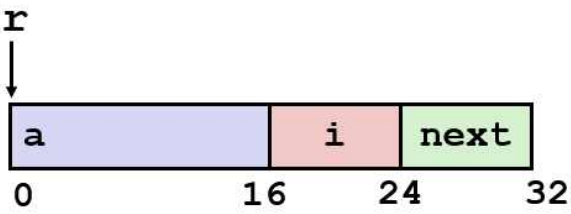
Machine Level Programming Ⅳ: Data

1. C 언어 구조체

1-1. 구조체 (Structure)

구조체(Structure)도 배열과 마찬가지로 요소들이 메모리 공간에 연속적으로 할당되며 (다만 정렬 원칙을 따르기 위해 요소들 사이에 빈 공간이 삽입될 수는 있음), **할당이 되는 순서는 구조체 내에서 선언되는 순서를 그대로 따른다.** 컴파일러는 해당 구조체의 전체 크기와 내부 요소들 각각의 위치 정보(오프셋)를 파악한 뒤 이를 바탕으로 구조체를 할당하고 접근하는 기계어 코드를 만들어 낸다. 따라서 기계어 수준의 프로그램은 구조체라는 것의 존재를 특별히 알지 못한다. 그저 메모리 상에 데이터들을 연속적으로 할당하고 그것에 접근할 뿐이다.

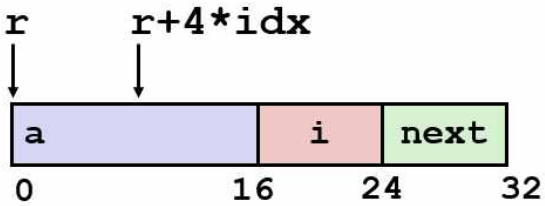
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



1-2. 구조체 접근 (Structure Access)

결국, **구조체도 배열과 마찬가지로 구조체의 시작 주소에 해당 요소의 오프셋을 더함으로써 접근하고자 하는 요소의 주소를 계산**한다. 앞서 말했듯 구조체 요소의 접근을 위한 오프셋 정보들은 컴파일러 타임에 컴파일러에 의해 파악(결정)된다. 다음은 위에서 예시로 보여줬던 구조체 내에 선언된 배열 a에 대하여 a[idx] 요소의 주소를 반환하는 함수의 코드를 나타낸다.

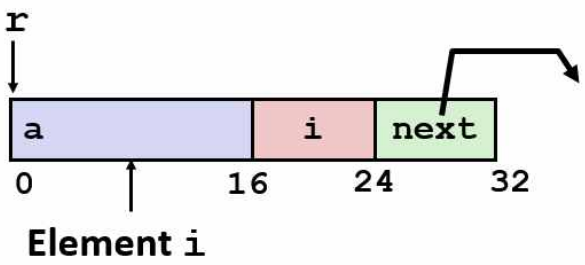
```
int *get_ap
  (struct rec *r, size_t idx)
{
  return &(r->a[idx]);
}
```




```
  # r in %rdi, idx in %rsi
  leaq   (%rdi,%rsi,4), %rax
  ret
```

다음은 **구조체로 이뤄진 연결 리스트**가 있을 때, 각 구조체 내에 존재하는 배열 a와 정수 i에 대하여 a[i]의 값을 val의 값으로 바꾸는 코드를 나타낸다. 참고로 movslq는 4바이트의 데이터를 Sign Extension 한 뒤 8바이트 공간에 이동시키는 명령어이다. 앞서 살펴본 movzbl와 유사한 성격의 명령어이다. 다음 코드를 통해 구조체 접근이 어떻게 구현되는지 확실하게 이해하고 넘어가도록 하자.

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```



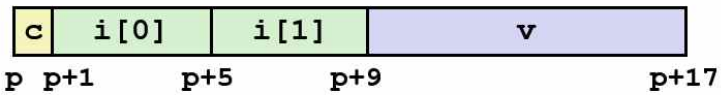

```
.L11:                           # loop:
  movslq  16(%rdi), %rax        #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)   #   M[r+4*i] = val
  movq    24(%rdi), %rdi        #   r = M[r+24]
  testq   %rdi, %rdi            #   Test r
  jne     .L11                  #   if !=0 goto loop
```

1-3. 정렬 원칙 (Alignment Principle)

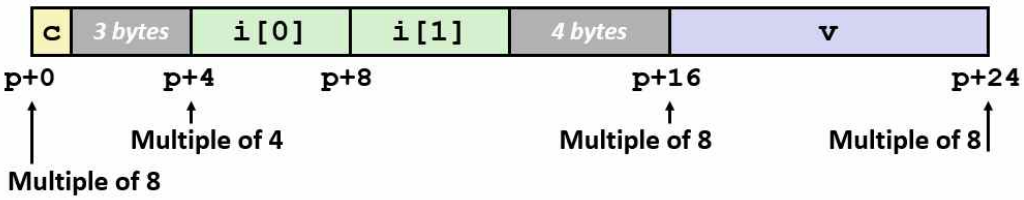
앞서 잠깐 언급했지만, 사실 구조체 내 요소들은 완전히 연속적으로 할당되지는 않고 그 사이사이에 빈 공간이 존재한다. 왜 그럴까? 이는 성능 향상을 위한 **정렬 원칙(Alignment Principle)**을 준수하기 위해서이다. 정렬이 무엇인가를 이해하기 위해 간단한 예시를 살펴보자. 다음의 첫 번째 그림은 요소들이 정렬되지 않은 구조체, 두 번째 그림은 요소들이 정렬된 구조체이다.

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

■ **Unaligned Data**



■ **Aligned Data**



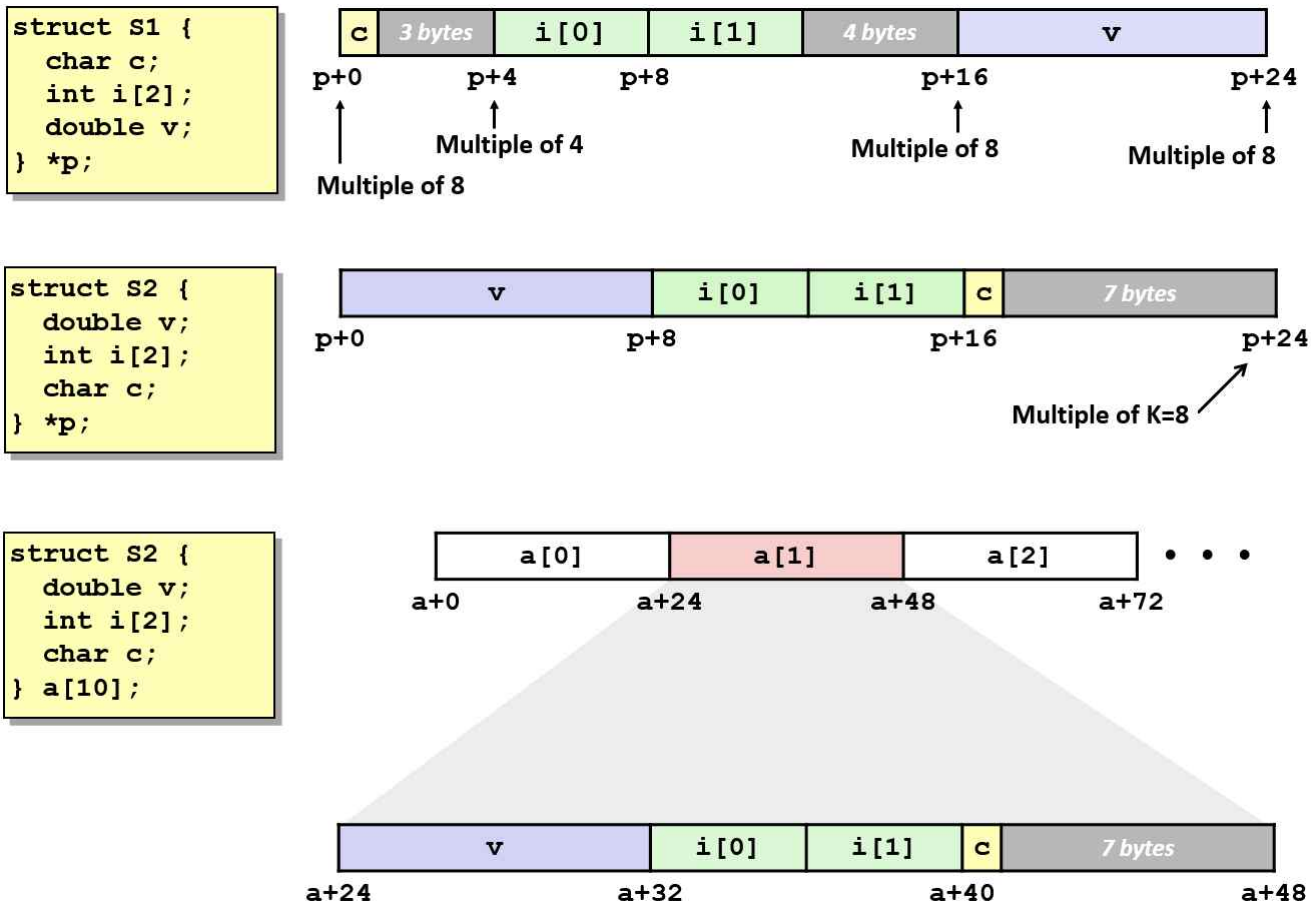
1-3-1. 구조체 내 각 요소들의 정렬 원칙

K 바이트 크기의 자료형이라면 시작 주소도 K 바이트여야 한다. 예를 들어,x86-64 기준으로 double, long 데이터 혹은 포인터는 8 바이트 데이터이므로 그 데이터의 시작 주소도 8의 배수여야 한다. 참고로 2^n의 배수임을 판별하는 방법은 하위 비트에 0이 n 개 있는지 확인하는 것이다.

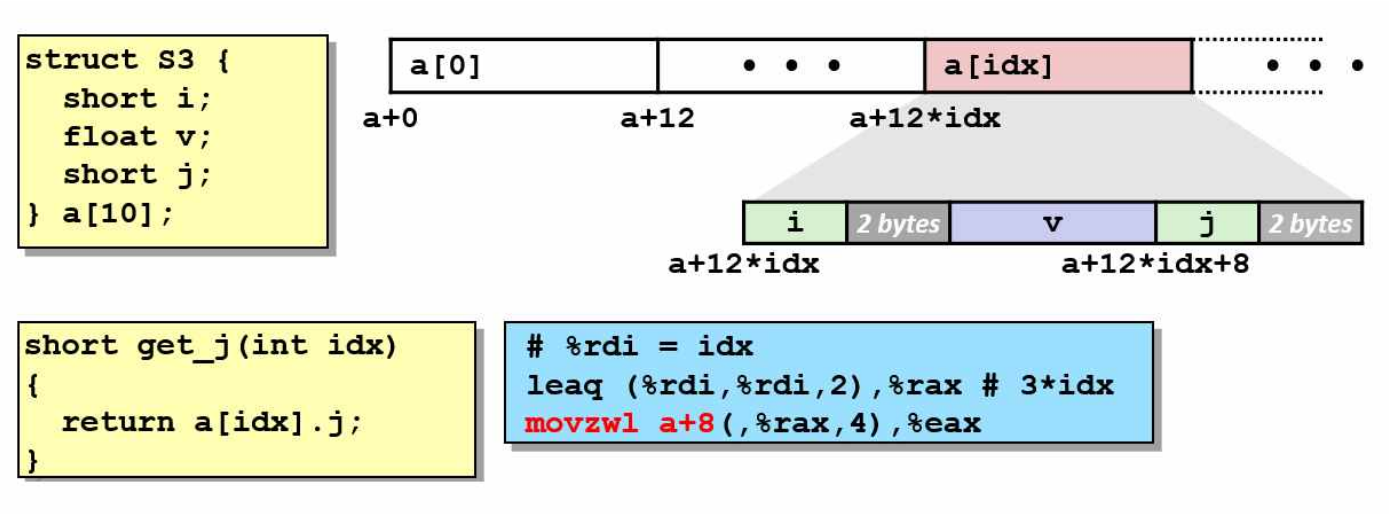
1-3-2. 구조제 전체의 정렬 원칙

구조체 내 요소 중 가장 큰 데이터의 크기가 K 바이트라 할 때, 해당 **구조체의 시작 주소와 길이는 K의 배수여야 한다.** 그러면 이와 같은 **정렬은 왜 필요한 것일까?** 메모리에 접근할 때 실제로 가져오는 것은 메모리 상에 연속적으로 할당되어 있는 4바이트 혹은 8바이트 단위의 청크인데, 하나의 데이터가 여러 청크에 걸쳐 있으면 메모리 접근의 효율이 상당히 떨어진다. 또한 가상 메모리 기술에서도 하나의 데이터가 여러 페이지(Page)에 걸쳐 있으면 성능이 저하된다. 이러한 이유로 특정 시스템에서는 구조체의 정렬 원칙을 권장하고 있다. 물론 그렇지 않은 시스템의 경우에는 정렬 원칙을 지킬 필요가 없다. 참고로x86-64의 경우 구조체의 정렬 원칙을 지킬 것을 권장하고 있다. 구조체의 정렬 원칙이 권장되는 시스템에서는 컴파일러의 기본 옵션으로 구조체 정렬이 설정되어 있다. 이러한 경우 **컴파일러는 구조체 내 요소들과 구조체 전체의 정렬을 보장하기 위해서 구조체 내 요소들 사이사이에 적절히 갭을 삽입**하게 된다.

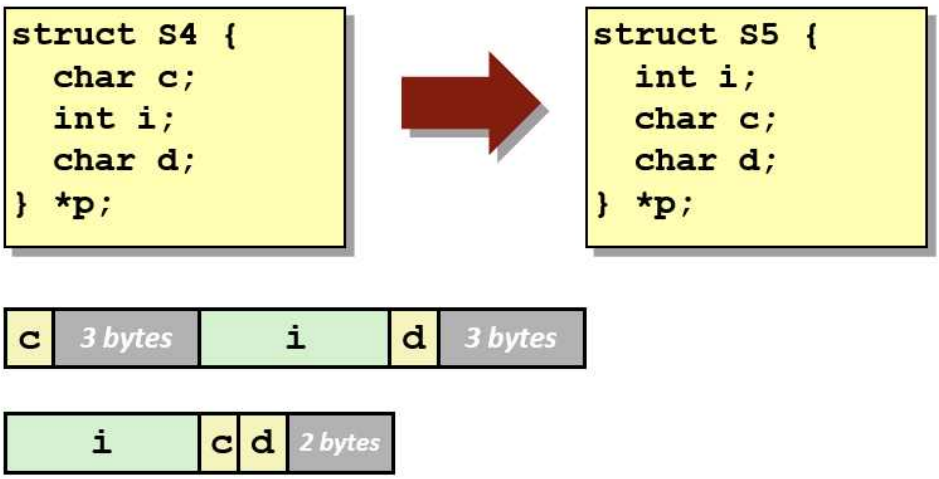
다음 그림들을 보며 구조체 정렬이 어떻게 이뤄지는지 직접 한 번 살펴보자.



다음은 조금 더 심화된 예시로, S3 구조체로 이뤄진 배열 a에 대하여 a[idx]의 멤버 j에 접근하는 코드를 나타낸다. 참고로 빨간색으로 표시된 a+8은 컴파일 타임에 결정할 수 없는 값에 해당하며, 나중에 링커에 의해 링킹이 진행될 때 올바른 값으로 채워진다.



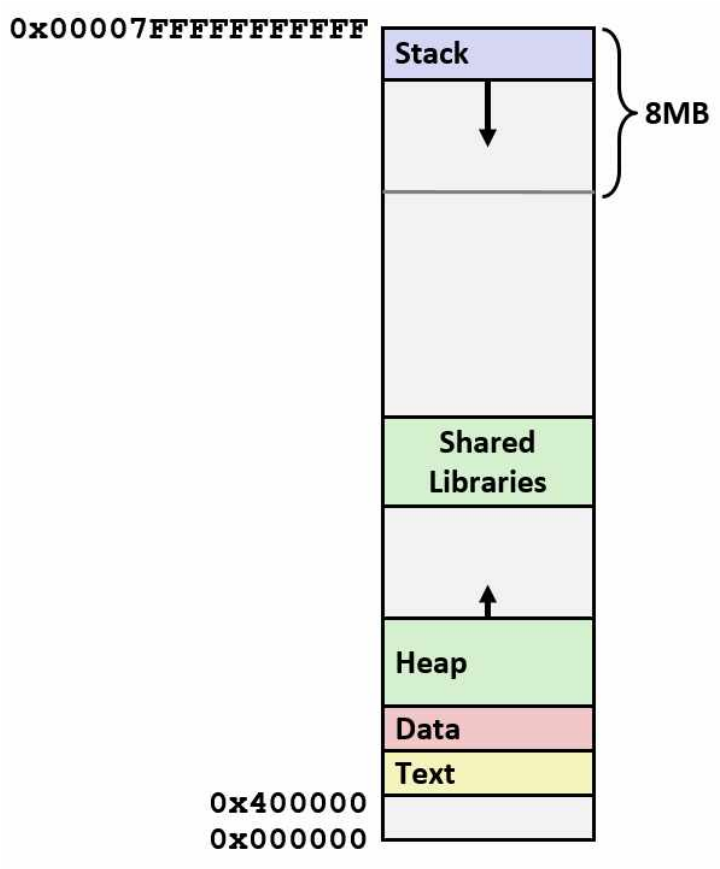
지금까지 이야기한 구조체의 정렬 원칙에 따르면, **구조체 내 요소들을 크기가 큰 것부터 선언해야 공간을 가장 효율적으로 쓸 수 있다**는 결론에 도달한다. 다음 그림을 보며 이 점을 이해해 보기 바란다.



Machine Level Programming V: Advanced Topics

1. 메모리 레이아웃

1-1. x86-64 메모리 레이아웃 (Memory Layout)



메모리 레이아웃(Memory Layout)이란 프로그램이 실행될 때 해당 프로그램의 데이터와 코드가 어느 곳에 어떻게 할당되는지를 나타내는 도식이다. 엄밀하게는 프로그램의 데이터와 코드가 로드되는 **가상 주소 공간(Virtual Address Space)의 구조**를 나타낸다. 물론 이는 시스템마다 다르다. 예를 들어 x86-64에서 컴파일되는 프로그램을 실행하면 해당 프로그램의 데이터와 코드가 오른쪽 그림과 같이 가상 주소 공간에 로드된다.

1-1-1. 스택 (Stack)

다른 말로 런타임 스택이라고도 하며, 8MB의 공간 제한이 있다. 지역 변수 등의 함수 내 지역 데이터가 저장되는 영역이다.

1-1-2. 힙 (Heap)

동적으로 할당되는 데이터들이 위치하는 영역이다. C 언어의 malloc(), calloc() 함수 등이 호출될 때 사용된다.

1-1-3. 데이터 (Data)

정적으로 할당되는 데이터들이 위치하는 영역이다. 전역 변수와 static 변수, 그리고 문자열 등의 상수가 이곳에 저장된다.

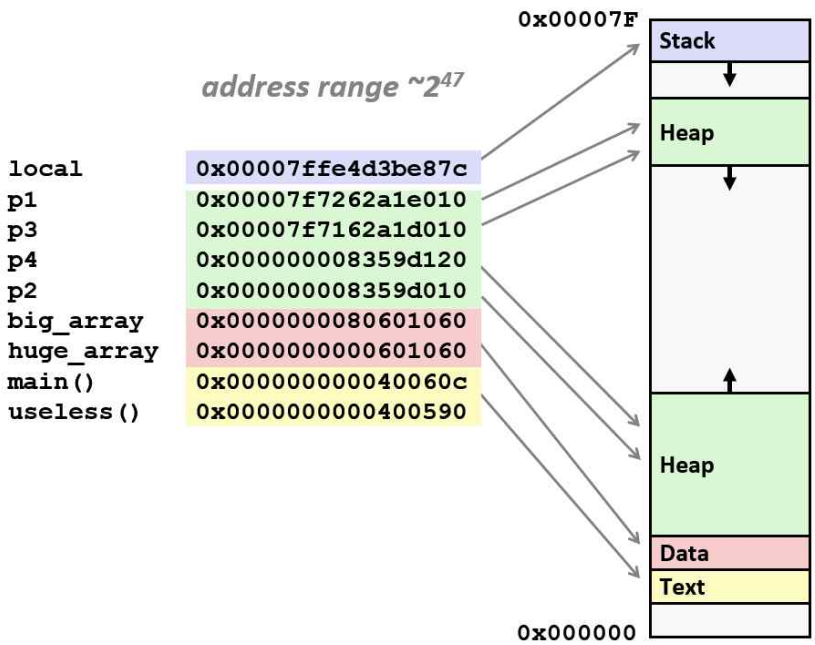
1-1-4. 텍스트 (Text), 공유 라이브러리 (Shared Libraries)

프로그램 코드에 해당하는 명령어들이 순차적으로 저장되는 읽기 전용(Read-only) 영역이다. 프로그램 실행 파일의 코드는 텍스트 영역, 공유 라이브러리의 코드는 별도의 공간에 로드된다.

1-2. 데이터/코드 할당 예시

이해를 돕기 위해 다음 예시를 살펴보자. 전역 변수에 해당하는 big_array, huge_array, global은 데이터 영역에 로드되고, 지역 변수에 해당하는 local은 스택 영역에 로드된다. 그리고 malloc으로 동적 할당된 데이터들(각각 p1, p2, p3, p4가 가리킴)은 힙 영역에 로드가 된다. 마지막으로 함수에 해당하는 useless()와 main()의 코드는 텍스트 영역에 로드가 된다.

```
char big_array[1L << 24];  /* 16 MB */
char huge_array[1L << 31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
    /* Some print statements ... */
}
```



2. 버퍼 오버플로우

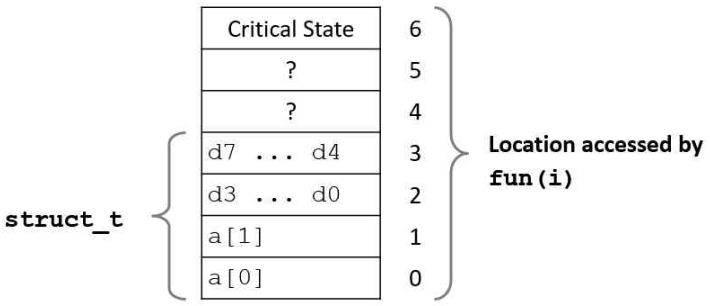
2-1. 버퍼 오버플로우 (Buffer Overflow)

버퍼 오버플로우(Buffer Overflow)는 메모리를 대상으로 하는 대표적인 시스템 해킹 기법 중 하나로, **메모리에 할당된 공간을 의도와 다르게 접근하여 조작**하는 것을 말한다. 이해를 돕기 위해 간단한 예시를 살펴보자. 오른쪽 그림은 현재의 스택 프레임에 struct_t 타입의 구조체 지역 변수가 할당되어 있는 구조를 보여준다. 원칙대로라면 구조체 멤버에 해당하는 배열 a는 길이가 2이

므로 a[0]과 a[1]만 올바른 접근이라고 할 수 있다. **하지만 컴파일러는 a가 int 형 데이터로 이뤄진 배열의 첫 번째 요소를 가리키는 포인터라는 것만 알고, 그 배열의 길이는 따로 기억하지 않는다.** 따라서 a[2], a[3]도 정상적으로 컴파일이 되고 실행으로까지 이어진다. 이러한 경우 3.14에 해당하는 값을 d 자리에 올바르게 저장했는데도 불구하고 배열 a에 대한 잘못된 접근으로 인해 d 자리에 잘못된 값이 쓰이게 된다. 이처럼 **허용된 공간을 초과하여 메모리에 접근함으로써 메모리를 조작**하는 것이 버퍼 오버플로우의 대표적인 사례이다.

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0)  → 3.14
fun(1)  → 3.14
fun(2)  → 3.1399998664856
fun(3)  → 2.00000061035156
fun(4)  → 3.14
fun(6)  → Segmentation fault
```



2-2. 버퍼 오버플로우 발생 이유

그렇다면 메모리 보안을 위협하는 버퍼 오버플로우 공격이 이뤄질 수 있는 이유는 무엇일까? 앞서 간단히 언급했지만, 가장 큰 이유는 **배열의 길이 정보를 컴파일러가 기억하지 않기 때문**이다. 따라서 특정 배열에 일련의 값들 혹은 문자열을 저장하는 경우 그 길이를 검사할 수 없는 것이다. 이 때문에 해당 배열의 길이를 넘는 데이터를 의도적으로 삽입하여 해커의 의도대로 메모리를 조작할 수 있게 된다.

2-3. 버퍼오버플로우 예시

그렇다면 앞서 든 예시보다 조금 더 실제 버퍼 오버플로우 공격에 가까운 예시를 하나 살펴보자. 우선 다음은 C 언어의 내장 함수인 **gets()의 내부 구현**을 보여준다. dest는 입력받을 문자열이 저장될 공간의 주소를 나타내는데, gets() 입장에서는 그 공간의 크기가 얼마나 되는지 알 방법이 없기 때문에 파일의 끝이나 개행 문자를 만날 때까지 문자 하나를 읽고 저장하기를 반복한다. 이는 또 다른 C 언어 내장 함수인 strcpy(), strcat(), scanf(), fscanf(), sscanf() 등에서도 발생하는 문제이다.

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

이제 이러한 get() 함수의 특성을 이용하여 버퍼 오버플로우 공격을 행하는 경우를 살펴보자. call_echo() 함수가 echo() 함수를 호출하면, **키보드로부터 문자열을 입력받아 그것을 4바이트만큼 할당된 배열 buf에 저장**하게 된다. 실행 결과에서도 볼 수 있듯이 **4바이트 이상의 문자열을 입력하여도 문제없이 동작**하는 것을 볼 수 있다. 다만 심각한 수준의 메모리 침범은 내부적으로 탐지하는 방법이 있어서(설명 생략), 이러한 경우 Segmentation Fault 예외가 발생하게 된다.

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
void call_echo() {
    echo();
}
```

실행 결과

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901 23
012345678901234567890123
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123 4
Segmentation Fault
```

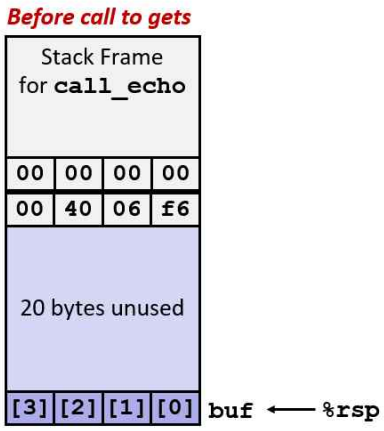
위의 코드를 어셈블리어로 번역한 결과와 메모리 할당 구조는 다음과 같다. echo() 함수의 스택 프레임에는 4바이트 길이의 buf 배열과 20바이트만큼의 사용하지 않는 빈 공간이 할당된다. 따라서 24바이트 길이의 문자열까지는 입력 시 별다른 문제가 발생하지 않는다. 그렇다면 이제 gets() 함수가 호출되어 키보드로부터 특정 길이의 문자열을 입력받은 뒤의 상황을 대략 세 가지 경우로 나눠서 살펴보자.

echo:

```
00000000004006cf <echo>:
  4006cf:  48 83 ec 18           sub    $0x18,%rsp
  4006d3:  48 89 e7              mov    %rsp,%rdi
  4006d6:  e8 a5 ff ff ff        callq  400680 <gets>
  4006db:  48 89 e7              mov    %rsp,%rdi
  4006de:  e8 3d fe ff ff        callq  400520 <puts@plt>
  4006e3:  48 83 c4 18           add    $0x18,%rsp
  4006e7:  c3                    retq
```

call_echo:

```
  4006e8:  48 83 ec 08           sub    $0x8,%rsp
  4006ec:  b8 00 00 00 00        mov    $0x0,%eax
  4006f1:  e8 d9 ff ff ff        callq  4006cf <echo>
  4006f6:  48 83 c4 08           add    $0x8,%rsp
  4006fa:  c3                    retq
```

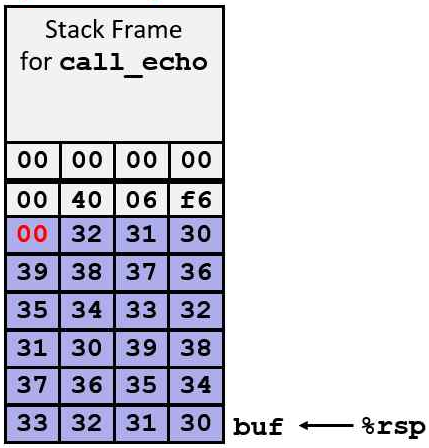


2-3-1. 문자열의 길이 ≤ 24바이트

첫 번째 경우는 24바이트보다 작거나 같은 길이의 문자열이 입력된 경우이다. 이러한 경우 echo() 함수의 복귀 주소(Return Address)를 손상하지 않기 때문에 별다른 문제없이 프로그램이 정상 동작한다.

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
```

After call to gets

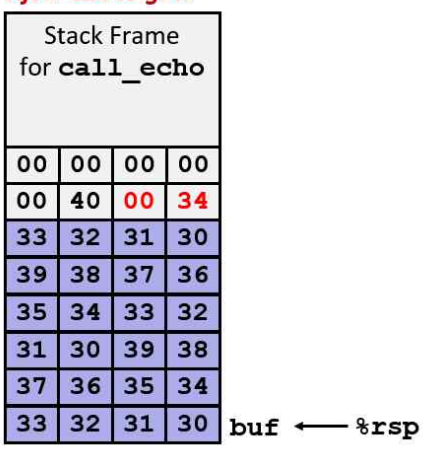


2-3-2. 문자열의 길이 > 24바이트 (① Segmentation Fault 발생 O)

두 번째는 비정상적으로 긴 문자열이 입력되어 복귀 주소가 완전히 훼손되고, Segmentation Fault 예외가 발생하는 경우이다.

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault
```

After call to gets



2-3-3. 문자열의 길이 > 24바이트 (② Segmentation 발생 X)

마지막은 24바이트보다 긴 문자열이 입력됨에도 불구하고 Segmentation Fault가 발생하지 않고 복귀 주소를 일부 망가뜨리는 경우이다. 이러한 경우 echo() 함수의 복귀 주소가 원래의 값을 잃어버리므로 예상치 못한 코드로 리턴하게 된다.

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
```

After call to gets

Stack Frame for **call_echo**

00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

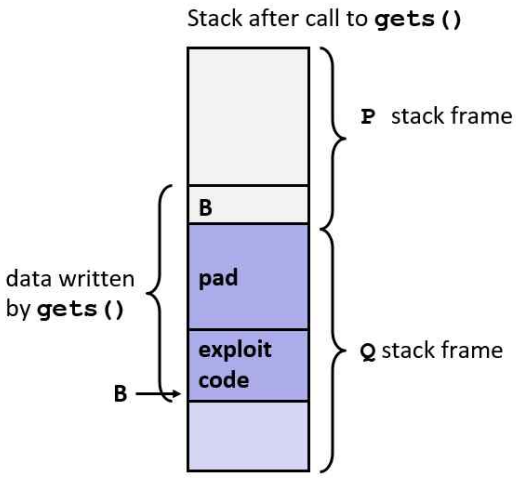
2-4. 코드 인젝션 (Code Injection)

위에서 기술한 버퍼 오버플로우 공격의 대표적인 사례가 바로 **코드 인젝션(Code Injection)**이다. 다음 예를 보자. P() 함수가 Q() 함수를 호출하면 키보드로부터 문자열을 입력받아 64바이트 길이의 배열 buf에 저장하게 된다. 다음 그림에서 A로 표시된 부분은 Q()가 P()로 돌아가기 위한 복귀 주소인데, 버퍼 오버플로우 공격을 통해 그 값을 B로 바꿔버릴 수 있다. 그리고 B는 악의적인 목적으로 입력된 문자열이 저장되는 공간에 위치하는 코드들의 시작 주소가 되게 하는 것이다. 그러면 Q()는 P()로 올바르게 리턴하지 못하고 **해커가 의도한 코드로 리턴함으로써 잘못된 코드가 실행**이 되는 것이다.

```
void P(){
  Q();
  ...
}
```

address A

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

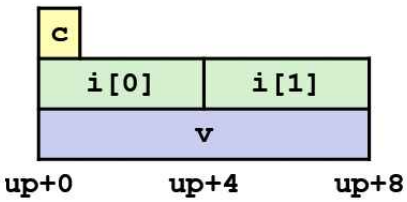


3. 공용체, 바이트 오더링

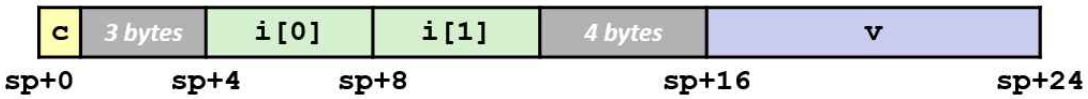
3-1. 공용체 (Union)

C언어의 공용체(Union)은 구조체(Structure)와 여러 모로 비교가 많이 되는 자료구조이다. 구조체는 멤버 변수들이 메모리 상에 선언된 순서대로 할당이 되지만, 공용체는 하나의 공간을 여러 변수가 공유하여 사용한다.

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```

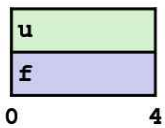



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



그렇다면 공용체 내 멤버 변수는 어떻게 접근이 이뤄질까? **공용체의 가장 큰 장점은 하나의 공간을 여러 자료형의 관점에서 사용할 수 있다는 것**이다. 예를 들어 "int a"로 선언된 a라는 공간에는 2의 보수로 표현되는 정수 값만 저장될 수 있다. 만약 다른 자료형의 값을 넣으려고 하면 int 형으로 자동 형 변환을 수행하거나, 에러가 발생하게 된다. 그러나 공용체는 그렇지 않다. 위의 예시와 같이 선언된 공용체가 있다고 가정할 때, c에 접근하면 문자를 저장할 수 있고 v에 저장하면 실수를 저장할 수 있다.

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```




```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

Same as (float) u ? → No!

```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Same as (unsigned) f ? → No!

다음 예를 살펴보자. 이를 이해할 수 있다면 위에서 기술한 개념을 전부 이해했다고 보아도 무방하다. bit2float() 함수의 경우, 매개변수로 전달되는 정수 값을 공용체에 저장한 뒤 그 값을 float 형으로 반환한다. **즉 비트 상에는 u에 해당하는 정수 값이 2의 보수로 표현되어 있을 텐데, 그 비트 배열은 그대로 둔 채 해석만 float 방식으로 하겠다**는 것을 의미한다. 이는 명백하게 "(float) u"와 다르다. 이는 u에 해당하는 값을 똑같이 표현할 수 있는 실수 값을 나타내도록 비트의 배열을 완전히 바꿀 것이기 때문이다. float2bit() 함수도 마찬가지로 해석하면 된다.

3-2. 바이트 오더링 (Byte Ordering)

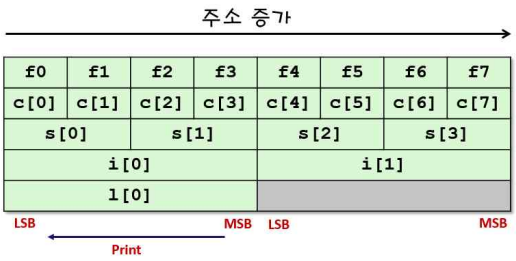
바이트 오더링(Byte Ordering)이란 메모리에 연속적으로 할당되는 데이터들의 비트 배열 방향을 나타낸다. **주소가 가장 작은 부분이 MSB이면 Big Endian 방식, 주소가 가장 큰 부분이 MSB이면 Little Endian 방식**이라고 한다.(사람이 읽기 편한 방식은 Big Endian이라고 할 수 있다. 보통 비트 배열은 왼쪽에서 오른쪽으로 쓰고, 주소도 오른쪽으로 증가한다고 보는 게 편하기 때문이다.) 바이트 오더링은 기계들끼리 이진 데이터를 주고받기 위해 통일되어야 하는 특성으로서, 시스템마다 다르다. **x86-64의 경우 Little Endian 방식을 채택**하고 있다. 참고로 두 방식을 혼합하여 사용하는 Bi Endian이라는 방식도 존재한다.

다음은 한 공용체 내에 선언된 char 형 배열, short형배열, int형배열, long형배열을 통해 시스템의 바이트 오더링이 무엇인지 파악하기 위한 코드를 나타낸다. 직접 노트를 펴고 메모리 구조도를 그려보면서 다음 코드를 꼭 이해하고 넘어가기 바란다. 이를 이해하면 바이트 오더링은 대부분 이해한 것이다.(참고로 이 셋 중 x86-64만 64비트, 나머지는 32비트 ISA에 해당한다.)

```
union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l[1];
} dw;
```

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 == [0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
       dw.c[0], dw.c[1], dw.c[2], dw.c[3], dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
       dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
       dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
       dw.l[0]);
```

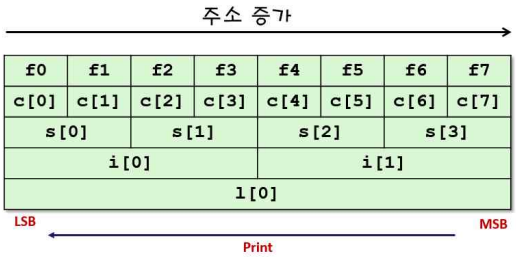
Little Endian (IA32)



Output:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf3f2f1f0]
```

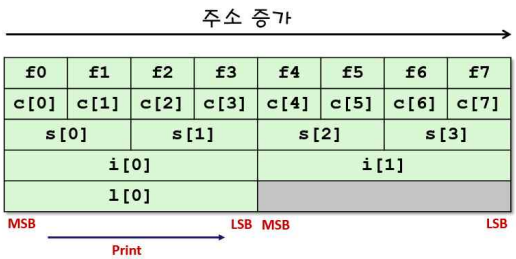
Little Endian (x86-64)



Output:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

Big Endian (Sun)



Output on Sun:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```

Program Optimization

1. 여러 가지 프로그램 최적화 기법

1-1. 비효율적인 함수 호출 최소화

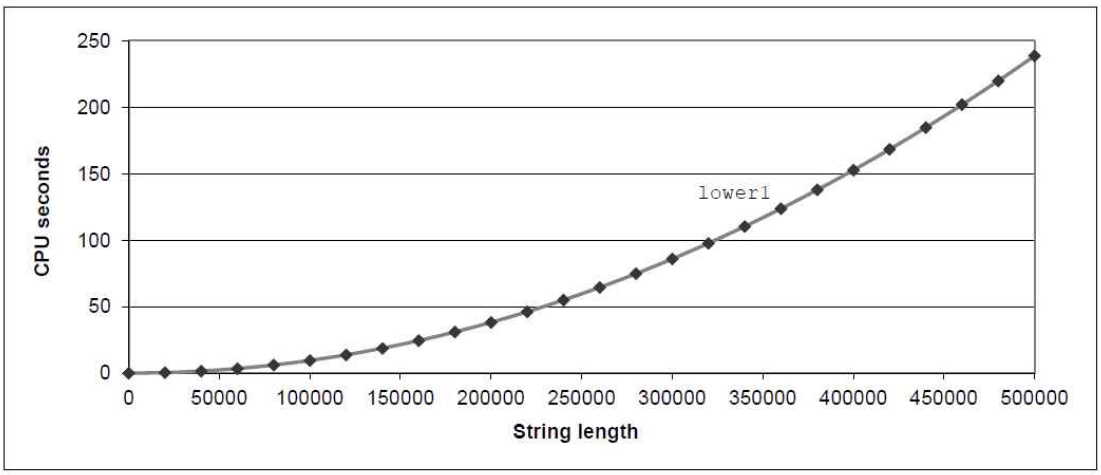
아래 코드의 문제점은 무엇일까?

```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

strlen 함수가 for문이 한 번씩 반복될 때마다 매번 호출되는 것이 문제다. 보통은 컴파일러가 이러한 경우도 최적화할 수 있다고 생각하지만, 생각보다 모든 최적화를 해주지는 않는다. 위 코드를 아래처럼 고치면 최적화할 수 있다.

```
void lower(char *s)
{
  size_t i;
  size_t len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

이 둘의 성능 차이가 얼마나 심할지 그래프를 통해 살펴보자.



이만큼 차이가 난다. 왜 이만큼 차이가 날까? **strlen은 문자열 s의 길이 n에 비례하는 실행 시간을 가지고, lower 함수 내의 for문은 문자열 s의 길이 n만큼 반복한다.** 결과적으로 **첫 번째 lower 함수는 n의 제곱에 비례하는 실행 시간을 가진다.** 두 번째 lower 함수는 strlen 함수 호출과 반복문을 분리했고, **결과적으로 문자열 길이 n에 비례하는 실행 시간을 가진다.**

1-2. 비효율적인 메모리 참조 최소화

아래 코드의 문제점은 무엇일까? 어셈블리 코드도 같이 확인해보자. 명령어 옵션 –O1으로 gcc를 실행한 결과다.

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
# sum_rows1 inner loop
.L4:
        movsd   (%rsi,%rax,8), %xmm0        # FP load
        addsd   (%rdi), %xmm0               # FP add
        movsd   %xmm0, (%rsi,%rax,8)        # FP store
        addq    $8, %rdi
        cmpq    %rcx, %rdi
        jne     .L4
```

세상에... 루프마다 메모리 액세스를 세 번이나 한다. 왜 **컴파일러는 매 루프마다 b[i]의 값을 메모리에서 로드하고 저장하게 만들어놨을까?** 답은 아래와 같은 경우가 발생할 수 있어서다.

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

Value of B:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 22, 16]
```

```
i = 2: [3, 22, 224]
```

a와 b가 같은 메모리 영역을 참조하고 있다. **이러한 상황을 memory aliasing이라고 부른다.** 매 루프마다 b 배열에 값을 저장하는 행위가 a 배열에도 영향을 줄 수 있다. 그래서 우리는 컴파일러에게 그러지 말라고 알려줄 필요가 있다. 지역변수를 도입하자.

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

```
# sum_rows2 inner loop
.L10:
        addsd   (%rdi), %xmm0       # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L10
```

이제 메모리 액세스는 루프 당 하나뿐이다.

1-3. 최적화 예제

vector 자료형에 대한 연산이 정의된 combine 함수를 여러 기법을 사용해 최적화시키겠다. 가장 첫 번째로 작성된 함수, combine1이다. OP는 연산자다. combine4는 이를 앞의 두 기법으로 최적화한 코드이다.

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

다음은 위 두 코드의 성능을 비교한 표이다. 확연히 다른 모습을 확인할 수 있다.

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

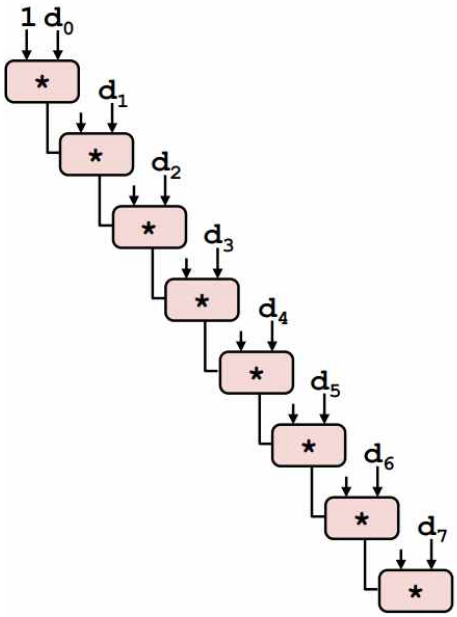
1-4. 루프 풀기 (Loop Unrolling)

매 루프마다 계산되는 원소의 수를 증가시켜서 루프 반복실행 횟수를 줄이는 방법이다. 이 방법을 통해 얻는 이득은 두 가지다.

1. 루프 인덱스 계산, 조건부 분기와 같은 연산과 직접적인 연관이 없는 연산을 줄인다.
2. 전체 연산의 수를 줄이는 관점에서 최적화를 적용할 수 있다.

combine4 함수에 loop unrolling을 적용해보자. 한 루프에 2개의 연산을 하도록 loop unrolling을 적용하면 다음과 같은 코드와, 기계어, 그리고 도식이 나온다. 이러한 변환을 2 * 1 loop unrolling이라고 부른다.

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```



이 기법의 CPE는 다음과 같다. 정수 덧셈 연산에 대해서는 성능이 향상됐지만 다른 연산에서는 큰 성능 향상이 보이지 않는다.

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

1-5. 재결합 변환(Reassociation Transform)

병렬성 향상을 위해 코드를 변환하는 방법이다. 위에서 loop unrolling한 함수 중 일부 코드를 다음과 같이 변형해보자.

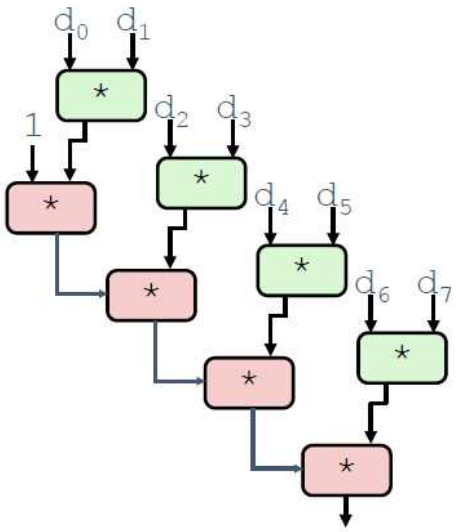
```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Compare to before

```
x = (x OP d[i]) OP d[i+1];
```

괄호 위치만 바뀌었는데 성능에 큰 차이가 생길까? 결과부터 얘기하면 그렇다. 왜 그럴지 살펴보자. 둘의 차이점을 비교하면, 인스트럭션은 하나가 늘었다. 그리고 사용하는 레지스터도 하나 늘었다. 눈에 보이는 인스트럭션 차이는 중요하지 않다. 이전의 코드대로는 x에 대한 첫 번째 곱연산이 끝날 때까지 두 번째 곱연산을 할 수 없었다. 순차적 의존성 때문이다. 그러나 지금 코드는 x에 대한 곱연산을 하는 동안, 다음 루프의 x에 대한 곱연산을 실행할 수 있다. 그러므로 프로세서의 병렬 처리가 제대로 이루어진다면 꽤 많은 실행시간을 절약할 수 있다. 그렇다면 CPE는 어떻게 나올까? 아래와 같다. 거의 2배에 가까운 성능 향상을 보여준다.

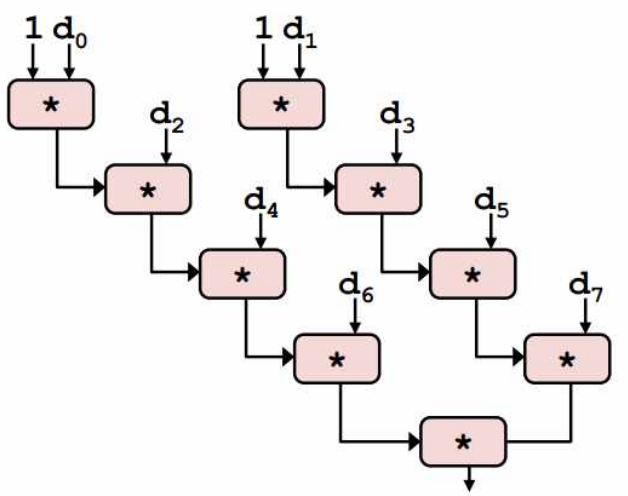
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50



1-6. 병렬 누산기(Separate Accumulators)

지역변수를 명시적으로 여러 개 설치해서 순차적 의존성을 깰 수도 있다. 이전에 재결합 변환을 통해 최적화한 코드를 누산기를 적용하여 다음과 같이 변형해보자.

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

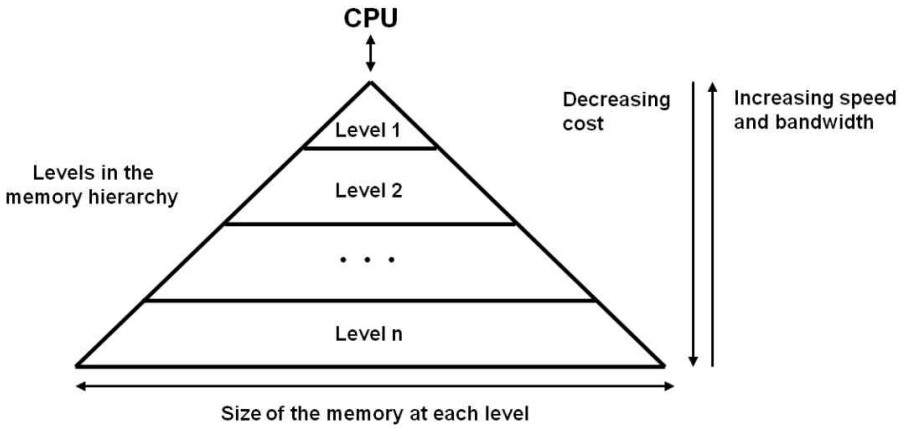


무엇이 다른걸까? 지금의 코드는 서로 독립적인 두 가지 stream을 따라 코드가 작동함을 볼 수 있다. 다음은 위 코드의 CPE이다.

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

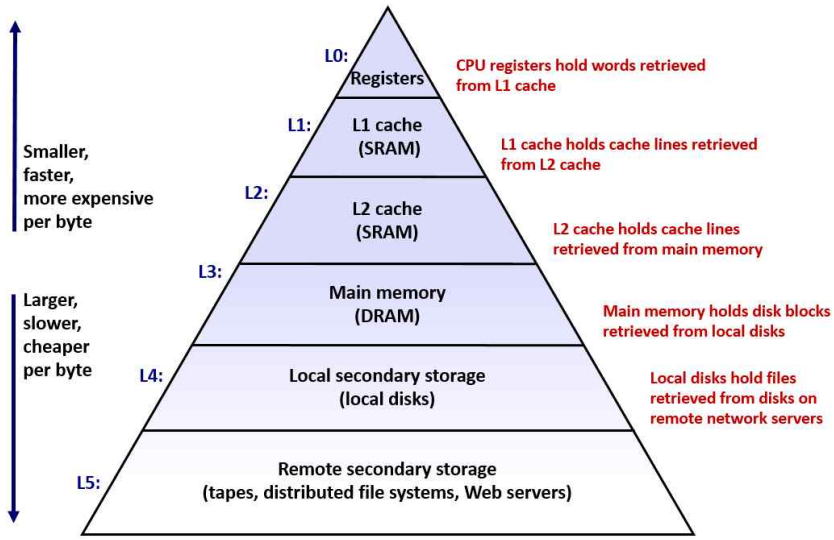
Memory Hierarchy

1. 메모리 계층 (Memory Hierarchy)



DRAM은 비트 당 가격이 저렴하지만 접근 속도가 매우 느리고, SRAM은 비트 당 가격이 비싸지만 접근 속도가 매우 빠르다. 이때 뒤에서 설명할 **지역성(Locality)을 잘 활용하면, DRAM과 SRAM을 동시에 사용하여 비트 당 가격은 DRAM만큼 저렴하고 접근 속도는 SRAM만큼 빠른 메모리 소자가 있는 듯한 효과를 만들어낼 수 있다.** SRAM으로 만들어지는 캐시 메모리가 DRAM에서 자주 참조되는 부분들을 저장하고, CPU는 접근 속도가 더 빠른 SRAM에 먼저 접근함으로써 메모리 참조의 성능을 엄청나게 향상시킬 수 있다. 이와 같이 메모리는 DRAM 하나로만 되어 있지 않고 메모리 참조의 성능을 개선시키기 위한 여러 계층으로 구성되어 있는데, 이를 **메모리 계층(Memory Hierarchy)**이라고 한다. 메모리 계층의 구조는 위 그림과 같다. 접근 속도가 빠르지만 비싼 메모리 소자는 CPU와 가까운 계층에 위치시키고, 접근 속도는 느리지만 저렴하여 큰 용량으로 만들 수 있는 메모리 소자는 CPU와 먼 계층에 위치시킨다. 그리고 N번째 계층의 메모리 소자는 (N+1)번째 계층의 메모리 소자에서 자주 참조되는 부분들을 저장하도록 한다. 이렇게 하면, CPU는 메모리 참조가 필요할 때마다 자신과 가까운 계층에 존재하는 메모리 소자부터 접근함으로써 메모리 참조를 더욱 효율적으로 수행할 수 있게 된다.

실제로 현대 컴퓨터의 메모리 계층은 대략 다음과 같은 구조로 나타낼 수 있다. 위에서 설명한 것보다 훨씬 더 다양한 계층이 존재한다는 것을 볼 수 있다. 물론 당장 이 그림을 이해하는 것은 불가능에 가깝다. 그러니 그냥 이런 게 있구나 하고 간단히 보고만 넘어가자. 나중에 공부를 마친 뒤 돌아오면 안 보이던 게 보일 것이다.



2. RAM (Random Access Memory)

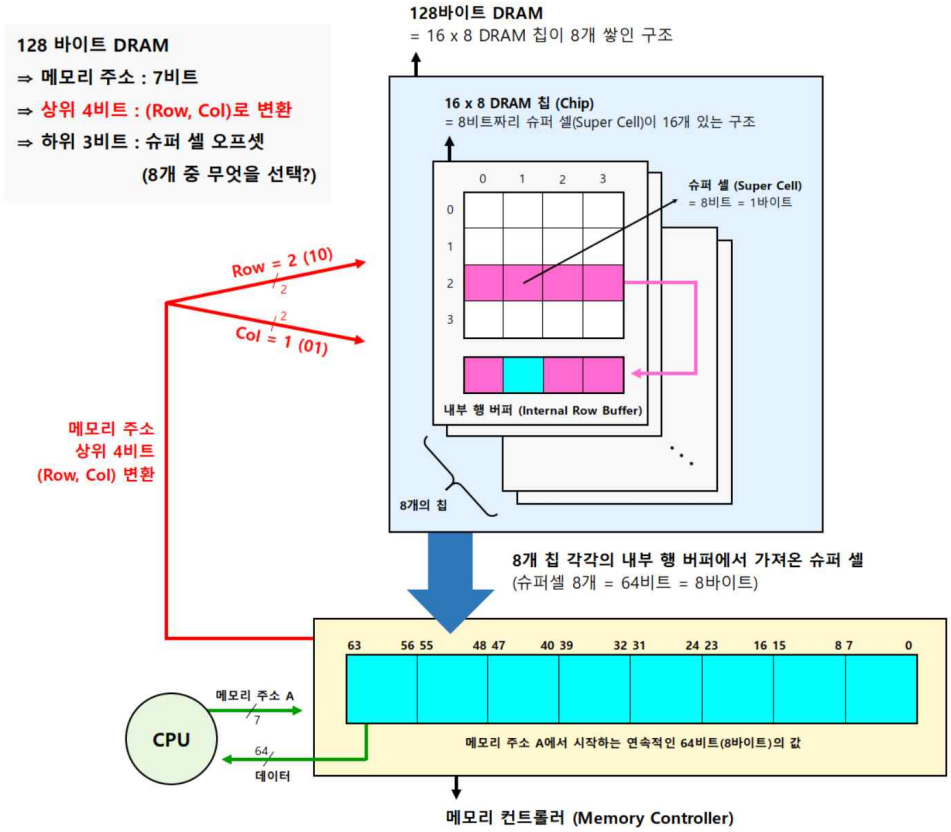
2-1. DRAM vs SRAM

RAM(Random Access Memory)은 컴퓨터에서 보조 기억 장치로 사용되는 메모리 소자를 의미한다. 기본적으로 각 셀(Cell)은 1비트를 저장하고, 여러 셀이 모여서 하나의 칩(Chip)을 이룬다. 그리고 이러한 칩을 여러 개 쌓아서 최종적으로 RAM을 만들게 된다. 자세한 구조는 아래에서 설명하는 내용을 참고하자.

DRAM(Dynamic RAM)의 경우, 각 셀은 하나의 축전기를 사용하여 1비트를 저장하며, 접근 시에는 하나의 트랜지스터가 사용된다. 따라서 SRAM에 비해서는 **비트 당 가격이 저렴하다.** 그러나 SRAM보다 **접근 속도가 느리고**, 방사선 등의 외부 방해 요소에 취약하다. DRAM에 저장되는 값은 오래 두면 휘발될 수 있기 때문에 **10-100ms 간격으로 계속 Refresh 작업을 수행해줘야 한다.** 일반적으로 DRAM은 메인 메모리로서 사용된다.

SRAM(Static RAM)의 경우, 각 셀은 무려 4~6개의 트랜지스터로 이뤄진 회로를 사용하여 1비트를 저장한다. 따라서 DRAM에 비해서는 **비트 당 가격이 비싸다.**그러나 DRAM보다 **접근 속도가 빠르고**, 방사선 등의 외부 방해 요소에 크게 영향받지 않는다. SRAM에 저장되는 값은 **전력이 계속 공급되는 한 휘발되지 않고 안전하게 유지된다.** 일반적으로 SRAM은 캐시 메모리를 만드는 데 사용된다.

2-2. DRAM의 구조와 인터페이스



왼쪽 그림은 DRAM의 전반적인 구조를 나타낸다. 128바이트 크기의 DRAM을 가정하고 있다. 위 그림을 기준으로 DRAM의 데이터를 참조하는 과정을 이해해 보자. 먼저, **CPU가 접근하고자 하는 메모리의 물리 주소를 메모리 컨트롤러에 입력하면, 메모리 컨트롤러는 하위 3비트를 제외한 나머지 비트를 (Row, Col)로 변환**한다. 이는 각 DRAM 칩에서 어떤 행과 어떤 열에 위치한 슈퍼 셀을 고를 것인지를 나타내는 것이다. 그렇게 변환한 후에는 우선 Row 정보를 바탕으로 각 DRAM 칩에서 행을 하나씩 골라 해당 칩의 내부 행 버퍼로 가져온다. 그리고 Col 정보를 바탕으로 각 내부 행 버퍼에서 알맞은 슈퍼 셀을 고른 뒤, 그렇게 칩의총개수만큼 선택된 슈퍼 셀들을 메모리 컨트롤러로 가져온다. 그러면 메모리 컨트롤러에는 8바이트 크기의 데이터가 들어오게 된다. 그러면 이제 마지막으로 **CPU가 입력한 메모리 주소의 하위 3비트 정보를 바탕으로 몇 번째 바이트부터 읽을 것인지 결정하고, 그곳에서부터 몇 바이트를 읽어야 하는지를 나타내는 제어 신호의 정보를 바탕으로 몇 바이트만큼 읽을지 결정**한다. 그렇게 결정된 범위의 데이터를 이제 CPU로 전달해주면 메모리 참조는 끝이 난다.

여기서 유추할 수 있는 사실이 하나 있다. **만약 모든 데이터가 메모리 상에 정렬 원칙(Alignment Principle)에 따라 배치되어 있다면, 적은 횟수의 참조만으로도 메모리의 데이터를 효율적으로 가져올 수 있다**는 것이다. 만약 정렬 원칙을 따르지 않고 데이터가 메모리 상에 존재한다면 8바이트 데이터를 읽을 때에도 두 번의 참조가 필요할 수도 있다. (메모리 컨트롤러로 가져오는 데이터의 단위를 블록이라고 칭한다고 가정할 때) 한 데이터가 두 블록에 걸쳐 있을 수도 있기 때문이다. 이는 상당히 비효율적이다.

2-3. DRAM의 발전

위에서 설명한 **기본적인 DRAM의 내부 구조와 참조 동작 방식은 초창기와도 크게 다르지 않을 만큼 유지**되어 왔다. 그러나 **인터페이스의 로직이나 입출력 작업의 속도는 날이 갈수록 발전**되었다. 예를 들어 SDRAM(Synchronous DRAM)은 Row 주소를 재활용 가능하도록 개선하였고, 기존의 비동기적 제어 대신에 클락 신호를 사용한 동기적 입출력을 수행함으로써 클락의 Rising-edge 때 값을 읽거나 쓸 수 있도록 하였다. 또한 DDR SDRAM(Double Data Rate Synchronous DRAM)은 클락의 Rising-edge 때뿐만 아니라 Falling-edge 때도 값을 읽거나 쓸 수 있도록 하여 입출력 수행 속도를 배로 향상시켰다.

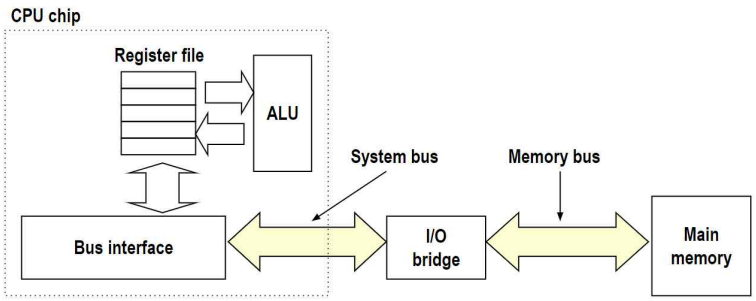
2-4. 비휘발성 메모리

앞서 설명한 DRAM은 휘발성 메모리에 속한다. 전력 공급이 중단되면 저장하고 있던 정보를 모두 잃기 때문이다. 반면 **비휘발성 메모리**도 존재한다. 이는 **전력 공급이 중단되어도 저장하고 있는 정보를 그대로 유지**한다. 비휘발성 메모리의 대표적인 사례를 몇 가지 알아보자.

첫째는 ROM(Read-only Memory)이다. 기본적으로 제조 단계에서 모든 정보를 저장하며, 이후 내용 수정이 불가능하다. 둘째는 PROM(Programmable ROM)으로, 사용자의 입맛대로 최초에 한 번은 프로그래밍(원하는 정보를 저장)할 수 있는 ROM을 의미한다. 셋째는 EPROM(Erasable PROM)으로, 자외선이나 X-Ray 등으로 정보를 지울 수 있는 PROM을 의미한다. 넷째는 EEPROM(Electrically Erasable PROM)으로, 전기적 특성을 이용하여 정보를 지울 수 있는 PROM을 의미한다. 마지막은 EEPROM과 유사한 특성을 가진 플래시 메모리(Flash Memory)이다.

비휘발성 메모리는 여러 분야에서 활용된다. 기본적인 펌웨어 프로그램들(EX. 바이오스)이 ROM에 주로 저장되며, 무엇보다 하드 디스크를 대신하는 주기억 장치로 떠오르고 있는 **SSD가 플래시 메모리 기반의 저장 장치**이다. 스마트폰, MP3 플레이어, 태블릿, 노트북 등의 단말기에서 사용되는 저장 장치를 만드는 데 핵심적인 역할을 수행하고 있다.

2-5. CPU와 메모리의 통신 구조 (인터페이스)



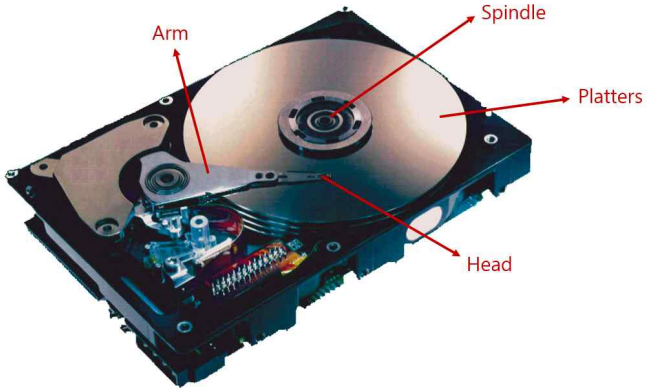
버스(Bus)란 간단히 말해서 주소, 데이터, 컨트롤 신호 등을 운반하는 도선을 의미한다. 버스를 통해서는 한 번에 해당 CPU의 워드 사이즈만큼의 데이터를 교환할 수 있다. 예를 들어 64비트 CPU의 버스는 64개의 도선이 하나의 버스를 이룬다. 물론 버스는 데이터의 교환을 위한 수단이기 때문에, 여러 장치가 공유하여 사용할 수 있다. 위 그림을 기준으로 메모리의 값을 읽을 때와 메모리에 값을 쓸 때의 동작 방식을 간단히 알아보자.

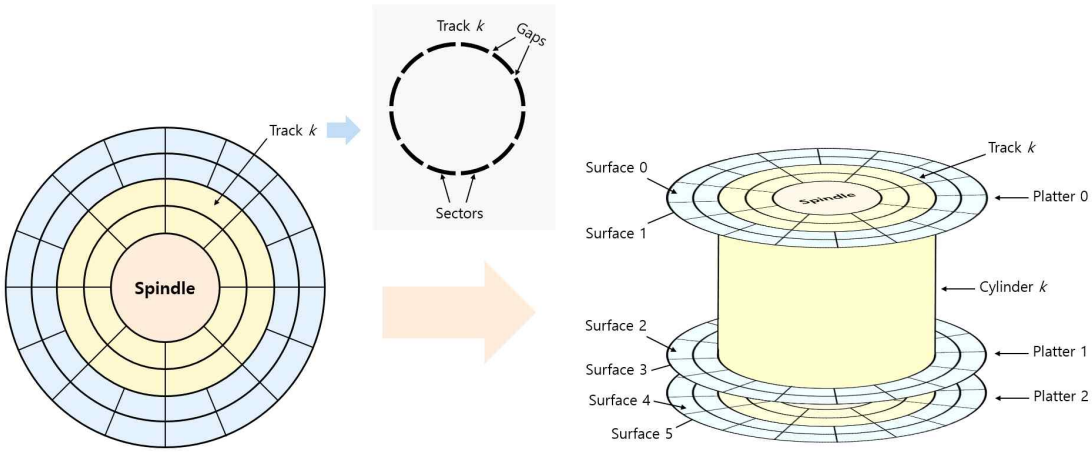
먼저 **메모리의 값을 읽을 때**이다. 예를 들어 movq A, %eax 명령어를 실행한다고 해보자. 먼저, CPU가 메모리 주소 A를 시스템 버스에 올려서 메모리 버스로 전달한다. 그러면 메인 메모리가 메모리 버스에서 A를 읽고 그 주소에 위치한 값 X를 로드하여 다시 메모리 버스에 올린다. 이제 CPU가 다시 버스에서 X를 읽어서 %eax에 복사하면 명령어 실행이 마무리된다.

다음으로 **메모리에 값을 쓸 때**이다. 예를 들어 movq %eax, A 명령어를 실행한다고 해보자. 먼저, CPU가 값을 쓸 메모리 주소 A를 시스템 버스에 올려서 메모리 버스로 전달한다. 그러면 메인 메모리는 메모리 버스에서 A를 읽고 CPU가 쓸 값을 줄 때까지 기다린다. 그러다가 CPU가 쓸 값 Y를 버스에 올리면, 메인 메모리는 버스에서 Y를 읽어서 A가 가리키는 공간에 저장함으로써 명령어 실행을 마무리한다.

3. 하드 디스크 (Hard Disk)

3-1. 하드 디스크 구조





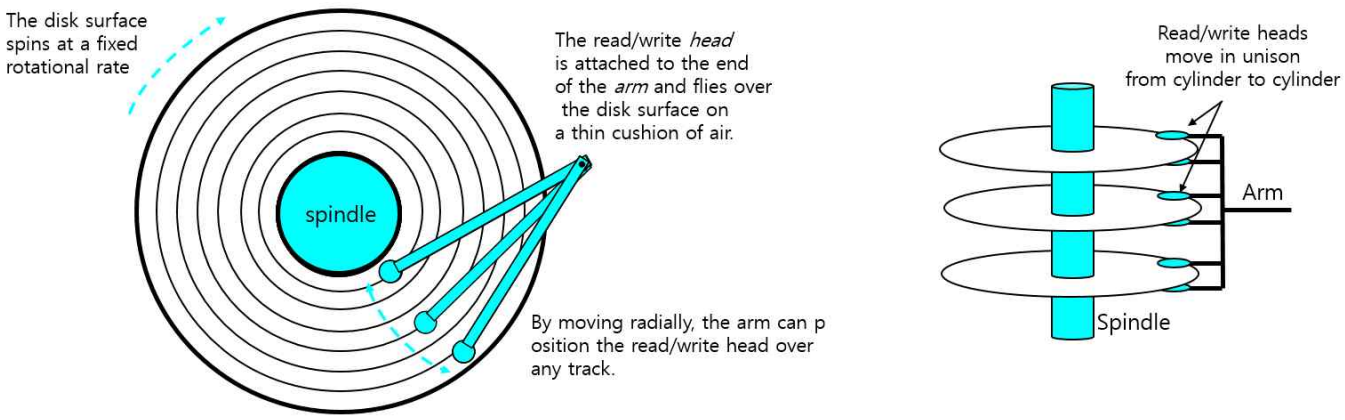
3-2. 용량 계산

용량(Capacity)은 하드 디스크가 저장할 수 있는 최대 비트 수를 의미하며, 일반적으로 GB 혹은 GiB 단위로 표현한다. 1GB는 10^9 바이트를 의미하고, 1GiB는 2^30 바이트를 의미한다. 하드 디스크의 용량을 결정짓는 대표적인 요인으로는 대략 세 가지가 있다. 첫 번째는 **Recording Density**로, 한 트랙의 1인치 세그먼트에 들어가는 비트의 수를 의미한다. 두 번째는 **Track Density**로, 1인치 길이의 세그먼트에 들어가는 트랙의 수를 의미한다. 마지막은 **Areal Density**로, 1제곱인치 면적에 들어가는 비트의 수를 의미한다. 이는Recording Density와 Track Density를 곱해서 얻을 수 있다.

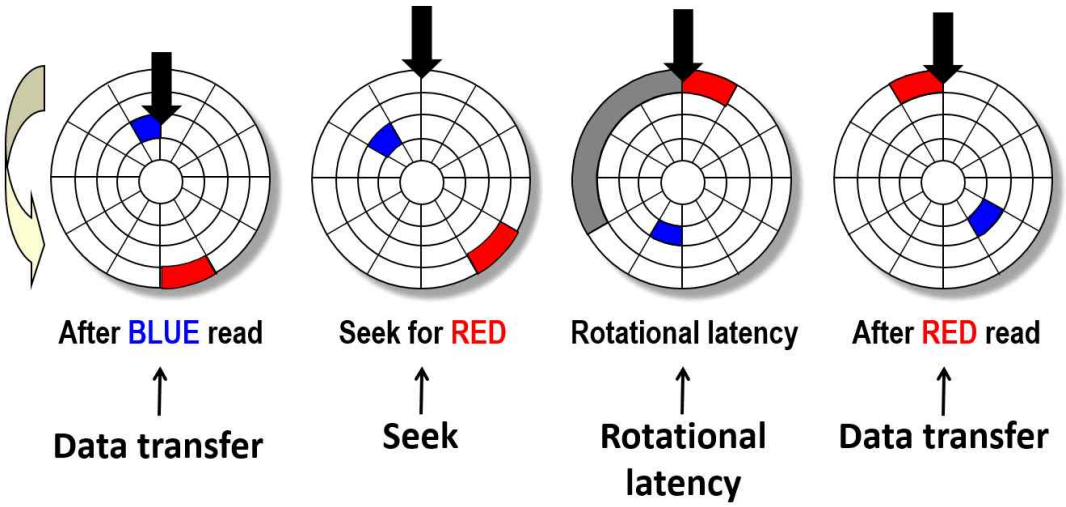
참고로 **하드 디스크는 몇 개의 Recording Zone으로 나눠져 있고, 각 Recording Zone에는 고정된 개수의 섹터들이 존재**한다. 위 그림을 기준으로, 노란색 영역과 하늘색 영역이 각기 다른 Recording Zone을 의미한다. 동일한 Recording Zone에서는 트랙 당 섹터의 개수도 동일한 반면, Recording Zone이 다르면 트랙 당 섹터의 개수도 다르다는 것을 관찰할 수 있다. Recording Zone의 트랙 당 섹터의 개수는 가장 안쪽에 위치한 트랙에 의해 결정지어진다. 이렇듯 Recording Zone에 따라 트랙 당 섹터의 개수가 다르기 때문에, **하드 디스크의 용량을 계산할 때는 트랙 당 평균 섹터의 개수를 먼저 계산하여 이를 사용**하게 된다.하드 디스크의 용량을 계산하는 방법은 다음과 같다.

⇒ **Capacity = (# bytes/sector) x (평균 # sectors/track) x (# tracks/surface) x (# surfaces/platter) x (# platters/disk)**

3-3. 접근



하드 디스크의 데이터에 접근하는 과정은 이러하다. 먼저, 해당 데이터를 포함하는 섹터가 위치한 트랙을 찾는다. 이를 위해 암(Arm)은 끝 부분에 붙어 있는 헤드(Head)가 해당 데이터를 포함하는 트랙의 위에 놓일 때까지 위 그림과 같이 회전한다. 이렇게 트랙을 찾는 데까지 걸리는 시간을 **Seek Time**이라고 한다. 트랙을 찾고 나면, 이제 해당 데이터를 포함하는 섹터를 찾아야 한다. 이를 위해 플래터는 고정된 속도로 위 그림과 같이 반시계 방향으로 회전한다. 이렇게 해당 데이터를 포함하는 섹터의 첫 번째 비트를 찾는 데까지 걸리는 시간을 **Rotional Latency**라고 한다. 이제 찾은 섹터 내에 존재하는 데이터를 읽기만 하면 된다. 이때 걸리는 시간을 Transfer Time이라고 한다. 하드 디스크 접근 과정을 그림으로 요약하면 다음과 같다.



위에서 설명한 내용을 바탕으로, 간단한 예시를 통해 **하드 디스크 접근 시간**을 계산해보도록 하자. 플래터의 회전 속도가 7,200 RPM이고, 평균 Seek Time은 9ms이며, 트랙 당 평균 섹터의 개수는 400개인 하드 디스크를 가정해보자. 그러면 평균 Rotational Latency와 Transfer Time은 다음과 같이 계산할 수 있다. 그리고 그 둘과 Seek Time을 더하면 최종적인 접근 시간을 알아낼 수 있다.

⇒ **평균 Rotational Latency = (1/2) x (60 secs / 7200 RPM) x (1000 ms/sec) = 4ms**

⇒ **평균 Transfer Time = (60 secs / 7200 RPM) x (1/400 secs/track) x (1000 ms/sec) = 0.02ms**

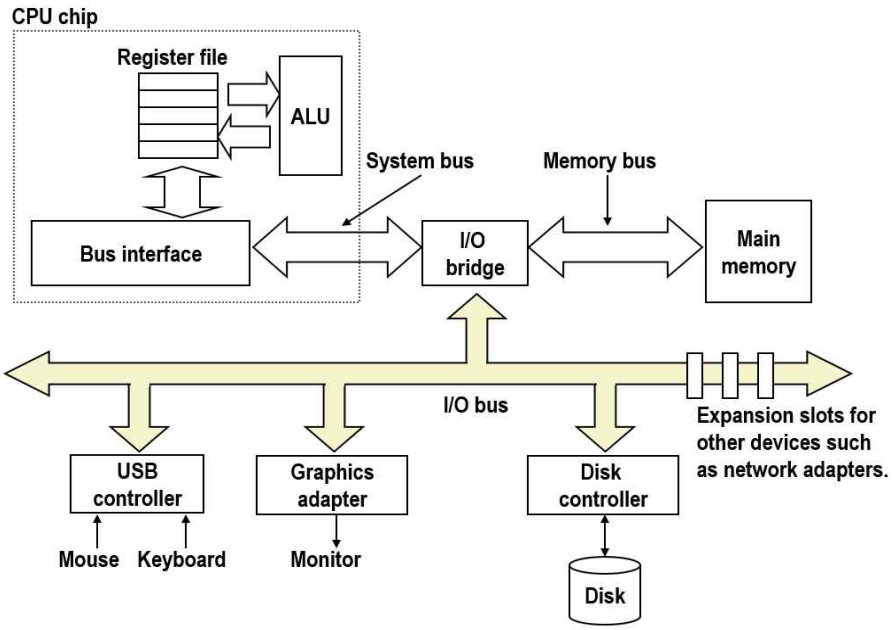
⇒ **평균 Access Time = 9ms + 4ms + 0.02ms = 13.02ms**

여기서 주목할 것은 두 가지이다. 첫 번째, 접근 시간은 Seek Time과 Rotational Latency의 영향력이 지배적이다. 따라서 이 둘을 줄일 수 있다면 접근 시간도 상당히 많이 감소할 것이다. 두 번째, 이미 알려진 사실과 같이 하드디스크의 접근 속도는 SRAM이나 DRAM에 비해 눈에 띄게 느리다는 것이다. 하드디스크는 SRAM보다 대략 40,000배 느리며, DRAM보다는 2,500배 느리다.

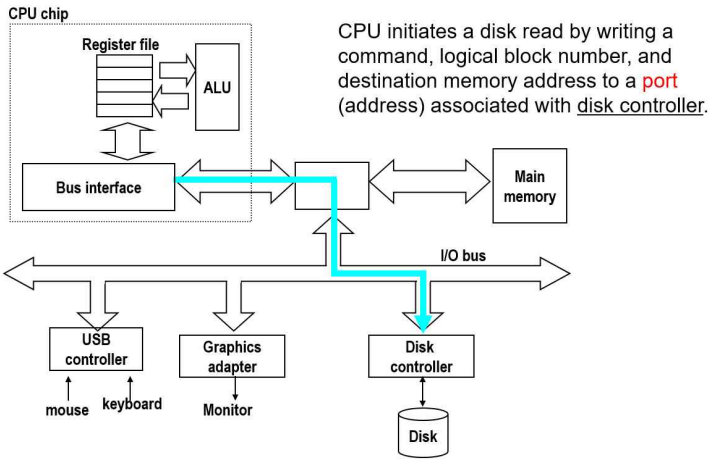
3-4. CPU와 하드 디스크의 통신 구조 (인터페이스)

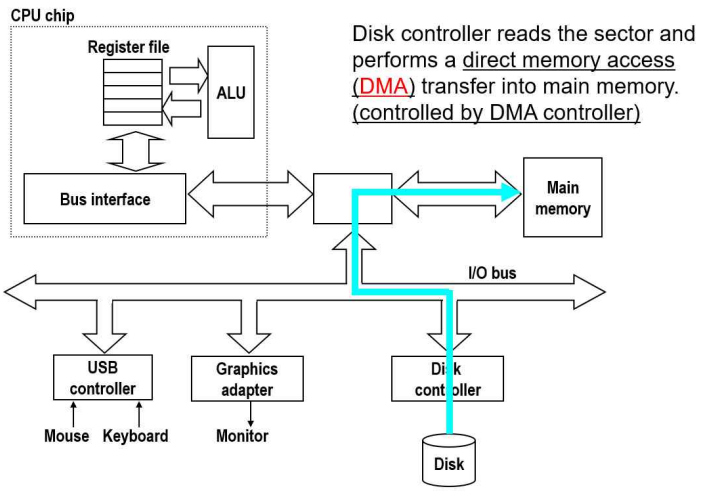
지금까지 하드 디스크의 구조와 접근 원리를 살펴보았는데, 하드 디스크를 사용하는 입장에서 이런 것들까지 다 알고 있기란 쉽지 않다. 그래서 **섹터 단위의 데이터들의 집합을 B개의 논리적 블록들로 모델링하여 추상화**하게 된다. 그리고 논리적 블록과 실제(물리적) 섹터들 사이의 맵핑은 **디스크 컨트롤러**라 불리는 하드웨어/펌웨어 장치에 의해서 유지된다. 즉, **외부에서 특정 데이터의 참조를 위한 논리적 블록을 요청하면 이를 (Surface, Track, Sector) 정보로 변환하여 하드 디스크에 전달함으로써 요청된 데이터를 포함하는 섹터를 반환**하도록 하는 것이다. 이는 앞서 살펴보았던 메모리 컨트롤러의 역할과 유사하다. 메모리 컨트롤러도 메모리의 복잡한 내부 구조를 감추고 외부에서는 단순히 메모리 주소만을 가지고도 원하는 데이터를 참조할 수 있도록 추상화해주기 때문이다.

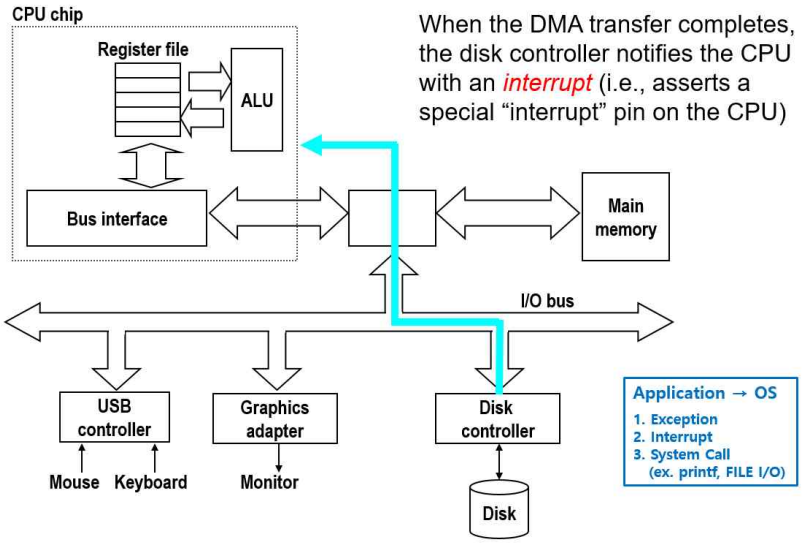
다음 그림은 앞에서 살펴본 **메모리 인터페이스를 포함하여, 하드 디스크를 비롯한 외부 입출력 장치들과의 인터페이스**를 보여주고 있다. 메모리와 외부 입출력 장치들은 I/O 브릿지를 경유지로 삼아서 CPU와 통신하며, I/O 버스에는 여러 개의 입출력 장치들을 장착할 수 있는 슬롯들이 마련되어 있다. 참고로 x86 CPU는 메모리와 통신하기 위한 명령어로서 mov, pop, push 등을 갖추고 있고, 외부 입출력 장치들과 통신하기 위한 명령어는 별도로 갖추고 있다. 즉 외부 입출력 장치에 접근할 때는 아래 그림에서 볼 수 있듯이 메모리에 접근할 때와는 다른 버스를 사용하게 된다. 이는 메모리 주소 공간의 일부를 입출력 장치들에게 부여하는 방식과는 또 다른 방식에 해당한다.



위 그림을 기준으로 CPU가 하드 디스크의 데이터를 참조하는 과정을 살펴보자. 먼저, 외부 입출력 장치와의 통신을 위해 별도로 마련되어 있는 명령어를 CPU가 실행한다. 해당 명령어에는 접근하고자 하는 입출력 장치의 인터페이스 포트(주소), 참조하고자 하는 데이터의 논리적 블록 정보, 데이터를 로드하여 위치시킬 메모리 주소, 로드할 데이터의 크기 등의 정보들이 담겨 있다. 그러면 디스크 컨트롤러는 요청된 데이터를 포함하는 섹터를 읽은 후, DMA(Direct Memory Access) 컨트롤러를 통해서 메인 메모리에 그 데이터를 로드하게 된다. 이 과정에 CPU는 개입하지 않기 때문에 시스템 버스는 Free 한 상태가 된다. 이제 DMA가 데이터를 성공적으로 메모리에 로드하고 나면, CPU에게 인터럽트를 걸어서 해당 작업이 완료되었음을 알리게 된다.

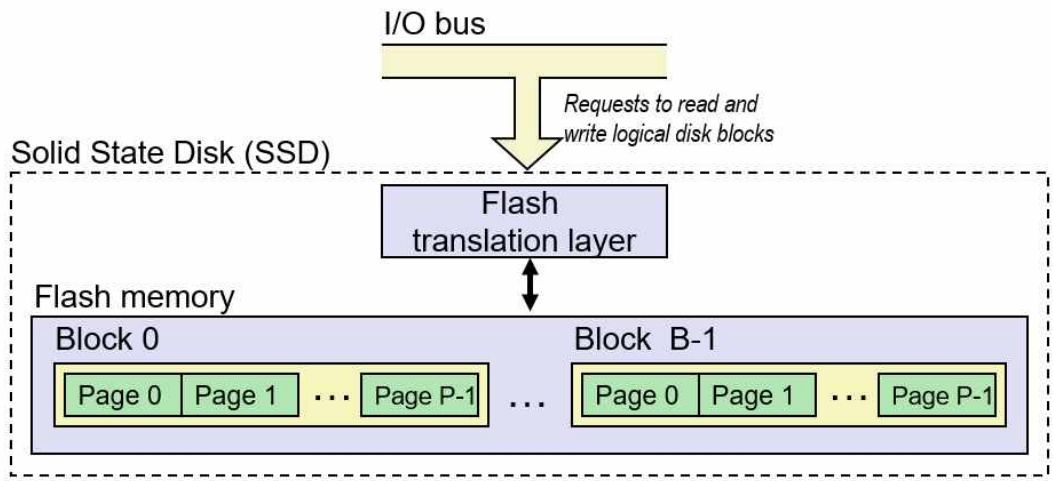






4. SSD (Solid State Disks)

4-1. SSD 구조 및 특징



SSD(Solid State Disk)는 플래시 메모리 기반의 주기억 장치로, 하드 디스크와 마찬가지로 논리적 디스크 블록 단위에 기초한 인터페이스를 갖추고 있다. 바깥층에는 Flash Translation Layer가 위치하고, 안쪽에는 플래시 메모리가 위치한다. **플래시 메모리 내 데이터는 블록 단위로 나눠지며, 각 블록 내 데이터는 다시 페이지 단위로 나눠진다.** 보통 블록은 32개~128개의 페이지를 가지고 있으며, 한 페이지는 대략 512B~4KB의 크기를 가진다. **SSD를 대상으로 한 읽기/쓰기 작업은 페이지 단위**로 이뤄지며, 특히 **쓰기 작업의 경우 한 페이지를 수정하기 위해서는 해당 페이지가 속한 블록 내 모든 페이지를 지워야 한다.** 또한 **특정 횟수(EX. 10만번) 이상의 쓰기 작업이 이뤄진 블록은 수명이 다하여** 더 이상 사용할 수 없게 된다.

4-2. 성능 평가

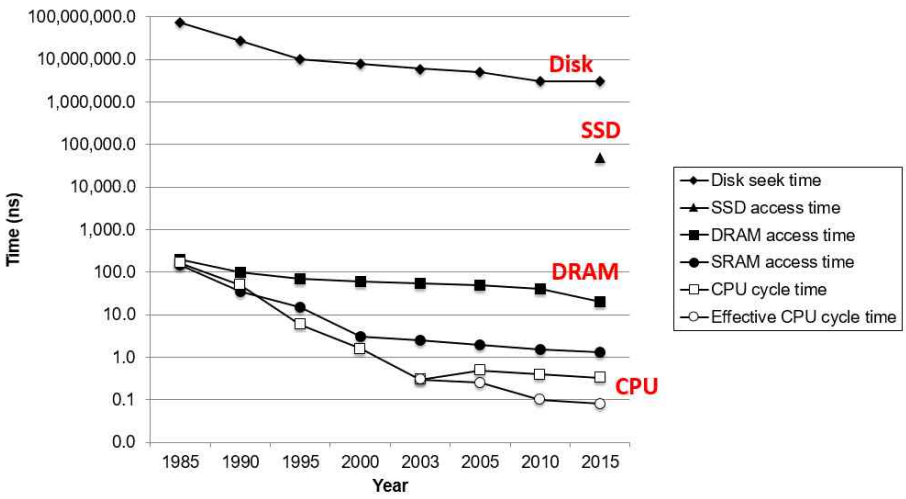
Sequential read tput	**550 MB/s**	**Sequential write tput**	**470 MB/s**
Random read tput	**365 MB/s**	**Random write tput**	**303 MB/s**
Avg seq read time	**50 us**	**Avg seq write time**	**60 us**

위 그림에서 볼 수 있듯이, 일반적인 메모리 소자와 유사하게 읽기/쓰기 작업의 경우 Random Access보다는 Sequential Access가 좋은 성능을 보인다. **특히, 쓰기 작업의 경우 SSD의 특성상 Random Access는 성능이 많이 떨어진다. 한 블록을 지우는 작업이 적지 않은 시간을 필요로 하는데, 한 페이지를 수정하려면 그 페이지가 속한 블록의 다른 페이지들을 전부 삭제해야 하기 때문이다.** 지금은 그래도 괜찮은 편이지만, 초창기 SSD는 이러한 이유 때문에 읽기 작업과 쓰기 작업의 성능 격차가 상당히 컸다.

하드 디스크와는 달리 물리적인 움직임을 필요로 하는 부분이 없기 때문에, 처리 속도가 더 빠르고 전력 소모가 적으며 무엇보다 외부의 충격에 대해 더욱 안전하다. 하지만 앞서 언급했듯, 유한한 횟수의 쓰기 작업을 하고 나면 해당 블록의 수명이 다 하는 문제가 있다. 이러한 문제는 **Flash Translation Layer에 특별한 로직을 추가**함으로써 어느 정도 완화가 가능하다. 실제 사례로, 인텔이 출시한 SSD 730은 그러한 로직을 추가함으로써 닳기 전까지 무려 128PB만큼의 데이터를 쓸 수 있도록 하였다. 최근 SSD는 MP3 플레이어, 스마트폰, 노트북 등에 주기억 장치로서 많이 사용되고 있으며, 데스크탑과 서버에도 사용되는 사례가 계속해서 증가하는 추세이다. 점점 더 많은 부분에서 하드 디스크를 대체하고 있다.

5. 지역성 (Locality)

5-1. CPU와 메모리 기술의 격차



위 그림에서 볼 수 있듯이, **CPU와 DRAM의 속도 차이는 점점 벌어지고 있다.** CPU의 속도는 눈에 띄게 빨라지고 있지만, DRAM의 속도는 상대적으로 천천히 빨라지고 있기 때문이다. 본 글의 서론에서 언급한 **메모리 계층(Memory Hierarchy)**은 이러한 문제를 해결하기 위해 등장했다. 속도가 빠른 CPU와 속도가 느린 DRAM 사이에 SRAM이라는 중간 계층을 두어서 CPU와 DRAM의 속도 격차를 줄여보자는 것이다. 그러나 단순히 중간 계층에 SRAM을 둔다고 해서 그 속도 격차가 줄어드는 건 아니다. 여기에는 특별한 아이디어가 하나 필요하다. 바로 **지역성(Locality)**이라는 것이다.

5-2. 지역성의 개념

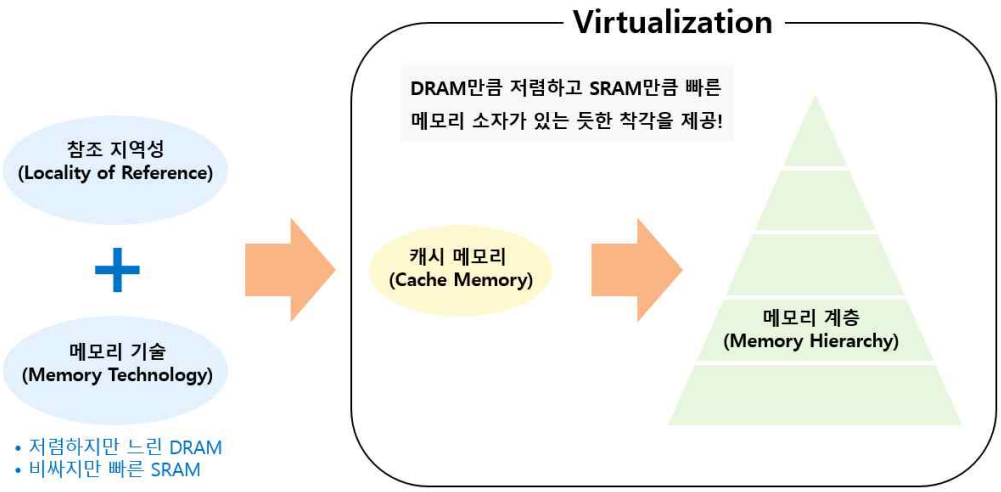
지역성(Locality)이란, 프로그램이 시간적 혹은 공간적으로 가까운 데이터/명령어를 다시 참조할 가능성이 높은 현상을 가리키는 말이다. 지역성은 두 가지로 나뉜다. 첫 번째는 **시간적 지역성(Temporal Locality)으로, 어떤 데이터를 참조하면 가까운 미래에 그 데이터를 다시 참조할 확률이 높다는 것**을 가리킨다. 두 번째는 **공간적 지역성(Spatial Locality)으로, 어떤 데이터를 참조하면 가까운 미래에 그 주변의 데이터를 참조할 확률이 높다는 것**을 가리킨다.

다음과 같은 간단한 C 언어 코드를 예시로 시간적 지역성과 공간적 지역성을 살펴보도록 하자. 먼저, 변수 i와 sum은 매 루프마다 계속해서 참조가 되고 있기 때문에 시간적 지역성을 만족한다. 다음으로, 배열 a의 각 요소는 루프에 따라 순서대로 참조가 이뤄지기 때문에 공간적 지역성을 만족한다. 실제로 이와 같이 지역성을 만족하는 코드는 상당히 많기 때문에, 이러한 지역성을 적절히 잘 활용하면 엄청난 성능의 향상을 기대할 수 있다.

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

5-3. 메모리 계층에서의 활용 (캐시 메모리)

지금까지 이야기한 내용을 종합해 보자. **CPU와 DRAM의 속도 차이는 점점 더 증가**하고 있기 때문에, 이에 대한 해결 방법이 필요하다. **DRAM보다 훨씬 빠른 SRAM**이 존재하긴 하지만, SRAM으로 DRAM을 완전히 대체하기에는 SRAM의 비트 당 가격이 너무 비쌌다. 이때, 많은 프로그램들의 코드들을 살펴본 결과 **지역성(Locality)**이라는 패턴이 존재한다는 것을 알게 되었다. 이러한 아이디어를 바탕으로 새로 고안된 SRAM 기반의 메모리가 바로 **캐시 메모리(Cache Memory)**이다. 캐시 메모리는 CPU와 DRAM 사이에 위치하여 메모리 참조의 성능을 혁신적으로 향상시켰으며, CPU와 DRAM만 존재하는 단순한 메모리 구조를 혁파하고 오늘날과 같은 **메모리 계층(Memory Hierarchy)**을 확립하였다.



그렇다면 캐시 메모리는 지역성을 어떻게 활용하여 메모리 참조의 성능을 향상시켰을까? 크게 시간적 지역성과 공간적 지역성의 관점에서 분석하면 이렇다. 캐시 메모리는 DRAM에서 자주 참조되는 부분을 가져와서 저장하게 되는데, 이때 참조한 데이터만 캐시 메모리로 가져오는 것이 아니라 **블록 단위로 그 주변의 데이터까지 한 번에 캐시 메모리로 가져옴으로써 공간적 지역성을 활용**한다. 또한, 그렇게 가져온 **블록들을 최근에 참조된 순서대로 저장**하고 캐시 메모리가 꽉 차는 경우에는 **가장 과거에 참조된 블록을 빼내는 방식**을 취함으로써 시간적 지역성을 활용한다. 이어지는 설명에서 캐시 메모리의 동작 원리를 간단하게 살펴보도록 하고, 더 자세한 내용은 추후 포스팅을 참고하길 바란다.

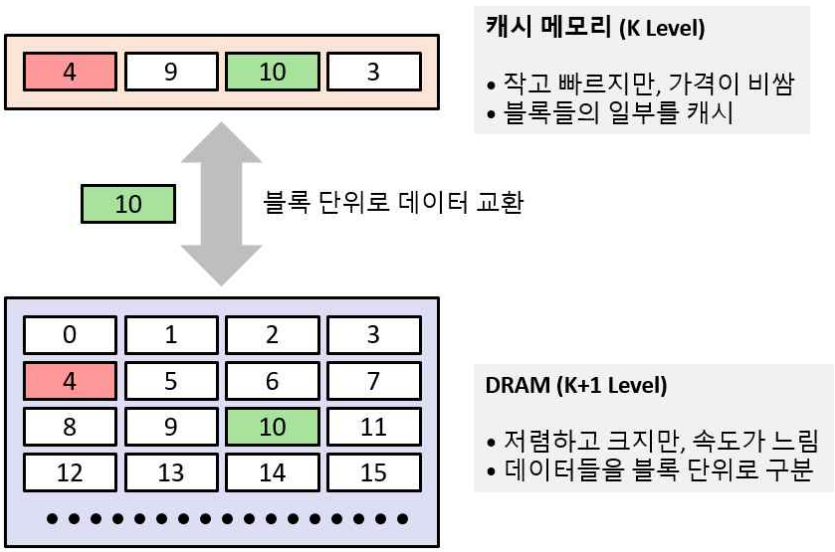
6. 캐시 메모리 (Cache Memory)

6-1. 개념

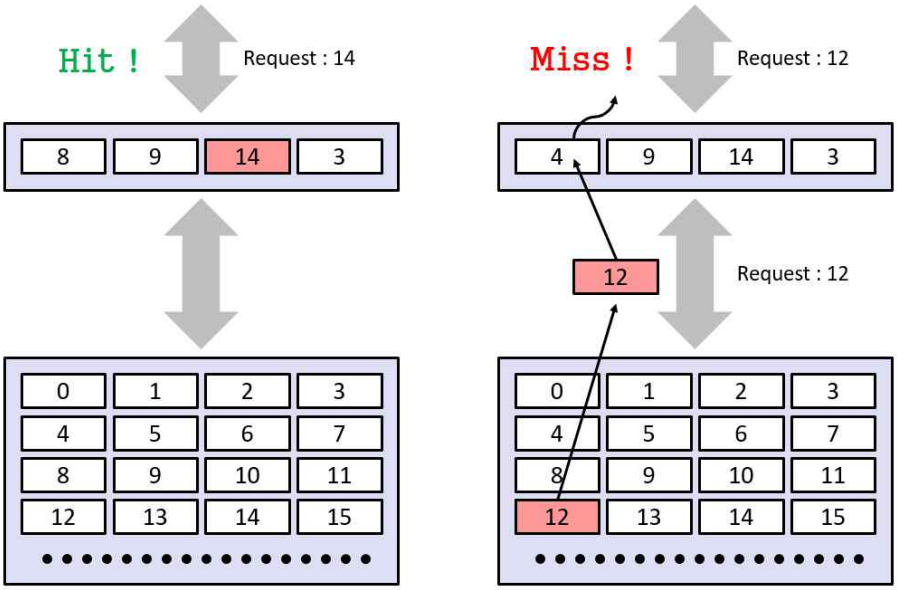
넓은 의미에서의 **캐시(Cache)란, 크고 느린 저장 장치가 가지고 있는 데이터들 중 접근 빈도가 높은 데이터들 일부를 저장하고 있는 작고 빠른 저장 장치**를 의미한다. 가장 대표적인 게 DRAM의 캐시로 사용되는 캐시 메모리(Cache Memory)이다. 참고로, 캐시 메모리가 DRAM의 캐시에 해당한다면, DRAM은 하드 디스크의 캐시에 해당한다. 이는 뒤에서 가상 메모리(Virtual Memory)를 다룰 때 설명하도록 하겠다.

캐시 메모리는 DRAM과 SRAM으로 이뤄지는 메모리 기술에 두 종류의 지역성을 결합시킨 산물이라고 할 수 있다. 캐시 메모리의 핵심 아이디어는 **메모리 기술에 대한 가상화(Virtualization)**이다. 즉, **가격은 DRAM만큼 저렴하지만 접근 속도는 SRAM만큼 빠른 메모리 장치를 사용하는 듯한 착각을 만들어 내는 것**이다. 그러면 캐시 메모리의 동작 원리에 대해 간단히 짚고 넘어가 보자.

6-2. 기본 동작 원리



위 그림은 캐시 메모리의 기본적인 동작 원리를 나타낸다. DRAM의 데이터들을 일정한 크기의 **블록(Blcok)**들로 구분하며, 캐시 메모리는 DRAM에서 자주 참조되는 블록들 일부를 저장한다. 그리고 CPU는 메모리 참조가 필요할 시 캐시 메모리를 먼저 확인한다. 이때, **찾으려는 데이터가 캐시 메모리에 존재하면 캐시 히트(Hit), 존재하지 않으면 캐시 미스(Miss)**라고 한다. 다음 두 개의 그림은 각각 캐시 히트와 캐시 미스가 발생하는 경우를 나타낸다.



캐시 미스가 발생하는 경우에는 고려할 점이 두 가지 있다. 첫 번째는 **Placement Policy**로, DRAM에서 새로 가져온 블록을 어느 위치에 저장해야 하는지 결정하는 것을 말한다. 두 번째는 **Replacement Policy**로, 캐시 메모리가 이미 꽉 차있는 경우 어떤 블록을 내쫓아야 하는지 결정하는 것을 말한다. 둘은 다른 것이니 혼동하지 말자. 전자는 캐시 메모리가 꽉 차있지 않은 경우, 후자는 꽉 차있는 경우를 가정한다. 참고로 Replacement Policy의 대표적인 예시는 가장 과거에 참조되었던 블록을 내쫓는 것이다. 시간적 지역성을 활용하기 위함이다.

6-3. 캐시 미스의 종류

캐시 미스가 발생할 수 있는 경우는 다음과 같이 크게 세 경우이다.

6-3-1. Cold(Compulsory) Miss

이는 해당 블록에 처음 접근할 때 발생하는 캐시 미스이다. 처음 접근하는 경우라면 해당 블록이 캐시 메모리에 없을 수밖에 없기 때문에, 이는 불가피하게 최소 한 번은 발생할 수밖에 없다.

6-3-2. Conflict Miss

이를 이해하기 위해서는 먼저 충돌이라는 것이 무슨 현상인지 알아야 한다. **일반적으로 (K+1)단계에 위치하는 각 블록들은 K단계의 특정 자리에만 들어갈 수 있도록 되어 있다.** 예를 들어 위 예시에서는 DRAM의 i번째 블록이 캐시 메모리의(i mod 4)번째 블록 자리에만 들어갈 수 있다. 이는 캐시 메모리가 꽉 차있지 않은 상황에도 똑같이 적용된다. 즉, 공간이 남아도 정해진 자신의 자리로 들어간다는 것이다. 참고로, **캐시 메모리에서는 각각의 자리를 Set이라고 한다.** 따라서 위 예시는 4개의 Set으로 이루어진 캐시 메모리를 나타낸다. Set은 위 예시와 같이 한 블록만 들어갈 수 있는 크기일 수도 있고, 여러 개의 블록이 들어갈 수 있는 크기일 수도 있다. **중요한 것은 Set의 크기가 유한하다는 것**이다. 따라서 동일한 Set에 맵핑되는 블록들이 반복적으로 참조되면 어느 순간 Set이 꽉 차서 해당 Set에 들어가 있는 특정 블록을 내쫓아야 하는 상황이 발생한다. 이 현상을 **충돌(Conflict)**이라고 한

다. 그리고 충돌로 인해 발생하는 캐시 미스가 바로 Conflict Miss이다. 예를 들어, 위 예시에서 0번째 블록과 8번째 블록이 반복적으로 참조되면 계속해서 캐시 미스만 발생할 것이다.

※ 각 블록이 들어갈 수 있는 자리가 정해져 있는 이유

메모리 참조가 필요해지면 CPU는 캐시 메모리부터 확인한다. 이때 CPU는 찾고자 하는 데이터가 캐시 메모리에 있는지 어떻게 확인할까? 만약 DRAM의 각 블록이 캐시 메모리의 어디든 들어갈 수 있도록 되어 있다면, CPU는 캐시 메모리의 데이터 전부를 일일이 뒤지는 수밖에 없을 것이다. 그러나 DRAM의 각 블록은 자신이 캐시 될 수 있는 위치가 정해져 있기 때문에, CPU는 그 위치만 확인해서 해당 데이터의 캐시 유무를 판단할 수 있다. 조금 더 구체적으로 설명하자면, DRAM의 각 블록들은 자신의 메모리 주소를 기준으로 자신이 캐시 될 수 있는 Set이 정해진다. 따라서 CPU는 특정 데이터의 캐시 유무를 판단해야 할 때 메모리 주소를 기준으로 해당 데이터가 캐시 될 수 있는 Set을 찾아가서 탐색을 시도하면 된다. 이는 빠른 시간 내에 탐색할 수 있는 해싱(Hashing) 알고리즘을 적용한 것으로, 각 Set의 크기는 유한하기 때문에 탐색의 시간 복잡도는 O(1)이 된다.

6-3-3. Capacity Miss

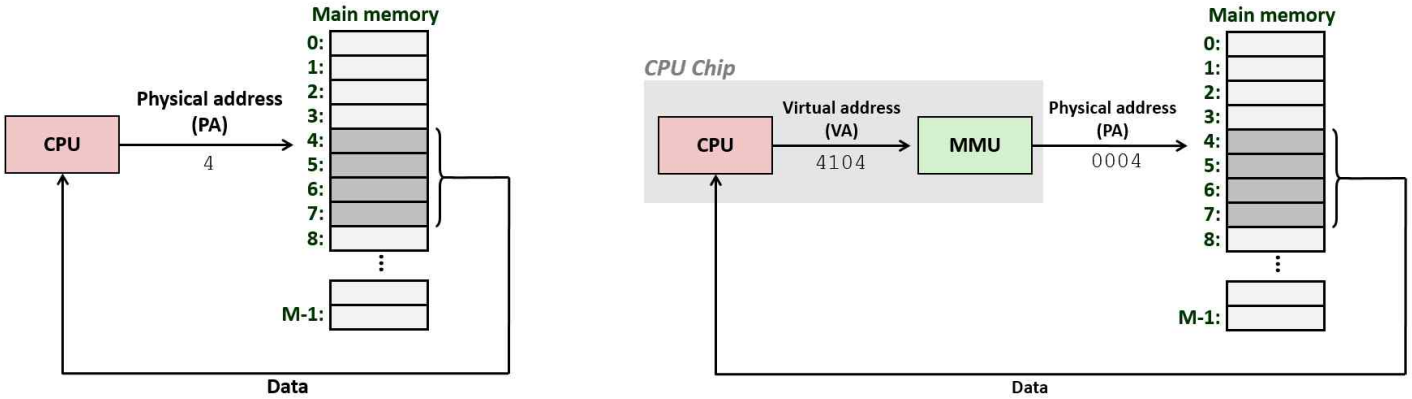
단순히 캐시 메모리의 용량이 적기 때문에 발생하는 미스를 의미한다. 뒤에서 알아볼 용어를 사용한다면, Working Set(Set of active cache blocks)의 크기가 캐시 메모리의 용량보다 큰 경우에 발생하는 미스이다. 예를 들어, 순차적으로 접근하는 배열의 전체 크기가 캐시 메모리의 크기보다 크다면 필연적으로 캐시 미스가 발생할 것이다.

Virtual Memory: Concepts

1. Introduction

1-1. 가상 메모리 (Virtual Memory)

몇몇 단순한 임베디드 마이크로컨트롤러와 같이 가상 메모리 기술을 사용하지 않는 시스템에서는 메모리 참조 방식이 아래 왼쪽 그림과 같다. 즉, CPU가 물리 주소를 메인 메모리에 바로 입력하여 메모리 참조를 진행하는 것이다. 반면 대부분의 현대 데스크탑, 서버, 노트북과 같이 가상 메모리 기술을 사용하는 시스템에서는 메모리 참조 방식이 아래 오른쪽 그림과 같다. 가상 메모리(Virtual Memory) 시스템에서는 **각 프로그램이 가상의 주소를 사용하도록 하며, CPU가 메모리 참조를 시도할 때는 MMU(Memory Management Unit)라는 하드웨어 장치를 이용하여 해당 가상 주소(Virtual Address)를 실제 메인 메모리의 물리 주소(Physical Address)로 변환**하여 메모리 참조를 진행한다.



[Figure] 가상 메모리 기술을 사용하지 않는 시스템(왼쪽), 가상 메모리 시스템(오른쪽)

1-2. 가상 메모리의 필요성

그렇다면 이와 같은 가상 메모리 기술은 왜 사용하는 것일까? 크게 세 가지 이유로 나눠서 생각해볼 수 있다. 각각에 대한 자세한 내용은 바로 이어지는 세 개의 큰 주제(Caching, Memory Management, Memory Protection)에서 설명하는 부분을 읽어보도록 하자.

먼저, **메인 메모리를 효율적으로 사용하기 위해서**이다. 가상 메모리 시스템에서는 각 프로그램이 사용하는 가상 주소 공간(Virtual Address Space)을 우선 디스크에 저장해 두고, 그중에서 자주 사용되는 부분만 메인 메모리로 가져와서 사용한다. 즉 **메인 메모리를 디스크의 캐시로 사용**하는 것이다. 이렇게 하면 하나의 메인 메모리에 여러 프로그램의 데이터와 코드를 로드하는 것이 가능해진다.

다음으로, **메모리 관리를 단순화**한다. 가상 메모리 시스템에서는 **각 프로세스가 완전히 동일한 포맷의 가상 주소 공간**을 가진다. 이것이 가능한 이유는 실제로 메모리 참조를 수행할 때는 가상 주소를 물리 주소로 변환하는 작업이 진행되기 때문이다. 따라서 각각의 프로세스에게 혼자서 메인 메모리를 사용한다는 듯한 착각을 제공한다. 이는 링커 및 로더의 구현과 메모리 할당 방식 등을 단순화한다.

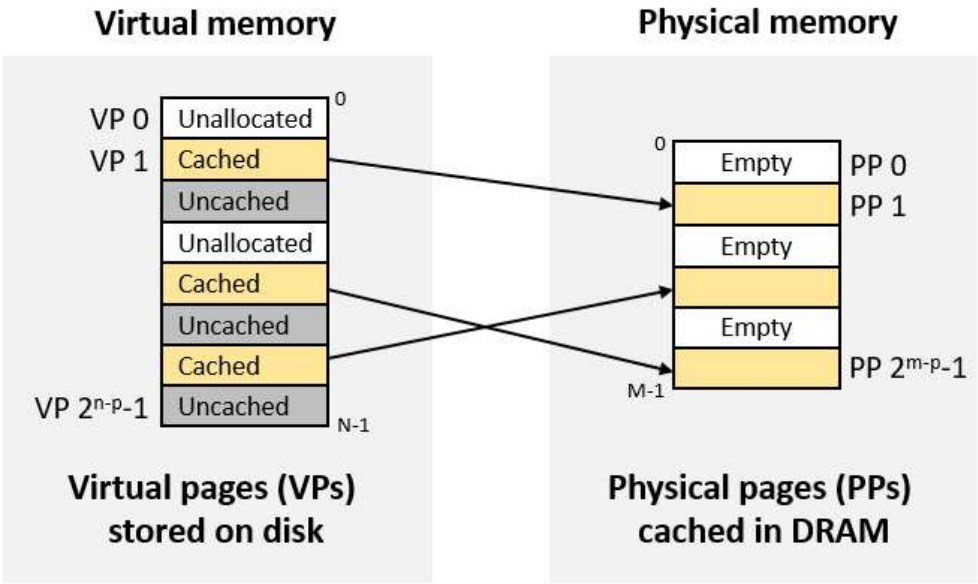
마지막으로, **메모리 보호 메커니즘을 단순화**한다. 가상 메모리 시스템에서는 한 프로세스가 다른 프로세스의 주소 공간에 접근하는 것을 쉽게 막을 수 있다. 또한, 가상 주소를 물리 주소로 변환할 때 참조하는 **맵핑 테이블의 각 엔트리에는 해당 가상 주소에 대한 접근 권한이 명시**된다. 따라서 유저 프로세스가 커널 영역에 접근하는 것처럼 허용되지 않은 주소 공간에 접근하는 것도 쉽게 막을 수 있다.

2. VM for Caching

2-1. 가상 메모리와 DRAM 캐시

가상 메모리는 디스크에 저장되는 N개의 연속적인 바이트들로 이뤄진 배열을 의미하며, 해당 바이트 배열의 일부는 메인 메모리(DRAM 캐시)에 캐시 된다. 캐시 메모리가 메인 메모리의 캐시로 사용된다면, 메인 메모리는 디스크의 캐시로 사용되는 셈이다. 그리고 캐시 메모리와 DRAM의 관계에서 블록(Block)이라는 단위를 사용하듯이, 메인 메모리와 디스크의 관계에서는 **페이지(Page)**라는 단위를 사용한다. 디스크에 위치하는 **가상 주소 공간(Virtual Address Space)의 각 페이지는 가상 페이지(Virtual Address)**, **메인 메모리에 위치하는 물리 주소 공간(PhysicalAddress Space)의 각 페이지는 물리 페이지(Physical Page)**라고 부른다.

다음은 디스크의 가상 페이지들이 물리 페이지들에 맵핑된 예시를 보여준다. Unallocated는 프로그램에 의해 사용되지 않는 가상 주소 공간에 해당하는 가상 페이지이고, Uncached는 프로그램에 의해 사용되지만 아직 물리 페이지에 맵핑되지 않은 가상 페이지이며, Cached는 프로그램에 의해 사용되고 물리 페이지에 맵핑까지 된 가상 페이지이다.



한편, **DRAM 캐시(= 디스크의 캐시)는 캐시 메모리(= 메인 메모리의 캐시)에 비해 Miss Penalty가 매우 크다**는 특징이 있다. DRAM과 SRAM의 속도 차이는 10배 정도인 반면, 디스크와 DRAM의 속도 차이는 거의 10,000배에 이르기 때문이다. 이로 인해 캐시 메모리와 DRAM 캐시는 다음과 같이 크게 두 가지 측면에서 차이점을 가진다.

2-1-1. Miss Penalty 처리 방식

캐시 메모리의 경우 DRAM과 SRAM의 속도 차이가 그렇게 크지는 않기 때문에 하드웨어 수준에서만 처리해도 문제가 되지 않는다. 그러나 **DRAM 캐시의 경우 디스크와 DRAM의 속도 차이가 상당히 크기 때문에 하드웨어 수준에서만 처리하기에는 낭비되는 시간이 너무 많다. 따라서 소프트웨어적인 처리를 가미한다.** 대표적인 사례는 하드웨어가 DRAM 캐시의 Miss Penalty를 처리하는 동안 문맥 전환을 통해 잠시 다른 프로세스에게 제어를 넘겨주는 것이다. 이렇게 하면 DRAM 캐시의 Miss Penalty를 처리하는 긴 시간을 기다리기만 하면서 낭비하지 않아도 된다.

2-1-2. 캐시 구조 (Cache Organization)

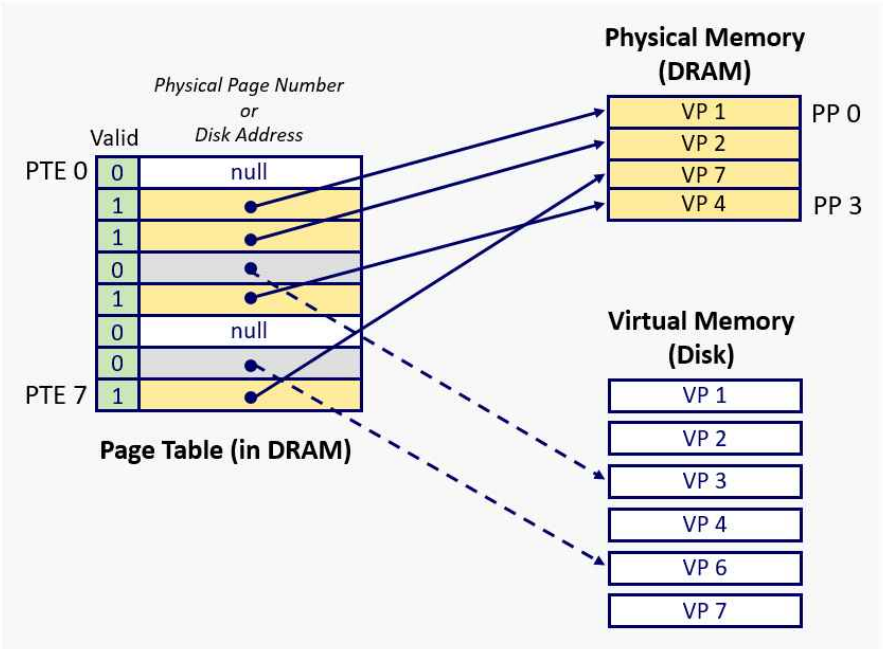
캐시 메모리에 비해 DRAM 캐시는 Miss Penalty가 상당히 크기 때문에 Miss Rate를 최소화하기 위한 구조가 필요하다. 먼저, **페이지 사이즈가 블록 사이즈에 비해 상당히 크다.** 페이지 사이즈는 일반적으로 4-8KB 정도이며, 때로는 4MB 정도까지 이르는 경우도 있다. 다음으로, **Fully Associative** 구조를 갖는다. 즉 각각의 가상 페이지는 어떠한 물리 페이지에도 맵핑될 수 있다. 단 이렇게 하면 DRAM 캐시에서 특정 가상 페이지의 캐싱 여부를 확인할 때 모든 물리 페이지를 확인해야 할 것이다. 이를 해결하기 위한 방법이 바로 맵핑 테이블이다. 맵핑 테이블은 입력된 가상 주소를 바탕으로 해당 가상 페이지가 DRAM 캐시에 존재하는지를 단번에 알아낼 수 있도록 한다. 다만 맵핑 테이블의 크기는 가상 페이지의 개수에 비례하기 때문에 맵핑 테이블의 크기를 최소화하는 전략도 필요하다. 이에 대해선 뒤에서 알아보도록 하자. 또한, **매우 정교한 Replacement 알고리즘을 사용한다.** Miss Rate를 최소화하기 위해 지금까지도 훌륭한 알고리즘이 많이 개발되어 왔다. 대표적인 알고리즘이 Second Chance 알고리즘이다. 마지막으로, **Write-back 방식을 채택한다.** Write-through 방식은 쓰기 작업을 수행할 때마다 디스크에 접근을 시도해야 해서 시간적으로 엄청난 낭비가 발생한다. 따라서 디스크의 접근을 최소화하기 위해 Write-back 방식을 채택한다.

※ Second Chance 알고리즘

LRU(Least Recently Used) 알고리즘을 근사적으로 구현한 알고리즘이다. FIFO 또는 순환 큐에 참조된 순서대로 PTE 정보가 저장되어 있고, 메모리 참조 시마다 해당 PTE의 참조 비트를 1로 설정한다. 그러다가 페이지 폴트가 발생하여 특정 페이지를 추방해야 하는 상황이 되면, 가장 옛날에 참조된 PTE부터 시작하여 다음과 같은 방법으로 추방할 페이지를 찾는다. 우선, 현재 보고 있는 PTE의 참조 비트를 확인한다. 만약 0이라면 그 페이지를 추방하면 된다. 반면 1이라면 그 값을 0으로 바꾸고(= 한 번 살아남을 기회를 주고) 다음 PTE를 확인한다. 그리고 추방할 페이지를 찾을 때까지 이 과정을 반복한다. 만약 모든 PTE의 참조 비트가 1이라면 한 바퀴를 돌아서 참조 비트를 0으로 바꿨던 최초의 PTE에 다시 도달하여 그 페이지를 추방하게 될 것이다. 이 알고리즘의 기본 아이디어는 이렇다. 참조 비트가 0으로 바뀐 뒤 한 번이라도 재 참조되면 참조 비트가 다시 1로 설정되므로 이후 추방할 페이지를 찾을 때 살아남게 된다는 것이다. 따라서 자주 참조되지 않는 페이지는 이후 추방할 페이지를 찾을 때에도 여전히 참조 비트가 0일 수 있고, 이러한 경우 살아남지 못하고 추방당하는 것이다.

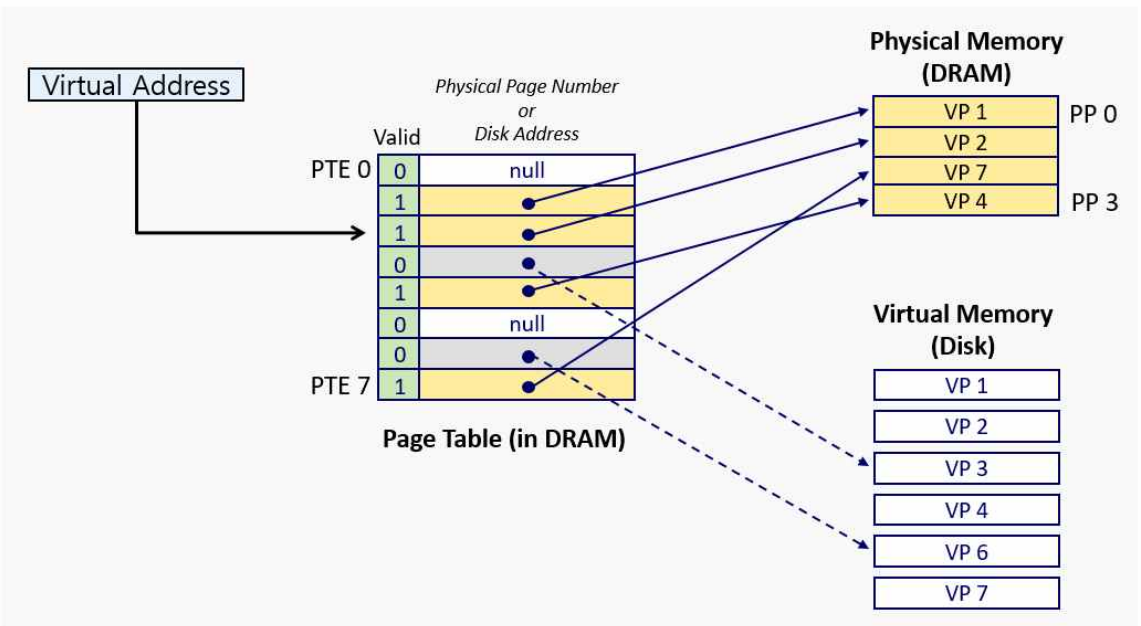
2-2. 페이지 테이블 (Page Table)

페이지 테이블(Page Table)은 가상 페이지와 물리 페이지 사이의 맵핑 정보를 담은 테이블로, 메인 메모리의 커널 영역에 저장되는 자료 구조 중 하나이다. 각 프로세스는 자신만의 페이지 테이블을 가지기 때문에, 문맥 전환을 수행할 때 저장 및 복원하는 문맥 정보에도 해당 프로세스의 페이지 테이블 정보가 포함된다. **페이지 테이블의 각 엔트리는 페이지 테이블 엔트리(Page Table Entry, PTE)**라고 부른다. **각 PTE는 해당 가상 페이지가 어떤 물리 페이지에 맵핑이 되어 있는지, 또는 맵핑이 되어 있지 않다면 디스크의 어느 위치에 존재하는지에 대한 정보를 담고 있다. 만약 프로그램이 사용하지 않아서 할당되지 않은 가상 페이지라면 null 값을 저장한다.** 이를 그림으로 나타내면 다음과 같다.



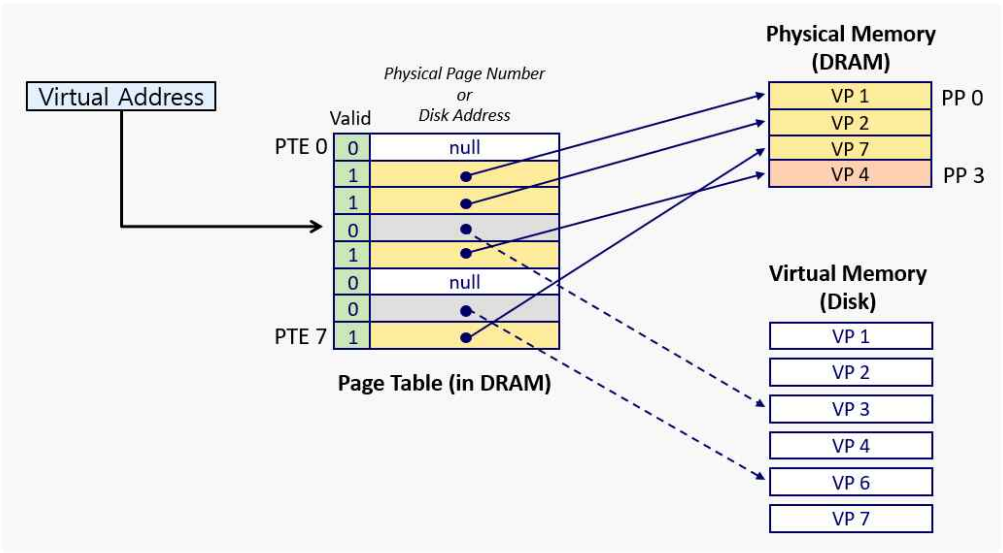
2-3. 페이지 히트 (Page Hit)

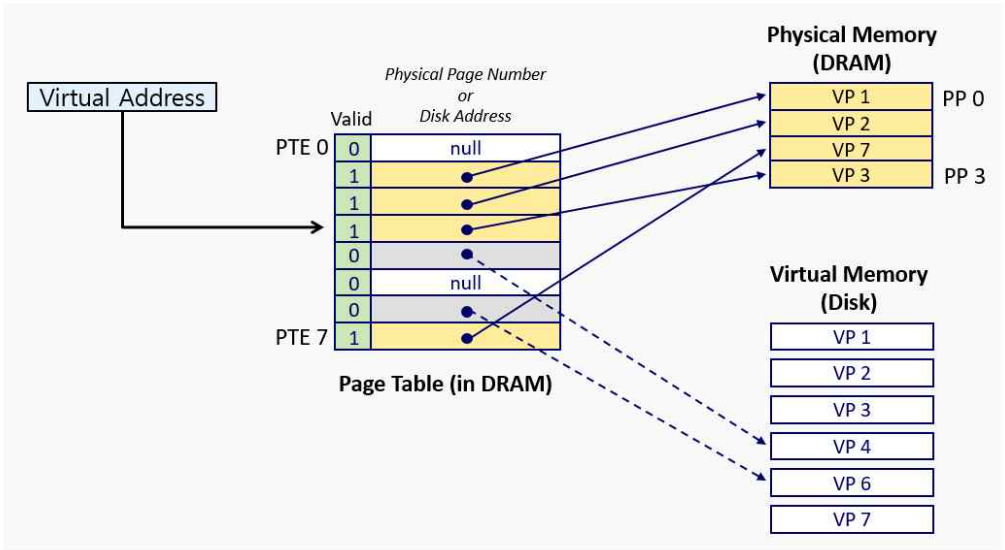
접근하고자 하는 가상 주소에 해당하는 PTE의 Valid 비트가 1이면, 해당 가상 페이지가 물리 페이지에 맵핑되어 있음을 의미한다. 이를 **페이지 히트(Page Hit) 또는 DRAM 캐시 히트**라고 부른다. 이제 해당 PTE의 정보를 바탕으로 가상 주소를 물리 주소로 변환하여 메인 메모리에 접근하기만 하면 된다. 이를 그림으로 나타내면 다음과 같다.



2-4. 페이지 미스 (Page Miss) → 페이지 폴트 (Page Fault)

접근하려는 가상 주소에 해당하는 PTE의 Valid 비트가 0이고 그곳에 디스크의 특정 위치 정보가 저장되어 있다면, 해당 가상 페이지가 물리 페이지에 맵핑되어 있지 않음을 의미한다. 이를 **페이지 미스(Page Miss) 또는 DRAM 캐시 미스**라고 부른다. 이러한 경우 **페이지 폴트(Page Fault) 예외가 발생하여 페이지 폴트 핸들러(Page Fault Handler)가 호출**된다. 그러면 **페이지 폴트 핸들러는 현재 메인 메모리에서 특정 물리 페이지를 선택하여 추방하고, 디스크에게 요청된 가상 페이지를 가져오도록 명령**한 뒤 문맥 전환을 통해 잠시 다른 프로세스에게 제어를 넘겨준다. 이후 디스크가 가상 페이지를 메인 메모리에 로드하는 작업을 완료하면, 인터럽트를 발생시켜서 문맥 전환을 통해 다시 페이지 폴트 핸들러의 프로세스로 제어를 옮긴다. 그러면 **페이지 폴트 핸들러는 추방된 물리 페이지와 새로 들어온 물리 페이지의 정보를 바탕으로 PTE를 갱신하고, 페이지 폴트를 일으켰던 명령어의 위치로 다시 리턴하여 해당 명령어를 재실행**한다. 그러면 이번에는 페이지 히트가 발생하므로 위에서 설명한 방식대로 메모리 참조를 진행하면 된다. 이를 그림으로 나타내면 다음과 같다.



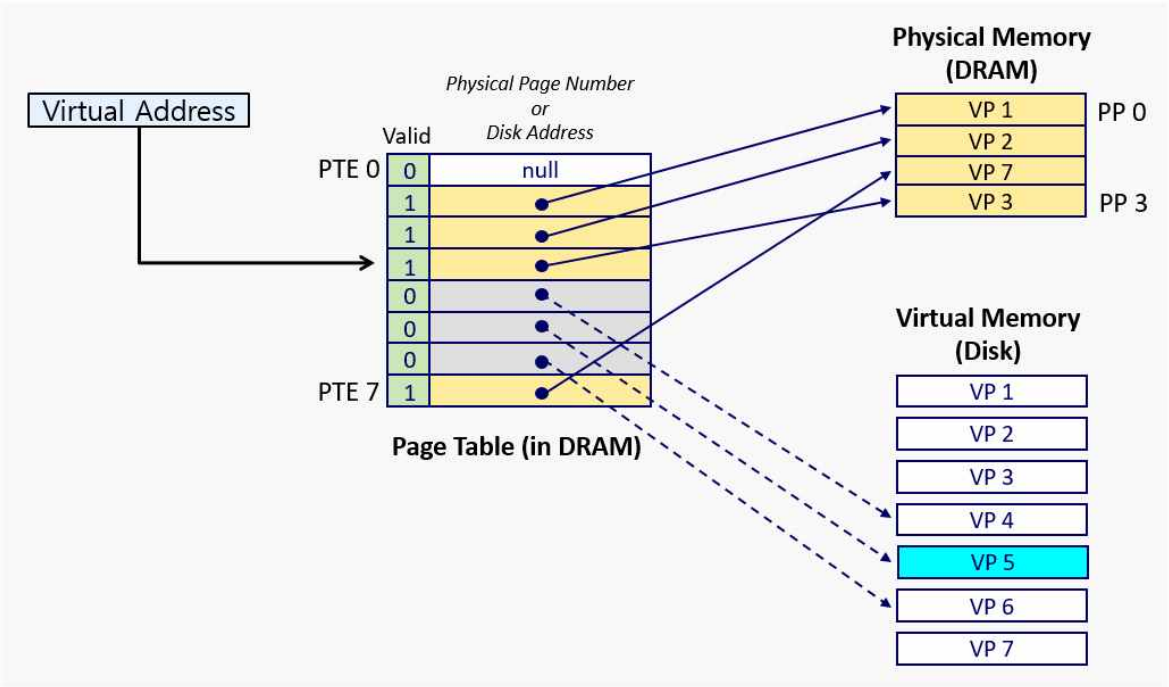


※ 추방하려는 물리 페이지의 Dirty 비트가 1이라면 해당 물리 페이지 내용을 먼저 디스크에 반영해줘야 한다. (Write-back)

※ 페이지 폴트 핸들러는 커널 영역에 항상 상주하고 있다. 이곳에서 페이지 폴트가 발생하면 곤란할 것이다.

2-5. 페이지 할당 (Allocating Pages)

유저 프로세스로 실행되는 프로그램이 **추가적으로 힙 영역을 할당하려고 시도하면(EX. malloc함수), 커널은 적절한 크기의 연속적인 가상 페이지들을 디스크에 할당**하고 그것들에 해당하는 PTE들을 페이지 테이블에 새로 만들어 준다. 그렇게 생성된 각각의 PTE는 Valid 비트가 0이며, 해당 가상 페이지가 디스크의 어느 위치에 할당되어 있는지에 대한 정보를 담게 된다. 이를 그림으로 나타내면 다음과 같다.



2-6. 작업 집합 (Working Set)

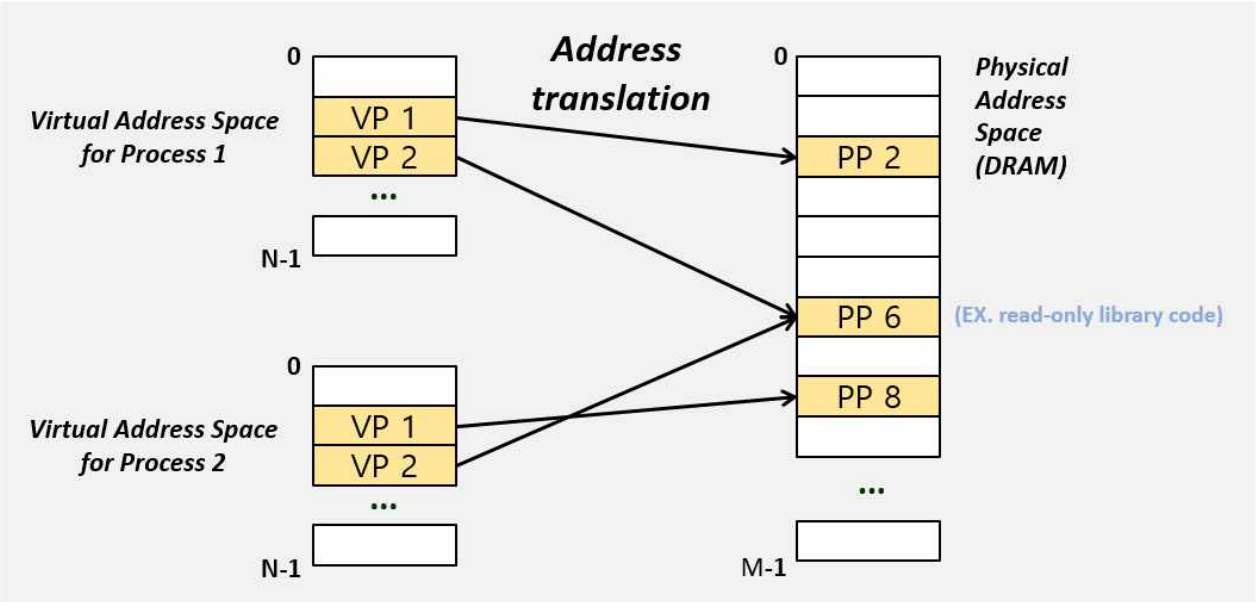
이와 같은 가상 메모리 기술이 훌륭한 성능을 보이는 것은 프로그램의 **지역성(Locality)**을 잘 활용했기 때문이라고 할 수 있다. 실제로, **각 프로그램은 특정 한 시점에 몇 개의 활성화된 가상 페이지들에 접근하는 경향**이 있다. 그러한 가상 페이지들의 집합을 **작업 집합(Working Set)**이라고 한다. 쉽게 말해서 작업 집합은 프로그램이 자주 사용하는 가상 주소 공간을 의미한다. 만약 프로그램이 시간적 지역성(Temporal Locality)을 잘 활용할 수 있도록 짜여 있다면, 작업 집합의 크기는 작을 것이다. 만약 **작업 집합의 크기가 메인 메모리의 크기보다 작다면, 최초의 미스(Compulsory Miss) 이후에는 메모리 참조에 있어서 굉장히 좋은 성능**을 보일 것이다. 반면, **작업 집합의 총 크기가 메인 메모리의 크기보다 크다면,** 특정 가상 페이지가 물리 페이지에 맵핑이 되었다가 해당 물리 페이지가 다시 추방당하는 과정이 계속 반복될 것이다. 이러한 현상을 **트래싱(Thrashing)**이라고 하며, 트래싱이 발생하면 대부분의 시간이 페이지 폴트를 처리하는 데 사용되므로 메모리 참조 성능이 심각하게 저하된다.

3. VM for Memory Management

3-1. 메모리 관리 (Memory Management)

가상 메모리 기술은 메모리 관리(Memory Management) 측면에서도 중요한 역할을 수행한다. 핵심적인 아이디어는 이렇다. **각 프로세스는 자신만의 가상 주소 공간을 가지기 때문에, 메인 메모리를 혼자서 사용하는 것처럼 생각해도 전혀 문제가 없다**는 것이다. 다만 실제로는 그것이 불가능하기 때문에 맵핑 함수가 각 프로세스의 몇몇 가상 페이지들만 물리 페이지에 적절히 맵핑하고, 그 정보를 페이지 테이블에 기록하는 것이다. 그러면 각 프로세스는 자신의 페이지 테이블 정보를 바탕으로 특정 가상 페이지에 해당하는 물리 페이지를 요청하면서 메모리 참조를 진행할 수 있다.

이때, **각각의 가상 페이지는 어떠한 물리 페이지에도 맵핑될 수 있으며, 동일한 가상 페이지라 하더라도 맵핑 시점에 따라 다른 물리 페이지에 맵핑될 수 있다.** 이는 DRAM 캐시의 Fully Associativity 특성에 기인한다. 따라서 맵핑 함수만 적절히 잘 선택된다면 메모리를 더 단순하게 할당하거나 관리하는 것도 가능해진다. 심지어는 서로 다른 가상 페이지들을 하나의 물리 페이지에 맵핑함으로써 프로세스들 간의 코드 및 데이터의 공유를 가능하게 할 수도 있다. 이를 그림으로 나타내면 다음과 같다.



3-2. 링킹 및 로딩의 단순화 (Simplifying Linking and Loading)

3-2-1. 링킹의 단순화

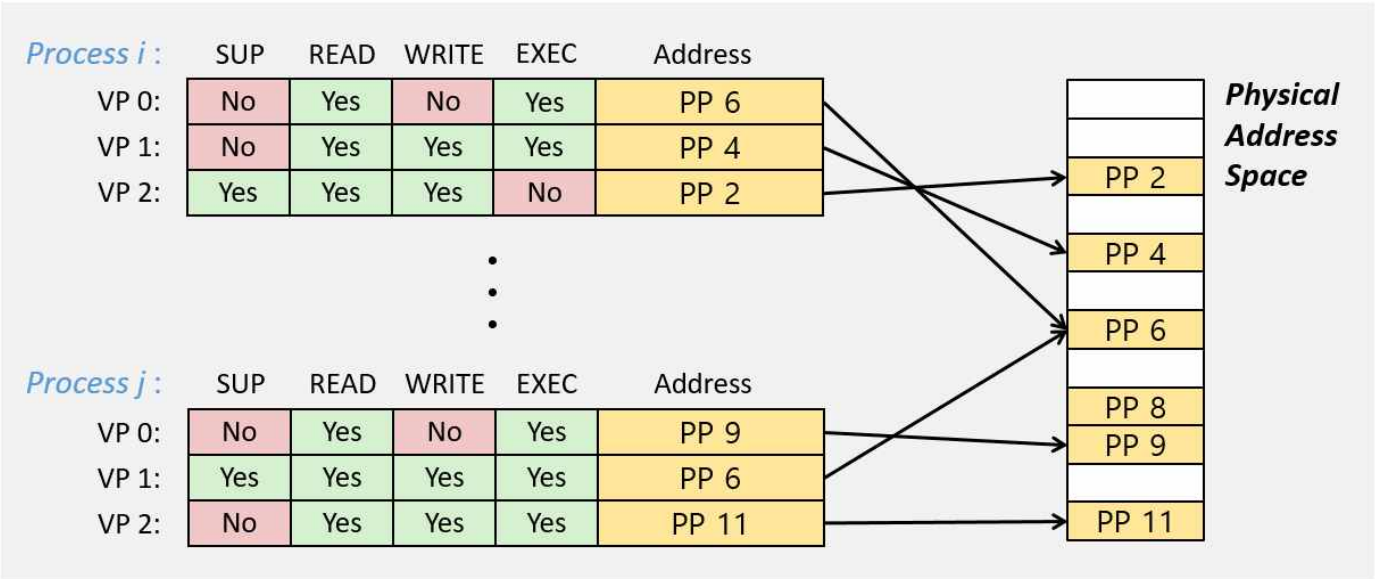
가상 메모리 시스템에서는 각 프로그램이 똑같은 포맷의 가상 주소 공간을 가진다. 예를 들어, 모든 프로세스의 가상 주소 공간은 코드 세그먼트, 유저 스택, 공유 라이브러리 영역이 동일한 주소에서 시작하게 되어 있다. 이러한 통일성으로 인해, **링커를 디자인하고 구현하는 것이 매우 단순화된다.** 가령 링커가 완전한 하나의 실행 가능한 파일을 만들어내는 과정에서, 해당 프로그램의 코드와 데이터가 실제로 메인 메모리의 어느 위치에 맵핑이 될지 전혀 고려할 필요가 없어진다.

3-2-2. 로딩의 단순화

로더에 해당하는 execve 함수는 .text 섹션과 .data 섹션을 위한 가상 페이지들을 디스크에 할당하고, 그것들에 대한 PTE들을 새로 만들어서 Valid 비트를 0으로 설정하고 디스크 내 적절한 위치를 가리키도록 한다. 여기서 주의할 점은 로더는 가상 페이지들을 메인 메모리에 직접 올리지 않는다는 것이다. 대신, 나중에 CPU가 특정 명령어를 명령어 메모리에서 Fetch 하거나 데이터 메모리에 접근해야 하는 명령어를 수행할 때, 가상 메모리 시스템에 의해 요청된 가상 페이지들이 메인 메모리에 올라가고 해당 PTE가 적절히 수정된다.

4. VM for Memory Protection

가상 메모리 기술은 메모리 보호(Memory Protection) 메커니즘의 구현을 용이하게 해준다. 가상 메모리 시스템에서는 CPU가 특정 가상 주소를 발생시킬 때마다 그것에 해당하는 PTE를 반드시 읽을 수밖에 없다. 따라서 **각 PTE에 해당 가상 페이지에 대한 접근 권한 정보를 추가해주면 프로세스별로 각 가상 페이지에 대한 접근을 통제하는 것이 매우 쉬워진다.** 이를 그림으로 나타내면 다음과 같다.



만약 PC에 저장된 값이 실행 권한이 없는 가상 페이지의 가상 주소이거나, 특정 명령어가 읽기/쓰기 권한이 없는 가상 페이지에 대하여 읽기/쓰기 작업을 수행하려 하면 하드웨어 수준에서 예외가 발생하여 예외 핸들러로 제어가 넘어간다. 예를 들어, x86-64 리눅스 시스템에서 읽기 권한이 없는 가상 페이지를 읽으려고 시도하면 General protection fault 예외가 발생하여 "Segmentation faults" 메시지가 출력된다. 또한 할당되지 않은 가상 페이지(Valid 비트 = 0 & 디스크 위치 정보 없음)에 접근할 때도 마찬가지 결과가 나타난다.

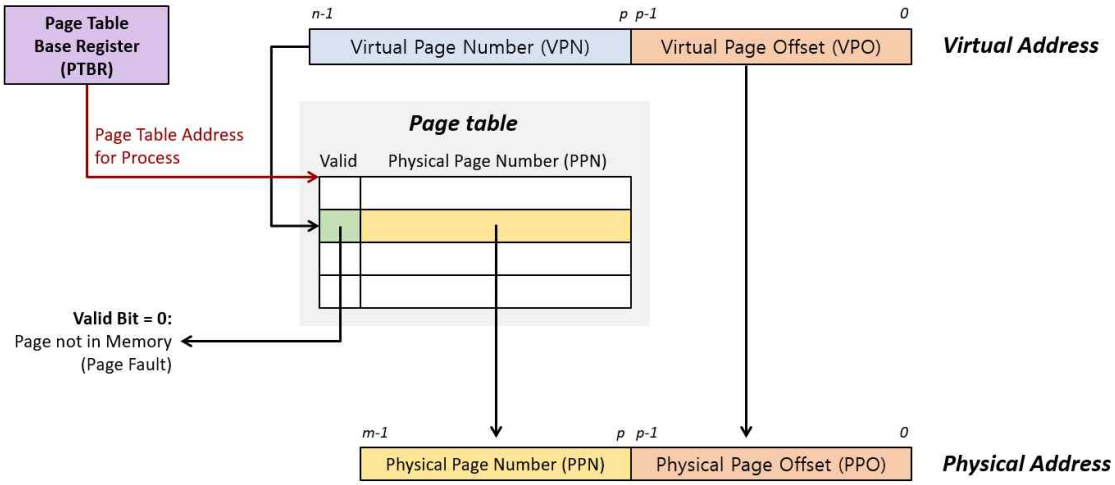
5. Address Translation

5-1. 용어 정리

용어/기호		의미
N = 2^n		가상 주소 공간 내 주소의 개수
M = 2^m		물리 주소 공간 내 주소의 개수
P = 2^p		페이지 사이즈 (바이트 단위)
V = {0, 1, 2, . . . , N-1}		가상 주소 공간
P = {0, 1, 2, . . . , M-1}		물리 주소 공간
MAP : V →P∪{∅}		가상 주소 VA에 대하여, 1) VA가 PA에 맵핑되어 있는 경우 : MAP(VA) = PA 2) VA가 물리 주소 공간에 맵핑되지 않은 경우 : MAP(VA) = ∅
가상 주소 (Virtual Address)	VPO (Virtual Page Offset)	가상 페이지 내 오프셋
	VPN (Virtual Page Number)	가상 페이지 번호
	TLBI (TLB Index)	TLB 집합 번호
	TLBT (TLB Tag)	동일한 TLB 집합에 맵핑되는 PTE들을 구별해주는 태그
물리 주소 (Physical Address)	PPO (Physical Page Offset) = VPO	물리 페이지 내 오프셋
	PPN (Physical Page Number)	물리 페이지 번호
	CO (Byte Offset within Cache Line)	캐시 블록 내 오프셋
	CI (Cache Index)	캐시 집합 번호
	CT (Cache Tag)	동일한 캐시 집합에 맵핑되는 캐시 블록들을 구별해주는 태그

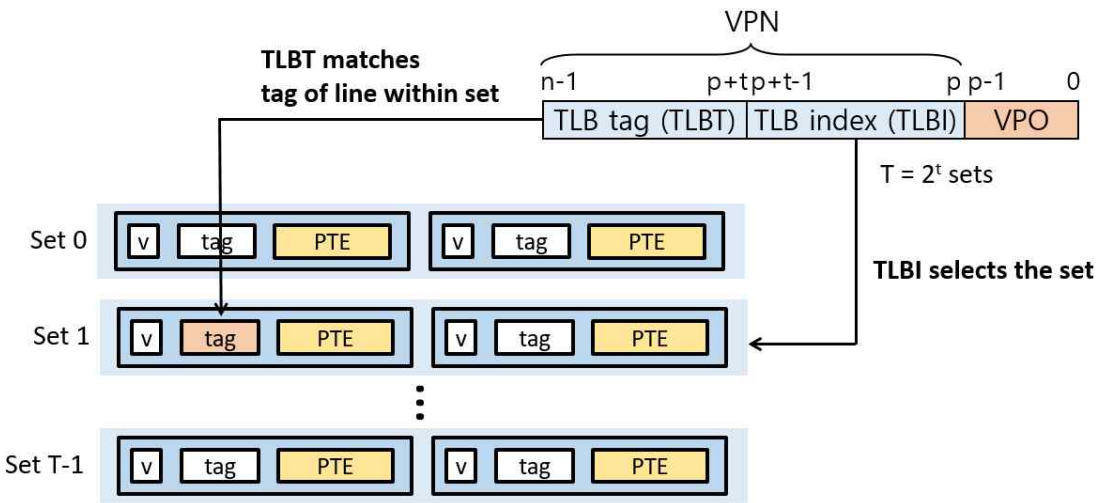
5-2. 주소 변환 (Address Translation)

페이지 테이블 엔트리(PTE) 정보를 바탕으로 가상 주소(Virtual Address)를 물리 주소(Physical Address)로 변환하는 과정은 다음과 같다. 먼저 변환하고자 하는 가상 주소에 해당하는 PTE를 찾아간다. 만약 PTE의 Valid 비트가 0인데 그곳에 디스크의 특정 위치 정보가 저장되어 있다면, 해당 가상 페이지가 물리 페이지에 맵핑되어 있지 않음을 의미하므로 페이지 폴트 예외가 발생한다. 반면 PTE의 Valid 비트가 1이면 해당 가상 페이지가 물리 페이지에 맵핑되어 있음을 의미하므로, 해당 PTE에서 맵핑된 물리 페이지의 번호(PPN)를 알아낸다. 그리고 여기에 가상 주소의 페이지 오프셋 정보(VPO)를 덧붙여 주면 물리 주소가 완성된다. 참고로 페이지 테이블 자체의 시작 주소는 CPU 내 페이지 테이블 베이스 레지스터(Page Table Base Register, PTBR)라는 곳에 저장이 되어 있다.



5-3. TLB (Translation Lookaside Buffer)

TLB(Translation Lookaside Buffer)는 메인 메모리에 존재하는 페이지 테이블의 캐시로서, MMU 내부에 존재하는 작은 하드웨어 버퍼 장치를 의미한다. 즉 **TLB에는 참조되는 빈도가 높은 페이지 테이블 엔트리(PTE)들이 저장된다.** 캐시 메모리가 메인 메모리의 캐시로 사용되는 것과 마찬가지 관계이다. 따라서 **TLB 히트가 발생하면 페이지 테이블 참조를 위해 메인 메모리에 찾아갈 필요가 없어진다.** 만약 TLB가 없다면 페이지 테이블을 참조하는 과정에만 최소한 1번의 메모리 참조를 진행해야 하며, 또 다른 메모리 참조로 인해 캐시 메모리 내에서 페이지 테이블 일부가 추방이라도 당하면 추후 캐시 메모리의 Miss Penalty로 인해 주소 변환 과정은 더욱 느려질 것이다.



TLB에 접근하여 PTE를 가져오는 과정은 위 그림과 같다. 앞에서 알아보았듯이 각 PTE의 주소는 가상 주소의 VPN을 통해 알아낸다. 따라서 TLB에 접근할 때도 가상 주소의 VPN을 입력한다. 그리고 **VPN 중 TLB 집합의 번호를 의미하는 TBLI 부분을 통해 해당 PTE가 위치할 수 있는 집합에 찾아간다.** 그곳에서 만약 **TBLT와 동일한 태그를 갖는 유효한(Valid 비트 = 1) 라인이 발견되면 TLB 히트**이므로, 해당 라인에서 PTE를 가져오면 된다. 반면 **TLBT와 동일한 태그를 갖는 유효한 라인이 없다면 TLB 미스**이므로, 어쩔 수 없이 메인 메모리로 찾아가서 PTE를 직접 가져와야 한다. 전체적으로 캐시 메모리와 동작 방식이 거의 유사함을 알 수 있다.

※ 문맥 전환 시 TLB에 존재하는 모든 라인의 Valid 비트는 0으로 세팅된다.

※페이지 폴트 핸들러는 추방된 물리 페이지와 새로 들어온 물리 페이지의 정보를 바탕으로 PTE와 TLB의 내용을 함께 갱신한다. 특히, 추방되는 물리 페이지에 해당하는 TLB 내 PTE는 Valid 비트를 0으로 설정한다. 이를 통해 TLB 히트가 발생하면 반드시 해당 가상 페이지가 물리 페이지에 맵핑되어 있음을 보장하게 된다.

5-4. 메모리 참조 전체 과정★★★

지금까지 논의한 내용을 종합해 보자. 우선, 프로그램에서 메모리를 접근하는 경우는 두 가지이다. 하나는 PC에 저장된 값을 바탕으로 명령어 메모리에서 명령어를 읽어오는 경우이고, 나머지 하나는 데이터 메모리를 접근하는 특정 명령어(EX.popq, pushq, movq)를 실행하는 경우이다. 그리고 이러한 메모리 접근은 전부 가상 주소에 근거하여 이뤄진다. **이제 CPU가 메모리 접근을 위해 특정 가상 주소(VPN)를 발생시켰다고 가정해 보자. 그러면 우선 PTE를 가져오기 위해 TLB를 방문하게 된다.**

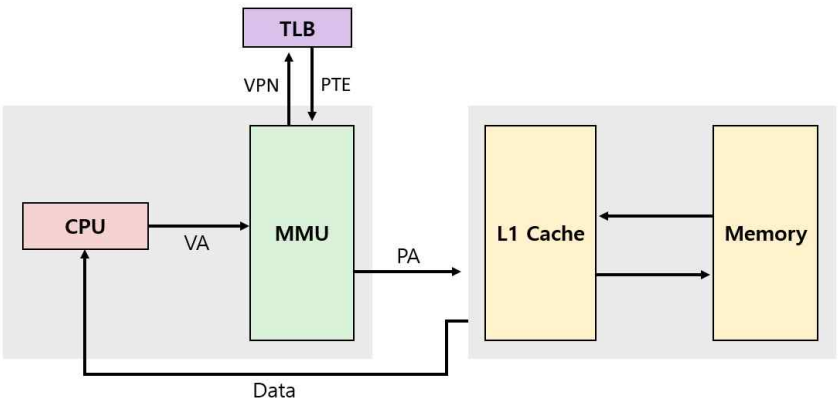
만약 **TLB 히트**라면, 그곳에 존재하는 PTE를 바탕으로 가상 주소를 물리 주소로 변환한다. 그러면 이제 해당 물리 주소를 바탕으로 메모리 계층(L1 캐시 → L2 캐시 → 메인 메모리)에 접근하면 된다(이 과정에서 발생하는 미스들은 전부 하드웨어 수준에서 처리). 참고로, **TLB 히트이면 해당 가상 페이지가 물리 페이지에 반드시 맵핑되어 있음이 보장**된다. (곧 알아보겠지만) TLB 미스가 발생하여 TLB에 PTE를 캐시 하는 시점에는 이미 해당 가상 페이지가 물리 페이지에 맵핑되어 있는 상태이며, 페이지 폴트 핸들러에 의해 특정 물리 페이지가 추방되는 경우에는 이에 해당하는 TLB 내 PTE의 Valid 비트도 0으로 세팅되기 때문이다.

반면 **TLB 미스**라면, 다시 두 가지 경우로 나뉜다. 먼저, **해당 가상 페이지가 물리 페이지에 맵핑되어 있는 경우**이다. 이러한 경우 메모리 계층에 접근하여 해당 가상 페이지에 대한 PTE를 가져오고, 이를 TLB에 반영해준다(이 과정은 하드웨어 혹은 소프트웨서 수준에서 처리). 그리고 해당 PTE를 바탕으로 가상 주소를 물리 주소로 변환하여 메모리 계층에 접근하면 된다.

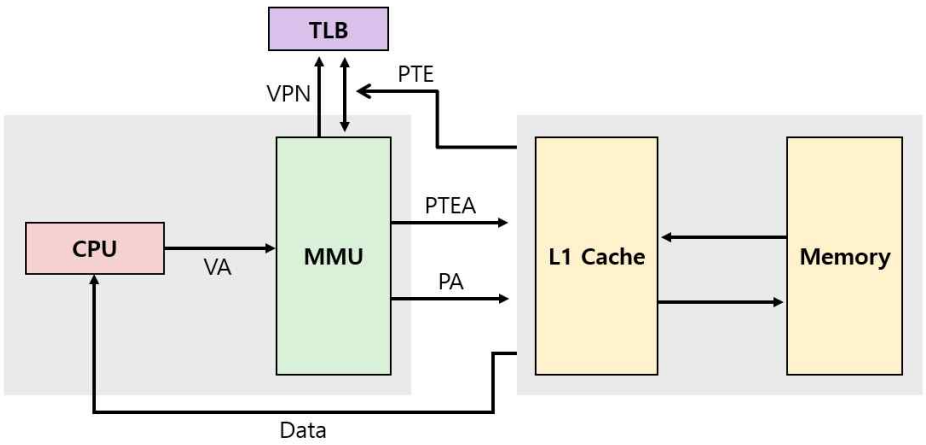
다음으로, **해당 가상 페이지가 물리 페이지에 맵핑되어 있지 않은 경우**이다. 이러한 경우 메모리 계층에 접근하여 가져오는 PTE의 정보를 보고 페이지 폴트 예외를 발생시킨다. 그러면 페이지 폴트 핸들러는 앞서 설명했듯이 특정 페이지를 추방하고 새로운 페이지를 가져온 뒤 이에 맞게 PTE와 TLB의 정보를 갱신해준다. 이제 다시 페이지 폴트를 유발했던 명령어를 재실행하면 바로 앞에서 설명했던 'TLB 미스인데 해당 가상 페이지가 물리 페이지에 맵핑되어 있는 경우'가 된다.

위에서 설명한 세 가지의 경우를 각각 그림으로 나타내면 다음과 같다.

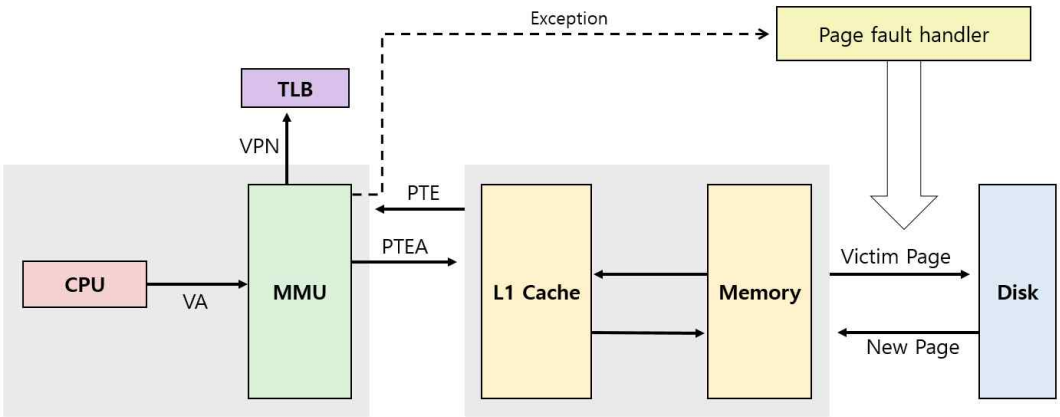
5-4-1. TLB 히트 (→ 페이지 히트)



5-4-2. TLB 미스, 페이지 히트



5-4-3. TLB 미스, 페이지 폴트



※ Miss Handling

1) 캐시 메모리 미스 : 하드웨어 수준의 처리

2) 페이지 폴트 : 소프트웨어 수준의 처리 (예외 메커니즘)

3) TLB 미스 : 하드웨어 또는 소프트웨어 수준의 처리 (하드웨어 수준의 처리는 성능이 좋고 빠르지만, 소프트웨어 수준의 처리는 여러 알고리즘의 개발 가능성이 열려 있으므로 Flexibility가 매우 높다는 특징이 있음)

Linking

1. Introduction

1-1. 링킹 (Linking)

링킹(Linking)이란 프로그램 코드 및 데이터의 조각들을 결합하여 메모리에 로드되어 실행될 수 있는 하나의 실행 파일을 만드는 과정을 의미한다. 이는 컴파일 타임에 수행될 수도 있고, 로드 타임(프로그램이 로더에 의해 실행되어 메모리에 로드될 때)에 수행될 수도 있으며, 혹은 애플리케이션 프로그램에 의해서 런타임에 수행될 수도 있다. **링커(Linker)는 링킹을 수행하는 프로그램**을 의미한다.

링커는 **각 모듈의 독립적인 컴파일**을 가능하게 함으로써 소프트웨어 개발에 혁신적인 변화를 가져왔다. 커다란 프로그램을 하나의 소스 파일로 개발하는 것이 아니라, 관리하기 용이한 작은 단위의 모듈들로 나누어서 개발할 수 있게 된 것이다. 특정 모듈이 나중에 수정된다면, 해당 모듈만 다시 컴파일하여 이미 컴파일되어 있는 다른 모듈들과 링킹을 수행하면 된다.

1-2. 링킹을 공부해야 하는 이유

그렇다면 링킹은 왜 공부해야 할까? 크게 다섯 가지의 이유가 있다. **첫째, 큰 규모의 프로그램을 개발하게 되면 링킹과 관련된 에러들에 많이 부딪힐 수 있기 때문이다.** 이때 링킹의 원리를 모른다면 디버깅에 곤란을 겪을 수 있다. **둘째, 위험한 프로그래밍 에러를 피하기 위해서이다.** 프로그래머가 예상치 못한 결과가 나올 수 있는 코드임에도 링커가 에러나 워닝을 띄우지 않는 경우가 있다. 예를 들어, 링커가 동일한 이름의 Weak 심볼을 여러 개 선언하는 것을 허용한다는 것을 모른다면 디버깅에 어려움을 겪을 수 있다. **셋째, 프로그래밍 언어의 스코프 규칙이 어떻게 구현되는지 이해할 수 있기 때문이다.** 예를 들어, 링킹의 원리를 알면 전역 변수와 지역 변수의 차이, 그리고 static 선언이 어떠한 의미를 지니는지 이해할 수 있게 된다. **넷째, 중요한 시스템 관련 개념들을 이해하기 위해서이다.** 예를 들어, 링킹은 프로그램의 로딩 및 실행 과정, 가상 메모리 기술, 페이징, 메모리 맵핑 등의 개념들과 아주 밀접하게 연관되어 있다. **마지막으로, 공유 라이브러리를 활용할 줄 알아야 하기 때문이다.** 과거에 비해 오늘날에는 공유 라이브러리의 동적 링킹이 핵심적인 프로그래밍 기술로 대두되고 있다. 링킹을 이해하면 더욱 능력을 인정받는 프로그래머가 될 수 있을 것이다.

1-3. 핵심 주제

이번 챕터의 핵심적인 주제는 다음과 같이 세 가지이다. 첫 번째는 **컴파일 타임에 이뤄지는 정적 링킹(Static Linking)**이고, 두 번째는 **로드 타임 시 이뤄지는 공유 라이브러리의 동적 링킹(Dynamic Linking of Shared Libraries at Load Time)**이며, 마지막은 **런타임 시 이뤄지는 공유 라이브러리의 동적 링킹(Dynamic Linking of Shared Libraries at Run Time)이다. 우리는 이러한 내용을 x86-64 리눅스 시스템에서 오브젝트 파일 포맷으로서 ELF-64를 사용하는 환경을 기준으로 설명할 것이다. 그러나 ISA, 운영체제, 그리고 오브젝트 파일 포맷이 다른 환경에서도 링킹의 핵심적인 원리는 크게 달라지지 않는다는 것을 기억하도록 하자.**

2. Compiler Drivers

(a) main.c

```
int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

(a) sum.c

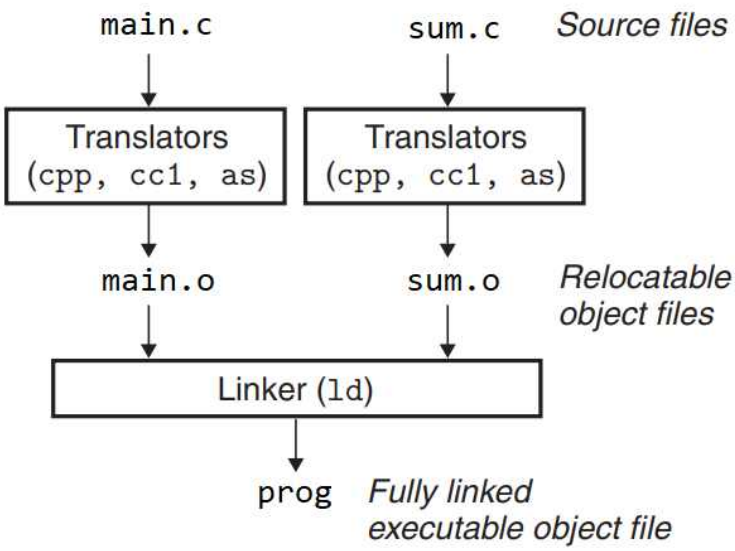
```
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

[Figure 1] C 언어 예시 코드

2-1. 컴파일 방법

대부분의 컴파일 시스템은 선행 처리기, 컴파일러, 어셈블러, 링커를 호출하는 컴파일러 드라이버를 제공한다. 예를 들어, [Figure 1]의 예시 코드는 다음과 같이 **GNU 컴파일 시스템의 GCC 드라이버**를 호출함으로써 컴파일할 수 있다. 그러면 GCC 드라이버는 아래 그림과 같이 선행 처리기(cpp), C 컴파일러(cc1), 어셈블러(as), 링커(ld)를 차례대로 호출함으로써 최종적으로 prog라는 하나의 완전한 실행 파일을 만들어 낸다.

```
linux> gcc -Og -o prog main.c sum.c
```



위 과정은 다음과 같이 개별적으로 진행할 수도 있다.

```
linux> cpp [other arguments] main.c /tmp/main.i
linux> cc1 /tmp/main.i -Og [other arguments] -o /tmp/main.s
linux> as [other arguments] -o /tmp/main.o /tmp/main.s
linux> ld -o prog [system object files and args] /tmp/main.o /tmp/sum.o
```

2-2. 실행 방법

그렇게 만들어진 실행 파일을 실행하는 방법은 다음과 같다. 그러면 쉘(Shell)은 로더라는 OS의 함수를 호출하는데, 이는 실행 파일의 코드와 데이터를 메모리에 로드한 뒤 CPU의 제어를 해당 프로그램의 시작 주소로 바꿔주게 된다.

```
linux> ./prog
```

3. Static Linking

리눅스의 LD 프로그램과 같은 **정적 링커(Static Linker)는 여러 개의 재배치 가능 오브젝트 파일들을 입력으로 받아서 하나의 완전한 실행 파일을 만들어 낸다.** 입력으로 들어오는 각각의 재배치 가능 오브젝트 파일들은 자신만의 코드 섹션과 데이터 섹션을 가진다. 이때 각 오브젝트 파일들은 그저 바이트 블록들의 배열에 지나지 않는다는 것을 기억하자. 어떤 바이트 블록은 프로그램 코드이고, 어떤 바이트 블록은 프로그램 데이터이며, 어떤 바이트 블록은 링커와 로더가 필요로 하는 정보들로 이뤄진 자료구조일 뿐인 것이다. **링커는 그러한 블록들을 적절히 합쳐주고, 각 블록에게 런타임에 필요한 가상 주소들을 알맞게 부여하며, 이에 맞게 코드 블록들과 데이터 블록들에 존재하는 메모리 주소들을 적절히 수정해준다.** 그리고 링커가 이러한 작업을 수행하는 데 필요한 대부분의 정보들은 컴파일러와 어셈블러가 만들어서 재배치 가능 오브젝트 파일 안에 담아준다. 링커는 그 정보들을 읽어서 링킹을 수행할 뿐, 타겟 머신에 대해서는 잘 알지 못한다. 방금 말한 링커의 작업을 두 단계로 요약하면 다음과 같다. 각 단계에 대한 자세한 내용은 뒷부분에 나오는 "***7. Symbol Resolution***"과"**8. *Relocation***" 에서 설명이 될 것이다.

3-1. 심볼 해석 (Symbol Resolution)

재배치 가능 오브젝트 파일들에서 등장하는 **각각의 심볼 참조(Symbol Reference)를 정확히 하나의 심볼 정의(Symbol Definition)에 연결하는 작업**을 말한다. 여기서 심볼(Symbol)이란 함수, 전역 변수, static 변수 등을 의미하며, non-static 지역 변수는 포함하지 않는다. 각 모듈은 특정 심볼을 정의하거나 참조하게 된다.

3-2. 재배치 (Relocation)

컴파일러와 어셈블러는 기본적으로 0번지에서 시작하는 코드 섹션과 데이터 섹션을 만들어 낸다. 링커는 **이러한 섹션들을 적절한 가상 주소로 재배치하고, 이에 따라 각 심볼 정의에도 알맞은 가상 주소를 부여**한다. 그리고 **각 심볼 참조가 올바른 심볼 정의를 가리킬 수 있도록 수정**해준다. 이러한 작업은 어셈블러에 의해 만들어져서 재배치 가능 오브젝트 파일 안에 담기는 **재배치 엔트리(Relocation Entry)**들의 정보를 바탕으로 수행된다.

4. Objects Files

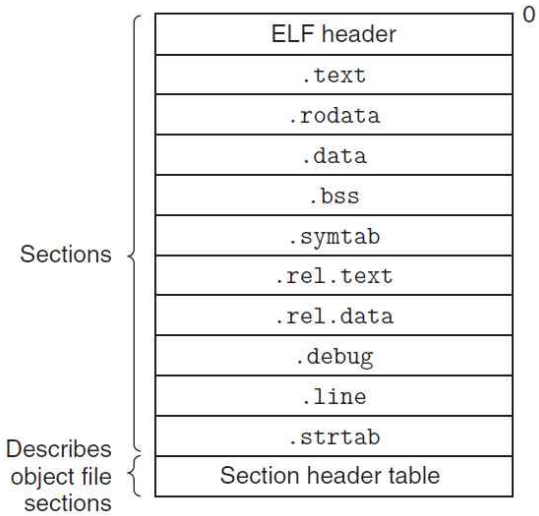
4-1. 오브젝트 파일 유형

오브젝트 파일의 유형은 다음과 같이 세 가지이다. 첫째, **재배치 가능 오브젝트 파일(Relocatable Object File)**이다. 이는 정적 링킹 시에 다른 재배치 가능 오브젝트 파일들과 결합되어 하나의 완전한 실행 파일을 만들어 내는 데 사용되는 오브젝트 파일을 의미한다. 다음은 **실행 가능 오브젝트 파일(Executable Object File)**이다. 간단하게 실행 파일이라고도 부르며, 코드와 데이터가 바로 메모리에 로드되어 실행될 수 있는 오브젝트 파일을 의미한다. 마지막은 **공유 오브젝트 파일(Shared Object File)**로, 로드 타임이나 런타임에 동적으로 메모리에 로드되어 링킹 될 수 있는 재배치 가능 오브젝트 파일을 의미한다. 이 중에 재배치 가능 오브젝트 파일(또는 공유 오브젝트 파일)은 컴파일러와 어셈블러에 의해 만들어지며, 실행 파일(=실행 가능 오브젝트 파일)은 링커에 의해 만들어진다.

4-2. 오브젝트 파일 포맷

각 오브젝트 파일은 특정 오브젝트 파일 포맷을 가진다. 오브젝트 파일 포맷은 컴파일러와 어셈블러가 만들어내는 정보들을 재배치 가능 오브젝트 파일 안에 어떤 구조로 담을지 결정한다. 오브젝트 파일 포맷은 시스템마다 각기 다르며, 우리는 **ELF(Executable and Linkable Format)**라는 오브젝트 파일 포맷을 기준으로 설명할 것이다. 그러나 오브젝트 파일 포맷이 달라져도 기본적인 링킹의 원리는 크게 달라지지 않는다는 것을 기억하도록 하자.

5. Relocatable Objects Files



왼쪽 그림은 **ELF 포맷을 따르는 재배치 가능 오브젝트 파일의 내부 구조**를 나타낸다. 파일 내부의 바이트 블록들은 역할에 따라 위 그림과 같이 몇 개의 영역으로 구분된다. 각 영역의 역할은 다음과 같다.

구분		역할
ELF 헤더 (ELF Header)		시스템의 워드 사이즈나 바이트 오더링과 같은 **시스템의 속성 정보**가 저장된다. 그리고 **링커가 이 파일을 읽어서 분석할 때 알아야 하는 정보들**도 저장된다. 예를 들어, ELF 헤더의 크기, 오브젝트 파일의 유형(EX. 재배치 가능 오브젝트 파일, 실행 파일, 공유 오브젝트 파일), 타겟 머신의 유형(EX. x86-64), 섹션 헤더 테이블의 파일 오프셋, 섹션 헤더 테이블에 존재하는 엔트리의 개수와 각 엔트리의 크기 등이 이곳에 저장된다.
섹션 (Section)	.text	컴파일된 프로그램의 명령어에 해당하는 **기계어 코드**들이 저장된다.
	.rodata	문자열 등의 상수나 switch 점프 테이블과 같은 **읽기 전용 값**들이저장된다.
	.data	**0이 아닌 값으로 초기화되는 전역 변수 및 static 변수**들이 저장된다. 참고로 non-static 지역 변수는 런타임 시에 스택 영역에서 저장 및 관리되므로, 오브젝트 파일에는 저장되지 않는다.
	.bss	**0으로 초기화되거나 초기화가 되지 않는 전역 변수 및 static 변수**들이 저장된다. 이 섹션은 재배치 가능 오브젝트 파일에서는 전혀 공간을 차지하지 않으며, 실행 파일에서도 이곳에 저장될 데이터들의 총사이즈 정보를 저장할 만큼의 공간만 차지한다. 프로그램이 실행되면 그때서야 실행 파일에 저장되어 있던 사이즈 정보를 바탕으로 .bss 섹션을 메모리에 할당한다. 해당 섹션은 어차피 전부 0으로 초기화될 것이기 때문에 미리 공간을 차지하고 있을 필요가 없다.
	.symtab	**이 모듈에서 정의하거나 참조하는 모든 심볼(함수, 전역 변수, static 변수 등)들의 정보**가 저장된다. 이는 모든 재배치 가능 오브젝트 파일들이 가지는 기본적인 섹션으로, -g 컴파일 옵션을 줘야 포함되는 컴파일러의 심볼 테이블과는 다른 것이다.
	.rel.text	**링킹 시 재배치가 필요한 .text 섹션 내 메모리 로케이션들의 정보**가 저장된다. 예를 들어, 외부 함수를 호출하거나 전역 변수를 참조하는 명령어들을 재배치가 필요하므로 해당 재배치 작업을 위한 정보들을 이곳에 저장한다. 반면, 동일 모듈 함수를 호출하는 명령어의 경우 상대 주소를 이용하면 재배치가 필요 없으므로 별다른 정보를 저장하지 않는다. 이 섹션은 특별한 컴파일 옵션을 주지 않는 한 재배치 가능 오브젝트 파일에만 포함된다.
	.rel.data	**링킹 시 재배치가 필요한 .data 섹션 내 메모리 로케이션들의 정보**가 저장된다. 예를 들어, 전역 변수나 외부 함수의 주소값으로 초기화되는 전역 변수들은 재배치가 필요하므로 해당 재배치 작업을 위한 정보들을 이곳에 저장한다. 참고로, .bss 섹션의 데이터들은 어차피 모두 0으로 초기화되므로 재배치 작업이 필요 없다(애초에 재배치 가능 오브젝트 파일과 실행 파일에는 .bss 섹션이 아직 할당되어 있지도 않음).
	.debug	프로그램에서 정의된 지역 변수 및 typedef의 정보들을 저장하는 디버깅 심볼 테이블, 해당 프로그램에서 정의하거나 참조하는 전역 변수들, 원본 C 소스 파일 등과 같이 **디버깅을 위한 정보**가 저장된다. 이 섹션은 -g 컴파일 옵션을 줄 때만 포함된다.
	.line	**원본 C 소스 파일의 라인들과 .text 섹션에 존재하는 기계어 코드들의 맵핑 정보**가 저장된다. 이 섹션은 -g 컴파일 옵션을 줄 때만 포함된다.
	.strtab	.symtab 섹션과 .debug 섹션의 심볼 테이블에 존재하는 **심볼 이름들에 해당하는 문자열들**과 섹션 헤더 테이블에 존재하는 **섹션 이름들에 해당하는 문자열들**이 저장된다. 예를 들어, 심볼 테이블의 각 엔트리는 해당 심볼의 이름에 해당하는 문자열의 .strtab 섹션 내 오프셋을 저장한다. 물론 각 문자열은 널 문자로 끝나는 형태로 저장된다.
섹션 헤더 테이블 (Section Header Table)		**각 섹션의 크기와 위치 정보**가 저장된다. 각 섹션에 대해 고정된 크기의 엔트리를 갖는다.

6. Symbols and Symbol Tables

6-1. 심볼의 종류

각각의 재배치 가능 오브젝트 파일들은 자신이 정의하거나 참조하는 심볼들의 정보를 담고 있는 심볼 테이블을 가지고 있다. 심볼 테이블에 저장되는 심볼의 종류는 다음과 같이 크게 **전역 심볼(Global Symbol)**과**지역 심볼(Local Symbol)**로 구분된다.

종류	설명	예시
전역 심볼 (Global Symbol)	Case 1)모듈 m에 의해 정의되고 다른 모듈에 의해 참조될 수 있는 심볼	non-static 함수 non-static 전역 변수
	Case 2)모듈 m에 의해 참조되지만 다른 모듈에 정의되어 있는 심볼	
지역 심볼 (Local Symbol)	모듈m에 의해 정의되고 오직 모듈m에 의해서만 참조될 수 있는 심볼	static함수 static전역/지역변수

※**static 지역 변수**: 런타임 스택 영역에서 저장 및 관리되는 일반적인 지역 변수와 달리, static 지역 변수는 컴파일러에 의해 고유한 이름을 부여받고 심볼 테이블에 지역 심볼로서 저장이 된다. 그리고 초기화 여부에 따라 .data 섹션 또는 .bss 섹션에 들어가게 된다. 고유한 이름을 부여받는 이유는 서로 다른 함수 내에 동일한 이름의 static 지역 변수가 선언될 수 있기 때문이다.

6-2. 심볼 테이블 엔트리

어셈블러는 컴파일러가 .s 파일에 담은 심볼들의 정보를 바탕으로 심볼 테이블을 구성한 뒤, 이를 재배치 가능 오브젝트 파일의 .symtab 섹션에 저장한다. 심볼 테이블에 저장되는 각 엔트리의 구조를 C 언어의 구조체로 표현하면 다음과 같다.

```
typedef struct {
    int   name;       /* String Table Offset */
    char  type:4,     /* Function or Data (4 bits) */
          binding:4;  /* Global or Local (4 bits) */
    char  reserved;   /* Unused */
    short section;    /* Section Header Table Index */
    long  value;      /* Section Offset or Absolute Address */
    long  size;       /* Object Size in Bytes */
} Elf64_Symbol;
```

***name* 필드**는 심볼 이름에 해당하는 문자열의 .strtab 섹션 내 오프셋을 저장한다. ***type* 필드**는 Function 또는 Data를 저장한다. ***binding* 필드**는 심볼의 종류를 나타내는 것으로, Global 또는 Local을 저장한다. ***section* 필드**는 심볼이 할당될 섹션을 나타내는 것으로, 해당 섹션의 섹션 헤더 테이블 내 인덱스를 저장한다. ***value* 필드**는 심볼의 위치를 나타내는 것으로, 재배치 가능 오브젝트 파일이라면 섹션 내 오프셋을 저장하고 실행 파일이라면 절대 가상 주소를 저장한다. ***size* 필드**는 심볼 데이터의 바이트 단위 크기를 저장한다.

section 필드는 실제로 존재하지 않는 Pseudo 섹션을 가리킬 수도 있다. **Pseudo 섹션은 섹션 헤더 테이블에 엔트리가 없는 섹션을 의미하는 것으로, 재배치 가능 오브젝트 파일에만 존재하고 실행 파일에는 존재하지 않는다.** 대표적인 Pseudo 섹션으로는 다음과 같이 세 가지이다. 첫 번째, 재배치(Relocation)가 이뤄지면 안 되는 심볼임을 나타내는 **ABS 섹션**이다. 두 번째, 다른 모듈에 정의되어 있는 심볼임을 나타내는 **UNDEF 섹션**이다. 마지막으로 세 번째, 초기화가 되지 않는 전역 변수들을 위한 **COMMON 섹션**이다. 이때 의아한 점이 생길 수도 있다. 위에서 ELF 포맷을 설명할 때는 초기화되지 않는 전역 변수들이 .bss 섹션에 저장되는 것이라고 했기 때문이다. COMMON 섹션과 .bss 섹션을 구분한 이유에 대해서는 뒷부분에 나오는 ***"7-2. 전역 심볼 중복 문제 처리 (COMMON vs .bss)"*** 에서 설명하도록 하겠다.

※ 참고로 COMMON 섹션에 해당하는 심볼의 경우 *value* 필드가 정렬 요구 조건(Alignment Requirement)을 저장하며, *size* 필드는 심볼 데이터의 최소 크기를 저장한다고 알려져 있다.

6-3. 심볼 테이블 예시

다음은 [Figure 1]의 예시 코드에 해당하는 심볼 테이블을 보여준다. 전역 심볼인 main 함수는 24바이트의 크기를 가지고, .text 섹션(1)에 할당되며, .text 섹션 내에서도 0번째 위치에 할당된다는 것을 알 수 있다. 그리고 전역 변수인 array 배열은 총 8바이트의 크기를 가지고, .data 섹션(3)에 할당되며, .data 섹션 내에서도 0번째 위치에 할당된다는 것을 알 수 있다. 마지막으로 함수 sum은 해당 모듈 내에 정의되어 있지 않으므로 UNDEF 섹션으로 표기되어 있음을 볼 수 있다.

```
Num:      Value          Size  Type     Bind    Vis      Nds  Name
  8: 0000000000000000    24  FUNC     GLOBAL  DEFAULT    1  main
  9: 0000000000000000     8  OBJECT   GLOBAL  DEFAULT    3  array
 10: 0000000000000000     0  NONTYPE  GLOBAL  DEFAULT  UND  sum
```

7. Symbol Resolution★★★

7-1. 기본

심볼 해석(Symbol Resolution)이란 링커가 입력으로 들어오는 재배치 가능 오브젝트 파일들의 심볼 테이블 정보를 바탕으로 각각의 심볼 참조를 정확하게 하나의 심볼 정의에 연결시키는 작업을 뜻한다. 심볼 정의란 곧 해당 심볼을 정의하는 심볼 테이블 엔트리를 의미하며, 심볼 참조란 코드 상에서 해당 심볼을 참조하는 부분을 의미한다.

7-2. 지역 심볼의 해석

지역 심볼(Local Symbol)의 경우 심볼 해석이 아주 쉽다. **지역 심볼은 정의되는 모듈 내에서만 참조가 가능할 뿐만 아니라(= Scope가 모듈 내로 한정됨), 컴파일러는 한 모듈 내에서 각 지역 심볼의 정의가 고유하다는 것을 보장**하기 때문이다. 실제로,

static 전역 변수는 동일한 이름으로 여러 번 정의되면 컴파일 에러가 발생한다. 또한 static 지역 변수도 하나의 함수 내에서는 동일한 이름으로 여러 번 정의되면 컴파일 에러가 발생하며, 서로 다른 함수에서 동일한 이름으로 정의가 되더라도 컴파일러에 의해 각기 다른 이름을 부여받아 심볼 테이블에 들어가게 된다. 따라서, **지역 심볼의 참조는 동일한 모듈의 심볼 테이블 상에 존재하는 심볼 정의를 그대로 연결하기만 하면 된다.**

7-3. 전역 심볼의 해석

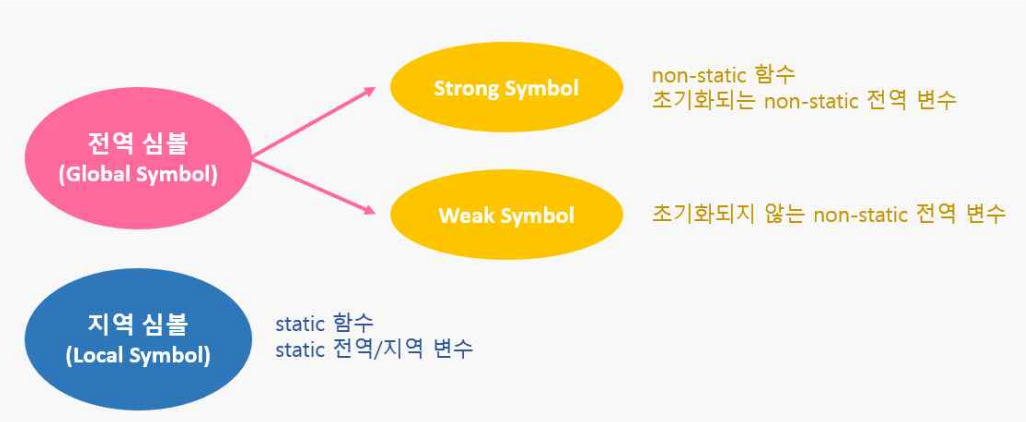
반면 ***전역 심볼(Global Symbol)***은 심볼 해석이 다소 까다롭다. **전역 심볼의 경우 정의되는 모듈 외부에서도 참조가 가능하기 때문이다.** 이로 인해 전역 심볼의 참조에 대응되는 심볼 정의를 현재 모듈의 심볼 테이블에서 찾지 못할 수도 있다. 이러한 경우 컴파일러는 해당 전역 심볼이 다른 모듈에 정의된 것이라고 가정하고, 섹션이 UNDEF인 심볼 테이블 엔트리를 만들어서 심볼 테이블에 저장한다. 그리고 나중에 **링커가 섹션이 UNDEF인 각각의 심볼 테이블 엔트리에 대해 올바른 심볼 정의를 다른 모듈의 심볼 테이블에서 찾아 심볼 해석을 진행**하게 된다. 이때 만약 링커가 입력으로 들어오는 재배치 가능 오브젝트 파일의 심볼 테이블들을 전부 확인했는데도 해당 심볼 참조에 대한 심볼 정의를 찾지 못하면, 에러를 발생시키고 즉시 종료한다.

또한, **전역 심볼의 경우 여러 모듈이 동일한 이름으로 정의하는 것도 가능하다**. 컴파일 단계에서는 다른 모듈에서 동일한 이름의 전역 심볼을 정의했는지 알 방법이 없기 때문이다. 만약 여러 모듈이 동일한 이름의 전역 변수를 정의했다면, **링커는 심볼 해석을 위해 전역 심볼의 중복 문제를 먼저 처리해야 한다.** 상황에 따라서 에러를 발생시키고 즉시 종료할 수도 있고, 동일한 이름으로 정의된 여러 개의 전역 심볼들 중 하나를 선택하고 나머지는 폐기할 수도 있다. **전역 심볼 중복 문제를 처리하고 나면, 각 전역 심볼은 고유한 이름을 가진다는 것이 보장되므로 해당 이름의 심볼 정의를 심볼 테이블에서 찾아 심볼 해석을 진행하면 된다.**

7-4. 전역 심볼 중복 문제 처리 (COMMON vs .bss)

그렇다면 링커는 전역 심볼의 중복 문제를 어떻게 처리하는 것일까? 이를 이해하기 위해서는 먼저 Strong 심볼과 Weak 심볼의 개념을 알아야 한다. 다음 그림과 같이 **전역 심볼(Global Symbol)은 초기화 여부에 따라 다시 Strong 심볼과 Weak 심볼로 구분된다.** non-static 함수와 초기화되는 non-static 전역 변수는 Strong 심볼에 해당하며, 초기화되지 않는 non-static 전역 변수는 Weak 심볼에 해당한다. **그리고 Strong 심볼과 달리 Weak 심볼은 동일한 이름으로 여러 번 정의될 수 있다는 특징이 있다.**

Strong 심볼 vs Weak 심볼



컴파일러는 컴파일 단계에서 각 심볼들의 정보를 파악한 뒤 이를 .s 파일에 담아서 어셈블러에게 전달한다. 이때, 자기 자신 모듈에서 정의되는 전역 심볼의 경우에는 Strong 심볼인지, 아니면 Weak 심볼인지를 파악하여 이 정보도 함께 어셈블러에게 전달한다. 그러면 어셈블러는 그렇게 전달받은 심볼들의 정보를 바탕으로 심볼 테이블을 구성하여 재배치 가능 오브젝트 파일을 만들게 된다. 따라서 나중에 **링커가 링킹을 수행하는 시점에는 이미 심볼 테이블에 각 전역 심볼의 유형(Strong 또는 Weak) 정보가 담겨 있기 때문에, 링커는 이러한 정보를 바탕으로 전역 심볼 중복 문제를 처리**하게 된다.

우리가 전제한 리눅스 시스템의 링커는 다음과 같은 세 가지 기준으로 전역 심볼 중복 문제를 처리한다. 첫째, 동일한 이름의 Strong 심볼이 여러 개 정의되어 있다면 에러를 발생시키고 즉시 종료한다. 둘째, 동일한 이름의 Strong 심볼 하나와 Weak 심볼 여러 개가 정의되어 있다면 Weak 심볼들은 전부 폐기하고 Strong 심볼을 선택한다. 셋째, 동일한 이름의 Weak 심볼이 여러 개 정의되어 있다면 그중에 아무거나 선택한 뒤 나머지 Weak 심볼들은 전부 폐기한다. **심볼을 선택한다 함은 그 이름의 심볼 참조들을 해당 심볼 정의에 연결하게 된다는 것**을, 심볼을 폐기한다 함은 심볼 테이블에서 해당 심볼 정의를 삭제한다는 것을 의미한다고 이해하면 된다.

이름 규칙

- 동일한 이름의 Strong 심볼 여러 개 → 링커 에러
- 동일한 이름의 Strong 심볼 하나 + 동일한 이름의 Weak 심볼 여러 개 → Strong 심볼 선택
- 동일한 이름의 Weak 심볼 여러 개 → 아무거나 선택

※ Weak 심볼로 인해 프로그래머가 겪을 수 있는 어려움

Weak 심볼이 여러 개 정의되는 경우, 링커는 그것들 중 하나를 임의로 선택하며 이 과정에서 특별한 메시지를 만들지 않는다. 따라서 링커의 원리를 알지 못하는 프로그래머는 이와 같은 상황에서 디버깅에 큰 어려움을 겪을 수 있다. 심지어, 동일한 이름으로 정의되는 Weak 심볼들이 자료형까지 다르다면 문제는 더욱 심각해진다. 예를 들어, A 모듈에서 int형 변수 x를 선언하고 B 모듈에서 double형 변수 x를 선언했는데 둘 다 초기화를 하지 않았다고 해보자. 그리고 B 모듈에 "x = -0.0;"과 같은 코드가 있었다고 해보자. 이때 링커에 의해 심볼 x로서 A 모듈의 변수 x가 선택이 된다면 무슨 일이 일어날까? A 모듈을 컴파일할 때 컴파일러는 x가 double 변수일 것이라고 가정하여 명령어를 만들어냈을 것이다. 그러나 실제로는 A 모듈의 변수 x가 선택되었으므로, 특정 실수를 표현하는 8바이트 배열이 4바이트 공간에 덮어쓰기 될 것이다. 이 경우 링커가 워닝 메시지를 만들긴 하지만, 대부분의 프로그래머는 큰 규모의 프로그램을 개발할 때 작은 워닝들은 무시하는 경향이 있기 때문에 디버깅에 어려움을 겪을 수 있다. 그래서 링커의 원리를 어느 정도 알아야 하는 것이다.

그렇다면 어셈블러는 심볼 테이블에 각 전역 심볼의 유형(Strong 또는 Weak) 정보를 어떤 방식으로 저장할까? 이는 곧 링커가 심볼 테이블을 보고 각 전역 심볼의 유형을 어떻게 파악할 수 있는지와 동일한 물음이다. **어셈블러는 컴파일러로부터 전달받은 심볼들의 정보를 바탕으로 심볼 테이블을 구성할 때, 각 심볼 테이블 엔트리에 해당 심볼이 가상 주소 공간의 어떤 섹션에 할당될 것인지 표기**해야 한다. 예를 들어, 함수들은 .text 섹션으로 표기하면 되고, 0이 아닌 값으로 초기화가 되는 전역 변수들은 .data 섹션으로 표기하면 된다.

그러나 **Weak 심볼**을 마주쳤을 때 문제가 발생한다. 초기화되지 않는 전역 변수니까 .bss 섹션으로 표기하면 되지 않냐고 반문할 수도 있다. 하지만 그렇지 않다. **어셈블러는 심볼 테이블을 만드는 시점에 해당 Weak 심볼이 나중에 링커에 의해 선택될지 미리 알 수 없다.** 링커에 의해 선택된다는 것이 보장되면 .bss 섹션으로 표기해도 되지만, 선택되지 않을 수도 있기 때문에 그렇게 함부로 단정하면 안 된다. 따라서, 어셈블러와 링커는 다음과 같은 약속을 하게 된다. **어셈블러는 Weak 심볼에 해당하는 심볼 테이블 엔트리에 COMMON 섹션으로 표기하고, 나중에 링커는 COMMON 섹션으로 표기되어 있는 동일한 이름의 심볼들을 따로 모아서**

(위에서 설명한 방식에 근거하여) 전역 심볼 중복 문제를 처리하도록 하는 것이다. 결국, 어셈블러는 각 전역 심볼의 유형을 명시적으로(Explicitly) 각 심볼 테이블 엔트리에 저장하는 것이 아니라 섹션 정보를 빌려서 링커에게 Weak 심볼의 존재를 암묵적으로(Implicitly) 알리게 되며, 이 과정은 컴파일러, 어셈블러, 그리고 링커가 입을 모아서 맞춘 하나의 약속에 기반한다는 것을 알 수 있다.

이쯤에서 다시 한번 상기해야 할 중요한 사실이 하나 있다. 이는 **전부 전역 심볼(Global Symbol)에만 해당하는 내용**이라는 것이다. 앞서 한 차례 설명했듯이, 지역 심볼(Local Symbol)에 해당하는 static 지역/변수들은 Scope가 해당 모듈 내로 한정되며 각기 고유한 이름을 가지고 있다는 것이 보장되기 때문에 (COMMON 섹션으로 표기할 필요가 없이) 바로 .data 또는 .bss 섹션으로 표기해도 된다. COMMON 섹션은 폐기될 수도 있는 심볼들을 처리하기 위한 수단일 뿐이기 때문이다. 지역 심볼은 폐기될 일이 없다.

위와 같이 COMMON 섹션으로 표기된 전역 심볼들의 중복 문제를 처리하고 나면, 최종적으로 선택된 심볼 정의들의 총사이즈가 실행 파일의 .bss 섹션에 저장될 값에 더해지고, 실행 파일에는 더 이상 COMMON 섹션이 존재하지 않게 된다.(단, 동적 라이브러리의 코드/데이터에 대한 심볼 참조가 있는 경우에는 여전히 COMMON 심볼이 남아있을 수도 있다.)실행 파일의 .bss 섹션에는 데이터들 자체가 아니라 그 데이터들의 총사이즈가 저장되기 때문이다.

⇒ 링크 결과 실행 파일의 .bss 섹션에 저장될 값 = .bss 심볼 정의들의 총사이즈 + 선택된 COMMON 심볼 정의들의 총 사이즈

7-5. 정적 라이브러리 링킹

대부분의 컴파일 시스템은 **서로 연관된 여러 오브젝트 모듈들을 하나의 정적 라이브러리(Static Library) 파일로 패키지화**하는 기능을 제공한다. 그렇게 만들어지는 **정적 라이브러리 파일은 링킹을 수행할 때 링커에 입력으로 들어가게 되며, 실제로 프로그램이 참조하는 심볼을 정의하는 오브젝트 모듈만 실행 파일 안에 포함**된다. 리눅스 시스템을 기준으로 정적 라이브러리는 Archive라는 이름의 파일 형식으로 저장되며, 파일 확장자는 .a이다. 그리고 Archive 파일은 각 멤버 오브젝트 모듈의 크기와 위치 정보를 명시하는 헤더를 가진다.

7-5-1. 정적 라이브러리의 필요성

정적 라이브러리는 왜 필요할까? 설명을 위해 실제 C 언어 정적 라이브러리를 두 개 소개하겠다. 하나는 표준 입출력, 문자열 조작, 수학과 관련된 함수들을 포함하는 libc.a 파일이고, 다른 하나는 실수를 다루는 수학과 관련된 함수들을 포함하는 libm.a 파일이다. 만약 이러한 정적 라이브러리가 존재하지 않는다고 하면 해당 라이브러리가 포함하는 함수들을 어떻게 사용할 수 있을지 생각해보자. 다음과 같이 크게 세 가지 방법을 떠올릴 수 있다.

먼저, **컴파일러가 특정 함수에 대한 호출 문을 발견하면 해당 함수의 코드를 자신이 직접 만들어 내는 방법**이 있다. 실제로 표준 함수의 개수가 적은 Pascal 등의 언어에서는 이 방식을 채택한다. 그러나 C 언어의 경우 적합하지 않다. C 언어 표준 함수의 개수가 상당히 많을뿐더러, 컴파일러의 무게가 지나치게 무거워지며, 함수가 추가/수정/삭제될 때마다 새로 컴파일러를 만들어야 하기 때문이다. 물론 프로그래머 입장에서는 편할 수도 있다. 코드를 작성하기만 하면 알아서 필요한 함수들의 코드를 만들어 주기 때문이다.

다음으로,**C 언어의 모든 표준 함수들을 하나의 재배치 가능 오브젝트 파일 안에 담는 방법**이 있다. 이렇게 하면 프로그래머들은 실행 파일을 만들 때 단 하나의 파일만 입력시켜주면 되기 때문에 편할 것이다. 또한 첫 번째 방법과 달리 표준 함수들의 구현과

컴파일러의 구현을 분리한다는 점에서 긍정적이다. 그러나 너무나도 큰 문제가 하나 있다. 바로 디스크 및 메모리 낭비가 너무나도 심하다는 것이다. 그 많은 함수들의 코드가 모든 실행 파일에 포함될 뿐만 아니라, 프로그램이 실행될 때도 메모리에 적재되기 때문이다. 또한 함수를 하나 추가/수정/삭제할 때마다 전체를 다시 컴파일해야 한다는 번거로움으로 인해 유지보수 및 개발 효율이 상당히 저하될 수 있다.

마지막으로, **각 표준 함수별로 재배치 가능 오브젝트 파일들을 만들어 놓고 그것들을 잘 알려진 디렉토리에 저장해두는 방식**이다. 하지만 이렇게 하면 프로그래머들은 실행 파일을 만들 때마다 필요한 함수에 해당하는 모듈들을 직접 찾아서 입력으로 넣어줘야 한다. 이는 에러에 상당히 취약한 방법일 뿐만 아니라 시간이 굉장히 많이 소요되는 번거로운 작업이므로 좋다고 평가하기 힘들 것이다.

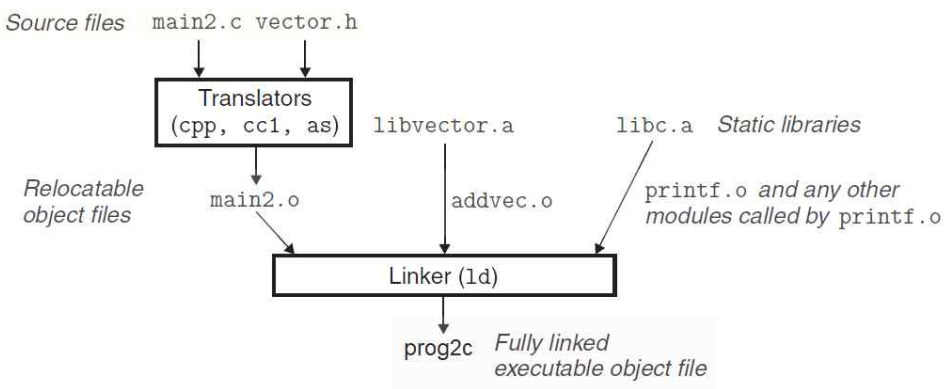
이러한 문제점들을 모두 해결해주는 것이 바로 정적 라이브러리이다. 정적 라이브러리를 사용하면 프로그래머들은 필요한 함수가 포함되어 있는 라이브러리 파일의 이름만 커맨드 라인에 적으면 된다. 정적 라이브러리 파일을 입력해도 **실제로 실행 파일에 복사되어 들어가는 부분은 프로그램이 실제로 참조한 심볼이 존재하는 오브젝트 모듈뿐**이기 때문에 디스크와 메모리의 낭비를 막을 수 있다. 또한 많이 사용되는 표준 함수들을 포함하는 libc.a 등의 **중요한 정적 라이브러리 파일들은 대부분 컴파일러 드라이버가 알아서 입력**시켜주기 때문에 프로그래머의 수고도 덜어줄 수 있다.

7-5-2. 정적 라이브러리 파일을 만드는 방법

addvec 함수를 포함하는 addvec.c 파일과 multvec함수를 포함하는 multvec.c 파일을 가지고 libvector.a라는 이름의 정적 라이브러리 파일을 만드는 방법은 다음과 같다. 참고로 -static은 메모리에 로드된 뒤 더 이상의 동적 링킹이 필요 없이 완전하게 실행될 수 있는 실행 파일을 만들어야 한다는 것을 나타내는 옵션이고, -lvector는 libvector.a의 약칭이며, -L은 libvector.a가 현재 디렉토리에 위치함을 나타내는 옵션이다.

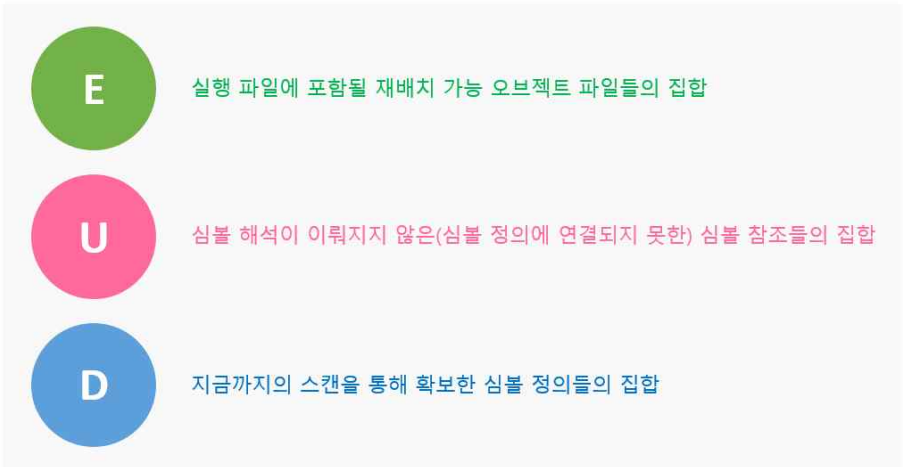
```
linux> gcc -c addvec.c multvec.c
linux> ar rcs libvector.a addvec.o multvec.o
linux> gcc -c main2.c
linux> gcc -static -o prog2c main2.o ./libvector.a 또는
       gcc -static -o prog2c main2.o -L. -lvector
```

이때 만약 main2.o 모듈이 addvec.o의 addvec 함수만 참조한다면, addvec.o는 실행 파일에 포함되지만 multvec.o은 실행 파일에 포함되지 않게 된다. 지금까지의 과정을 그림으로 나타내면 다음과 같다.



7-6. 심볼 해석 전체 과정

이제 본격적으로 **심볼 해석(Symbol Resolution)**의 진행 과정을 자세히 알아보도록 하자. 링커는 **커맨드 라인에 입력되는 순서대로(왼쪽 → 오른쪽) 재배치 가능 오브젝트 파일들과 아카이브 파일들을 하나씩 스캔**한다. (참고로 커맨드 라인에 확장자가 .c인 파일이 입력되면 그것은 자동으로 .o가 확장자인 파일로 먼저 번역된다.)링커는 스캔 과정에서 다음과 같은 세 개의 집합을 내부적으로 관리한다. 초기에는 셋 다 빈 집합이다.



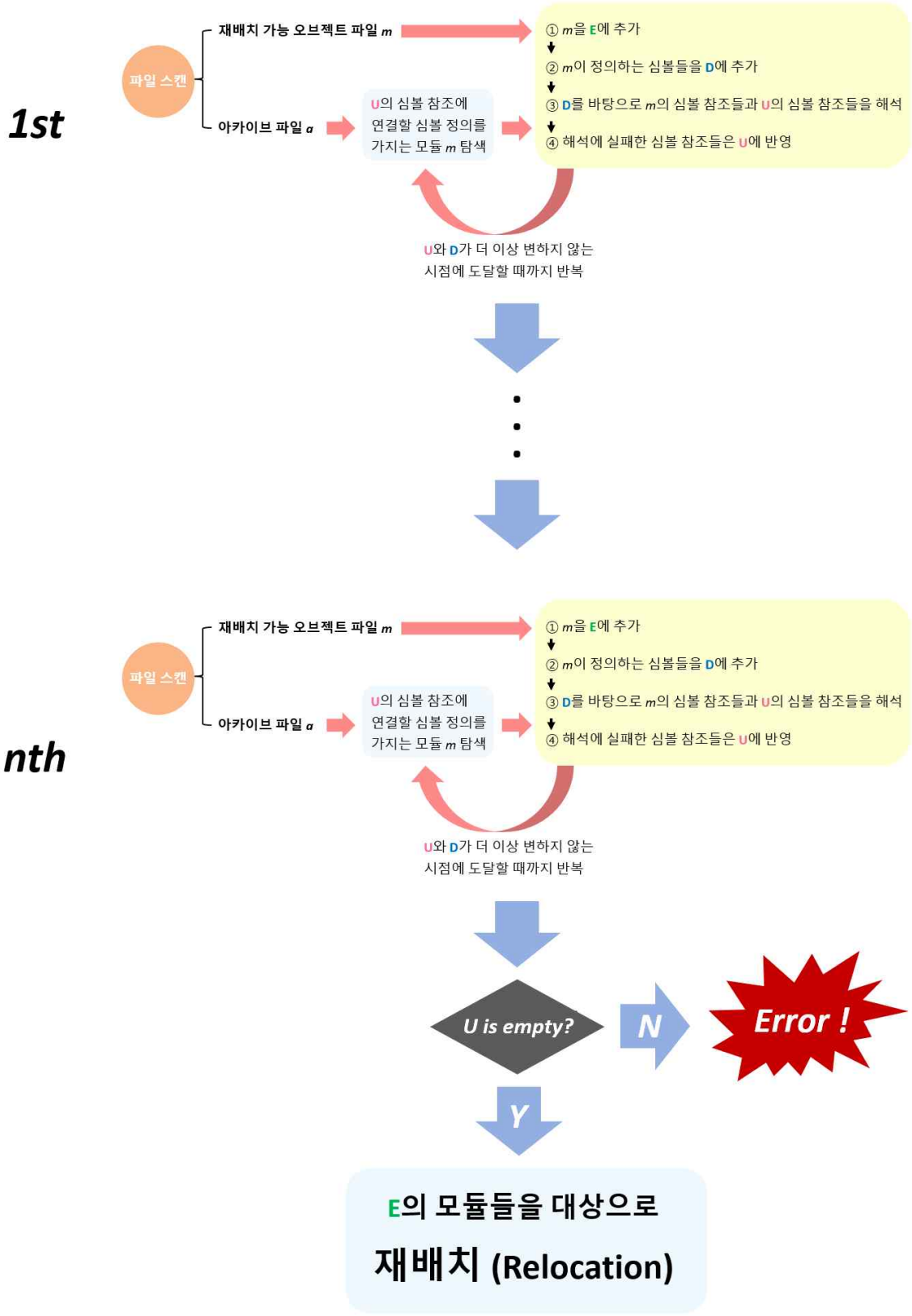
링커는 하나의 파일을 스캔할 때마다 다음과 같은 동작을 수행한다. 읽은 파일이 재배치 가능 오브젝트 파일인지, 아니면 아카이브 파일인지 판단한다. 그리고 그 판단 결과에 따라 다음 그림과 같은 동작을 수행하는 것을 반복하면서 모든 심볼 참조들을 정확히 하나의 심볼 정의(하나의 심볼 테이블 엔트리)에 연결하는 것을 시도한다. 만약 커맨드 라인 입력 순서대로 모든 파일을 스캔했음에도 해석되지 않은 심볼 참조가 있다면 링커는 에러 메시지와 함께 즉시 종료한다. 그렇지 않고 모든 심볼 참조를 성공적으로 하나의 심볼 정의에 연결했다면 이제 E의 모듈들을 대상으로 재배치(Relocation)를 진행한다.

이러한 링커의 동작 방식으로 인해 주의해야 할 점이 하나 있다. 바로 커맨드 라인의 입력 순서이다. **심볼 정의가 포함된 정적 라이브러리 파일을 심볼 참조가 포함된 재배치 가능 오브젝트 파일보다 앞에(왼쪽) 위치시키면 해당 심볼 참조를 해석하는 데 실패한다.** 재배치 가능 오브젝트 파일과 달리, 정적 라이브러리 파일은 U에 존재하는 심볼 참조에 대응되는 심볼 정의를 가지고 있는 모듈이 가지고 있는 심볼 정의들만 D에 넣기 때문이다. **따라서 정적 라이브러리 파일들은 맨 뒤에 두는 것을 원칙으로 한다.** 그리고 정적 라이브러리 파일들끼리의 의존성도 존재한다면 마찬가지 원리로 순서를 맞춰줘야 할 것이다. 참고로, 순환적 의존성 문제를 해결하기 위해 커맨드 라인에 동일한 파일을 두 번 이상 입력하는 것도 허용된다.

EX) p.o가 libx.a의 심볼을 참조, libx.a가 liby.a의 심볼을 참조, liby.a가 libx.a의 심볼을 참조, libx.a가 p.o의 심볼을 참조

```
linux > gcc p.o libx.a liby.a libx.a
```

※ 마지막에 p.o을 안 써도 되는 이유 : 정적 라이브러리 파일과 달리 p.o가 정의하는 심볼들은 이미 D에 들어가 있기 때문


1st
파일 스캔
재배치 가능 오브젝트 파일 m
아카이브 파일 a
U의 심볼 참조에 연결할 심볼 정의를 가지는 모듈 m 탐색
① m을 E에 추가
② m이 정의하는 심볼들을 D에 추가
③ D를 바탕으로 m의 심볼 참조들과 U의 심볼 참조들을 해석
④ 해석에 실패한 심볼 참조들은 U에 반영
U와 D가 더 이상 변하지 않는 시점에 도달할 때까지 반복
nth
파일 스캔
재배치 가능 오브젝트 파일 m
아카이브 파일 a
U의 심볼 참조에 연결할 심볼 정의를 가지는 모듈 m 탐색
① m을 E에 추가
② m이 정의하는 심볼들을 D에 추가
③ D를 바탕으로 m의 심볼 참조들과 U의 심볼 참조들을 해석
④ 해석에 실패한 심볼 참조들은 U에 반영
U와 D가 더 이상 변하지 않는 시점에 도달할 때까지 반복
U is empty?
N
Error !
Y
E의 모듈들을 대상으로
재배치 (Relocation)

7-7. 심볼 해석 보충 설명 (참고)

지금부터는 위에서 설명한 심볼 해석 전체 과정을 조금 더 구체적으로 분석해 보자. 여기서 설명하는 것은 서적에 나와 있지 않지만 필자가 공부하면서 나름대로 유추해본 내용들이다. 따라서 다소 부정확한 내용이 있을 수 있다는 점 참고해주기 바란다.

우선, 지금까지 이야기했던 '심볼 정의'와 '심볼 참조'라는 것들은 재배치 가능 오브젝트 파일 안에 어떤 방식으로 저장이 될까? 링커는 재배치 가능 오브젝트 파일의 정보만으로도 무엇이 심볼 참조이고 무엇이 심볼 정의인지 명확히 알아야 이를 바탕으로 심볼 해석을 진행할 수 있을 것이다. 예상컨대 **링커의 관점에서 심볼 정의(Symbol Definition)는 .symtab 섹션의 심볼 테이블에 존재하는 심볼 테이블 엔트리를 가리키며, 심볼 참조(Symbol Resolution)는 .rel.data 섹션과 .rel.text 섹션에 존재하는 재배치 엔트리(Relocation Entry)를 가리키는 것** 같다.

재배치 엔트리는 특정 심볼 참조의 정보를 저장하며, 해당 심볼 참조가 가리키는 심볼 정의의 최종적인 가상 주소를 확정할 수 없는 경우에만 만들어진다. 컴파일 단계에서 최종적인 가상 주소가 어떻게 결정될지 확정할 수 있는 심볼 참조는 링커가 처리할 필요가 없으므로 굳이 재배치 엔트리를 만들지 않는다. 그리고 **각 재배치 엔트리는 해당 심볼 참조가 어떤 심볼 정의를 가리키는지를 나타내는*symbol* 필드를 가지고 있으며**, 여기에는 가리키는 심볼 정의에 대응되는 심볼 테이블 엔트리의 심볼 테이블 내 인덱스가 저장된다. 결국, **심볼 해석의 목표는 모든 재배치 엔트리들의 *symbol* 필드 값을 올바르게 채워주는 것**이 아닐까 하고 유추해 볼 수 있다.이 작업이 끝나고 나면 모든 심볼 참조는 정확히 하나의 심볼 정의만을 가리키게 되므로 이제 재배치(Relocation)만 수행해주면 링킹은 끝이 나게 된다.

이를 바탕으로 위의 심볼 해석 전체 과정에서 ③에 해당하는 심볼 참조 해석 방법도 조금만 더 구체적으로 알아보자. 현재 모듈*m*을 E에 추가했고, 모듈 *m*이 정의하는 심볼 정의들도 D에 추가한 상태라고 가정하자. 이때 링커가 해석을 시도할 심볼 참조들은 U에 들어가 있는 상태이다. 그러면 U에 존재하는 각각의 심볼 참조들에 대해 연결할 심볼 정의는 다음과 같이 탐색할 것이다. 먼저 해당 심볼 참조에 대응되는 심볼 테이블 엔트리를 현재 모듈의 심볼 테이블에서 찾고(D의 심볼 정의들 중 현재 모듈에 해당하는 것들을 탐색), 그것의 *binding* 필드를 확인하는 것이다. 만약 지역 심볼이라면 (*binding*== Local) 해당 심볼 정의에 바로 연결하고, 전역 심볼이라면 (*binding*== Global) *section* 필드를 확인할 것이다. 만약 UNDEF가 아니라면 해당 심볼 정의에 바로 연결할 것이다. 이미 Weak 심볼과 관련한 전역 심볼 중복 문제를 처리했다고 가정할 때, 모든 전역 신볼들은 각기 고유한 이름을 가지고 있을 것이기 때문이다. 그러나 UNDEF라면 다른 모듈의 심볼 테이블들을 확인하여(D의 심볼 정의들 중 다른 모듈에 해당하는 것들을 탐색) 올바른 심볼 테이블 엔트리를 찾아 연결할 것이다.

8. Relocation★★★

8-1. 기본

심볼 해석이 완료되면, 각 심볼 참조는 정확히 하나의 심볼 정의(심볼 테이블 엔트리)에 연결되어 있다. 이제 재배치(Relocation) 작업을 수행할 준비가 된 것이다. 재배치는 다음과 같이 두 단계로 진행된다.

먼저, **섹션들과 그 안의 심볼 정의들을 재배치**한다. 같은 유형의 섹션들은 한 섹션으로 합쳐서 실행 파일에 담고, 그렇게 탄생하는 새로운 섹션들에 런타임 가상 주소를 할당한다. 그리고 이에 맞춰 각 섹션 안에 존재하는 심볼 정의들에도 알맞은 가상 주소를 부여한다. 따라서 이 단계가 끝나면 모든 함수, 전역 변수, 그리고 static 변수들의 고유한 런타임 가상 주소를 알게 된다.

다음으로, **심볼 참조들을 재배치**한다. 앞서 심볼 정의들에게 부여한 가상 주소 정보들을 활용하여, 각 섹션 안에 존재하는 심볼 참조들이 올바른 가상 주소를 가리키도록 수정해준다. 이 단계에서 필요한 정보는 재배치 가능 오브젝트 파일의.rel.text섹션과.rel.data섹션에 담기는 재배치 엔트리(Relocation Entry)들에 담긴다.

8-2. 재배치 엔트리 (Relocation Entry)

어셈블러는 가리키는 심볼 정의가 무엇인지 모르거나, 가리키는 심볼 정의의 최종적인 가상 주소를 확정할 수 없는 심볼 참조를 마주칠 때마다 재배치 엔트리(Relocation Entry)를 만들어서 재배치 가능 오브젝트 파일에 담는다. 여기에는 나중에 링커가 **해당 심볼 참조를 어떻게 수정(재배치)해줘야 하는지에 대한 정보**들이 담긴다. .text섹션에 존재하는 심볼 참조에 해당하는 재배치 엔트리는 .rel.text 섹션에 담기며, .data 섹션에 존재하는 심볼 참조에 해당하는 재배치 엔트리는 .rel.data 섹션에 담긴다. ELF 포맷의 재배치 가능 오브젝트 파일에 저장되는 재배치 엔트리의 구조를 C 언어의 구조체로 표현하면 다음과 같다.

```
typedef struct {
    long offset;     /* Offset of the Reference to Relocate */
    long type:32,    /* Relocation Type */
         symbol:32;  /* Symbol Table Dndex */
    long addend;     /* Constant Part of Relocation Expression */
} Elf64_Rela;
```

***offset* 필드**는 해당 심볼 참조의 섹션 내 오프셋을 저장한다. ***type* 필드는**해당 심볼 참조를 어떻게 수정(재배치)해줘야 하는지에 대한 정보를 저장한다. ***symbol* 필드**는 해당 심볼 참조가 가리켜야 하는 심볼 정의를 나타낸다. ***addend* 필드**는 특정 재배치 타입에 한하여 심볼 참조를 수정(재배치)해줄 때 사용해야 하는 부호 있는 상수 값을 저장한다. 여기서 *type* 필드에 의해 결정되는 재배치 타입의 경우 ELF 포맷을 기준으로 32개나 되지만, 우리는 다음과 같은 두 개의 기본적인 재배치 타입만 살펴볼 것이다.

재배치 타입 (type 필드)	의미
R_X86_64_PC32	32비트 PC-relative 주소 방식을 사용하여 재배치(주소를 계산)해야 하는 심볼 참조
R_X86_64_32	32비트 절대 주소 방식을 사용하여 재배치(주소를 계산)해야 하는 심볼 참조

※ 위 두 개의 재배치 타입은 코드와 데이터의 총 사이즈가 2GB보다 작아서 32비트 PC-relative 주소 방식으로 런타임에 접근이 가능한 프로그램의 경우에만 사용할 수 있다. 만약 사이즈가 2GB보다 큰 프로그램이라면 별도의 컴파일 옵션을 줘서 링킹 해야 한다.

8-3. 심볼 참조 재배치 알고리즘

다음 Pseudo 코드는 **심볼 참조를 재배치하는 알고리즘**을 나타낸다. 각 섹션s는 링커의 메모리 상에서 바이트 배열로 표현되어 있고, 각 재배치 엔트리 r은 위에서 보여주었던 Elf64_Rela 타입의 구조체로 표현되어 있다고 가정하자. 또한 아래 알고리즘을 수행할 때 링커는 이미 각 섹션 s의 런타임 가상 주소 (ADDR(s))와 각 심볼 정의의 런타임 가상 주소 π(ADDR(r.symbol))를 계산한 상태라고 가정하자.

```
for each section s {
    for each relocation entry r {
        /* Pointer to reference to be relocated */
        refptr = s + r.offset;

        /* Relocate a PC-relative reference */
        if (r.type == R_X_86_64_PC32) {
            refaddr = ADDR(s) + r.offset;
            *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr;
        }

        /* Relocate an absolute reference */
        if (r.type == R_X_86_64_32)
            *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
    }
}
```

각 섹션 s의 각 재배치 엔트리 r에 대하여, 다음과 같은 동작을 수행한다. 먼저, 수정되어야 하는 심볼 참조가 s배열 내에서 어디에 위치하는지 계산한다. 이는 곧 재배치 작업에 의해 수정될 부분을 나타낸다. 이후 재배치 엔트리의 type필드를 통해 해당 심볼 참조를 어떻게 수정(재배치)해줘야 하는지 파악한다. 각 경우에 대해 심볼 참조를 수정할 값을 계산하는 방법은 다음과 같다.

① 32비트 PC-relative 주소 방식

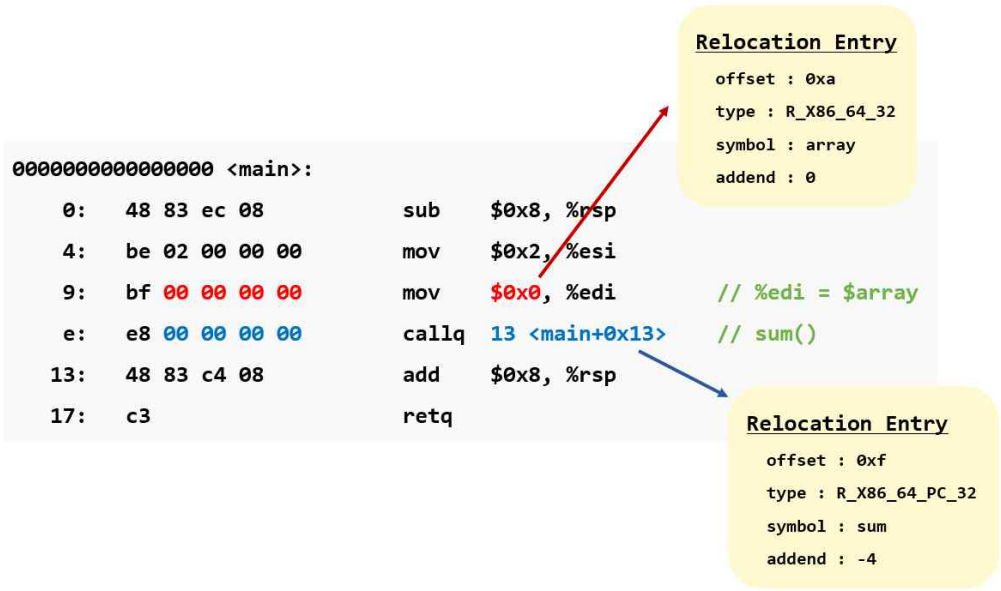
⇒ [심볼 정의의 가상 주소] + [재배치 엔트리의 addend 필드 값] - [심볼 참조의 가상 주소]

② 32비트 절대 주소 방식

⇒ [심볼 정의의 가상 주소] + [재배치 엔트리의 addend 필드 값]

8-4. 재배치 예시

다음은 [Figure 1]의 예시 코드가 링커에 의해 재배치되기 직전의 모습을 보여준다.



빨간색으로 표시된 부분은 함수의 인자로 배열 *array*를 전달하기 위해 %edi에 배열 *array*의 주소를 집어넣는 명령어에 존재하는 심볼 참조이다.재배치 엔트리의 정보에 의하면 재배치 타입이 32비트 절대 주소 방식이기 때문에, 빨간색으로 표시된 4바이트(= 32비트) 부분에는 배열 *array*의 절대 주소가 들어가야 한다는 것을 유추할 수 있다. 그 값을 계산하는 과정은 다음과 같다. 참고로 계산된 값을 거꾸로 쓰는 이유는 X86-64의 바이트 오더링이 Little Endian 방식이기 때문이다. 바이트 오더링은 값을 메모리에 쓰는 방식을 말하는 것으로, MSB가 주소가 작은 쪽에 위치하면 Big Endian 방식, 큰 쪽에 위치하면 Little Endian 방식이다.

① 가정

ADDR(r.symbol) = ADDR(array) = **0x601018**

② 계산 과정

*refptr = ADDR(r.symbol) + r.addend = 0x601018 + 0 = **0x601018**

③ 심볼 참조 재배치 결과

```
4004d9 :    bf 18 10 60 00    mov  $0x601018, %edi
```

파란색으로 표시된 부분은 *sum* 함수를 호출하는 명령어에 존재하는 심볼 참조이다. 재배치 엔트리 정보에 의하면 재배치 타입이 32비트 PC 주소 방식이기 때문에, 파란색으로 표시된 4바이트(= 32비트) 부분에는 호출하고자 하는 함수(*sum*)와의 상대적인 주소가 들어가야 한다는 것을 유추할 수 있다. 그 값을 계산하는 과정은 다음과 같다. 참고로 *callq* 명령어를 실행하는 순간에는 (CPU 내부적으로 명령어를 처리하는 절차적 특성에 의해) PC 값이 이미 다음 명령어(add)의 주소로 증가해 있다. 따라서 파란색 부분의 값은 (*sum* 함수의 주소 –*add* 함수의 주소)가 되는 것이 맞다.

① **가정**

ADDR(s) = ADDR(.text) = **0x4004d0**

ADDR(r.symbol) = ADDR(sum) = **0x4004e8**

② **계산 과정**

refaddr = ADDR(s) + r.offset = 0x4004d0 + 0xf = **0x4004df**

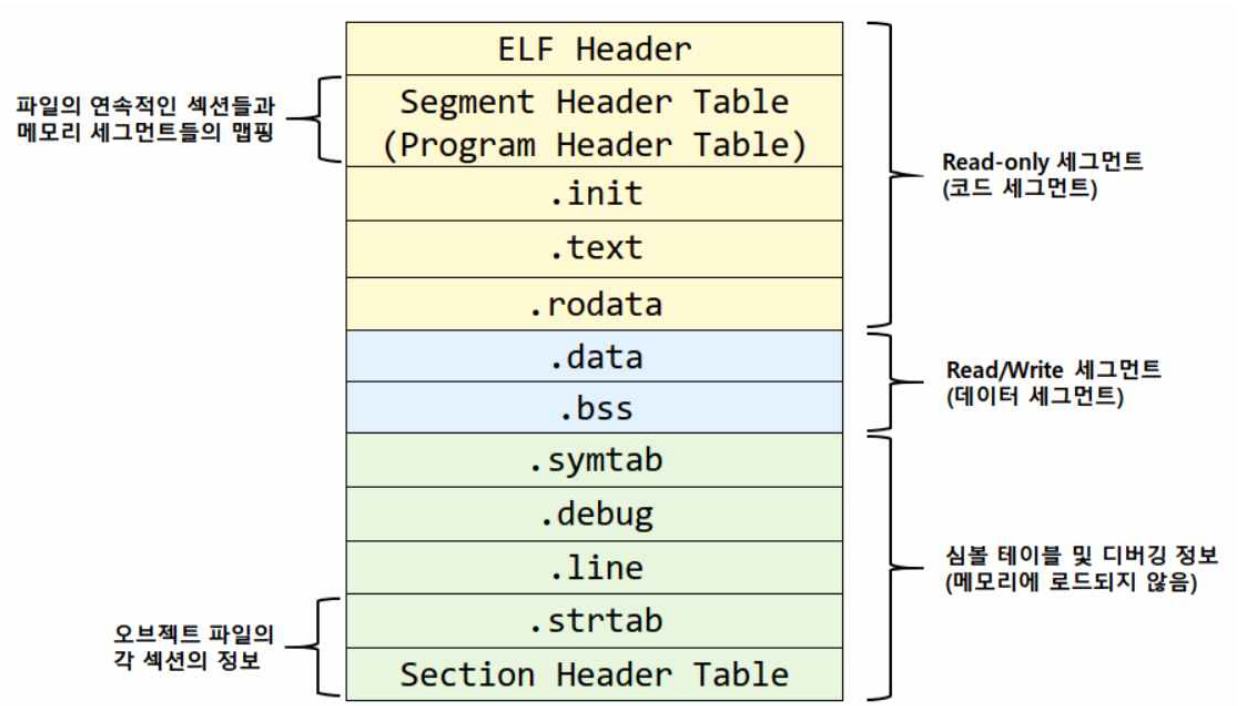
*refptr = ADDR(r.symbol) + r.addend + refaddr = 0x4004e8 + (-4) - 0x4004df = **0x5**

③ **심볼 참조 재배치 결과**

```
4004de :    e8 05 00 00 00    callq  4004e8 <sum>
```

9. Executable Object Files

ASCII 텍스트 파일들의 집합으로 시작한 C 프로그램은 컴파일러, 어셈블러, 그리고 링커를 거쳐서 하나의 완전한 실행 파일로 변환된다. 실행 파일은 메모리에 로드되어 실행되기 위한 정보들을 담은 하나의 이진 파일이다. 다음 그림은 **ELF 포맷을 따르는 실행 파일의 구조**를 나타낸다.



전반적으로 재배치 가능 오브젝트 파일과 유사한 구조를 가지고 있음을 볼 수 있다. **ELF 헤더**는 해당 오브젝트 파일의 전반적인 포맷 정보를 저장하며, 프로그램을 실행할 때 실행해야 하는 첫 번째 명령어의 주소인 Entry Point도 이곳에 저장된다. **.text, .rodata, .data 섹션**은 재배치 가능 오브젝트 파일과 거의 유사하다. 다만 이러한 섹션들이 링커에 의해 최종적인 런타임 가상 주소로 재배치가 이뤄졌다는 것만 다르다. **.init 섹션**은_init이라는 이름의 작은 함수를 하나 정의하는데, 이는 프로그램의 초기화 코드에 의해 호출되는 함수이다. 한편 실행 파일은 Fully Linked, 즉 이미 재배치가 수행된 완전한 실행 파일이기 때문에 **.rel.text 섹션**과 **.rel.data 섹션**은 존재하지 않는다.

ELF 실행 파일은 메모리에 로드되기 쉽도록 디자인된다. 즉, 실행 파일 내의 연속적인 바이트 청크들이 연속적인 메모리 세그먼트에 맵핑이 되도록 하는 것이다. 이러한 맵핑 정보는 **프로그램 헤더 테이블(세그먼트 헤더 테이블)**에 저장된다. 다음은 [Figure 1] 예시 코드에 해당하는 프로그램 prog의 프로그램 헤더 테이블 일부이다.

```
Read-only Code Segment
LOAD off    0x0000000000000000 vaddr 0x0000000000400000 paddr 0x0000000000400000 align 2**21
     filesz 0x000000000000069c memsz 0x000000000000069c flags r-x
Read/Write Data Segment
LOAD off    0x0000000000000df8 vaddr 0x0000000000600df8 paddr 0x0000000000600df8 align 2**21
     filesz 0x0000000000000228 memsz 0x0000000000000230 flags rw-
```

코드 세그먼트는 읽기 및 실행 권한이 부여되고, 메모리 상에서 가상 주소 0x400000에서 시작하며, 메모리에서 차지하는 총사이즈는 0x69c 바이트이며, 실행 파일의 0x000 ~ 0x69c 바이트 부분으로 초기화된다는 것을 알 수 있다. ELF 헤더부터 시작하여 .rodata 섹션까지가 이 세그먼트에 해당한다.

데이터 세그먼트는 읽기 및 쓰기 권한이 부여되고, 메모리 상에서 가상 주소 0x600df8에서 시작하며, 메모리에서 차지하는 총사이즈는 0x230 바이트이며, 실행 파일의 0xdf8 ~ 0x228 바이트 부분으로 초기화된다는 것을 알 수 있다. 나머지 8바이트는 .bss 섹션에 해당하는 8바이트이며, 런타임 시에 모두 0으로 초기화된다.

10. Loading Executable Object Files

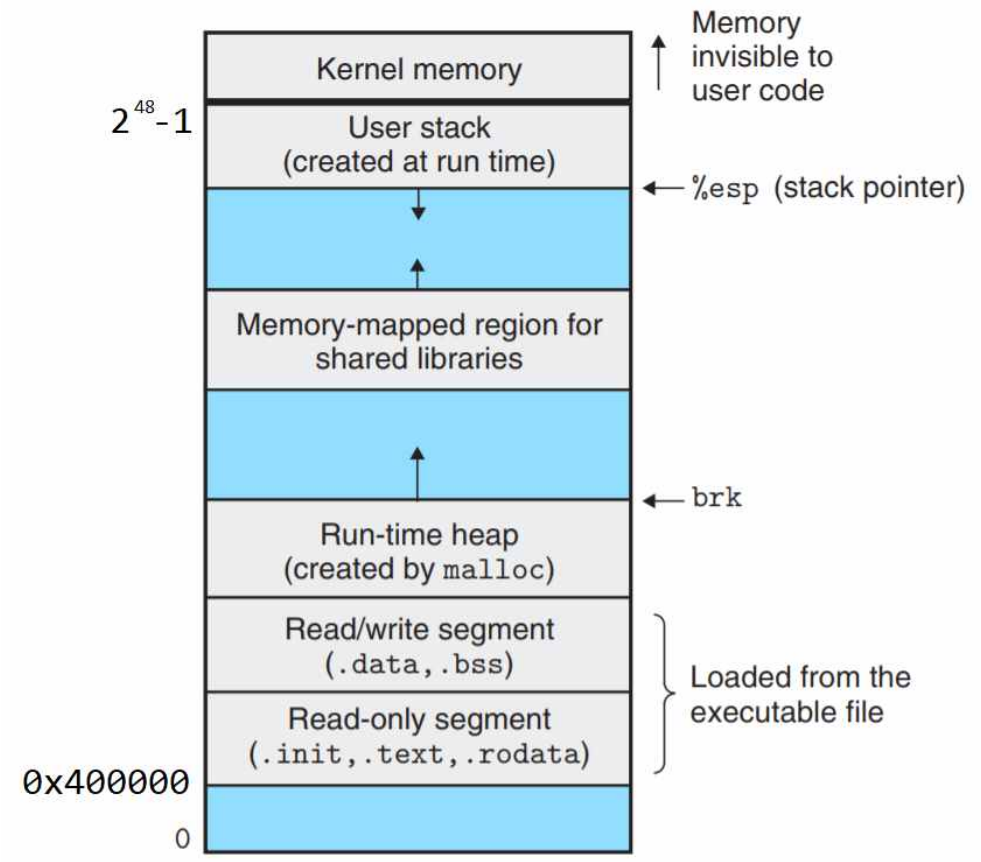
10-1. 로딩 (Loading)

```
linux > ./prog
```

리눅스 쉘에 위와 같이 실행 파일 이름을 입력하면 해당 프로그램을 실행할 수 있다. prog는 시스템 내장 쉘 커맨드가 아니기 때문에, 쉘은 그것이 실행 파일의 이름이라 판단한다. 그리고 **로더(Loader)**라고 불리는 (메모리에 언제나 상주하는) OS의 코드를 실행함으로써 해당 프로그램의 실행을 개시한다. 리눅스 프로그램은 execve라는 함수를 호출함으로써 로더를 실행하도록 되어 있다. 로더는 **디스크에 있는 실행 파일로부터 프로그램의 코드와 데이터를 메모리에 복사**하고, **Entry Point에 해당하는 첫 번째 명령어의 주소로 점프함으로써 해당 프로그램을 실행**한다. 이러한 과정을 **로딩(Loading)**이라고 부른다.

10-2. x86-64 리눅스 프로그램 런타임 이미지

x86-64 리눅스에서 실행되는 모든 프로그램은 다음 그림과 같은 런타임 메모리 이미지를 가진다.



코드 세그먼트는 0x400000에서 시작하며, 그 위에는 **데이터 세그먼트**가 등장한다. 그리고 데이터 세그먼트 위에는 등장하는 영역은**런타임 힙(Heap)**으로, malloc 라이브러리의 함수들을 호출할 때마다 주소가 큰 방향으로 확장된다. 그리고 그 위는 **공유 라이브러리(Shared Libraries)**들을 위해 예약해둔 영역이다. **유저 스택(User Stack)**영역은 유저가 접근할 수 있는 가장 큰 주소(2^48-1)부터 시작하여 주소가 작은 방향으로 확장된다. 2^48부터 시작하는 영역은 **커널의 코드와 데이터**를 위해 예약되어 있으며, 언제나 메모리에 상주하는 **운영체제**에 해당한다.

10-3. 로더의 역할

로더는 실행되면 위 그림과 유사한 메모리 이미지를 만들어 낸다. 그리고 실행 파일의 프로그램 헤더 테이블에 적힌 정보를 바탕으로 실행 파일의 연속적인 바이트 청크들을 코드 세그먼트와 데이터 세그먼트에 복사한다. 다음으로, 로더는 _start함수(시스템 오브젝트 파일인 crt1.o에서 정의됨)의 시작 주소에 해당하는 Entry Point로 점프함으로써 해당 프로그램의 실행을 개시한다. 그리고 _start함수는 *system startup function*인 __libc_start_main 함수를 호출한다. 이는 실행 환경을 초기화하고, 유저 프로그램의 main 함수를 호출하며, 그것의 반환 값을 처리하고, 필요한 경우에는 제어를 커널로 옮기는 역할을 수행한다.