Project #3: 64비트 모드 전환 및 페이지 테이블 추가기능구현

신정은, 조재호, 조한주
Soongsil University

1. 구현 목표

우리의 목표는 MINT64 OS를 만드는 것이다. 이를 위해서는 64비트 모드로 전환해야 한다. 보호모드의 커널 및 부트로더의 스택 영역을 64비트 모드의 커널 공간으로 사용한다면 많은 기능이 있는 커널을 올릴수 없으므로 1MB이상의 메모리에 접근하기 위해 A20게이트를 활성화한다. 호환성 유지를 위해 쓰이는 A20게이트의 의미와 역할을 이해하고 이를 BIOS서비스와 시스템 컨트롤 포트로 활성화하는 방법을 알아본다. 또한 1A-32e모드 커널실행을 위해 메모리 크기를 확인하고 IA-32e 커널 이미지가 초기화되지 않은 영역을 포함하고 있지 않기 때문에 IA-32e 커널이 위치할 영역을 0으로 초기화한다.

IA-32e 모드는 페이징이 필수이므로 64GB까지 매핑하는 페이지 테이블을 생성하고, 이를 프로세서에 설정하여 페이징 기능을 활성화하는 방법을 이해한다. 이를 위해 2MB를 사용하는 4단계 페이징을 구현해 본다. CR3 레지스터에 설정된 PML4 테이블의 기준 주소로부터 PML4 엔트리 -> 페이지 디렉터리 포인터 엔트리 -> 디렉터리 엔트리 순으로 진행하는 것을 직접 구현해 보고, 이렇게 2MB 페이지의 기준주소를 찾은 후, 기준주소에 하위 21비트의 오프셋을 더해 선형주소를 물리주소로 변환하는 과정을 이해한다. 이 과정에서 페이지 테이블을 구성하고 공간을 할당하는 것을 이해해야 한다. 그 후, CR0, CR3, CR4레지스터를 통해 프로세서의 페이징기능을 활성화하는 방법을 알아본다.

IA-32e모드로 전환하기 전, IA-32e모드 지원여부를 검사한 후, IA-32e모드용 세그먼트 디스크립터를 추가하여 전환할 준비를 한다.
IA-32e모드 커널을 생성하고, IA-32e모드 커널의 커널 엔트리 포인트 파일은 오브젝트 파일의 형태로 컴파일되어 C언어 커널과 함께 링크되는 것을 유의하여 makefile을 생성해본다.
마지막으로 보호모드 커널과 IA-32e모드 커널을 통합하여 하나의 os이미지로 통합한다. 이 과정에서 부트로더 파일, 이미지메이커 프로그램을 수정하여 보호모드커널의 C언어 엔트리 포인트 파일 에서 IA-32e모드 커널 이미지를 2MB 어드레스로 복사하고, 64비트로 전환되는 것을 확인해 본다.

운영체제가 페이지 테이블을 통해 할 수 있는 기능 2가지를 구현해 본다.

첫째로, Double Mapping을 구현한다. 이 기능을 이용하면 가상주소와 물리 주소가 1:1로 매핑되는 것이 아니라 서로 다른 가상주소가 하나의 물리 메모리를 가리키도록 한다. 페이지 디렉토리 엔트리를 수정하여 0xAB8000에 비디오 메모리를 맵핑하는 과정을 통해 Double Mapping기능을 이해한다.

둘째로. Finer-Grained Page Table와 Read-only Page를 구현한다.
현재 만들고 있는 MINT64 OS는 IA-32e 모드 페이징에서 4단계 페이징 기법을 사용한다. 이는 페이징 단위로 2MB를 사용한다. 그리고 페이지 테이블 대신 페이지 디렉터리가 직접 해당 페이지의 시작 어드레스를 가리킨다. 이는 페이지 테이블로 페이지의 주소를 가리키는 것에 비해 상대적으로 coarse-grained하다. 따라서 보다 세밀하게 메모리를 관리하기 위해 4KB 단위로 메모리에 접근할 수 있는 페이지 테이블 엔트리로 구성된 페이지 테이블을 하나 만들어, 페이지 디렉터리 엔트리에 맵핑해보며 커널 자료구조의 일부분이 Finer-Grained Page Table이 되는 것을 확인한다. 이렇게 4KB 단위로 커널 자료구조에 접근할 수 있는 페이지 테이블을 구현한 후에, flag를 수정하여 커널 자료구조영역의 일부 영역을 Read-Only Page로 구현해 본다.

따라서 본 프로젝트에서 구현해야 할 구현 목표는 크게 3가지이다..

1) 페이지 테이블 생성
2) 64비트 모드로 전환
3) 페이지 테이블의 2가지 기능 추가구현

2. 구현 내용

2.1 페이지 테이블 생성

4단계 페이징을 구현하기 위해 페이지 테이블과 페이지 테이블을 구성하고 있는 페이지 엔트리 자료구조를 생성해야 한다. 페이지 엔트리는 64비트 어드레스를 표현하기 위해 8byte의 크기를 가진다. 하지만 우리가 구현한 OS는 아직 32비트 환경이므로 4byte 변수를 2개 사용하여 페이지 엔트리를 구현한다.

4단계 페이징에서는 PML4 테이블, 페이지 디렉토

리 포인터 테이블, 페이지 디렉터리 등 총 3가지 종류의 페이지 테이블이 필요하다. 각 테이블은 512개의 엔트리로 구성된다. PML4 테이블의 각 엔트리는 하위 레벨인 페이지 디렉터리 포인터 테이블의 정보를 가지고, 페이지 디렉터리 포인터 테이블의 각 엔트리는 페이지 디렉터리의 정보를 가진다. 마지막으로 페이지 디렉터리는 2MB 페이지에 대한 정보를 가진다. 페이지 디렉터리 하나로 관리할 수 있는 메모리 영역은 (512개 * 2MB)가 되어 1GB이다. 우리는 64GB의 물리 메모리를 매핑해야 하므로 PML4 테이블 1개, 페이지 디렉터리 포인터 테이블 1개, 페이지 디렉터리 64개가 필요하다. 각 테이블의 크기는 4KB (512개 * 8byte)이므로 264KB (66개* 4KB)의 메모리가 필요하다. 264KB는 매우 큰 크기이므로 0x100000 ~ 0x142000의 어드레스에 할당한다.

2.2 64비트모드로 전환

64비트 모드로 전환하기 전에 IA-32e 모드용 세그먼트 디스크립터를 생성한다. 보호 모드 커널 엔트리 포인트에서 생성한 GDT에 IA-32e 모드용 세그먼트 디스크립터를 추가한다. IA-32e 모드용 세그먼트 디스크립터가 보호 모드용 세그먼트 디스크립터의 앞에 있으므로 보호 모드로 전환할 때 사용했던 세그먼트 셀렉터의 값을 변경한다.

64비트 모드로 전환하기 위해 물리 메모리 확장 기능을 활성화 하고 페이지 테이블을 설정한다. 최상위 테이블의 어드레스를 프로세서에 알리는 역할을 하는 CR3 레지스터에 PML4 테이블의 어드레스를 설정한다. 또한 CR4 레지스터의 PAE, 비트를 1로 설정하여 페이징을 활성화한다. 프로세서의 확장 기능을 제어할 수 있는 IA32-EFER 레지스터의 LME비트를 1로 설정함으로써 IA-32e 모드를 활성화 한다. 마지막으로 CR0 레지스터의 NW,CD,PG 비트를 1로 설정하여 캐시와, 페이징을 활성화한다.

모든 준비를 끝냈으니 CS 세그먼트 셀렉터를 IA-32e 모드용 코드 세그먼트 디스크립터로 교체하고 IA-32e 모드 커널로 이동할 준비를 마친다.

이제 IA-32e 모드 커널의 엔트리 포인트와 C언어 커널 엔트리 포인트를 합쳐 IA-32e 모드용 커널 이미지를 만든다. 이 때 IA-32e 모드 커널의 커널 엔트리 포인트 파일은 오브젝트 파일의 형태로 컴파일 되어 C언어 커널과 함께 링크된다. 따라서 커널 엔트리 포인트 파일이 링크 목록의 가장 앞에 위치해야 한다.

부트 로더와 ImageMaker 프로그램을 수정하여 IA-32e 모드 커널을 이동하는데 필요한 정보를 추가하고 보모 모드 커널의 C언어 엔트리 포인터르 수정하여 IA-32e 모드 커널을 0x200000 어드레스에 복사 후 IA-32e 모드로 전환하면 된다.

엔트리 포인트가 C 코드를 실행하려면 아래 3가지 조건을 만족해야 한다.

1. C 라이브러리를 사용하지 않게 빌드해야 한다.
2. 0x10200 위치에서 실행해야 한다.
3. 코드, 데이터만 포함된 바이너리 파일이다.

이 세 조건이 만족하도록 makefile을 작성하고 링커 스크립트를 작성한다. 또한 make의 와일드카드 기능을 사용하여 .c 확장자의 파일만 추가되면 자동으로 포함하여 빌드하게 수정한다.

2.3 페이지 테이블의 2가지 기능 추가 구현

첫 번째 기능인 Double Mapping을 구현해 본다. 0xB8000에 비디오 메모리가 위치한다. 이 비디오 메모리는 0번째 페이지 디렉터리에 매핑된 페이지에 위치한다. 우리가 Double Mapping하려는 0xAB8000은 5번째 페이지 디렉터리에 매핑된 페이지에 위치한다. 이 페이지의 기준 주소는 0xA00000(10MB)이다. 이 페이지에서 0xB8000 만큼 떨어진 곳에 0xAB8000이 위치하므로 우리는 5번째 페이지 디렉터리와 0번째 페이지 디렉터리에 매핑된 페이지가 매핑되게 하면 된다.

```
for(i=0;i<PAGE_MAXENTRYCOUNT*64;i++)
{
    //Double Mapping을 위해
    if(i==5){
        kSetPageEntryData(&(pstPDEntry[5]),0, 0,PAGE_FLAGS_DEFAULT | PAGE_FLAGS_PS,0);
    }
    else{
        kSetPageEntryData(&(pstPDEntry[i]),(i*(0x200000 >> 20))>>12,
                dwMappingAddress,PAGE_FLAGS_DEFAULT | PAGE_FLAGS_PS,0);
    }
    dwMappingAddress += PAGE_DEFAULTSIZE;   //2MB씩 더한다
}
```

그리고 IA-32e모드의 kPrintString함수에서 0xB8000대신 0xAB8000를 통해 문자열을 출력해 본다.

```
MINT64 OS Boot Loader Start~!!
Current Time:  16:13:51
OS Image Loading... Complete~!!
Switch To Protected Mode Success~!!
C Language Kernel Start.....[Pass]
Minimum Memory Size Check.....[Pass]
IA-32e Kernel Area Initialize.....[Pass]
IA-32e PageTables Initialize.....[Pass]
Processor Vendor String.....[AuthenticAMD]
64bit Mode Support Check.....[Pass]
Copy IA-32e Kernel To 2MB address.....[Pass]
Switch To IA-32e Mode
Switch To Ia-32e Mode Success~!!
IA-32e C Language Kernel Start.....[Pass]
This message is printed through the video memory relocated to 0xAB8000
```

[출력결과 1]

두 번째 기능인 Finer-grained Page Table을 구현해 본다. Finer-grained Page Table을 만들어 0x1FF000번지를 Read-only Page를 구현한다. 0x1FF000 번지는 첫번째 페이지 디렉터리(1GB)의 첫번째 페이지 디렉터리 엔트리(2MB)의 범위에 속한다. 따라서 다음 코드와 같이 작성한

다. 56번 줄부터 62번 줄까지 코드가 첫 번째 페이지 디렉터리 엔트리가 새로 만들어진 페이지 테이블을 가리키도록 한다. 이 때 기존 코드와 다른 점은 0x142000(페이지 테이블의 주소)는 32비트를 넘지 않아 상위 dwUpperBaseAddress를 0으로 준 것과, 하위 dwLowerBaseAddress로 0x142000 를 준 것이다. 또, 페이지의 사이즈가 4KB이기 때문에 엔트리의 PS 필드를 0으로 설정하였다. PS 필드가 0일경우 페이지의 크기를 4KB 로 인식한다.

```
46     // 페이지 디렉터리 테이블 생성
47     // 하나의 페이지 디렉터리가 1GB까지 매핑 가능
48     // 여유있게 64개의 페이지 디렉터리를 생성하여 총 64GB까지 지원
49     pstPDEntry = ( PDENTRY* ) 0x102000;
50     dwMappingAddress = 0;
51     for( i = 0 ; i < PAGE_MAXENTRYCOUNT * 64 ; i++ )
52     {
53 
54         // 첫 번째 페이지 디렉터리 테이블이 새로 만들어지는
55         // 페이지 테이블을 가리키도록 한다.
56         if( i == 0 )
57         {
58             kSetPageEntryData( &( pstPDEntry[ i ] ), 0, 0x142000,
59                     PAGE_FLAGS_DEFAULT , 0 );
60             dwMappingAddress += PAGE_DEFAULTSIZE;
61             continue;
62         }
63 
64         // 32비트로는 상위 어드레스를 표현할 수 없으므로, MB 단위로 계산한 다음
65         // 최종 결과를 다시 4KB로 나누어 32비트 이상의 어드레스를 계산함
66         kSetPageEntryData( &( pstPDEntry[ i ] ), ( i * ( PAGE_DEFAULTSIZE >> 20 ) ) >> 12,
67                 dwMappingAddress, PAGE_FLAGS_DEFAULT | PAGE_FLAGS_PS, 0);
68         dwMappingAddress += PAGE_DEFAULTSIZE;
69     }
```

새로 만들어지는 페이지 테이블은 0번지부터 2MB 크기로 매핑한다. 우리가 읽기 전용으로 하고자하는 0x1FF000번지를 페이지 테이블 엔트리에 매핑해줄 때 PAGE_FLAGS_RW를 0으로 설정한다. R/W flag가 0이라면 읽기만 가능하며 쓰기를 시도할 경우 page fault가 예외가 발생한다. 따라서 해당 번지를 읽기 전용으로 설정할 수 있다.

```
    // 페이지 테이블 하나 생성
    // 하나의 페이지 테이블은 2MB까지 매핑한다.
    // 0x000000 번지부터 0x200000 번지 까지 매핑해준다.
    // 우리가 접근해야하는 0x1FF000 번지의 경우 읽기전용으로 만들어준다.
    pstPTEntry = ( PTENTRY* ) 0x142000;
    dwMappingAddress = 0;
    for( i = 0 ; i < PAGE_MAXENTRYCOUNT ; i++ )
    {

        if( dwMappingAddress == 0x1FF000 )
        {
            kSetPageEntryData( &( pstPTEntry[ i ] ), 0,
                dwMappingAddress, PAGE_FLAGS_P , 0 );
            dwMappingAddress += 0x1000;
            continue;
        }

        kSetPageEntryData( &( pstPTEntry[ i ] ), 0,
                dwMappingAddress, PAGE_FLAGS_DEFAULT, 0 );
        dwMappingAddress += 0x1000;
    }
```

CR0 컨트롤 레지스터의 WP 필드가 0이라면 페이지 속성에 상관없이 R/W 가 가능하다. 하지만 1이라면 페이지 속성의 R/W 플래그에 영향을 받는다. 따라서 IA-32e 모드로 스위치할 때 CR0 컨트롤 레지스터의 WP field를 1로 설정한다.

```
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ; CR0 컨트롤 레지스터를 NW 비트(비트 29) =  0, CD 비트(비트 30) = 0, PG 비트(비트 31) = 1로
    ; 설정하여 캐시 기능과 페이징 기능 활성화
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    mov eax, cr0
    ; 읽기 전용을 위해 WP 필드(비트 16) = 1 로 추가 설정
    or eax, 0xE0010000
    xor eax, 0x60000000
    mov cr0, eax
```

간단하게 IA-32e 커널에서 일부 메모리 영역에 대한 읽기와 쓰기가 정상으로 동작하는지 확인하는 함수를 작성한다.

이 함수는 Kernel64/main.c 의 main 함수에서 동작한다.

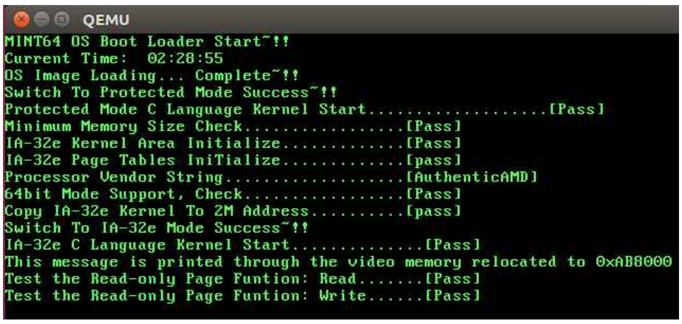
```
// Read_Only  Read 기능을 테스트하는 함수
BOOL ReadTest( void )
{
    DWORD testValue = 0xFF;

    DWORD* pdwCurrentAddress = ( DWORD* ) 0x1ff000;

    testValue = *pdwCurrentAddress;

    if(testValue != 0xFF)
    {
        return TRUE;
    }
    return FALSE;
}

// Read_Only  Write 기능을 테스트하는 함수
BOOL WriteTest( void )
{
    DWORD* pdwCurrentAddress = ( DWORD* ) 0x1ff000;
    *pdwCurrentAddress =0x12345678;

    if(*pdwCurrentAddress !=0x12345678){
        return FALSE;
    }
    return TRUE;
}
```

ReadTest 함수는 읽기 전의 변수값과 읽은 후의 변수값을 비교해서 다르다면 메모리로부터 정상적으로 읽었다고 판단한다.

WriterTest 함수는 메모리의 값을 변경하고 그 값이 정상적으로 변경되었을 때의 값과 비교해서 같다면 메모리에 정상적으로 쓰였다고 판단한다.

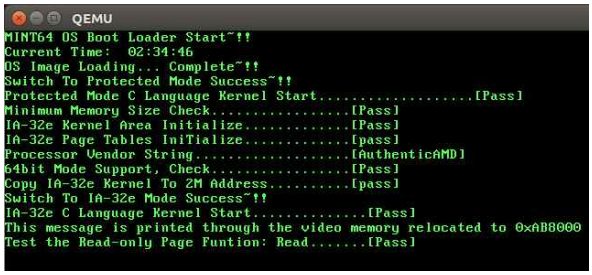
이제 다음과 같이 3 가지 경우를 확인한다.

1. 0x1FF000 번지를 읽기 전용으로 설정하는 코드를 주석 처리하는 경우



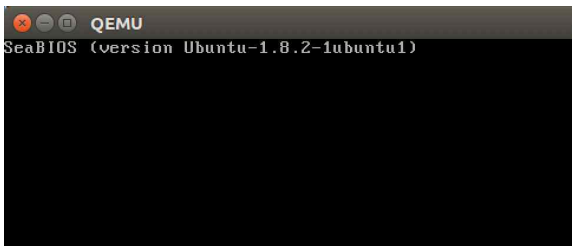
[출력결과2] 0x1FF000 번지를 정상적으로 읽고, 정상적으로 쓰는 것을 확인할 수 있다.

2. 0x1FF000 번지를 읽기 전용으로 설정하고, WriteTest를 주석처리 하는 경우



[출력결과3] 정상적으로 ReadTest까지 수행한다.

3. 0x1FF000 번지를 읽기 전용으로 설정한 경우



[출력결과4] Page Fault가 발생하여, MINT 64 OS가 계속 재부팅된다.

따라서 정상적으로 Finer-grained Page Table과 Read-only Page가 구현되었다는 것을 확인할 수 있다.

2.4 개발 시 경험한 문제점 및 해결점

- IA-32e 커널 이미지가 위치할 영역을 0 으로 왜 초기화하는지 이해하지 못한 문제점

'IA-32e 커널 이미지가 초기화되지 않은 영역을 포함하고 있지 않기 때문'이라는 것을 이해하지 못했는데, 구현을 하며 IA-32e 커널이 위치할 2MB ~ 6MB 뿐만 아니라 IA-32e 커널의 자료구조가 위치할 1MB 의 영역도 초기화를 해야 한다는 것을 알게 되었다.

- 페이징 기법에 대한 이해부족

페이징 디렉터리에 매핑되는 값이 가상 주소여서, 물리 메모리의 주소는 또 다른 방법으로 취하여야 하는 것으로 오해했다. 그래서 0xB8000 이라는 물리 메모리를 가리키는 가상 주소를 따로 구하려고 하였다. 그런데 사실 현재 우리가 만드는 MINT64 OS 는 가상 주소와 물리 주소가 선형으로 매핑 되어 있어 사실 가상 주소가 실제 물리주소를 가리키고 있다. 따라서 0xAB8000 의 페이지 테이블 엔트리 값의 baseaddress 를 0xB8000 과 같게 설정하여 문제를 해결했다.

- 0xAB8000로 비디오메모리에 접근하는 것을 32비트 모드에서 시도했던 점

Double Mapping을 위해 Page.c를 고친 후, 64 비트 모드가 아닌 32비트 모드의 kPrintString함수에 0xAB8000으로 비디오 메모리접근을 시도하고 있었다.

ModeSwitch.asm 파일의 kSwitchAndExecute64bitKernel 함수에서 페이징을 활성화하기 때문에 그 이후에 페이지 테이블의 기능을 사용해야 한다. 추가 기능구현 전에 넘겼던 부분을 다시 볼 수 있었다.

- flag 설정의 문제

페이지 테이블을 만들고 페이지 디렉터리에 맵핑도 제대로 해주었는데 계속 재부팅이 되는 문제가 있었다. 이는 새로 만든 페이지 테이블을 가리키는 디렉터리의 PAGE_FLAGS_PS 를 1로 잘못 설정해주어 페이지를 2MB로 읽어들여 잘못 접근하여 생기는 문제였다. 따라서 디렉터리와 페이지 테이블 엔트리의 flags를 제대로 설정해주니 문제가 해결되었다.

3. 조원별 기여도

신정은 34%, 조재호 33%, 조한주 33%