 핵심적인 역할을 하며, 논리 메모리를 페이지 단위로 나누고, 물리 메모리를 프레임 단위로 관리함으로써 가상 메모리 시스템의 효율성을 높인다. 페이지 테이블을 참조해 MMU(Memory Management Unit) 가 논리 주소를 물리 주소로 변환하며, 이를 통해 프로세스가 연속적이지 않은 물리 메모리에서도 동작할 수 있도록 지원한다.

결론적으로, 페이지 테이블은 가상 메모리의 원활한 작동을 위해 논리 메모리와 물리 메모리를 연결하고, 독립적인 메모리 관리를 가능하게 하는 필수적이다.

3-4. MMU(Memory Management Unit)

MMU 는 CPU 와 메모리 사이에서 CPU 가 생성한 가상주소(VA)를 물리주소(PA)로 변환하는 역할을 하는 하드웨어 장치이다. 가상 주소는 논리 메모리 페이지 번호와 offset 으로 구성된다. MMU 의 주소 변환 과정은 다음과 같다.

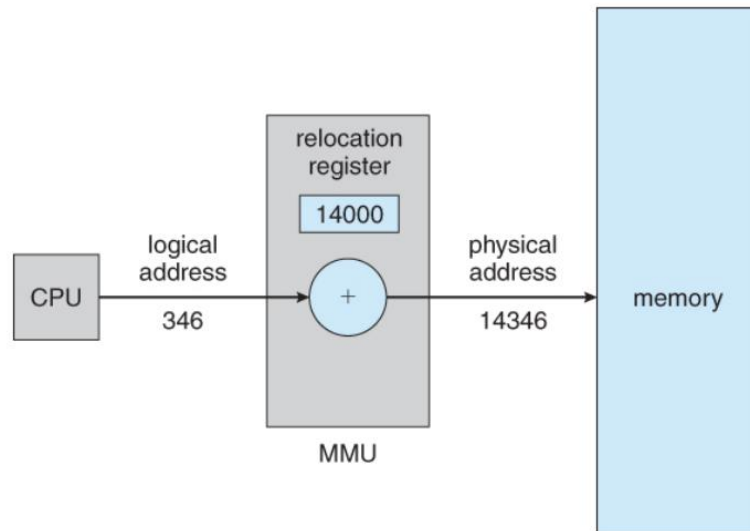


Fig 5. MMU

1. 가상 주소 분리: MMU 는 CPU 가 생성한 가상 주소를 받아 논리 메모리 페이지 번호와 offset 으로 분리한다.
2. 페이지 테이블 조회: MMU 는 페이지 테이블을 참조하여, 논리 메모리 페이지 번호에 매핑된 물리 프레임 번호를 확인한다.
3. 그와 매핑 되어있는 물리 메모리 프레임 번호를 확인한다.
4. 물리 프레임 번호에 맞는 프레임을 찾아 offset 을 더하고, 그 값을 가져온다.

MMU 는 주소 변환의 속도를 높이기 위해서 TLB(Translation Lookaside Buffer) 라는 고속 캐시를 사용할 수 있다. TLB 는 자주 사용되는 페이지의 매핑 정보를 저장하여, 페이지 테이블을 조회하지 않고도 물리주소를 빠르게 찾을 수 있다.

MMU 는 페이지 테이블에서 요청된 페이지가 물리 메모리에 없으면, 페이지 폴트를 감지하고, 이를 OS 에 알린다. 이를 감지한 OS 는 페이지 폴트처리를 적절하게 한다.

3-5. Demand Paging

앞서 살펴본 페이지징은 가상 메모리를 물리 메모리로 매핑해주는 메모리 관리 기법이다. 하지만 이 방식은 모든 페이지를 미리 물리 메모리에 적재하기 때문에 메모리 사용이 비효율적일 수 있다. 즉, 프로그램 실행 시 실제로 필요하지 않은 페이지까지 메모리에 적재되면서 불필요하게 메모리 사용량이 증가하고, 이로 인해 성능 저하가 발생할 가능성이 있다.

이러한 페이징의 비효율성을 해결하기 위해 Demand Paging 이 도입되었다. Demand Paging 은 페이징의 기본적인 구조를 따르면서, 프로세스가 필요로 하는 페이지만 물리 메모리에 동적으로 적재하여, 접근되지 않은 페이지는 물리 메모리에 적재하지 않고, 보조 메모리인 디스크에 적재하여 메모리의 자원을 효율적으로 관리하는 기법이다.

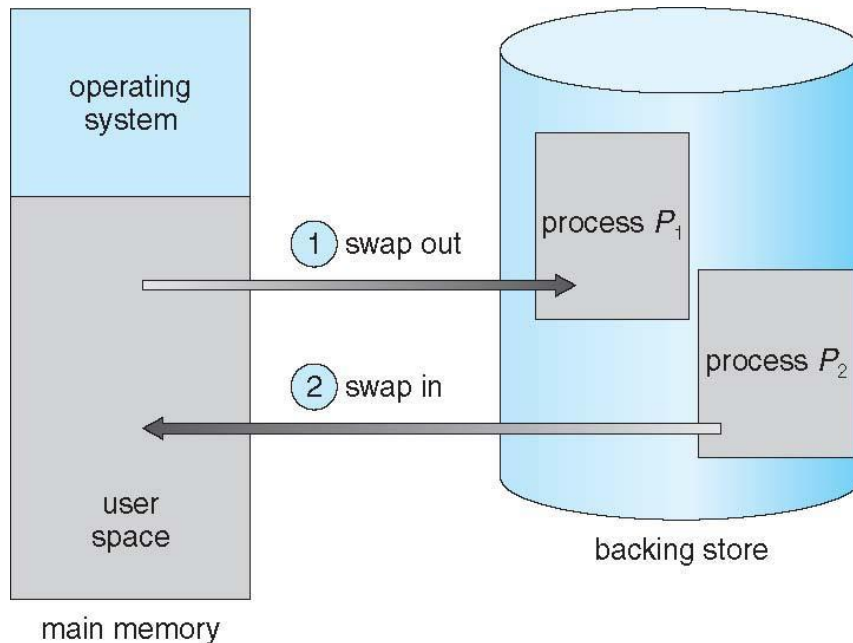


Fig 6. Demand Paging Process

CPU 가 가상 주소를 생성할 때, 해당 주소의 페이지가 물리 메모리에 저장되어 있지 않다면 Page Fault 가 일어나게 된다. 이때 페이지 테이블의 유효비트를 확인하여 페이지가 물리 메모리에 없으면 Invalid 상태로 간주되고, OS 가 디스크에서 해당 페이지를 가져와 메모리에 적재한다. 이러한 방식으로 실제 필요한 페이지만 물리 메모리에 적재하고, 물리 메모리에 없다면 디스크에 가져오는 방식으로 메모리 낭비를 줄인다.

Demand Paging 에서 페이지 폴트가 발생하면 디스크 I/O 작업이 필요하게 된다. 이 경우, 운영체제(OS)는 실행 중인 프로세스를 일시 중단(interrupt)하고, 디스크 I/O 작업이 완료될 때까지 대기한다. 작업이 완료되면 OS 는 페이지를 메모리에 적재한 뒤, 중단된 프로세스를 다시 재개한다. 그러나 페이지 폴트가 빈번하게 발생하는 경우, 디스크 I/O 가 과도하게 증가하여 시스템 성능이 크게 저하될 수 있다.

3-6. Page Fault

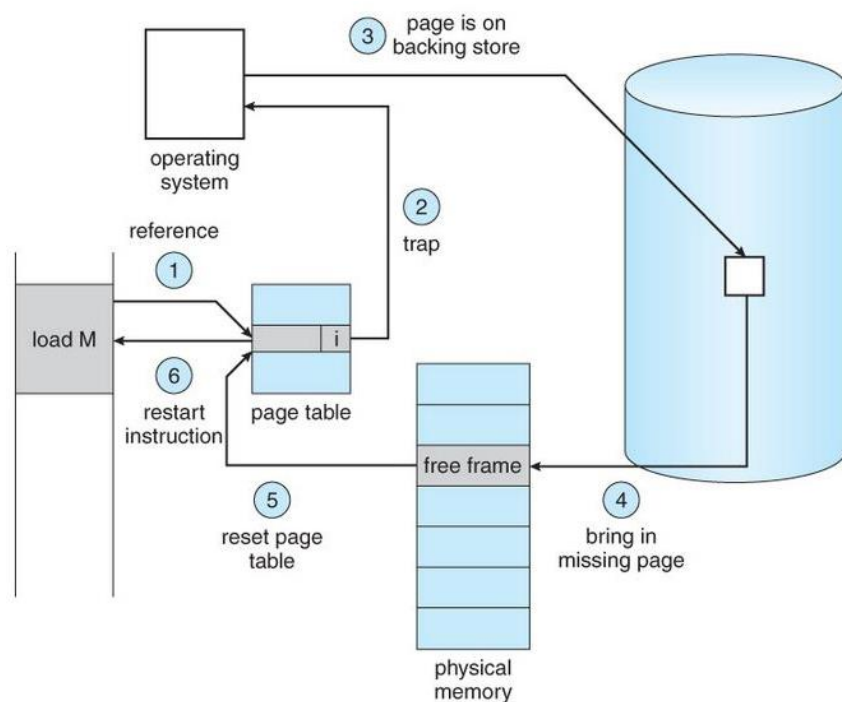


Fig 7. Handling Process of Page Fault

요구 페이징은 필요한 페이지만 물리 메모리에 적재하는 기법이다. 따라서 프로세스 실행 중 필요한 페이지는 물리 메모리에 적재 되어있고, 필요하지 않았던 페이지는 보조 기억 장치인 디스크에 적재 되어있다. 페이지 fault 는 페이지 테이블에서 매핑되어 있는 논리 메모리가 물리 메모리에 없을 때 발생한다. 즉, 필요한 페이지의 유효 비트가 Invalid 인 경우 발생하는 것이다. 페이지 폴트를 처리하는 과정은 Fig 7 과 같다.

1. MMU 가 페이지 테이블에 접근하여 필요한 페이지의 유효 비트를 통해 물리 메모리에 해당 페이지가 존재하는지 확인한다.
2. 유효 비트가 Invalid 로 설정되어 있다면 이를 OS 에게 알리고, OS 는 현재 실행중인 프로세스를 인터럽트를 발생시켜 멈춘다.
3. 페이지가 저장 되어있는 보조 기억 장치의 해당 페이지 주소로 접근한다.
4. 보조 기억 장치에서 해당하는 페이지를 비어 있는 물리 메모리 공간에 적재한다.
5. 해당 페이지가 물리 메모리에 적재되었으니 프로세스의 페이지 테이블의 유효비트를 Valid 로 재설정한다.
6. 해당 프레임을 가지고 오기 위해 멈춰 있었던 프로세스를 재개한다.

페이지 폴트는 가상 메모리 시스템의 성능에 중요한 영향을 미친다. 페이지 폴트는 보조 기억 장치에서 수행하는 작업에서 디스크 I/O 가 발생하고, 이는 물리 메모리에 접근하는 시간에 비해 훨씬 오래 걸리므로, 빈번한 페이지 폴트는 전체 시스템의 성능을 저하시킨다. 특히 페이지 폴트가 빈번하게 일어나면 프로세스들이 디스크와 메모리 간 데이터를 교환할 수 없게 되는 Thrashing 현상이 발생할 수 있다. 이를

방지하기 위해서는 페이지 교체알고리즘을 적절히 사용하거나, TLB 와 같은 고속 캐시를 사용하여 페이지 폴트를 줄이는 방법이 필요하다.

3-7. Free Frame List



Fig 8. Example of free frame list

Page Fault 가 일어나게 되면 OS 는 필요한 페이지를 보조 저장 장치에서 가져와 물리 메모리에 적재해야 한다. 이때 자유 프레임 리스트는 페이지 폴트가 일어났을 때 물리 메모리에서 사용 가능한 프레임을 찾기 위해서 사용되는 프레임 풀이다. OS 는 페이지 폴트가 발생하면 자유 프레임 리스트를 참조하여 비어 있는 물리 메모리의 프레임을 확인한다. 비어있는 곳이 있으면 보조 저장 장치에 접근하여 프레임에 적재하고, 비어 있는 곳이 없다면 스와핑이나 페이지 교체알고리즘을 사용해 프레임을 확보한다.

자유 프레임 리스트의 구조는 보통 간단하거나 효율적인 방식으로 구현된다. 일반적으로 Linked List 나 스택을 사용하여 Free Frame 을 관리하며, Fig 8 과 같이 각 노드나 스택의 항목은 사용 가능한 프레임의 정보를 포함한다.

두 가지 구현 방식의 장단점은 다음과 같다.

1. Linked List

동적인 크기의 리스트를 지원하므로 메모리 자원을 유연하게 관리할 수 있고, 삭제와 삽입이 효율적이며, 어느 위치이든 쉽게 연결할 수 있다. 하지만 특정 프레임을 찾으려면 리스트를 순차적으로 탐색해야 하므로 탐색 속도가 느릴 수 있다.

2. 스택

구조가 단순하며, 삽입과 삭제가 매우 빠르다. LIFO 방식이 특정 상황에서는 유리할 수 있다. 그러나 스택의 구조상 특정 프레임에 접근하거나 관리하는 것이 제한적이다. 또한 free frame 이 많아지면 관리가 어려워질 수 있고, 메모리 사용이 비효율적일 수 있다.

본 레포트에서는 빠른 접근 속도와 단순한 구조를 우선시하여 배열을 사용해 자유 프레임 리스트를 구현하였다. 배열은 고정된 크기의 메모리를 관리할 수 있으며 특정 프레임에 빠른 속도로 접근할 수 있다는 장점이 있다.

3-8 Swapping

앞서 페이지 폴트 섹션에서 CPU 가 필요로 하는 프레임이 메인 메모리에 존재하지 않을 때 페이지 폴트가 일어나고 해당하는 프레임을 보조 기억장치에서 찾아서 비어 있는 물리 메모리공간에 적재한다고 하였다. 하지만 이런 과정은 물리 메모리가 비어 있을 때 가능한 일이다. 물리 메모리 프레임에 빈공간이 존재하지 않는다면 빈자리를 만들고 필요한 페이지를 적재해야 할 것이다. 이것이 스와핑 기법이다.

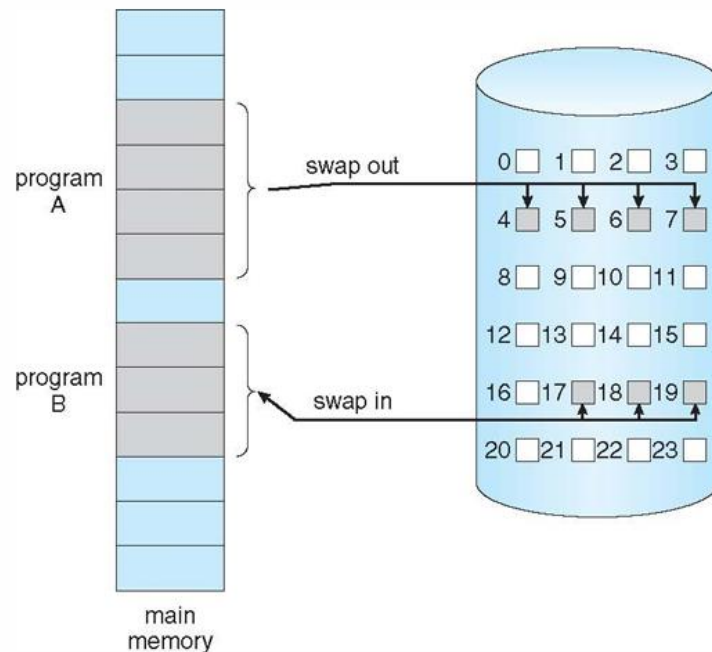


Fig 9. Swapping

스와핑은 Fig 9 와 같이 CPU 가 필요로 하는 프레임을 보조기억장치에서 찾았는데 해당 프레임을 적재할 물리 메모리의 공간이 없다면, 페이지 교체 알고리즘에 의해 물리 메모리에 적재 되어있던 프레임을 보조 기억장치로 쓰고, CPU 가 필요로 하는 프레임을 새로 물리 메모리에 적재한다. 여기서 기존에 물리메모리에 있던 것을 보조기억 장치로 보내는 행위를 swap out, 보조 기억장치에 있던 프레임을 물리 메모리로 가져오는 것을 swap in 이라고 한다. 이를 통해 필요한 페이지를 우선적으로 물리 메모리에 유지하고, 사용하지 않는 페이지를 디스크로 이동하여 메모리 공간을 최적화하고 페이지 폴트를 최소화할 수 있다.

스와핑은 물리 메모리 공간이 부족할 때 효과적인 메모리 관리 기법이지만, 디스크 I/O 가 빈번하게 발생하므로 성능에 영향을 미칠 수 있다. 따라서 적절한 페이지 교체 알고리즘을 선택하여 디스크 접근을 최소화하고 시스템 효율을 극대화해야 한다. 본 레포트에서 사용한 페이지 교체 알고리즘은 다음과 같다.

3-8-1. FIFO(First-In-First-Out)

페이지 교체 알고리즘 중에 가장 간단하고 구현이 쉬운 방식이다. 이 알고리즘은 가장 먼저 물리 메모리에 적재된 페이지를 교체하는 정책을 따른다. 페이지가 물리 메모리에 들어온 순서를 기준으로, 오래된 페이지를 먼저 제거한다. FIFO 는 페이지가 메모리에 적재된 순서를 기록하기 위해서 큐 자료구조만 사용하면 되기 때문에 구현이 쉽고, 페이지를 삽입하거나 삭제하는 것이 간단하다.

하지만 FIFO 는 구조적인 한계에서 비롯되는 단점이 존재한다. FIFO 는 페이지가 물리메모리에 적재된 순서만을 고려하기 때문에 오래된 페이지가 자주 참조되어 집에도 불구하고 교체될 수 있다. 또 새로 들어온 페이지를 유지하려는 특성상 필요한 페이지가 먼저 교체되고 불필요한 페이지가 남아 있을 수 있다. 이것을 Belady's anomaly 라고 한다. 따라서 메모리 프레임수가 증가함에도 불구하고 페이지 폴트율이 증가하는 상황이 발생할 수 있기 때문에 성능적인 측면에서 항상 좋지는 않다.

3-8-2. LRU(Least Recently Used)

최소시간 페이지 교체 정책은 물리 메모리안의 페이지 프레임 중 CPU 가 가장 오랫동안 접근하지 않은 페이지를 교체하는 방식이다. LRU 는 자주 사용되는 페이지를 물리 메모리에 유지할 수 있다. 따라서 페이지 폴트율이 낮다. 또 페이지 참조 이력을 기반으로 교체할 페이지를 선정하므로 메모리 프레임 수 증가에도 페이지 폴트율이 증가하는 Belady's anomaly 현상이 일어나지 않는다. 하지만 페이지마다 마지막으로 참조된 시간을 기록하고 관리하기 때문에 시간 복잡도가 증가하고, 규모가 큰 시스템에서 성능 저하로 이어질 수 있다.

3-9. Two-Level Paging

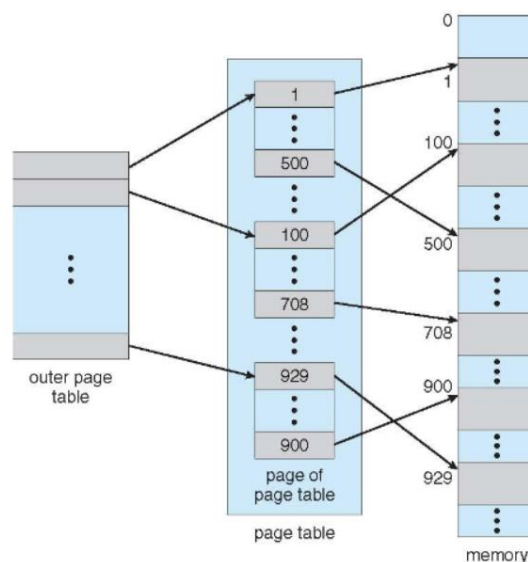


Fig 10. Two level paging

2 단계 페이징은 가상 메모리에서 페이지 테이블의 크기를 줄이고 메모리를 효율적으로 관리하기 위해서 사용하는 메모리 관리기법이다. 가상 주소의 크기가 커질수록 페이지 테이블의 크기는 커지게 되고, 이는 메모리의 낭비로 이어질 수 있다. 예를 들면 32 비트 주소 공간에 페이지 크기가 4KB 일 경우, 총 2^{20} 개의 페이지가 필요하게 되고, 페이지 테이블 크기도 이에 비례해 증가한다. 2 단계 페이징은 이러한 문제를 해결하기 위해서 페이지 테이블을 상위 테이블과 하위테이블로 분리하여 필요한 부분만 메모리에 적재하도록 설계한 것이 2 단계 페이징이다.

2 단계 페이징에서는 가상 주소를 세 부분으로 나눈다.

1. **상위 페이지 번호:** 1 단계 페이지 테이블의 인덱스로 사용된다. 해당 항목에는 2 단계 페이지 테이블의 시작주소가 저장된다
2. **하위 페이지 번호:** 상위 페이지 테이블에서 가져온 2 단계 페이지 테이블의 인덱스로 사용된다. 해당 항목에는 물리 메모리 프레임이 저장된다.
3. **오프셋:** 프레임 내에서 데이터의 정확한 위치를 나타내기 위해 사용된다.

2 단계 페이징에서는 CPU 에서 가상주소를 가져와 세부분으로 나눈다. Fig 10 과 같이 상위 페이지 번호를 이용해 1 단계 페이지 테이블을 조회하고 해당 항목의 2 단계 페이지 테이블의 시작 주소를 가져온다. 하위 페이지 테이블을 조회하여 물리 메모리의 프레임 번호를 가져오고, 프레임 번호와 페이지 오프셋을 결합하여 물리 주소를 계산하고 데이터에 접근하게 된다.

이 방식은 테이블을 계층적으로 나누어 필요한 부분만 메모리에 적재할 수 있기 때문에 메모리를 효율적으로 사용할 수 있다. 그러나 두 단계의 테이블을 참조하기 때문에 주소 변환 속도가 느려질 수 있으며, 이를 보완하기 위해서 TLB 가 같이 사용된다.

3-10. COW(Copy On Write)

Copy On Write 는 프로세스간 메모리 공유와 효율성을 높이기 위해 사용되는 메모리 관리 기법으로, 프로세스 생성시 메모리 낭비를 줄이기 위해서 사용된다. 가상 메모리에서 부모 프로세스는 자식을 생성하게 되는데, 자식은 생성되면서 프로세스의 메모리를 복사한다. 복사된 메모리는 즉시 수정되지 않고 단순히 읽기 작업만 수행되게 된다. 수정되지 않은 데이터를 복사하는 것은 불필요한 메모리 사용으로 인한 성능 저하로 이어질 수 있다.

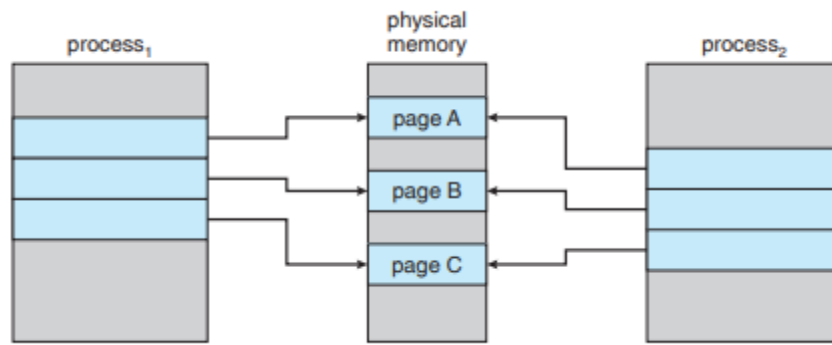


Fig 11. Before COW

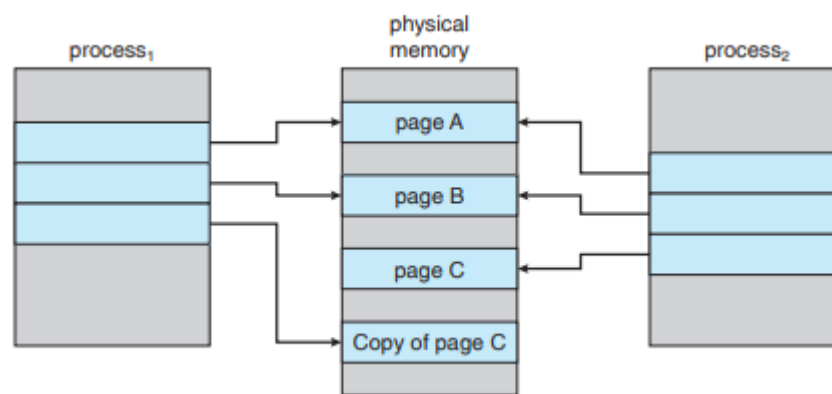


Fig 12. After COW

COW의 동작 과정을 Fig 11은 부모 프로세스와 자식 프로세스가 생성된 초기 상태이다. 이 상태에서는 Page A, Page B, Page C가 물리 메모리에 저장되어 있으며, 부모와 자식 프로세스가 동일한 메모리 페이지를 공유하고 있다. 이때 모든 메모리는 읽기 전용으로 설정되어 있으므로 두 프로세스가 메모리를 수정하지 않는다면 추가적인 메모리 복사는 불필요하다.

Fig 12는 자식 프로세스에서 Page C를 수정하려는 쓰기 작업이 발생한 상황을 나타낸다. 쓰기 작업이 시도되면, 해당 페이지는 읽기 전용 상태이기 때문에 페이지 폴트가 발생한다. 이 신호를 받은 OS는 Page C의 복사본을 물리 메모리의 새로운 공간에 생성하고, 자식 프로세스가 이 복사본을 수정할 수 있도록 설정한다. 이후 부모 프로세스는 기존의 Page C를 계속 참조하며, 자식 프로세스는 새로운 메모리 블록을 독립적으로 사용하게 된다. 이러한 방식은 데이터가 변경되는 시점에만 메모리를 복사하므로, 불필요한 메모리 사용과 시스템 자원 낭비를 방지할 수 있다.

Copy-On-Write는 메모리 효율성과 성능을 동시에 향상시키는 기법이다. 데이터가 변경되지 않는 동안 부모와 자식 프로세스는 동일한 메모리 페이지를 공유하므로 메모리 사용량을 줄일 수 있다. 또한, 데이터가 변경될 때만 필요한 페이지를 복사하여 독립적인 메모리를 제공하므로, 프로세스 간의 데이터 충돌을 방지하면서 메모리 관리의 효율성을 극대화한다. 다만, 데이터 변경이 빈번한 경우에는 복사 작업이 많아질 수 있으므로, 특정 상황에서는 성능 저하가 발생할 수 있다.

4. Implements

본 장에서는 Virtual Memory 프로그램을 구현하는데 있어서 설계한 아이디어에 대해 제시하고, 완성된 프로그램에 대한 코드를 작성한다.

4-1. Idea for Implementation

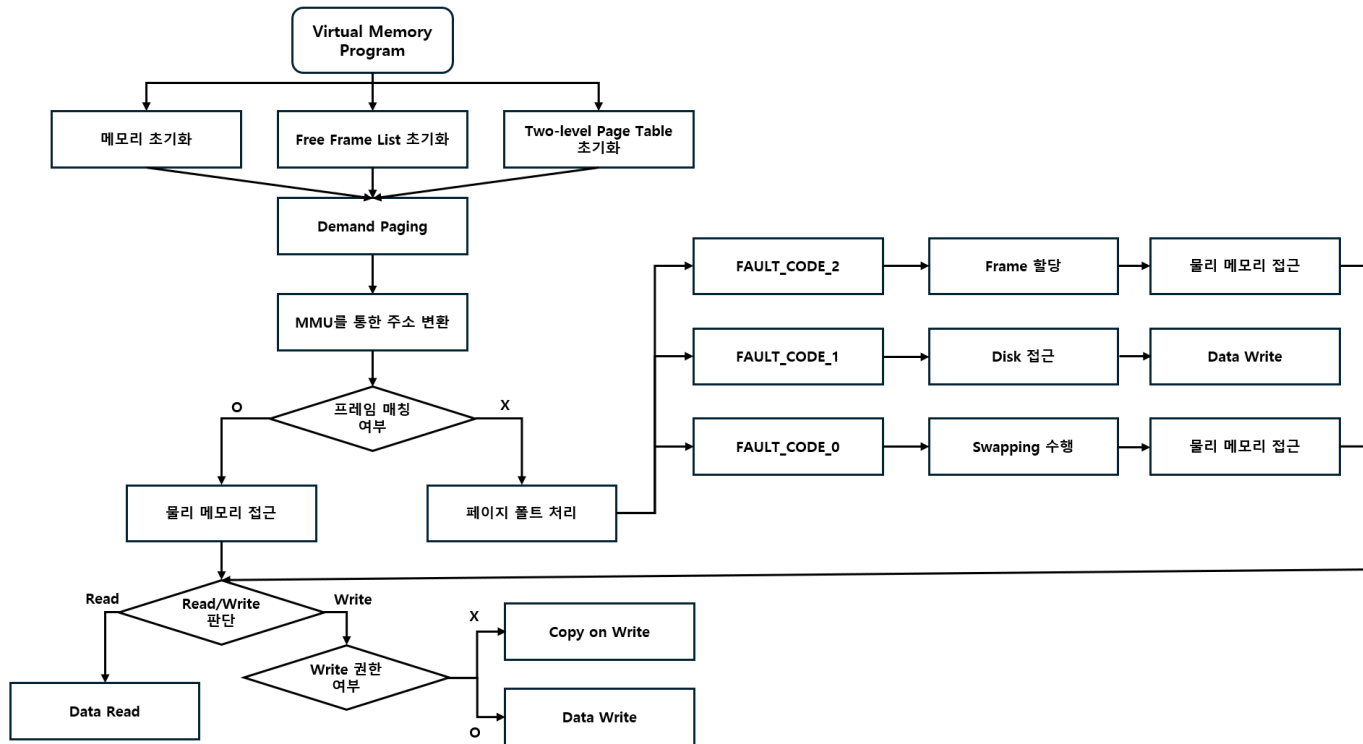


Fig 13. Process of Virtual Memory Program

Fig 13 에 본 레포트에서 구현하고자 하는 프로그램의 아이디어에 대한 순서도가 제시 되어있다. 먼저 Demand Paging 을 하기 전에 초기화를 수행한다. 이후 MMU 를 통해 주소를 변환하는데, 해당 변환 과정에서 가상 주소가 존재하는 페이지 테이블이 프레임과 매칭 되는 지를 확인한다. 이후, 매칭이 되어 있다면 물리 메모리에 접근하여 Read/Write 권한을 판단한다. 만약, Read 권한이라면 모든 페이지는 Read 권한이 부여되므로 바로 메모리에서 데이터를 읽는다. Write 권한이라면 Write 권한 존재 여부에 대해 판단하는데, 만약 존재하지 않는다면 Copy on Write 를 수행하고, 존재한다면 메모리에 데이터를 바로 작성한다. 프레임 매칭이 되지 않았더라면 페이지 폴트 처리를 하도록 한다. 페이지 폴트를 처리하면 특정 코드가 반환된다. Code 2 는 Free Frame List 의 유효 값이 존재하는데, 할당이 되지 않은 것이므로 할당한 후 물리 메모리에 접근하여 Read/Write 권한을 판단한다. Code 1 은 Disk 에 해당 가상 주소가 존재하므로 Disk 에 접근하여 데이터를 가져와 해당 페이지가 매칭 되어있었던 프레임에 바로 물리 메모리에 작성하도록 한다. Code 0 은 Free Frame List 가 존재하지 않는 경우 이므로, Swapping 후

프레임을 할당한다. 이후, Read/Write 권한을 판단하도록 한다.

4-2. Implement Program

본 절에서는 제시한 아이디어를 바탕으로 구현한 프로그램의 코드를 설명한다.

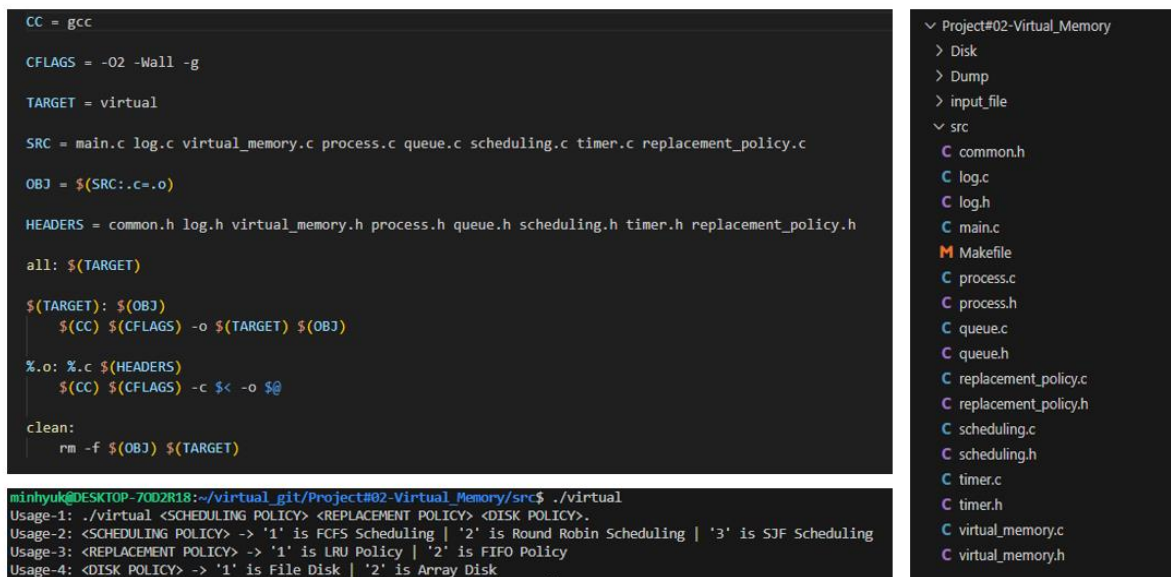
4-2-1. Build environment

먼저 구현한 코드 설명에 앞서 해당 프로그램을 개발한 IDE, 프로그램 실행하기 위한 환경은 다음과 같다.

- ✓ 개발 환경: Ubuntu 22.04, VSCode
- ✓ 개발 프로그래밍 언어: C
- ✓ 프로그램 실행 방식

STEP 1) make

STEP 2) ./virtual <SCHEDULING POLICY> <REPLACEMENT POLICY> <DISK POLICY>



```
CC = gcc

CFLAGS = -O2 -Wall -g

TARGET = virtual

SRC = main.c log.c virtual_memory.c process.c queue.c scheduling.c timer.c replacement_policy.c

OBJ = $(SRC:.c=.o)

HEADERS = common.h log.h virtual_memory.h process.h queue.h scheduling.h timer.h replacement_policy.h

all: $(TARGET)

$(TARGET): $(OBJ)
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJ)

%.o: %.c $(HEADERS)
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJ) $(TARGET)
```

```
minhyuk@DESKTOP-70D2R18:~/virtual_git/Project#02-Virtual_Memory/src$ ./virtual
Usage-1: ./virtual <SCHEDULING POLICY> <REPLACEMENT POLICY> <DISK POLICY>.
Usage-2: <SCHEDULING POLICY> -> '1' is FCFS Scheduling | '2' is Round Robin Scheduling | '3' is SJF Scheduling
Usage-3: <REPLACEMENT POLICY> -> '1' is LRU Policy | '2' is FIFO Policy
Usage-4: <DISK POLICY> -> '1' is File Disk | '2' is Array Disk
```

- Project#02-Virtual_Memory
 - > Disk
 - > Dump
 - > input_file
 - ✓ src
 - C common.h
 - C log.c
 - C log.h
 - C main.c
 - M Makefile
 - C process.c
 - C process.h
 - C queue.c
 - C queue.h
 - C replacement_policy.c
 - C replacement_policy.h
 - C scheduling.c
 - C scheduling.h
 - C timer.c
 - C timer.h
 - C virtual_memory.c
 - C virtual_memory.h

Fig 14. Build Environment

4-2-2. virtual_memory.h

```
#pragma once

#include "common.h"
#include "process.h"
#include "queue.h"
#include "replacement_policy.h"

#define PAGE_SIZE 4096
#define PHYSICAL_SIZE 0x400000
#define VIRTUAL_SIZE 0x800000
#define FRAME_NUM (PHYSICAL_SIZE / PAGE_SIZE)
#define PAGE_NUM (VIRTUAL_SIZE / PAGE_SIZE)

#define FAULT_CODE_2 2
#define FAULT_CODE_1 1
#define FAULT_CODE_0 0

typedef struct PT{
    int *matching_frame;
    bool *valid;
    bool *isRead;
    bool *isWrite;

    int *last_used;
} PageTable;

typedef struct PD{
    PageTable **page_table;
} PageDirectory;

typedef struct FFL{
    int *free_frame_list;
} FreeFrameList;
```

Fig 15. virtual_memory.h

Fig 15 에 본 레포트에서 구현하고자 하는 가상 메모리 프로그램의 헤더 파일이 제시 되어있다. 먼저 페이지 크기는 4KB 로 설정하였다. 물리 메모리 크기는 4MB 로 설정하였으며, 가상 메모리 크기는 8MB 로 설정하였다. 이후, 프레임의 수는 1024 개로 설정하였으며 페이지의 수는 2048 개로 설정하였다. 페이지 폴트 발생 시에 대한 에러 코드를 2, 1, 0 으로 설정하였다. 페이지 테이블은 프레임 매칭 정보, 페이지 유효 정보, Read/Write 권한을 설정하였으며, 추가로 last_used 를 두어 마지막 사용 시간에 대한 추적 변수를 두어 LRU Policy 가 정상적으로 작동하도록 설정하였다. 본 레포트에서는 2 단계 페이지 테이블을 통한 가상 메모리 프로그램을 구현할 것이기에 PageDirectory 구조체에 페이지 테이블을 이중 포인터로 두어 효율적으로 2 단계 페이지 테이블이 동작하도록 하였다. 마지막으로 자유 프레임 리스트를 두어 페이지 테이블과 매칭 되도록 설정하였다.

4-2-3. Initialize Part

```
void initialize_memory(int **physical_memory, int **virtual_memory){
    uint32_t buf;
    uint32_t idx = 0;

    *physical_memory = (int*)malloc(PHYSICAL_SIZE);
    *virtual_memory = (int*)malloc(VIRTUAL_SIZE);

    if(*physical_memory == NULL){
        fprintf(stderr, "Memory allocation failed for physical_memory\n");
        exit(EXIT_FAILURE);
    }
    if(*virtual_memory == NULL){
        fprintf(stderr, "Memory allocation failed for virtual_memory\n");
        exit(EXIT_FAILURE);
    }

    FILE *fp = fopen("input_file/input4.bin", "rb");
    if(fp == NULL){
        perror("Failed to open input4.bin!");
        exit(EXIT_FAILURE);
    }

    while(fread(&buf, 1, sizeof(int), fp) == 4){
        (*physical_memory)[idx / 4] = buf;
        idx += 4;
    }

    fclose(fp);
}
```

Fig 16. initialize_memory

Fig 16 에 initialize_memory 에 대한 코드가 제시 되어있다. 각 메모리가 초기화 되고 있으며, 바이너리 파일을 물리 메모리에 적재하여 이후, Memory Read 가 수행될 때 적절한 값을 읽어오도록 하였다. 추가로 메모리 정렬 기준을 맞추기 위해 4 Byte 단위로 설정하였다.

```

PageDirectory initialize_page_directory(){
    PageDirectory page_directory;
    page_directory.page_table = (PageTable**)malloc(PAGE_NUM * sizeof(PageTable *));

    if(page_directory.page_table == NULL){
        fprintf(stderr, "Memory Allocatiuon Failed for page directory\n");
        exit(EXIT_FAILURE);
    }

    for(int i = 0; i < PAGE_NUM; i++) page_directory.page_table[i] = NULL;

    return page_directory;
}

```

Fig 17. initialize_page_directory

Fig 17 에는 Page Table Level 1 에 대한 초기화가 이루어진다. 사전에 계산된 페이지 수만큼 1 Level 인덱스 수를 결정하였으며, 각 페이지 인덱스에 대한 부분을 NULL 로 초기화 하였다.

```

FreeFrameList initialize_free_frame_list(void){
    FreeFrameList free_page;

    free_page.free_frame_list = (int*)malloc(FRAME_NUM * sizeof(int));

    if(free_page.free_frame_list == NULL){
        fprintf(stderr, "Memory allocation failed for free_frame_list\n");
        exit(EXIT_FAILURE);
    }

    for(int i = 0; i < FRAME_NUM; i++) free_page.free_frame_list[i] = 1;

    return free_page;
}

```

Fig 18. initialize_free_frame_list

Fig 18 에 Free Frame List 에 대한 초기화가 이루어지는 코드가 제시 되어있다. 해당 부분 역시 사전에 정의된 프레임 수만큼 인덱스 수를 결정하였으며, 초기에는 모든 프레임이 사용 가능 상태이기에 1 로 사용 가능 여부를 할당하였다.

4-2-4. Memory Management Unit

```
int MMU(PageDirectory *page_directory, int virtual_address, bool *first_table_fault, bool *second_table_fault, bool *isRead, bool *isWrite){
    /* | FIRST TABLE IDX 10비트 | SECOND TABLE IDX 10비트 | OFFSET 12비트 */
    int dir_idx = (virtual_address >> 22) & 0x3ff;
    int table_idx = (virtual_address >> 12) & 0x3ff;
    int offset = virtual_address & 0xfff;

    if(page_directory->page_table[dir_idx] == NULL){
        *first_table_fault = true;
        *second_table_fault = true;
        return dir_idx;
    }

    if (page_directory->page_table[dir_idx]->valid == NULL || page_directory->page_table[dir_idx]->valid[table_idx] == false) {
        *first_table_fault = false;
        *second_table_fault = true;
        return table_idx;
    }

    int mmu_frame_num = page_directory->page_table[dir_idx]->matching_frame[table_idx];
    page_directory->page_table[dir_idx]->valid[table_idx] = true;
    page_directory->page_table[dir_idx]->last_used[table_idx] = ++timer;
    *isRead = page_directory->page_table[dir_idx]->isRead;
    *isWrite = page_directory->page_table[dir_idx]->isWrite;
    return (mmu_frame_num * PAGE_SIZE + offset) / 4;
}
```

Fig 19. MMU

Fig 19 에 Memory Management Unit 에 대한 코드가 제시되어 있다. 해당 코드는 먼저 Bit-Shift 를 통해 가상 주소를 분리한다. 1 단계 페이지 테이블에 대한 비트를 상위 10 bit 로 계산되며, 중간 10bit 는 2 단계 페이지 테이블에 대한 인덱스로 계산하였다. 이후 하위 12bit 는 offset 을 통해 2^{12} (4096, 4KB)로 두어 페이지마다 offset 을 통해 접근 가능하도록 하였다. 해당 코드에서는 먼저, 1 단계 페이지의 인덱스가 NULL 인지 확인한 후 처리한다. 1 단계 페이지 인덱스가 유효하다면, 2 단계 페이지 인덱스의 유효성을 검사한다. 만약 유효하다면 최종적으로 페이지 테이블 업데이트를 수행한 후 물리 주소 변환 후 반환하도록 하였다.

4-2-5. Handle Page Fault Function

```
int handle_page_fault(PageDirectory *page_directory, FreeFrameList *free_list, int virtual_address, int pid){
    int fault_dir_idx = (virtual_address >> 22) & 0x3ff;
    int fault_table_idx = (virtual_address >> 12) & 0x3ff;

    if(page_directory->page_table[fault_dir_idx] == NULL){
        page_directory->page_table[fault_dir_idx] = (PageTable *)malloc(sizeof(PageTable));

        if(page_directory->page_table[fault_dir_idx] == NULL){
            fprintf(stderr, "Page table allocation failed.\n");
            exit(EXIT_FAILURE);
        }

        page_directory->page_table[fault_dir_idx]->matching_frame = (int *)malloc(sizeof(int) * PAGE_NUM);
        page_directory->page_table[fault_dir_idx]->valid = (bool *)malloc(sizeof(bool) * PAGE_NUM);
        page_directory->page_table[fault_dir_idx]->last_used = (int *)malloc(sizeof(int) * PAGE_NUM);

        if(page_directory->page_table[fault_dir_idx]->matching_frame == NULL || page_directory->page_table[fault_dir_idx]->valid == NULL || page_directory->page_table[fault_dir_idx]->last_used == NULL){
            fprintf(stderr, "Fail to allocation Page Table Entry\n");
            exit(EXIT_FAILURE);
        }

        for(int i = 0 ; i < PAGE_NUM; i++) page_directory->page_table[fault_dir_idx]->matching_frame[i] = -1;
        for(int i = 0 ; i < PAGE_NUM; i++) page_directory->page_table[fault_dir_idx]->valid[i] = false;
        for(int i = 0 ; i < PAGE_NUM; i++) page_directory->page_table[fault_dir_idx]->last_used[i] = 0;

        page_directory->page_table[fault_dir_idx]->isRead = true;
        page_directory->page_table[fault_dir_idx]->isWrite = false;
    }

    if(page_directory->page_table[fault_dir_idx]->valid[fault_table_idx] == false){
        for(int i = 0; i < FRAME_NUM; i++){
            if(free_list->free_frame_list[i] == 1){
                page_directory->page_table[fault_dir_idx]->matching_frame[fault_table_idx] = i;
                page_directory->page_table[fault_dir_idx]->valid[fault_table_idx] = true;
                page_directory->page_table[fault_dir_idx]->last_used[fault_table_idx] = ++timer;

                free_list->free_frame_list[i] = 0;

                if(replacement_policy == 2) fifo_enqueue(fault_dir_idx, fault_table_idx);

                return FAULT_CODE_2;
            }
        }
    }

    gettimeofday(&start, NULL);
    int isFindDisk = find_from_disk(pid, fault_dir_idx, fault_table_idx);
    gettimeofday(&end, NULL);
    disk_time_ns += (end.tv_sec - start.tv_sec) * 1000000000LL +
        (end.tv_usec - start.tv_usec) * 1000;

    if(isFindDisk != -1) return FAULT_CODE_1;

    return FAULT_CODE_0;
}
```

Fig 20. Page Fault Handle

Fig 20 에 페이지 폴트를 처리하는 함수가 제시 되어있다. 먼저, 1 단계 페이지가 NULL 이라면 발생한 폴트 인덱스에서 메모리 할당이 이루어진다. 초기에는 Read 권한만 있기에 Read 권한을 설정해주고, Write 권한은 false 로 한다. 이후 사용 가능한 자유 프레임 리스트가 존재한다면 폴트가 발생한 인덱스에 할당해주고, 업데이트를 해준다. 만약 할당 가능한 프레임 리스트가 존재하지 않는다면 디스크에 저장 여부를 확인한다. 디스크에 저장되지 않으면 FAULT_CODE_0 을 반환하도록 해준다.

4-2-6. Swapping Function

```
int swapping(PageDirectory *page_directory, FreeFrameList *free_list, int virtual_address, int pid){
    int replace_dir_idx = -1;
    int replace_table_idx = -1;
    int fault_dir_idx = (virtual_address >> 22) & 0x3ff;
    int fault_table_idx = (virtual_address >> 12) & 0x3ff;

    switch (replacement_policy){
        case 1: find_lru_page(page_directory, &replace_dir_idx, &replace_table_idx); break;
        case 2: find_fifo_page(&replace_dir_idx, &replace_table_idx); break;
        default: perror("No Replacement"); break;
    }

    if(replace_dir_idx == -1 || replace_table_idx == -1){
        fprintf(stderr, "No page available to swap out!\n");
        exit(EXIT_FAILURE);
    }

    int frame_to_replace = page_directory->page_table[replace_dir_idx]->matching_frame[replace_table_idx];
    int *frame_data = &physical_memory[frame_to_replace * PAGE_SIZE / 4];

    free_list->free_frame_list[frame_to_replace] = 1;

    page_directory->page_table[fault_dir_idx]->matching_frame[fault_table_idx] = frame_to_replace;
    page_directory->page_table[fault_dir_idx]->valid[fault_table_idx] = true;
    page_directory->page_table[fault_dir_idx]->last_used[fault_table_idx] = ++timer;
    page_directory->page_table[fault_dir_idx]->isRead = true;
    page_directory->page_table[fault_dir_idx]->isWrite = false;

    if(replacement_policy == 2) fifo_enqueue(fault_dir_idx, fault_table_idx);

    gettimeofday(&start, NULL);
    write_page_to_disk(page_directory, free_list, pid, replace_dir_idx, replace_table_idx, frame_data);
    gettimeofday(&end, NULL);
    disk_time_ns += (end.tv_sec - start.tv_sec) * 1000000000LL +
                   (end.tv_usec - start.tv_usec) * 1000;

    return frame_to_replace;
}
```

Fig 21. Swapping

Fig 21 에 Swapping Function 에 대한 코드가 제시 되어있다. 먼저 인자로 입력된 교체 정책으로부터 교체 알고리즘을 선택한다. 이후 교체될 인덱스를 반환 받은 후 해당 페이지 테이블의 정보로 업데이트 해준다. 이후 교체될 페이지는 디스크로 이동한다.

4-2-7. Replacement Function

```
void lru_replacement(PageDirectory *page_directory, int *replace_dir_idx, int *replace_table_idx){
    int oldest_time = INT_MAX;
    bool page_found = false;

    for(int dir_idx = 0; dir_idx < 2; dir_idx++){
        PageTable *page_table = page_directory->page_table[dir_idx];
        if(page_table == NULL) continue;

        for(int table_idx = 0; table_idx < PAGE_NUM; table_idx++){
            if(page_table->valid[table_idx]){
                if(page_table->last_used[table_idx] < oldest_time){
                    oldest_time = page_table->last_used[table_idx];
                    *replace_dir_idx = dir_idx;
                    *replace_table_idx = table_idx;
                    page_found = true;
                }
            }
        }
    }

    if(!page_found){
        *replace_dir_idx = -1;
        *replace_table_idx = -1;
    }
}

void find_lru_page(PageDirectory *page_directory, int *replace_dir_idx, int *replace_table_idx){
    lru_replacement(page_directory, replace_dir_idx, replace_table_idx);
}
```

Fig 22. Replacement Function – LRU Policy

Fig 22 에 LRU Policy 에 대한 코드가 제시 되어있다. 해당 알고리즘은 기존에 접근 시간 추적을 위한 변수를 통해 접근 시간이 가장 오래된 페이지를 찾아 해당 페이지의 인덱스를 설정해준다. 만약 존재하지 않는다면 -1 로 인덱스를 설정하여 오류 처리가 되도록 하였다.


```

void fifo_dequeue(int *dir_idx, int *table_idx){
    if (is_fifo_empty()) return;

    *dir_idx = fifo_queue.dir_index[fifo_queue.front];
    *table_idx = fifo_queue.table_index[fifo_queue.front];
    fifo_queue.front = (fifo_queue.front + 1) % FIFO_CAPACITY;
    fifo_queue.size--;

    if (fifo_queue.size == 0){
        fifo_queue.front = -1;
        fifo_queue.rear = -1;
    }
}

void find_fifo_page(int *replace_dir_idx, int *replace_table_idx){
    if(is_fifo_empty()){
        *replace_dir_idx = -1;
        *replace_table_idx = -1;
        return;
    }

    fifo_dequeue(replace_dir_idx, replace_table_idx);
}

```

Fig 23. Replacement Function - FIFO Policy

Fig 23 에 FIFO Policy 에 대한 코드가 제시 되어있다. 해당 알고리즘은 단순 Dequeue 기능을 통해 먼저 큐로 입력된 페이지가 반환되도록 하였다. 또한 모듈러 연산을 통해 큐를 순회하도록 하였다.

4-2-8. Copy Page Function (Copy on write)

```
int copy_page(PageDirectory *page_directory, int virtual_address, FreeFrameList *free_list, int pid){
    int fault_dir_idx = (virtual_address >> 22) & 0x3ff;
    int fault_table_idx = (virtual_address >> 12) & 0x3ff;

    if(page_directory->page_table[fault_dir_idx] == NULL || page_directory->page_table[fault_dir_idx]->valid[fault_table_idx] == false){
        fprintf(stderr, "Copy page called for invalid page!\n");
        exit(EXIT_FAILURE);
    }

    int original_frame = page_directory->page_table[fault_dir_idx]->matching_frame[fault_table_idx];
    page_directory->page_table[fault_dir_idx]->last_used[fault_table_idx] = ++timer;

    int new_frame = -1;

    for(int i = 0; i < FRAME_NUM; i++){
        if(free_list->free_frame_list[i] == 1){
            new_frame = i;
            free_list->free_frame_list[i] = 0;
            break;
        }
    }

    if(new_frame == -1) new_frame = swapping(page_directory, free_list, virtual_address, pid);

    page_directory->page_table[fault_dir_idx]->matching_frame[fault_table_idx] = new_frame;
    page_directory->page_table[fault_dir_idx]->valid[fault_table_idx] = true;
    page_directory->page_table[fault_dir_idx]->isRead = true;
    page_directory->page_table[fault_dir_idx]->isWrite = true;
    page_directory->page_table[fault_dir_idx]->last_used[fault_table_idx] = ++timer;

    return new_frame;
}
```

Fig 24. Copy Page (Copy on write)

Fig 24 에 Copy Page Function 에 대한 코드가 제시 되어있다. 먼저, 해당 함수가 호출이 되면 자유 프레임 리스트를 검색하여 사용 가능한 프레임을 확인한다. 만약 존재하지 않으면 swapping 함수를 통해 프레임을 할당 받은 후, Write 권한을 설정해주어 물리 메모리에 데이터를 작성 가능하도록 한다.

4-2-9. Disk Function

```
void write_page_to_disk(PageDirectory *page_directory, FreeFrameList *free_list, int pid, int dir_idx, int table_idx, int *data){
    if(disk_policy == 1){
        FILE *disk_file = fopen(DISK_FILE, "a");
        if(!disk_file){
            fprintf(stderr, "Failed to open disk.txt for writing!\n");
            exit(EXIT_FAILURE);
        }
        fprintf(disk_file, "%d, %d, %d, 0x%x\n", pid, dir_idx, table_idx, *data);
        fclose(disk_file);
    }
    else if(disk_policy == 2){
        if(disk_idx >= DISK_SIZE){
            printf("DISK CAPACITY is OVER\n");
            exit(EXIT_FAILURE);
        }

        disk[disk_idx++] = pid;
        disk[disk_idx++] = dir_idx;
        disk[disk_idx++] = table_idx;
        disk[disk_idx++] = *data;
    }
    page_directory->page_table[dir_idx]->valid[table_idx] = false;
}
```

Fig 25. Disk Write Function

Fig 25 에 디스크에 대한 Write Function 이 제시 되어있다. 본 레포트에서는 디스크를 파일을 통한 구현 및 배열을 통한 구현을 하였다. 만약 파일로 디스크를 제어한다면, fprintf 함수를 통해 페이지 정보 및 데이터를 작성하도록 한다. 배열로 디스크를 제어한다면, 인덱스를 통해 데이터를 작성한다.

```

int find_from_disk(int pid, int dir_idx, int table_idx){
    int line_number = 0;
    int file_pid, file_dir_idx, file_table_idx;

    if(disk_policy == 1){
        FILE *disk_file = fopen(DISK_FILE, "r");
        if(!disk){
            fprintf(stderr, "Disk file not found!\n");
            exit(EXIT_FAILURE);
        }

        char line[256];
        while(fgets(line, sizeof(line), disk_file)){
            line_number++;
            char data[128];
            data[0] = '\0';

            if(sscanf(line, "%d, %d, %d, %s", &file_pid, &file_dir_idx, &file_table_idx, data) == 4){
                if(file_pid == pid && file_dir_idx == dir_idx && file_table_idx == table_idx){
                    fclose(disk_file);
                    return line_number;
                }
            }
        }
        fclose(disk_file);
    }
    else if(disk_policy == 2){
        for(int i = 0; i < DISK_SIZE; i += 4){
            line_number++;
            file_pid = disk[i];
            file_dir_idx = disk[i+1];
            file_table_idx = disk[i+2];
            int data = disk[i+3];

            if(file_pid == pid && file_dir_idx == dir_idx && file_table_idx == table_idx){
                return line_number;
            }
        }
    }
    return -1;
}

```

Fig 26. Disk Find Function

Fig 26 에 Disk 에서 데이터를 찾는 함수가 제시 되어있다. 먼저, 파일로 구현한 디스크에서 데이터를 찾을 시 sscanf 함수를 통해 페이지 정보를 가져와 찾고자 하는 페이지가 맞는 지를 확인한다. 배열로 구현한 디스크에서는 인덱스 정보를 통해 페이지를 찾는다. 페이지를 찾는다면, 해당 줄 번호를 반환하도록 한다.

```

int read_page_from_disk(int line_num) {
    int data = 0;

    if(disk_policy == 1){
        FILE *disk_file = fopen(DISK_FILE, "r");
        FILE *temp = fopen(TEMP_FILE, "w");
        if (!disk_file || !temp) {
            fprintf(stderr, "Error opening file!\n");
            exit(EXIT_FAILURE);
        }

        char line[256];
        int current_line = 0;
        int data = 0;
        int found = 0;

        fseek(disk_file, line_num, SEEK_SET);
        if(fgets(line, sizeof(line), disk_file)){
            if(sscanf(line, "%d, %d, %d, %x", &data) == 1) found = 1;
            fputs(line, temp);
        }

        fclose(disk_file);
        fclose(temp);

        if(found){
            remove(DISK_FILE);
            rename(TEMP_FILE, DISK_FILE);
            return data;
        }
        else{
            remove(TEMP_FILE);
            return EOF;
        }
    }
    else if(disk_policy == 2){
        data = disk[line_num + 3];
        disk[line_num] = 0; disk[line_num+1] = 0; disk[line_num+2] = 0; disk[line_num+3] = 0;
        return data;
    }
}

```

Fig 27. Disk Read Function

Fig 27 에 디스크에서 데이터를 읽는 함수가 제시 되어있다. 먼저 파일로 구현된 디스크에서는 fseek 함수를 통해 해당 줄 번호로 이동한 후 데이터를 가져온다. 이후, 읽은 데이터는 디스크에서 삭제하는 과정을 거친다. 배열로 구현된 디스크에서는 줄 번호 인덱스를 통해 데이터를 가져온 후, 해당 배열 부분에 대해 0 으로 초기화 하여 삭제한다. 이후 데이터를 반환한다.

4-2-10. Demand Paging

```
void demand_paging(freeFrameList *free_frame_list, Queue * queue){
    Node *current_process = queue->head;
    int physical_address = MMU(current_process->pcb->page_directory, virtual_address, &first_table_fault, &second_table_fault, &isRead, &isWrite);

    if(first_table_fault == false && second_table_fault == false){
        memory_data = physical_memory[physical_address];
    }
    else{
        fault_num = handle_page_fault(current_process->pcb->page_directory, free_frame_list, virtual_address, pid);

        if(fault_num == FAULT_CODE_2){
            physical_address = MMU(current_process->pcb->page_directory, virtual_address, &first_table_fault, &second_table_fault, &isRead, &isWrite);
            memory_data = physical_memory[physical_address];
        }
        else if(fault_num == FAULT_CODE_1){
            gettimeofday(&start, NULL);
            int line_num = find_from_disk(pid, dir_idx, table_idx);
            int disk_data = read_page_from_disk(line_num);
            gettimeofday(&end, NULL);
            disk_time_ns += (end.tv_sec - start.tv_sec) * 1000000000LL +
                (end.tv_usec - start.tv_usec) * 1000;

            request = "W";
            swapping(current_process->pcb->page_directory, free_frame_list, virtual_address, pid);
            isSwap = true;
            physical_address = MMU(current_process->pcb->page_directory, virtual_address, &first_table_fault, &second_table_fault, &isRead, &isWrite);
            physical_memory[physical_address] = disk_data;
            log_memory_state(virtual_address, physical_address, current_process, first_table_fault, second_table_fault, write_data, i, request, isSwap, isCow);
            continue;
        }
        else if(fault_num == FAULT_CODE_0){
            swapping(current_process->pcb->page_directory, free_frame_list, virtual_address, pid);
            isSwap = true;
        }
    }
}
```

Fig 28. Demand Paging - MMU, Handle Fault

Fig 28 에 Demand Paging 에 대한 로직이 제시 되어있다. 먼저, 현재 스케줄링에서 실행 중인 프로세스를 가져온다. MMU 변환을 통해 물리 주소에 대한 정보를 얻는 데, 이 과정에서 페이지 테이블에 대한 인덱스 유효성을 평가한다. 유효하지 않다면 폴트를 처리하도록 한다. 만약, FAULT_CODE_2 라면 자유 프레임 리스트에서 할당 가능 프레임이 존재할 때, 프레임이 할당된 것이 아니기에 이를 할당한 후, 물리 주소 변환을 하도록 한다. FAULT_CODE_1 이라면 디스크에 해당 페이지에 대한 정보가 존재하기에 디스크에서 값을 읽은 후, 할당 가능 프레임이 존재하지 않기에 스와핑을 한 후 강제로 덮어 쓰도록 한다. 그리고, FAULT_CODE_0 이라면 디스크에도 존재하지 않기에 스와핑을 통해 프레임을 할당하는 작업을 수행한다.

```

if(memory_data == 0){
    request = "W";
    write_data = memory_data + (i+1);

    if(isWrite == true){
        physical_memory[physical_address] = write_data;
    }
    else{
        int new_frame = copy_page(current_process->pcb->page_directory, virtual_address, free_frame_list, pid);
        physical_address = (new_frame * PAGE_SIZE + (virtual_address & 0xfff)) / 4;
        physical_memory[physical_address] = write_data;
        isCow = true;
    }
    log_memory_state(virtual_address, physical_address, current_process, first_table_fault, second_table_fault, write_data, i, request, isSwap, isCow);
}
else{
    request = "R";
    log_memory_state(virtual_address, physical_address, current_process, first_table_fault, second_table_fault, &physical_memory[physical_address], i, request, isSwap, isCow);
}

```

Fig 29. Demand Paging - Read/Write

Fig 29 에 Demand Paging 의 Read/Write 코드가 제시 되어있다. 만약 물리 메모리의 데이터가 0 이라면 쓰기 작업을 수행하도록 설계하였다. 이후, 쓰기 권한이 존재한다면 물리 메모리에 바로 작성하도록 하고, 쓰기 권한이 없다면 Copy on write 를 수행하여 새로운 프레임에 데이터를 작성한다. 읽기 권한이라면 데이터를 바로 읽은 후 로그 파일에 작성하도록 한다.

5. Results

본 장에서는 프로그램의 실행 결과를 확인한다.

5-1. Log File

```
=====
[TIME TICK: 0] | [Process 0]
[0] Virtual Address: 0x55e114
[0] Physical Address: 0x445
1-Table Fault : 0
2-Table Fault : 0
Swapping : X
Copy On Write : 0
Matching Info : [Page 350 - Frame 1]
Write Data to Memory: 0x1
=====

[TIME TICK: 5000] | [Process 5]
[0] Virtual Address: 0x310de4
[0] Physical Address: 0xab79
1-Table Fault : X
2-Table Fault : X
Swapping : X
Copy On Write : X
Matching Info : [Page 784 - Frame 42]
Read Data from Memory: 0xb69e1df4
=====

[TIME TICK: 3724] | [Process 3]
[0] Virtual Address: 0x59e893
[0] Physical Address: 0x5b224
1-Table Fault : X
2-Table Fault : 0
Swapping : 0
Copy On Write : 0
Matching Info : [Page 414 - Frame 364]
Write Data to Memory: 0x1
=====

[TIME TICK: 9999] | [Process 4]
[9] Virtual Address: 0x36b181
[9] Physical Address: 0x1cc60
1-Table Fault : X
2-Table Fault : X
Swapping : X
Copy On Write : X
Matching Info : [Page 875 - Frame 115]
Write Data to Memory: 0xa
=====
```

Fig 30. Log File Results

Fig 30 에 Log File 에 대한 출력 결과를 확인할 수 있다. 먼저 Tick 0 부터 Tick 9999 까지 총 10000 Tick 이 발생하는 것을 확인할 수 있으며, 각 가상 메모리 요청 마다 특정 로직을 수행하는 것을 확인할 수 있다. 2 Level Page Table 에 대한 1-Table, 2-Table 의 Fault 정보를 확인할 수 있으며, Swapping 여부, Copy on Write 여부를 확인할 수 있고, 페이지와 매칭된 프레임 정보를 확인할 수 있다. 그리고, 물리 메모리에 Write 할 것인지, Read 할 것인지를 확인할 수 있으며, 해당 메모리에 Write 할 데이터와 Read 할 데이터를 확인할 수 있다.

5-2. Disk File

8, 1, 466, 0xa	8, 1, 466, 0xa	5, 1, 765, 0x1
8, 0, 335, 0xa	8, 0, 335, 0xa	5, 0, 926, 0x1
8, 1, 644, 0xa	8, 1, 644, 0xa	9, 0, 958, 0x1
9, 0, 650, 0xa	9, 0, 650, 0xa	9, 0, 129, 0x1
5, 1, 981, 0x5	8, 1, 827, 0xb4dffff	5, 0, 1020, 0x6
5, 1, 703, 0x2	8, 0, 583, 0x2	5, 1, 290, 0x0
5, 0, 862, 0x0	6, 1, 519, 0x0	5, 1, 140, 0x5
5, 0, 545, 0xffffb4d4	6, 1, 667, 0x0	5, 0, 1022, 0x0
	6, 0, 87, 0x4	5, 1, 771, 0x0

Fig 31. Disk File Results

Fig 31 에서 Disk File 에 대한 결과를 확인할 수 있다. 프로세스 ID, 1 단계 페이지 번호, 2 단계 페이지 번호, 데이터가 정상적으로 저장되는 것을 확인할 수 있다.

5-3. Disk File Based Result

<pre>===== RESULT ===== [1] SCHEDULING STATISTICS 1. AVG.WAIT TIME : 48.250(s) 2. AVG.TURNAROUND TIME : 61.865(s) [2] VIRTUAL MEMORY STATISTICS 1. PAGE TABLE LEVEL 1 HIT RATE: 99.998 2. PAGE TABLE LEVEL 2 HIT RATE: 65.836 3. SWAPPING COUNT : 32108 4. COPY ON WRITE COUNT: 31996 5. DISK ACCESS TIME: 2667444000(ns) =====</pre>	<pre>===== RESULT ===== [1] SCHEDULING STATISTICS 1. AVG.WAIT TIME : 35.675(s) 2. AVG.TURNAROUND TIME : 51.681(s) [2] VIRTUAL MEMORY STATISTICS 1. PAGE TABLE LEVEL 1 HIT RATE: 99.998 2. PAGE TABLE LEVEL 2 HIT RATE: 66.483 3. SWAPPING COUNT : 31399 4. COPY ON WRITE COUNT: 31289 5. DISK ACCESS TIME: 38902419000(ns) =====</pre>	<pre>===== RESULT ===== [1] SCHEDULING STATISTICS 1. AVG.WAIT TIME : 8.794(s) 2. AVG.TURNAROUND TIME : 61.156(s) [2] VIRTUAL MEMORY STATISTICS 1. PAGE TABLE LEVEL 1 HIT RATE: 99.998 2. PAGE TABLE LEVEL 2 HIT RATE: 65.324 3. SWAPPING COUNT : 32635 4. COPY ON WRITE COUNT: 32522 5. DISK ACCESS TIME: 38735803000(ns) =====</pre>
<pre>===== RESULT ===== [1] SCHEDULING STATISTICS 1. AVG.WAIT TIME : 55.103(s) 2. AVG.TURNAROUND TIME : 68.820(s) [2] VIRTUAL MEMORY STATISTICS 1. PAGE TABLE LEVEL 1 HIT RATE: 99.998 2. PAGE TABLE LEVEL 2 HIT RATE: 70.914 3. SWAPPING COUNT : 26887 4. COPY ON WRITE COUNT: 26468 5. DISK ACCESS TIME: 2177101000(ns) =====</pre>	<pre>===== RESULT ===== [1] SCHEDULING STATISTICS 1. AVG.WAIT TIME : 40.092(s) 2. AVG.TURNAROUND TIME : 49.231(s) [2] VIRTUAL MEMORY STATISTICS 1. PAGE TABLE LEVEL 1 HIT RATE: 99.998 2. PAGE TABLE LEVEL 2 HIT RATE: 70.671 3. SWAPPING COUNT : 27167 4. COPY ON WRITE COUNT: 26773 5. DISK ACCESS TIME: 2171722000(ns) =====</pre>	<pre>===== RESULT ===== [1] SCHEDULING STATISTICS 1. AVG.WAIT TIME : 11.330(s) 2. AVG.TURNAROUND TIME : 21.985(s) [2] VIRTUAL MEMORY STATISTICS 1. PAGE TABLE LEVEL 1 HIT RATE: 99.998 2. PAGE TABLE LEVEL 2 HIT RATE: 77.328 3. SWAPPING COUNT : 20293 4. COPY ON WRITE COUNT: 19810 5. DISK ACCESS TIME: 1329706000(ns) =====</pre>

Fig 32. Disk File Based Result
(TOP: LRU(FIFO, RR, SJF), BOTTOM: FCFS(FIFO, RR, SJF))

Fig 32 에 디스크를 파일로 구현하였을 때의 결과가 제시 되어있다.

5-4. Disk Array Based Result

<pre>===== RESULT ===== [1] SCHEDULING STATISTICS 1. AVG.WAIT TIME : 64.562(s) 2. AVG.TURNAROUND TIME : 77.764(s) [2] VIRTUAL MEMORY STATISTICS 1. PAGE TABLE LEVEL 1 HIT RATE: 99.998 2. PAGE TABLE LEVEL 2 HIT RATE: 85.783 3. SWAPPING COUNT : 11401 4. COPY ON WRITE COUNT: 11251 5. DISK ACCESS TIME: 9376750000(ns) =====</pre>	<pre>===== RESULT ===== [1] SCHEDULING STATISTICS 1. AVG.WAIT TIME : 33.982(s) 2. AVG.TURNAROUND TIME : 50.844(s) [2] VIRTUAL MEMORY STATISTICS 1. PAGE TABLE LEVEL 1 HIT RATE: 99.998 2. PAGE TABLE LEVEL 2 HIT RATE: 85.322 3. SWAPPING COUNT : 11846 4. COPY ON WRITE COUNT: 11716 5. DISK ACCESS TIME: 46587904000(ns) =====</pre>	<pre>===== RESULT ===== [1] SCHEDULING STATISTICS 1. AVG.WAIT TIME : 15.566(s) 2. AVG.TURNAROUND TIME : 60.067(s) [2] VIRTUAL MEMORY STATISTICS 1. PAGE TABLE LEVEL 1 HIT RATE: 99.998 2. PAGE TABLE LEVEL 2 HIT RATE: 85.774 3. SWAPPING COUNT : 11408 4. COPY ON WRITE COUNT: 11256 5. DISK ACCESS TIME: 9610902000(ns) =====</pre>
<pre>===== RESULT ===== [1] SCHEDULING STATISTICS 1. AVG.WAIT TIME : 59.415(s) 2. AVG.TURNAROUND TIME : 73.813(s) [2] VIRTUAL MEMORY STATISTICS 1. PAGE TABLE LEVEL 1 HIT RATE: 99.998 2. PAGE TABLE LEVEL 2 HIT RATE: 91.980 3. SWAPPING COUNT : 5476 4. COPY ON WRITE COUNT: 5063 5. DISK ACCESS TIME: 4416885000(ns) =====</pre>	<pre>===== RESULT ===== [1] SCHEDULING STATISTICS 1. AVG.WAIT TIME : 38.119(s) 2. AVG.TURNAROUND TIME : 50.273(s) [2] VIRTUAL MEMORY STATISTICS 1. PAGE TABLE LEVEL 1 HIT RATE: 99.998 2. PAGE TABLE LEVEL 2 HIT RATE: 91.605 3. SWAPPING COUNT : 5850 4. COPY ON WRITE COUNT: 5369 5. DISK ACCESS TIME: 40757708000(ns) =====</pre>	<pre>===== RESULT ===== [1] SCHEDULING STATISTICS 1. AVG.WAIT TIME : 5.419(s) 2. AVG.TURNAROUND TIME : 44.128(s) [2] VIRTUAL MEMORY STATISTICS 1. PAGE TABLE LEVEL 1 HIT RATE: 99.998 2. PAGE TABLE LEVEL 2 HIT RATE: 92.153 3. SWAPPING COUNT : 5312 4. COPY ON WRITE COUNT: 4877 5. DISK ACCESS TIME: 4503861000(ns) =====</pre>

Fig 33. Disk Array Based Result
(TOP: LRU(FIFO, RR, SJF), BOTTOM: FCFS(FIFO, RR, SJF))

Fig 33 에 디스크를 배열로 구현하였을 때의 결과가 제시 되어있다.

6. Evaluation

본 장에서는 5 장에서 확인한 결과를 바탕으로 프로그램에 대한 평가를 하도록 한다.

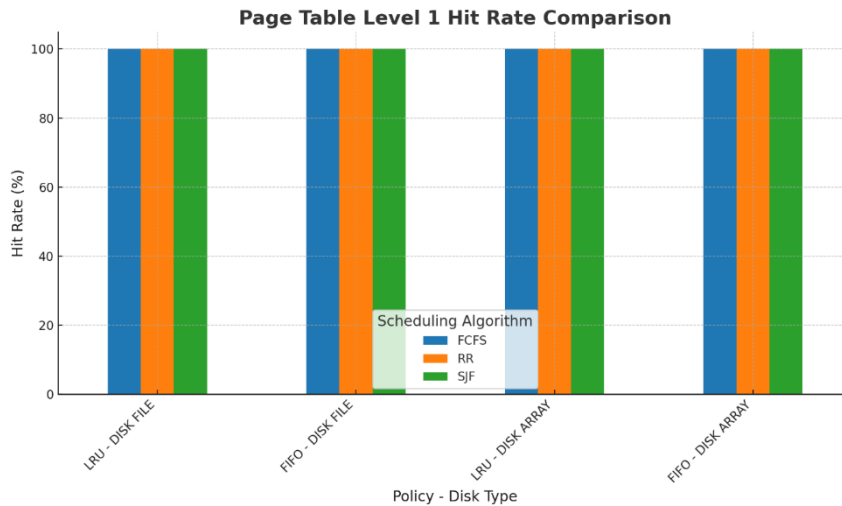


Fig 34. Bar Graph of PTL 1 Hit Rate

Fig 34 는 Page Table Level 1 에 대한 Hit Rate 이다. 해당 그래프를 통해 Replace Policy, Scheduling Algorithm, Disk Type 에 상관없이 같은 비율을 보여주는 것을 알 수 있다. 이를 통해 2 단계 페이지 테이블에서 Page Table Level 1 은 항상 같은 Hit Rate 를 유지하는 것을 알 수 있다.

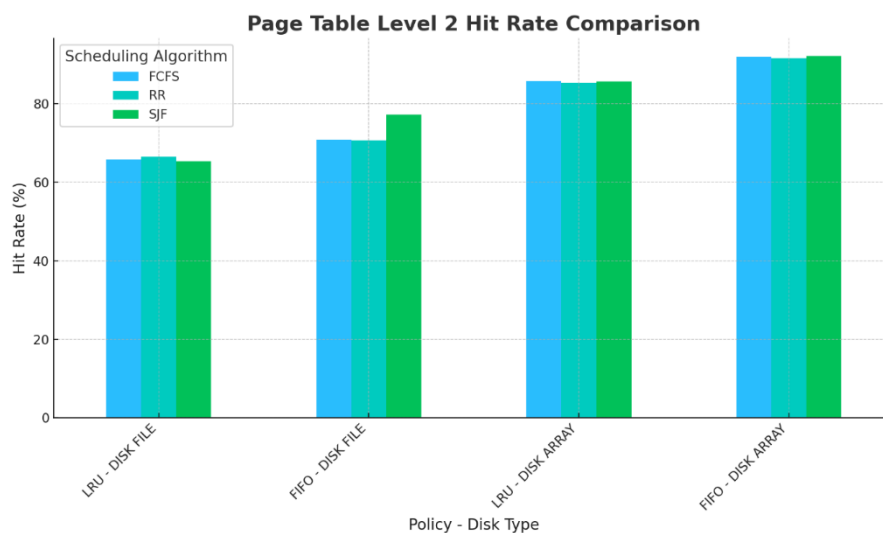


Fig 35. Bar Graph of PTL 2 Hit Rate

Fig 35 에서 Page Table Level 2 에 대한 Hit Rate 를 확인할 수 있다. 먼저, 같은 Disk Type 일 때 FIFO 알고리즘이 LRU 알고리즘 보다 더 높은 Hit Rate 를 보였다. 다른 Disk Type 에서는 Array 로 디스크를

구현하였을 때 더 높은 Hit Rate 를 도출한다는 것을 확인할 수 있다.

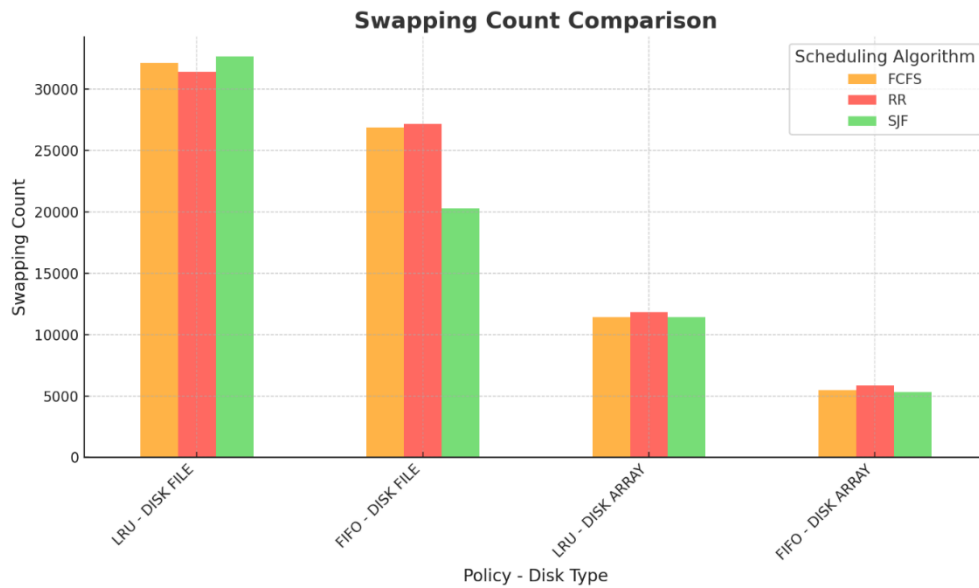


Fig 36. Bar Graph of Swapping count

Fig 36 에 Swapping count 를 막대 그래프로 시각화한 분석 결과가 나타나 있다. 먼저, 같은 Disk Type 일 때 FIFO 알고리즘의 Swapping 횟수가 더 적은 것을 확인할 수 있으며, 다른 Disk Type 일 때 Array 로 구현한 디스크가 더 적은 Swapping 횟수를 보이는 것을 확인할 수 있다.

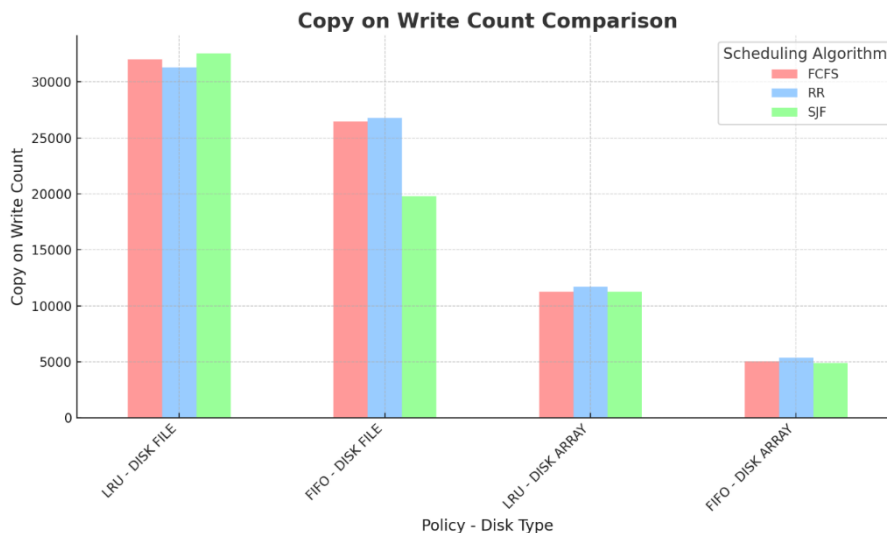


Fig 37. Bar Graph of Copy on Write count

Fig 37 은 Copy on Write count 에 대한 데이터 시각화 분석 그래프이다. 해당 분석 결과 또한 Swapping 과 비슷한 흐름을 보인다는 것을 확인할 수 있으며, 이를 통해 Swapping 과 Copy on Write 의

발생 빈도는 유사하다는 것을 알 수 있다.

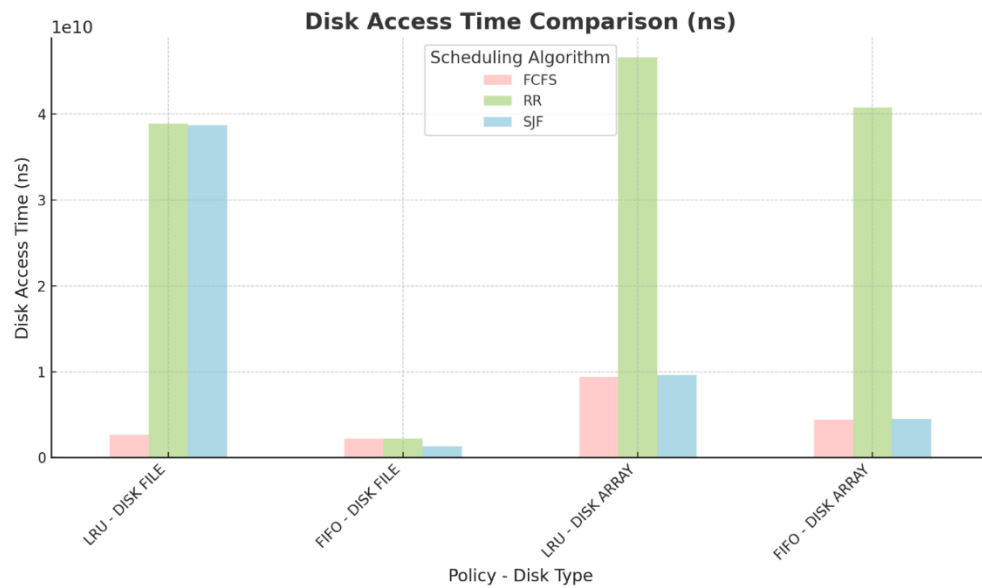


Fig 38. Bar Graph of Disk Access Time

Fig 38 은 Disk Access Time 에 대한 Bar Graph 이다. 해당 결과를 분석해보면 Round Robin 스케줄링 방식에서는 반복적으로 파일을 열고 닫는 과정이 많아 디스크 접근 시간이 평균적으로 큰 것으로 나타나 있다. 또한 Disk Array 방식이 Disk File 방식보다 평균적으로 긴 디스크 접근 시간을 보여주는 것을 알 수 있다. 대용량의 디스크 크기에서 인덱스를 통한 반복문 로직은 디스크 파일로 접근하는 것보다 비효율적이라는 것을 알 수 있다.

7. Conclusion

본 레포트에서는 Virtual Memory 프로그램을 구현을 목표로 하였다. 특히, 가상 메모리 기법 중 페이지징을 구현하는 데 초점을 두었다. 이를 위해 2장에서 요구 사항을 검토하고, 3장에서 프로그램을 구현하기 위한 관련 개념을 살펴본 뒤, 4장에서는 구현을 하기 위한 아이디어에 대한 순서도를 제시하였으며, 이를 바탕으로 코드를 작성한 후, 각 로직에 대한 설명하였다. 5장에서는 구현한 프로그램을 바탕으로 결과를 확인하였으며, 출력된 결과를 바탕으로 6장에서는 분석을 통한 평가를 하였다.

본 레포트에서는 가상 메모리 프로그램을 실제 가상 메모리 기법과 유사하게 구현하기 위해 2-Level Page Table, Swapping, Copy on Write를 추가적으로 구현하였다. 또한 디스크를 파일로 구현하고, 배열로 구현함으로써 디스크 구현 방식에 따른 성능 차이를 확인하였다.

결과적으로, 디스크를 파일로 구현할 경우 Swapping과 Copy on Write의 횟수가 증가하였지만 디스크 접근 시간이 감소하는 경향을 확인하였다. 반면, 디스크를 배열로 구현할 경우 Swapping과 Copy on Write의 횟수는 감소하였으나 디스크 접근 시간은 증가하는 것을 관찰할 수 있었다. 이는 디스크 접근 패턴과 디스크 구현 방식의 차이에서 기인하는 것으로 해석된다. 또한, Page Table Level 1의 Hit Rate는 모든 가상 메모리 정책에서 동일한 결과를 보였으며, Page Table Level 2의 Hit Rate는 배열 기반 디스크 방식에서 더 높은 값을 보이는 것을 확인하였다.

이 과정에서, 가상 메모리 개념을 이해한 뒤 이를 바탕으로 프로그램을 설계하고, 구현한 프로그램의 결과를 평가하며 시각화 된 데이터를 분석하는 경험은 가상 메모리 시스템의 동작 원리에 대한 이해를 넘어 더욱 깊은 통찰력을 제공하였다. 특히, 다양한 가상 메모리 정책 및 구현 방식의 장단점을 비교함으로써 각 기술의 적합성을 평가할 수 있었다.

추가적으로, 본 프로젝트는 가상 메모리 시스템의 이론적 개념을 실질적으로 구현하고 이를 평가할 수 있는 체계적인 접근 방식을 제공하였다. 이러한 과정은 가상 메모리 설계와 최적화에 대한 기초를 다지며, 현실 시스템에서의 가상 메모리 기술 활용 가능성을 확인할 수 있는 기회가 되었다.

마지막으로, 이번 프로젝트에서 멈추지 않고, 향후 더 다양한 교체 정책 알고리즘을 추가하고, 메모리 및 디스크의 크기를 다르게 함으로써 여러 실험을 통해 성능 차이를 확인하고, 최적화된 결과를 도출함으로써 프로젝트를 발전해 나갈 생각이다.