C언어 - HW1

임베디드스쿨1기
Lv1과정
2020. 08. 07
박성환

1. 복습(포인터 & 함수)

1. 포인터
1) 정의
- 주소를 값으로 담고 있는 변수

2) 어렵게 생각하지 않고 단순하게 생각하기

```
int a = 4;
int *ptr = &a; // 변수 a의 주소를 값으로 담고 있는 변수
int **pptr = &ptr; // ptr 포인터 변수의 주소를 값으로 담고 있는 변수
                   // &ptr => 이중포인터
                   // &&ptr => 존재x
```

2. 함수
1) 정의
- 특정 목적을 위한 기능을 하는 코드들을 묶어 하나의 명령어 처럼 사용하는 것

2) 장점
- 유지보수 및 가독성
- 재활용성

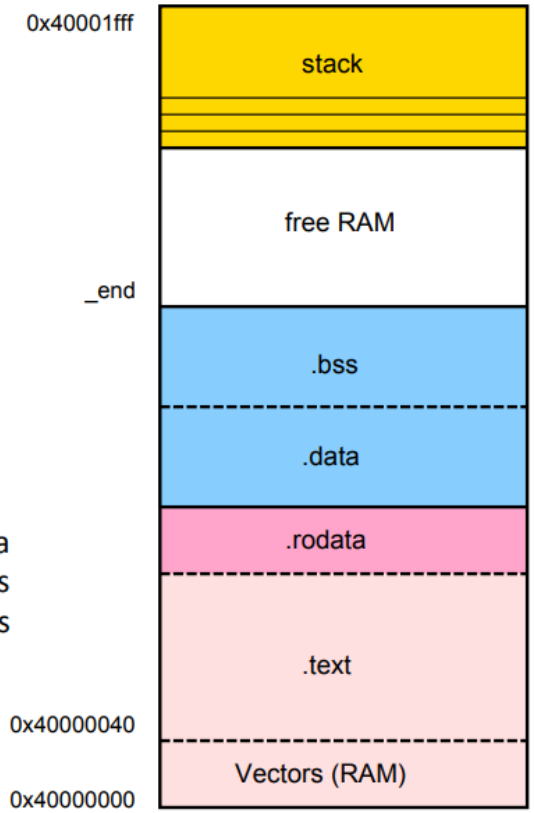
2. Keyword정리

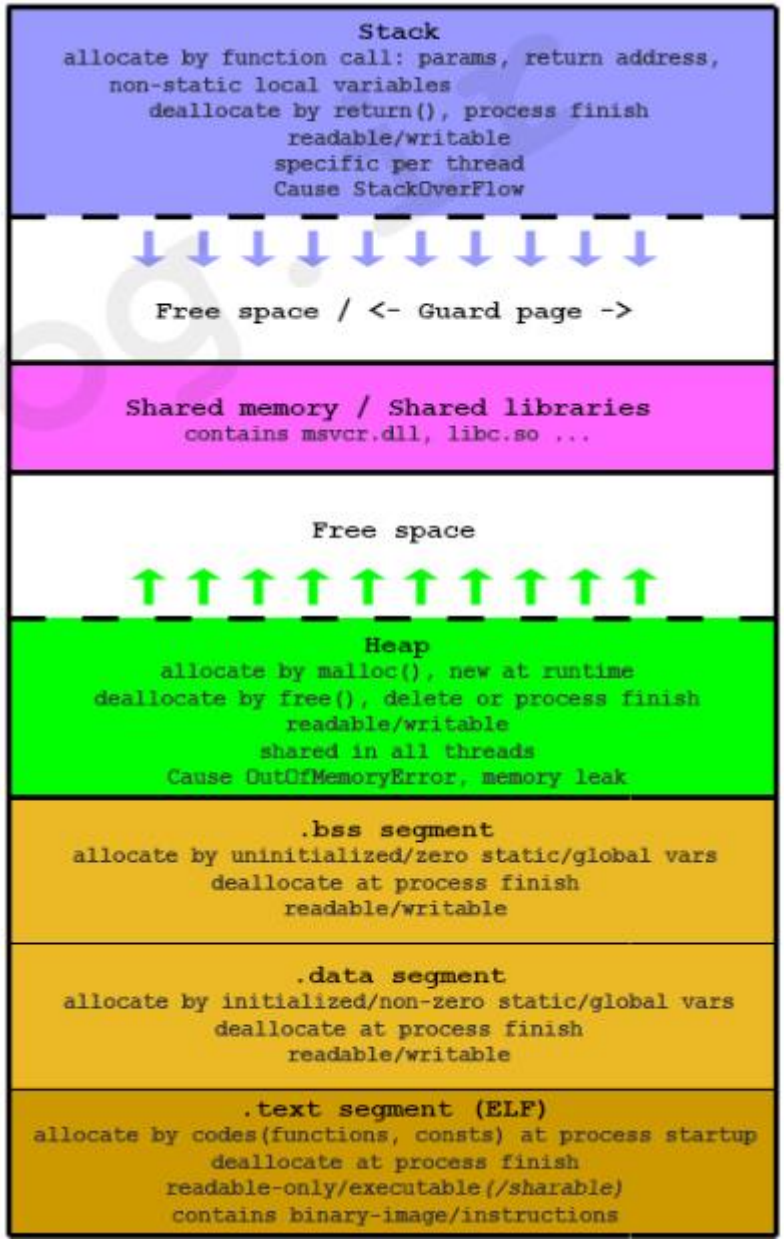
1. Data Segment

Sections:

- .text: Program code. Read only
- .rodata: constants (const modifier) and strings. Read only
- .data: Initialized global and static variables (startup value ≠ 0)
- .bss: Uninitialized global and static variables (zero value on startup)

The bootloader (bt2.exe) places the .text, .rodata and .data sections into the RAM and then orders the ARM CPU to jump to the reset vector (address 0x40000000)





2. Keyword정리

2. Big Endian vs Little Endian

1) Big Endian : 낮은 주소에 높은 바이트부터 저장하는 방식

- 메모리에 저장된 순서 그대로 읽을 수 있음, 이해하기 쉬움
- RISC CPU 계열에서 주로 이 방식으로 데이터를 저장하는 편

2) Little Endian : 낮은주소에 낮은 바이트부터 저장하는 방식

- 평소 우리가 생각하는 선형방식과 반대로 거꾸로 읽어야 함
- 대부분의 인텔 CPU 계열에서는 이 방식으로 데이터를 저장

EX) 0x12345678 있을 경우

2. Keyword정리

3) Big Endian VS Little Endian

- 단지 저장해야 할 큰 데이터를 어떻게 나누어 저장하는가에 따른 차이일 뿐,
  어느 방식이 더 우수하다고는 단정할 수 없음

- 물리적으로 데이터를 조작하거나 산술연산을 수행할 때에는 리틀 엔디안 방식이 더 효율적
  데이터의 각 바이트를 배열처럼 취급할 때에는 빅 엔디안 방식이 더 적합

- 인텔 기반의 윈도우는 리틀 엔디안 방식을 사용하지만
  네트워크를 통해 데이터를 전송할 때에는 빅 엔디안 방식이 사용됨
  따라서 인텔 기반의 시스템에서 소켓 통신을 할 때는 바이트 순서에 신경을 써서 데이터를 전달해야 함

```c
int i;
int test = 0x12345678;
char* ptr = (char*)&test; // 1 바이트만을 가리키는 포인터를 생성함.

for (i = 0; i < sizeof(int); i++)
{
    printf("%x", ptr[i]); // 1 바이트씩 순서대로 그 값을 출력함.
}
```

78563412 이면 리틀 엔디안 방식
12345678 이면 빅 엔디안 방식

감사합니다.