24S_OS_miniOS

1. **프로젝트 개요**
    a. 컴퓨터에서 프로그램을 가동시키기 위해선 메모리를 사용해야 함.
    b. 이러한 메모리를 어떤 방법으로 관리하고 관리하는 이유에 대해서 이해가 필요함.
    c. 메모리 관리 기법 중 하나인 페이징을 데모 버전으로 구현하여 추가적인 기능을 개선할 예정

2. **메모리 관리의 필요성**
    a. 우리가 사용하고자 하는 프로그램은 기본적으로 저장장치인 HDD, SSD에 저장되어 있음.
    b. 메모리의 용량은 보통 저장장치보다 적기 때문에 우리가 프로그램들은 모두 메모리에 올라와 있지 않음.
    c. 프로그램을 실행할 때 메모리에 프로그램이 올라오게 해야하며, 이때 효율적인 방법을 사용하여 메모리 낭비를 최소화 해야함.
    d. 또한 여러 프로그램이 동시에 실행이 될 때 다른 프로그램의 영역에 침범하지 않게 메모리 보호를 해야함.
    e. 사용가능한 메모리 블록이 부족해지는 메모리 단편화를 방지하기 위해서도 메모리 관리가 필수적임.

3. **페이징 - Paging**
    - 페이징은 메모리 관리 기법 중 하나로 물리적 메모리와 가상 메모리를 일정한 크기의 블록으로 나누어 사용하는 방식임.
    - 페이징을 통해 메모리 관리의 효율성을 높이고, 프로그램이 요구하는 메모리 공간을 보다 효율적으로 할당할 수 있음
    - 용어
        - 페이지(Page)
            - 가상 메모리를 일정한 크기(통상 4KB)의 블록으로 나눈 것. 각 페이지는 연속적인 가상 주소를 가짐

    - 프레임(Frame)
        - 물리적 메모리를 페이지와 동일한 크기의 블록으로 나눈 것. 프레임은 실제 물리적 메모리 주소를 가짐
    - 페이지 테이블(Page Table)
        - 가상 메모리의 페이지와 물리적 메모리의 프레임 간의 매핑 정보를 저장하는 데이터 구조임. 각 프로세스는 자신만의 페이지 테이블을 가짐
    - TLB
        - 일종의 캐시 메모리로 페이지 테이블의 일부 정보를 저장하여 주소 변환을 빠르게 수행할 수 있도록 도움.
- 장점
    - 메모리 보호 : 각 프로세스는 자신의 가상 메모리 공간만을 사용하므로, 다른 프로세스의 메모리 공간에 접근하지 못함
    - 메모리 효율성 : 물리적 메모리를 작은 블록으로 나누어 사용하므로, 메모리의 단편화를 줄이고, 보다 효율적으로 메모리를 사용할 수 있음.
    - 가상 메모리 지원 : 페이징을 통해 실제 물리적 메모리보다 큰 가상 메모리 공간을 제공할 수 있음 → 프로그램이 더 큰 주소 공간을 사용할 수 있게 함.
- 단점
    - 페이지 테이블 오버헤드 : 페이지 테이블을 관리하기 위한 추가적인 메모리 공간과 성능 오버헤드가 발생할 수 있음
    - 페이지 폴트 : 필요한 페이지가 물리적 메모리에 없을 때 발생하는 상황으로, 저장장치에서 페이지를 로드해야 하므로 성능 저하가 발생될 수 있음.

4. 현재 구성된 메모리 관리 코드

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

#define MAX_MEMORY_BLOCKS 10
#define BLOCK_SIZE 1024

typedef struct {
    bool is_free;
```

```
    char block[BLOCK_SIZE];
} MemoryBlock;

MemoryBlock memory_pool[MAX_MEMORY_BLOCKS];

// 초기화 함수
void init_memory_pool() {
    for (int i = 0; i < MAX_MEMORY_BLOCKS; i++) {
        memory_pool[i].is_free = true;
    }
}

// 메모리 할당 함수
void* allocate_memory() {
    for (int i = 0; i < MAX_MEMORY_BLOCKS; i++) {
        if (memory_pool[i].is_free) {
            memory_pool[i].is_free = false;
            return (void*)memory_pool[i].block;
        }
    }
    return NULL; // 메모리가 부족할 경우 NULL 반환
}

// 메모리 해제 함수
void free_memory(void* ptr) {
    for (int i = 0; i < MAX_MEMORY_BLOCKS; i++) {
        if (memory_pool[i].block == ptr) {
            memory_pool[i].is_free = true;
            return;
        }
    }
}

// 디버그 함수: 메모리 풀 상태 출력
void print_memory_pool_status() {
    for (int i = 0; i < MAX_MEMORY_BLOCKS; i++) {
        printf("Block %d: %s\n", i, memory_pool[i].is_free
? "Free" : "Allocated");
```

```
    }
}

int main() {
    init_memory_pool();

    // 테스트 코드
    void* mem1 = allocate_memory();
    void* mem2 = allocate_memory();
    print_memory_pool_status();

    free_memory(mem1);
    free_memory(mem2);
    print_memory_pool_status();

    return 0;
}
```

- memoruy_pool은 고정 크기의 메모리 블록 배열임
- allocate_memory 함수는 사용 가능한 메모리 블록을 찾아 할당함
- free_memory 함수는 할당된 메모리 블록을 해제함.
- print_memory_pool_status 함수는 메모리 풀의 현재 상태를 출력함

5. 향후 프로젝트 진행 계획
    a. 위의 코드에 페이징을 추가하여 효율적으로 메모리를 관리하는 매커니즘을 구현할 예정
    b. 페이징을 구현 후 TLB를 구현할 예정.
    c. 이후 LRU라는 페이지 교체 알고리즘을 구현할 예정.