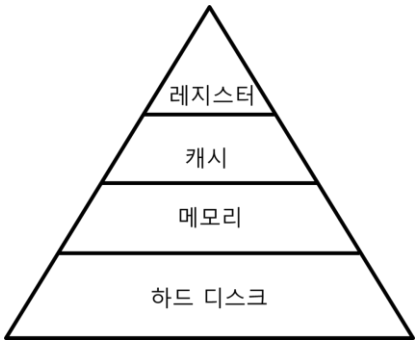
메모리 계층 구조

CPU가 메모리에 더 빨리 접근할 수 있도록 메모리를 나누어 놓은 것



[속도] 레지스터 속도가 가장 빠르다
디스크 속도가 가장 느리다
레지스터 > 캐시 > 메모리 > 디스크

[용량] 디스크 용량이 가장 크다
레지스터 용량이 가장 작다
디스크 > 메모리 > 캐시 > 레지스터

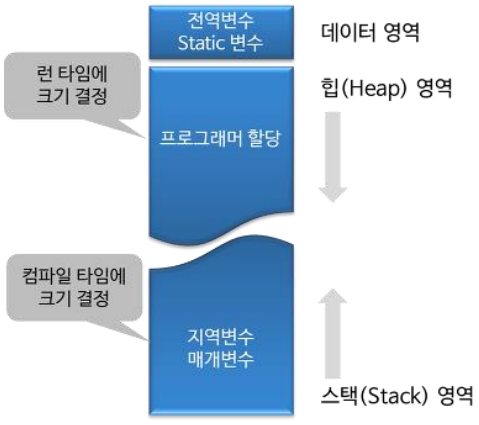
1. 왜 이런 차이가 발생할까?
레지스터와 캐시는 CPU 내부에 있어서 CPU가 빠르게 접근 가능하고
메모리(RAM)는 CPU 외부에 있어서 레지스터와 캐시보다 더 느리게 접근
하드 디스크는 CPU가 직접 접근 할 수 조차 없다
(CPU가 하드 디스크에 접근하기 위해서는 하드 디스크의 데이터를 메모리로 이동시키고, 메모리에서 접근해야 한다.
아주 느린 접근)

CPU내부에 메모리가 필요한 이유는 ALU의 고속 연산을 돕기 위함
레지스터는 연산의 결과, 함수의 반환 값을 저장함

2. 메모리 계층 구조의 필요성

- 디코딩 속도 (CPU는 작은 메모리에 더 빨리 접근할 수 있다)
- 자주 쓰이는 데이터는 전체 데이터 양에 비해 작은 양이다
(즉, 캐시는 메모리보다 작아도 되고 메모리는 디스크 보다 작아도 된다)
- 경제성(상층에 속할수록 비싸다)

가상 메모리 영역의 4가지 구조



프로그램이 실행될 때마다 운영체제는 프로그램 실행 시 필요
(지역변수, 전역변수 선언)한 메모리공간을 할당한다
할당되는 메모리 공간은 크게
코드, 스택(Stack), 힙(Heap), 데이터(Data)영역으로 나뉘어진다.

코드(Code)영역
실행할 프로그램의 코드, 함수를 저장하는 공간
CPU는 코드영역에 저장된 명령문을 하나씩 가져가서 실행한다

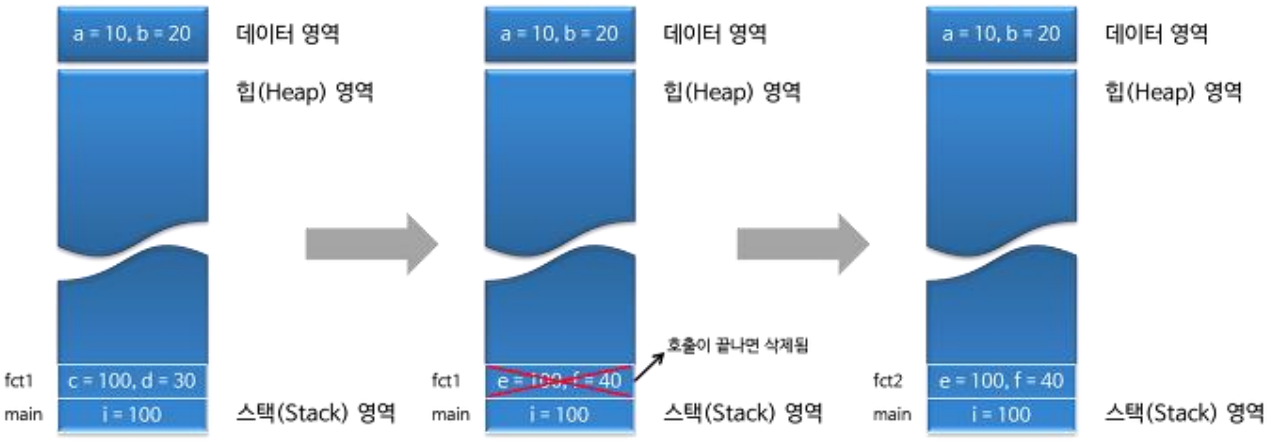
데이터(Data)영역
프로그램 시작부터 종료까지 유지해야 할 데이터 저장!
전역 변수와 static 변수가 할당되는 영역

프로그램의 시작과 동시에 할당되고,
프로그램이 종료되어야 메모리에서 소멸됨

스택(Stack) 영역
잠깐 사용하고 삭제할 데이터를 저장!
함수 호출 시 생성되는 지역 변수와 매개 변수가 저장되는 영역
함수 호출이 완료되면 사라짐
함수호출 위치를 저장한다

```
#include <stdio.h>
void fct1(int);
void fct2(int);
 int a = 10; // 데이터 영역에 할당 int b = 20; // 데이터 영역에 할당
int main() {
int i = 100; // 지역변수 i가 스택 영역에 할당
fct1(i);
fct2(i);
 return 0; }

void fct1(int c) {
int d = 30; // 매개변수 c와 지역변수 d가 스택영역에 할당
}
void fct2(int e) {
 int f = 40; // 매개변수 e와 지역변수 f가 스택영역에 할당
}
```



힙(Heap) 영역
필요에 의해 동적으로 메모리를 할당 할 때 사용
사용자의 요구에 맞게 메모리를 할당해 주기 위해서는(런타임에 메모리 크기를 결정하고 싶을 때) 메모리 동적 할당을 통해 힙 영역에 메모리를 할당해야 한다.

x86 어셈블리(아키텍처)

x86 아키텍쳐는 8개의 범용 레지스터 8개, 6개의 세그먼트 레지스터, 1개의 플래그 레지스터, 마지막으로 명령어 포인터를 갖는다.

x86 범용 레지스터

AX (accumulator) : 산술 연산에 사용(함수의 return값을 저장)
CX (counter): 시프트/회전 연산과 루프에서 사용(무언가 반복하고자 할 때 사용)
SP (stack Pointer). 스택의 최상단을 가리키는 포인터로 사용.
BP (stack Base Pointer). 스택의 베이스(기준점)를 가리키는 포인터로 사용.

명령어 포인터

IP (instruction pointer) : 다음에 실행할 명령어의 주소

[참고]
16비트 모드 : 위와 같다 AX
32비트 모드 : 앞에 'E'(*extended*)를 붙여 표시한다. EAX
64비트 모드: 앞에 'R'을 붙인다. RAX

STACK

자료구조의 하나
자료를 효율적으로 이용할 수 있도록 컴퓨터에 저장하는 방법을 **자료구조**(Data Structure)라고 한다

STACK은 한 쪽 끝에서만 자료를 넣거나(push) 뺄(pop)수 있는 선형 구조로 나중에 넣은 값이 먼저 나온다
즉, LIFO - Last In First Out 구조

```
mov ax, 006Ah   // ax의 값을 스택의 꼭대기에 push해라
mov bx, F79Ah   // bx에 대해 같은 작업 반복
                // 스택은  $006A 와 $F79A의 값을 갖는다
```

스택은 아래로 자란다

값이 쌓이면 스택은 - 의 주소를 가지게 된다
반대로 값이 빠지면 +

Stack(스택)을 제외하고는 나머지는 전부
정상적인 절차로 쌓이게 된다
쌓이면 + 빠지면 -

스택! 어디에 쓰일까?

레지스터가 부족하면 메모리에 값을 잠깐 저장했다가 필요하면 다시 레지스터로 불러서 쓴다
(엄밀하게는 함수 호출을 위한 용도)

[x86] 함수의 입력을 스택으로 전달한다.

[ARM] 함수 입력을 4 개까진 레지스터로 처리, 4 개가 넘어갈 경우 스택에 집어넣는다.
(그러므로 성능을 높이고자 한다면 ARM 에서는 함수 입력을 4 개 이하로 만드는 것이 좋다)

로드 스토어 아키텍쳐

메모리에 있는 데이터를 처리하기 위해 이 데이터는 메모리로부터 레지스터에 옮겨져서 내부 프로세서에 의해 처리되고, 이것이 다시 메모리에 쓰여진다.

우선 모든 프로세서는 레지스터 에서 레지스터로 연산이 가능하다.
x86 은 메모리 에서 메모리로 연산이 가능하다.
하지만 ARM 은 로드/스토어 아키텍처라고 해서 메모리 에서 메모리로 연산이 불가능하다.
그래서 ARM 은 먼저 메모리에서 레지스터로 값을 옮기고
다시 이 레지스터 값을 메모리로 옮기는 작업을 한다
로드하고 스토어하는 방식이라고 해서 로드/스토어 아키텍처라고함

어셈블리어 명령어 정리

데이터 이동: mov, lea
논리 연산: add, sub, inc, dec
흐름제어: cmp,jmp
프로시져: call, ret
스택조작: push, pop
인터럽트: int

mov SOURCE, DESTINATION
: SOURCE 위치에 들어있는 데이터를 복사하여 DESTINATION 위치에 저장
(모든 연산은 레지스터에 저장된 뒤 이루어짐)
메모리와 레지스터사이의 데이터 이동
레지스터와 레지스터 사이의 데이터 이동
값을 메모리나 레지스터에 대입할 때 사용

add opr1, opr2 레지스터나 메모리의 값을 덧셈할때 쓰임.
: opr1 + opr2 = opr2 (operand)

sub opr1, opr2 레지스터나 메모리의 값을 뺏셈할때 쓰임.
: opr1 - opr2 = opr1

call target 프로시저 호출시 쓰임
: call명령 다음에 오는 명령의 주소를 스택에 push하고 target으로 이동
즉, 다른 함수로 제어를 옮긴다 (=EIP를 변경한다)

ret
: (return) 호출한 함수 즉, 콜한 지점으로 복귀한다
즉, sp값(call당시 push되었던 주소)을 꺼내서(pop) EIP레지스터에 할당한다

pop DESTINATION sp레지스터를 조작하여 스택에 데이터를 저장한다
: 스택 맨 위에서 하나의 워드를 DESTINATION에 로드한다
 그리고 스택 포인터는 바로 전의 데이터를 가리킨다

push DESTINATION sp레지스터를 조작하여 스택에서 데이터를 꺼낸다
: 메모리 상 설정된 스택 공간에 데이터를 저장한다
스택의 가장 윗부분에 저장하며 스택 포인터(sp)는 워드 크기만큼 감소한다

프로그램 디버깅 및 어셈블리 분석

Dump of assembler code for function **main:**

push %rbp	기준점 rbp를 스택에 푸시하여 보관한다
mov %rsp,%rbp	rsp를 rbp와 같게 하여 새로운 기준점을 만든다
	여기까지가 스택 프레임 생성과정
sub $0x10,%rsp	rsp에서 16진수 10을 뺀다
movl $0x3,-0x8(%rbp)	rbp에서 push하여 3을 넣는다
mov -0x8(%rbp),%eax	방금 push하여 넣은 값 3을 eax에 복사한다
mov %eax,%edi	eax의 3을 edi에 복사한다
callq **0x400526** <myfunc>	<myfunc>을 호출한다
	함수의 주소는 0x400526
	callq다음 명령의 주소(복귀주소)인 **40054e**를 rip에 저장한다
	call= 복귀주소를 push + 함수호출 주소로 jmp

0x000000000**40054e** <+25>:	mov %eax,-0x4(%rbp)
	eax의 값 6을 rbp에서 4바이트 푸시하여 넣는다
mov -0x4(%rbp),%eax	방금 push하여 넣은 값6을 eax에 복사한다
mov %eax,%esi	eax의 값 6을 esi에 복사한다
mov $0x4005f4,%edi	$0x4005f4**(?)**의 값을 edi에 복사한다
mov $0x0,%eax	eax를 0으로 초기화한다
callq 0x400400 <printf@plt>	0x400400 위치에 있는 <printf@plt>함수를 호출한다
	callq다음 명령의 주소(복귀주소)인 **400565**를 rip에 저장한다
0x000000000**400565** <+48>:	mov $0x0,%eax
	eax를 0으로 초기화한다

에필로그 부분! 스택의 프레임을 해제한다. 컴파일시 컴파일러가 자동으로 함수의 마지막부분에 삽입한다

leaveq	mov esp, ebp
	pop ebp
retq	pop eip
	jmp eip

Dump of assembler code for function **myfunc:**

=> 0x0000000000**400526** <+0>: push %rbp	기준점 rbp를 스택에 푸시하여 보관한다
mov %rsp,%rbp	rsp를 rbp와 같게 하여 새로운 기준점을 만든다 스택 프레임 생성 부분!
mov %edi,-0x4(%rbp)	edi의 값 3을 rbp에서 4바이트 push하여 넣는다
mov -0x4(%rbp),%eax	방금 push하여 넣은 값 3을 eax에 복사한다
add $0x3,%eax	eax+3=eax 즉, eax는 6이된다
pop %rbp	에필로그 부분!
retq	

2진수-16진수 변환

10진수 144를 2진수로 변환

이진수 각자리의 가중치 128 64 32 16 8 4 2 1을 생각한다!

144 = 128 + 16 에서

1001 0000이다

2진수 1001 0000을 16진수로 변환

4자리씩 끊어준다!

즉, 1001 =9, 0000=0 에서

0x90이다

16진수 0x48932110을 2 진수로 변환

16진수 1자리가 2진수 4자리라는 것을 기억하자!

8421	8421	8421	8421	8421	8421	8421	8421
0100	1000	1001	0011	0010	0001	0001	0000

0100 1000 1001 0011 0010 0001 0001 0000(2)

메모리 용량을 나타내는 단위

기억장치에서 용량을 나타내는 단위는 바이트(byte) 혹은 워드(word, 단어)

(8비트 = 1바이트)

워드의 길이

CPU가 실행할 명령어의 길이 또는 내부연산에서 한번에 처리할 수 있는 데이터의 비트 수

일반적으로 8,16,32,64바이트

기억장치 용량

각 기억장치 위에는 고유 주소가 할당된다

대부분 주소지정단위는 바이트
주소비트의 수를 n개라 할 때, 주소가 기억되는 기억장소는M= 2^n개
바이트 단위로 주소 지정 시, 기억장치 용량은 M바이트

2^10 byte = 1 KB
2^10 KB = 1 MB
2^10 MB = 1 GB

32 bit = 2^32 = 4GB = 2^10 x 2^10 x 2^10 x 2^2

16진수 8자리가 표현할 수 있는 범위는 0x0000 0000 ~ 0xffff ffff 이다
4비트가 16진수 한자리를 표현한다
따라서, 32비트는 16진수 32/4= 8자리
64비트는 16진수 16자리
8비트는 16진수 2자리
8비트 시스템에서 표현할 수 있는 범위는 0x00 ~ 0xff 로 2^8=256개이다
(0x100 - 1 = 0xff)

포인터의 크기

포인터의 크기는 8 비트 시스템의 경우 1 byte, 16 비트는 2 byte, **32 비트는 4 byte, 64 비트는 8 byte**

왜 그럴까?
컴퓨터의 산술 연산이 ALU 에 의존적이기 때문이다.

ALU 의 연산은 범용 레지스터에 종속적이고
컴퓨터가 32비트라는 의미는 이들이 32 비트로 구성된 32비트 시스템이라는 것을 의미한다.

즉, CPU가 한번에 처리할 수 있는 명령어의 크기가 32비트
또는 한번에 전송(CPU와 메모리간)할 수 있는 데이터(포인터의 경우 메모리 주소)의 크기가 32비트인 시스템이라는 의미이다
(어드레스 버스의 폭이 32비트라는 의미)

포인터는 메모리 위치를 가리키기 위한 주소값를 담는 변수다.
따라서, CPU의 메모리 접근을 위한 주소가 32비트 만큼 가능하기 때문에 포인터도 32비트(4바이트)가 된다.
여기서, 메모리용량은 2의 32승 즉, 4G바이트

반면, 64비트 시스템이라면 CPU의 메모리 접근을 위한 주소가 64비트(8바이트) 만큼 가능하기 때문에 포인터도 64비트(8바이트)가 된다.

여기서, 메모리용량은 2의 64승

포인터 변수 자료형이 동일한 크기를 가지는 이유
일반적인 자료형(char, int, float, double)과 다르게 포인터 변수 선언 시, 자료형들은 동일한 크기를 가진다.

```
#include <stdio.h>

int main(void) {
	printf("sizeof(int *) = %lu₩n", sizeof(int *));
	printf("sizeof(double *) = %lu₩n", sizeof(double *));
	printf("sizeof(float *) = %lu₩n", sizeof(float *));
	return 0;
}
```

이 코드를 32비트 프로그램으로 컴파일 하면 4바이트
64비트의 경우 8바이트가 나온다

각각의 자료형은 해당 자료 표현을 위해 그 범위 만큼의 크기를 가진다. char 자료형은 1 Byte의 크기를 가진다. 즉, 1 Byte = 8 Bit이므로 표현할 수 있는 범위는0~255까지 256개이다
32 Bit 프로그램 기준으로 int 자료형은 4 Byte의 크기를 가지며, 이는 총 32 Bit으로 4,294,967,296개의 경우의 수를 가진다.

포인터 자료형의 경우, char, int, double 자료형과는 다르게 모든 포인터 자료형에 대하여 동일한 크기를 가진다. 포인터 변수는 주소 값을 저장하는 자료형이므로 OS Bit에 따른 해당 메모리 주소 값을 모두 포인터 변수로 넣을 수 있는 경우의 수가 가능한 자료형이어야 한다
32 Bit프로그램의 경우 표현 및 사용 가능한 메모리는 4,294,967,296 경우의 수이므로, OS에서 총 가용할 수 있는 메모리는 4 GB(4,294,967,296 Byte)가 된다.
따라서 32 Bit에서 포인터 자료형이 모두 동일하게 4 Byte인 이유는
가용할 수 있는 모든 메모리의 경우의 수를 표현하기 위해서는 4 Byte 크기의 자료형이어야 하기 때문이다.