|  |  |  |
| --- | --- | --- |
| **3주차 1번 과제** | **제출일 : 2024.2.25** | **이 선 용** |

스택프레임 개념에서 대해 학습하고 개념 정리하기

텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

자동 생성된 설명

## 스택 프레임

\* 메모리의 스택 영역은 함수의 호출과 관계되는 지역 변수 및 매개변수가 저장되는 영역임.

\* 스택 영역은 함수의 호출과 함께 할당되며, 함수의 호출이 완료되면 소멸한다.

\* 스택 프레임이란 함수가 호출될 때, 그 함수만의 스택 영역을 구분하기 위해 생성되는 공간임. 이 공간에는 함수와 관계되는 지역 변수, 매개변수가 저장되며, 함수 호출 시 할당되며, 함수가 종료되면서 소멸된다.

함수가 호출될 때 스택에는 함수의 \*\*매개변수\*\*, 함수 호출이 끝난 뒤 돌아갈 \*\*반화 주소값\*\*, 함수 내부 \*\*지역변수\*\* 가 저장된다. 이렇게 스택 영역에 차례대로 저장되는 함수의 호출 정보를 스택 프레임 이라고 한다.

## 스택 프레임의 동작 방식

[예시 코드]

int main() {

func1();

}

void func1(){

func2();

}

void func2() {

}

0) 프로그램이 실행된다.

1) main() 함수가 실행되고, main() 함수의 스택 프레임이 스택에 저장된다.

2) func1() 함수가 실행되고, func1() 함수의 스택 프레임이 스택에 저장된다.

3) func2() 함수가 실행되고, func2() 함수의 스택 프레임이 스택에 저장된다.

4) func2() 함수의 모든 작업이 완료되어 반환되면 func2() 함수의 스택 프레임이 스택에서 제거된다.

5) func1() 함수의 모든 작업이 완료되어 반환되면 func1() 함수의 스택 프레임이 스택에서 제거된다.

6) main() 함수의 모든 작업이 완료되어 반환되면 main() 함수의 스택 프레임이 스택에서 제거된다.

7) 프로그램이 종료된다.

## 함수 프롤로그

함수 프롤로그란 함수가 호출되면 그 함수의 영역을 설정하기 위해 2개의 명령어를 사용한다.

\* push ebp

\* mov ebp, esp

ebp를 push 함으로써 caller 함수(자신을 호출한 함수)의 ebp를 스택에 저장한다. 그리고 esp를 ebp에 복사함으로써 esp를 ebp 주소로 설정한다. ebp는 함수의 기준점을 나타내므로, ebp를 기준으로 return address나 argc, argv 그리고 함수의 지역변수 등의 위치를 쉽게 알 수 있다.

1. ESP는 데이터가 들어가고 나갈 때마다 움직임.

\* PUSH(데이터 들어감) - 스택의 끝자리에 데이터를 저장시키고, ESP를 다음 칸으로 변경함.

\* POP(데이터 나감) - 스택의 끝자리 데이터를 빼내고, ESP를 앞 칸으로 변경함.

2. EBP를 통해 지금 사용중인 스택의 기준점 잡음.

3. 루틴이 새롭게 호출되면, 스택 상에 기존에 사용된 루틴의 정보(RET, SFP)를 저장함.

\* RET - 루틴이 끝나고 진행해야 하는 명령이 기어된 곳.

\* SFP - 날 호출한 상위 루틴의 EBP(스택의 기준).

#### Register

1. EIP(Extended Instruction Pointer) : CPU한테 다음 '할 일'을 알려줌

\* '할 일' 이란? => CODE address

\* CPU는 계산밖에 할 줄 모름. CPU는 EIP가 알려주는 메모리 주소에 있는 일을 가져다 실행한다. CPU가 그 일을 해내는 동안, EIP는 =1이 된다. -> 다음 할 일을 알려준다.

2. EBP : 지금 사용중인 스택의 \*\*시작 지점\*\*

3. ESP : 지금까지 사용한 스택의 \*\*마지막 지점\*\*

### 함수의 호출 단계

1. 함수가 사용할 매개 변수를 넣고 함수 시작 지점으로 점프한다. -> \*\*함수 호출\*\*

2. 함수 내에서 사용할 스택 프레임을 설정한다. -> \*\*함수 프롤로그\*\*

3. 함수의 내용을 수행한다. -> \*\*함수 본체\*\*

4. 수행을 끝내면 처음 호출한 지점으로 돌아가기 위해 스택을 복원한다. -> \*\*함수 에필로그\*\*

## 스택 프레임의 구조

\* PUSH EBP : 함수 시작(EBP를 사용하기 전에 기존의 값을 스택에 저장)

\* MOV EBP, ESP : 현재의 ESP(스택 포인터)를 EBP에 저장

\* ... : 함수 본체. 여기서 ESP가 변경되더라도 ESP가 변경되더라도 EBP가 변경되지 않으므로 안전하게 로컬 변수와 파라미터를 엑세스할 수 있음.

\* MOV ESP, EBP : ESP를 정리(함수 시작했을때의 값으로 복원 시킴)

\* POP EBP : 리턴되기 전에 저장해 놓았던 원래 EBP 값으로 복원

\* RETN : 리턴. 함수 종료

### 함수 프롤로그

\* PUSH EBP : 함수 시작(EBP를 사용하기 전에 기존의 값을 스택에 저장)

\* MOV EBP, ESP : 현재의 ESP(스택 포인터)를 EBP에 저장

함수 수행을 마치면 다시 제자리로 돌아가기 위해 EBP의 값을 스택 PUSH 하여 저장해놓는다.

EBP에 ESP를 저장해줌으로써 스택 프레임을 생성해준다.

### 함수 에필로그

\* MOV ESP, EBP : ESP를 정리(함수 시작했을때의 값으로 복원 시킴)

\* POP EBP : 리턴되기 전에 저장해 놓았던 원래 EBP 값으로 복원

\* RETN : 리턴. 함수 종료

함수 수행을 마치면 호출을 했던 지점으로 돌아가기 위해 처음 함수가 시작되었을 때의 값으로 복원시킨다.

그 다음, 프롤로그에서 push 해놓았던 값을 다시 pop 하여 EBP에 넣어준다.

## Stack Overflow

앞서 함수의 재귀 호출이 무함히 반복되면, 해당 프로그램은 스택 오버플로우에 의하여 종료됨.

만약 재귀 호출이 무한히 반복되면, 재귀 호출에 의한 스택 프레임이 계속해서 쌓이게 됨.

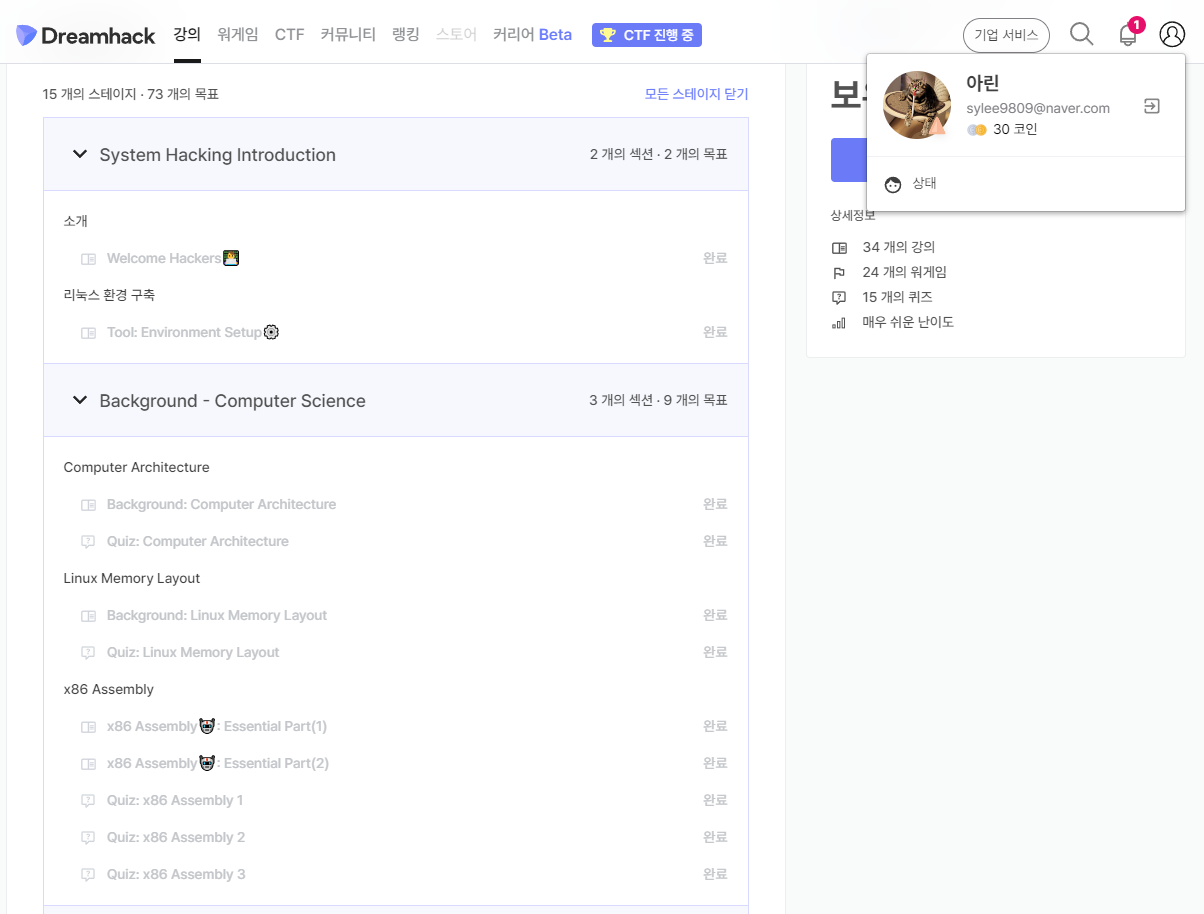
이렇게 계속 쌓이게 되면 스택의 모든 공간을 다 차지하고 난 후 더 이상의 여유 공간이 없게 되는데 이때 또 다시 스택 프레임을 저장하게 되면, 해당 데이터는 스택 영역을 넘어가서 저장된다.

이렇게 해당 스택 영역을 넘어가도 데이터가 저장될 수 있으면, 해당 프로그램은 오동작을 하게 되거나 보안상 큰 취약점을 갖게 된다.

C언어에서는 실행 중인 프로그램에서 스택 오버플로우가 발생하면, 에러를 발생시키고 강제 종료를 시키게 된다.

|  |  |  |
| --- | --- | --- |
| **3주차 2번 과제** | **제출일 : 2024.2.25** | **이 선 용** |

DreamHack의 System Hacking 로드맵의 1,2 스테이지 클리어하고 인증하기



|  |  |  |
| --- | --- | --- |
| **3주차 3번 과제** | **제출일 : 2024.2.25** | **이 선 용** |

C언어로 더블 링크드 리스트 구현하고 학습 노트에 정리하기

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

struct node {

char \*data;

struct node \*next;

struct node \*prev;

};

typedef struct node Node;

Node \*head;

Node \*tail;

int size=0;

void add\_after(Node \*pre, char \*item){

Node \*new\_node = (Node \*)malloc(sizeof(Node));

new\_node->data = malloc(strlen(item) + 1);

strcpy(new\_node->data, item);

new\_node->prev = NULL;

new\_node->next = NULL;

if(pre==NULL && head==NULL){

head=new\_node;

tail=new\_node;

}

else if(pre==NULL) {

new\_node->next = head;

head->prev = new\_node;

head = new\_node;

}

else if(pre==tail) {

new\_node->prev = tail;

tail->next = new\_node;

tail = new\_node;

}

else {

new\_node->prev=pre;

new\_node->next=pre->next;

pre->next->prev=new\_node;

pre->next=new\_node;

}

size++;

}

void insert\_at\_position(int position, char \*item){

Node \*p = head;

for(int i=1; i<position; i++){

if(p == NULL){

printf("입력한 위치가 리스트의 길이를 초과합니다.\n");

return;

}

p = p->next;

}

add\_after(p, item);

}

void remove\_node(Node \*p) {

if(head==p && tail==p) {

head=NULL;

tail=NULL;

}

else if(head==p) {

p->next->prev=NULL;

head=p->next;

}

else if(tail==p) {

p->prev->next=NULL;

tail=p->prev;

}

else{

p->prev->next=p->next;

p->next->prev=p->prev;

}

free(p->data);

free(p);

}

void add\_ordered\_list(char \*item) {

Node \*p = tail;

while(p!=NULL && strcmp(p->data, item) < 0)

p=p->prev;

add\_after(p, item);

}

void print\_list() {

Node \*p = head;

while (p != NULL) {

printf("%s ", p->data);

p = p->next;

}

printf("\n");

}

int main(void) {

int choice, position;

char input[100];

while (1) {

printf("\n1. 노드 추가\n2. 원하는 위치에 노드 추가\n3. 노드 삭제\n4. 종료\n선택: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("추가할 사람 이름을 입력하세요: ");

scanf("%s", input);

add\_ordered\_list(input);

break;

case 2:

printf("추가할 사람 이름을 입력하세요: ");

scanf("%s", input);

printf("원하는 위치를 입력하세요: ");

scanf("%d", &position);

insert\_at\_position(position, input);

break;

case 3:

printf("삭제할 사람 이름을 입력하세요: ");

scanf("%s", input);

Node \*p = head;

while (p != NULL) {

if (strcmp(p->data, input) == 0) {

remove\_node(p);

break;

}

p = p->next;

}

if (p == NULL) {

printf("해당 이름을 가진 사람이 없습니다.\n");

}

break;

case 4:

printf("프로그램을 종료합니다.\n");

printf("최종 리스트: ");

print\_list();

return 0;

default:

printf("잘못된 입력입니다.\n");

}

printf("현재 리스트: ");

print\_list();

}

return 0;

}

구조체는 하나 이상의 변수 (일반 변수, 포인터, 배열 등)를 그룹으로 묶어 새로운 자료형을 정의하는 것.

구조체에 담기는 각 변수들을 멤버라고 함.

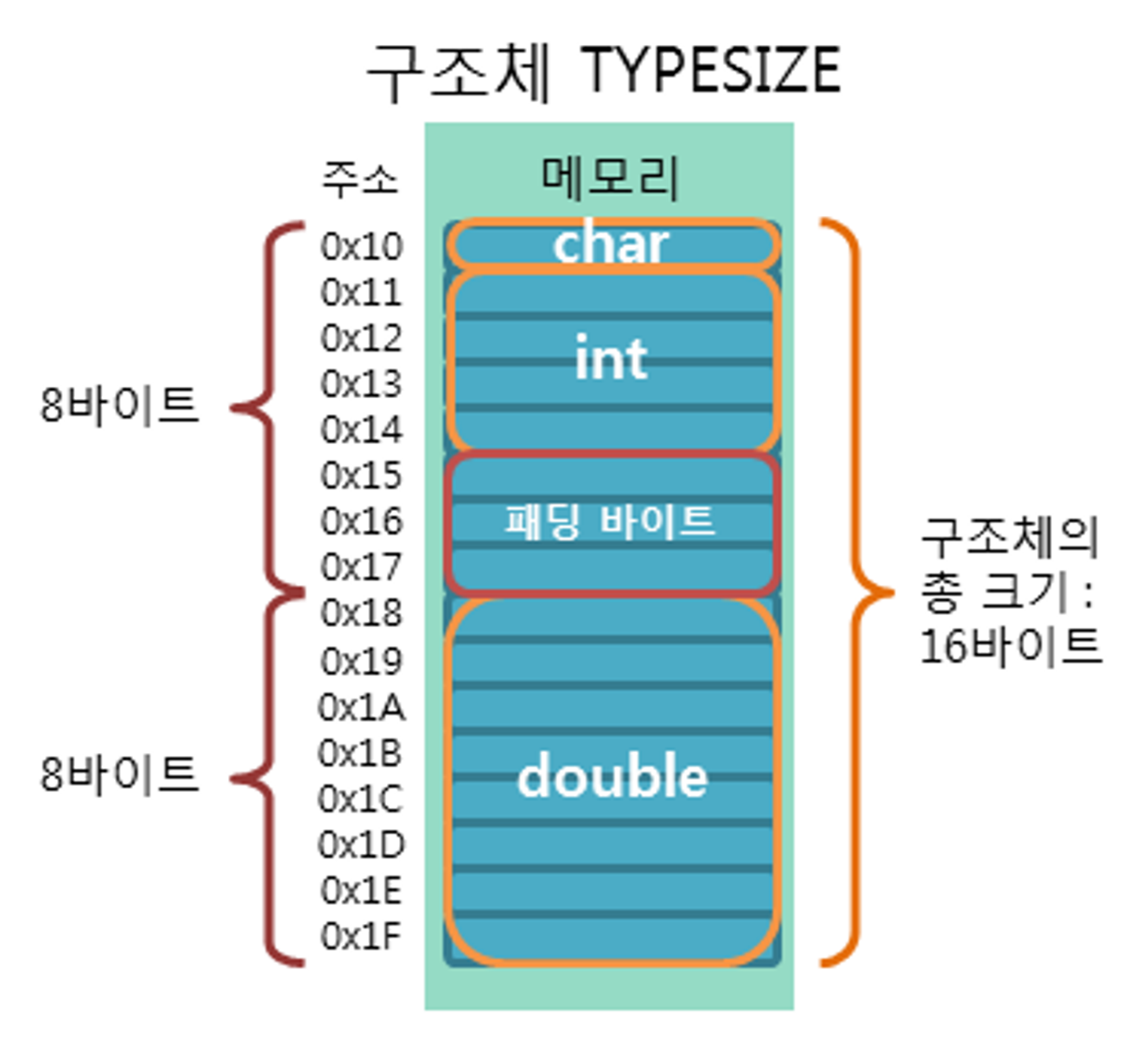
멤버들을 직접 지정해줘야지만 사용 가능함. 구조체 변수를 통으로 쓰는 건 아니고 멤버들을 멤버 접근 연산자를 이용해 하나씩 사용해야함.

* \* 연산자 . -> () [] 는 우선순위가 가장 높다.

멤버가 int 와 double일 때 12바이트의 크기를 가질 것으로 예상되는데 sizeof로 출력하면 사실 16바이트가 나온다. 이는 패딩 바이트가 들어가기 때문인데, 시스템은 cpu가 메모리에 접근할 때 접근 효율을 높이기 위해 일정한 바이트 단위로 접근한다. 멤버의 크기가 들쑥날쑥한 경우 멤버 사이에 패딩 바이트를 넣어 멤버들을 정렬 하는 것이다.

그러나 패딩 바이트의 경우 너무 많은 메모리를 잡아먹는다는 단점이 있어 심각한 문제를 발생시킬 수 있다.

순서만 바꿔도 데이터 낭비를 줄일 수 있으므로 구조체 멤버를 사용 시 순서를 적절히 조절해주면 된다.



구조체 멤버로는 다양한 자료형들을 쓸 수 있다. 멤버로 배열 선언도 가능하며 포인터도 넣을 수 있다. 구조체의 변수를 새로운 구조체의 멤버로도 사용이 가능하다.

구조체 포인터 선언

구조체 포인터는 동적 메모리 할당을 위해 사용한다. 동적 메모리 할당을 위해 malloc()을 사용해야 한다. 만약 동적 메모리 할당을 하지 않는다면 구조체 포인터 값은 0으로 초기화 된다.

Typedef는 이미 존재하는 타입에 새로운 이름을 붙일 때 사용함. 이를 사용하면 구조체를 새롭게 선언할 때 struct를 매번 사용하지 않아도 된다. 구조체 정의할 때 동시에 선언할 수도 있다.

구조체 멤버 변수 접근

.을 사용해서 접근할 수 있고, 포인터를 사용해서 구조체를 선언했다면 -> 또는 연산자 \*을 통해 접근할 수 있다. **참조 연산자는 멤버 연산자보다 우선순위가 낮으므로 반드시 괄호를 사용해야한다.**

구조체 포인터 변수로 선언하고, 구조체의 멤버가 포인터일 때 역참조 하는 방법은 다음과 같다.

* \*구조체 -> 멤버
* \*(\*구조체).멤버

구조체 정의.

각 노드는 문자열 데이터를 저장하는 data와 이전 노드를 가리키는 prev 포인터, 다음 노드를 가리키는 next 포인터가 있다.

typedef를 통해 node 구조체를 Node로 재정의한다.

전역 변수로 head는 리스트의 첫번째 노드를 가리키는 포인터, tail은 마지막 노드를 가리키는 포인터다.

size는 현재 리스트에 있는 노드의 수를 저장하는 변수다.

add\_node함수

pre 노드 뒤에 새 노드를 추가하는 함수. 새 노드를 할당하고 item 문자열을 복사하여 새 노드의 data에 저장함.

pre 노드가 리스트의 어디에 있는지에 따라 적절히 새 노드를 리스트에 연결

insert\_at\_position 함수

사용자가 입력한 위치에 새 노드를 삽입하는 함수

head부터 시작하여 position번째 노드를 찾아 add\_node 함수를 호출해서 새 노드를 삽입한다.

remove\_node 함수

노드를 리스트에서 제거하는 함수.

p가 어디 있는지에 따라 prev와 next 포인터를 재조정하여 p를 리스트에서 분리하고 p와 그 data를 메모리에서 해제

add\_ordered\_list 함수

item을 리스트에 순서대로 추가하는 함수.

리스트를 끝에서 시작하여 item이 들어갈 위치를 찾은 후 add\_node 함수를 호출하여 item을 삽입함.

print\_list 함수

모든 노드를 출력하는 함수.

head부터 next 포인터를 따라 리스트를 돌며 각 노드의 data를 출력함.

main 함수

여태 구현한 함수들을 바탕으로 실질적으로 굴러가는 코드 부분

메뉴 선택을 입력받고 해당하는 작업을 수행하고 현재 리스트를 출력함.

사용자가 종료하면 최종 리스트를 출력하고 종료하도록 구현함.

|  |  |  |
| --- | --- | --- |
| **3주차 4번 과제** | **제출일 : 2024.2.25** | **이 선 용** |

어셈블리로 Hello World 출력하기

; mac OS 에서는 구동이 안된다는 것을 확인하고 Linux 환경에서 작업했습니다. 파일 실행 시 명령어는 다음과 같이 했을때 작동을 확인했습니다.

; nasm -f elf64 -o hello.s hello.asm

; ld -o hello hello.s

; 파일 실행에 있어 편의를 위해 hello.asm으로 이름 바꿔서 실행하시는 것을 추천 드립니다.

section .data ; 데이터 섹션은 프로그램에서 사용하는 정적 데이터를 저장함.

msg db "Hello World!" ; msg에 Hello world를 저장함. db는 database가 아니라 define byte를 의미함. 각 문자를 1바이르토 정의

section .text ; 실질적인 코드가 시작되는 부분임.

global \_start ; start라는 라벨을 전역으로 선언. 프로그램의 실행 시작점을 가리킴.

\_start: ; 프로그램 시작.

mov rax, 1 ; syscall number 1을 rax 레지에 저장. 1은 write를 의미함.

mov rdi, 1 ; stdout number 1을 rdi에 저장함. 파일 디스크립터임.

mov rsi, msg ; 출력할 문자열이 담긴 msg의 주소를 rsi 레지에 저장함.

mov rdx, 12 ; rdx에 12 저장. 문자열 길이를 의미함.

syscall ; rax 값에 따라 시스템 콜을 호출.

mov rax, 60 ; syscall number 60을 rax에 저장. 60은 exit을 의미함.

mov rdi, 0 ; 0을 rdi에 저장. 프로그램 정상종료.

syscall ; rax에 저장된 syscall 호출. 정상 종료하는 시스템콜임.