Process A 실행 → Process B로
바꿔주는데, 바꿔주는 메커니즘을
컨텍스트 스위칭 이라고 한다.

프로세스와 컨텍스트 스위칭

- 프로세스 구조 deep dive
  → 상세하게 알아본다.

- 메커니즘을
이해하기 위해
프로세스 구조를 상세하게
알아야 한다.

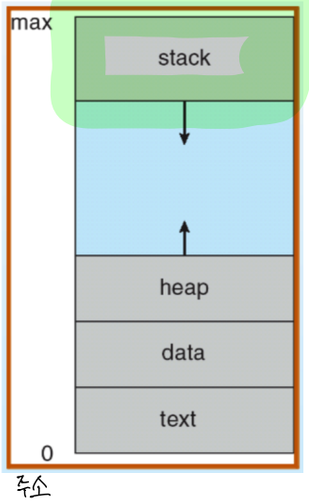
- 실제 프로그램의 구조를 알 필요가 있다
(현업에서 디버깅 시 같게 할 경우 필요)

프로세서과 주요 Register들의 관계,,

프로세스와 컨텍스트 스위칭

- 프로세스 (process) 는 일반적으로 어떻게 구성되어 있을까?
  - text(CODE): 코드
  - data: 변수/초기화된 데이터
  - stack: 임시 데이터(함수 호출, 로컬 변수등)
  - heap: 코드에서 동적으로 만들어지는 데이터

Stack Frame



> OS.xlsx -> ProcessStructure

Process

컴파일 된다고 가정한 프로세스 구조

③ STACK	Return Address : 0050h
	a=1
	b=2
④ HEAP	
DATA	c=0 ② 변수 저장된 공간
① 컴파일 된 코드 저장 CODE 사실상 기계어	def func(a,b)
	Print (a+b)
	c=0
	c= func (1,2)
	Print (c)

함수가 끝나는 다음 코드의 주소를 return address로 stack에 저장

0050h 주소

Program

파이썬은 interpreter 언어서 컴파일 없이 한줄씩 실행파일 X

```
def func(a,b)
    Print (a+b)
c=0
c= func (1,2)
Print (c)
```

함수 선언 — 함수 내 변수 (동적)

변수선언 → 주소를 가져옴. 메모리

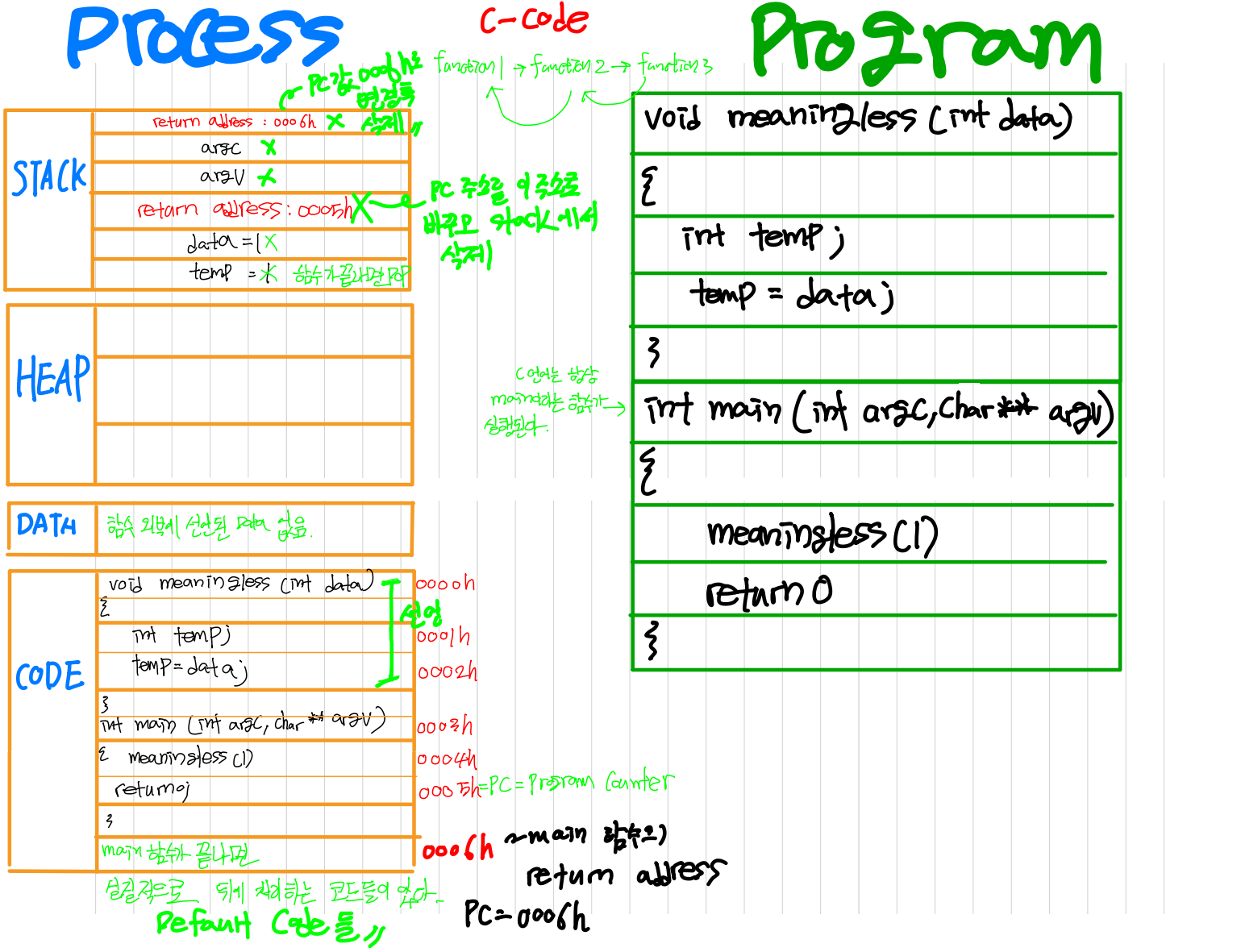
함수 실행 → 함수의 실행은 동적으로 처리를 해야함

↳ 함수에 선언된 변수는 함수 끝나면 사라지므로.

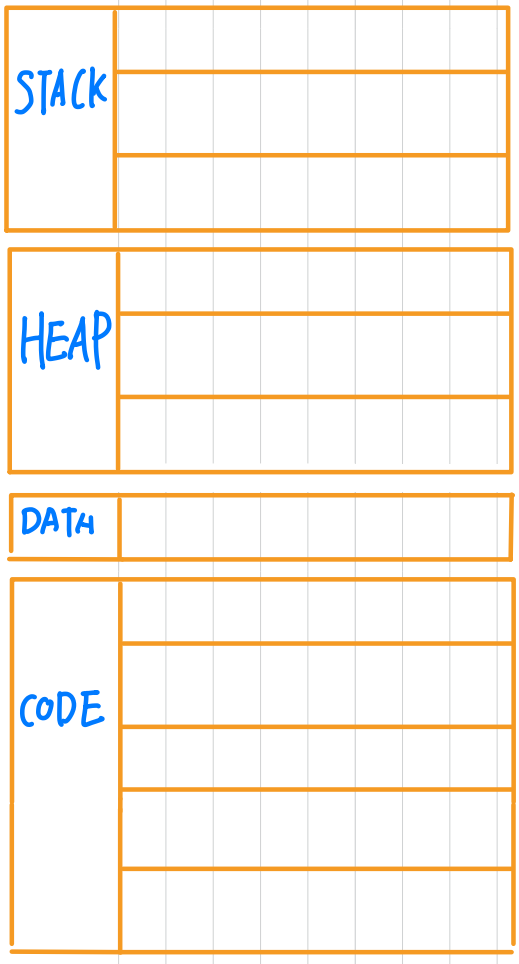
malloc() ← C 언어

↳ 동적으로 특정 메모리 공간을 생성

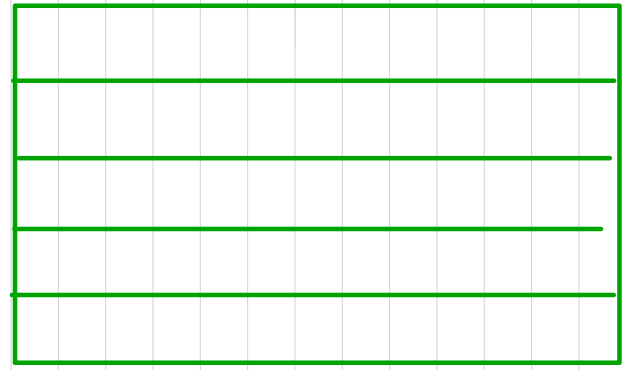
func (1,2) 실행이 끝나면 stack의 1,2 제거되고, return address로 가서 그 코드를 실행하게 된다.


Process
C-code
Program
function1 → function2 → function3
STACK
return address : 0006h
argc
argv
return address : 0005h
data = 1
temp = ?
HEAP
DATA
함수 외부에 선언된 Data 없음.
CODE
void meaningless (int data)
{
int temp;
temp = data;
}
int main (int argc, char** argv)
{
meaningless(1)
return 0
}
0000h
0001h
0002h
0003h
0004h
0005h
0006h
선언
main 함수 끝나면
실질적으로 뒤에 처리하는 코드들이 없다.
Default Code들
PC=0006h

Process



Program



여기서 잠깐 복습!

- 스택 (stack) Stack Frame (Stack이라는 자료구조를 이용하여 함수를 실행할 수 있는 구조를 만들것)

> 이 자료구조는 뭐에 쓰이고 왜 강조해서 배웠나?

> datastructure.xlsx -> Stack

PC, SP 컴퓨터 구조에서 주요한 Register

Q. Process가 곧 Code 예시 검색 해보기,,

프로세스와 컴퓨터 구조 복습

- 프로세스 (process) 는 일반적으로 어떻게 동작할까? - 컴퓨터 구조도 복습
  - text: 코드
  - data: 변수/초기화된 데이터
  - stack: 임시 데이터(함수 호출, 로컬 변수등)
  - heap: 코드에서 동적으로 만들어지는 데이터
- PC(Program Counter) + SP(Stack Pointer)

↳ Code를 한 줄 한 줄 가리키는 주소 레지스터

↳ Stack Frame의 최상단 주소를 가리키는 레지스터,

↳ 이 두 개가 프로세스 구조에서 어떻게 동작 되는지 알아보자.

> OS.xlsx -> ProcessWithCS

Heap 이란?

main 자체가 함수, 따라서 내부의 변수들은 local 변수처럼 취급이 된다.

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *data;
    data = (int *) malloc(sizeof(int));
    *data = 1;
    printf("%d\n", *data);

    return 0;
}
```

heap (32bit) → data

heap

free(data) 공간 해제!

정수형 Pointer

malloc으로 만든 메모리의 주소 return

컴파일러가 미리 작성 해놓은 함수

- 자세히 // 라는 함수를 갖고있는 기본 라이브러리가 있고,

main() // 이 내부에서 main() 이라는 함수 호출,

Process 종료() // 처리.

우리가 만든 C프로그램 실행

이 전체가 실행 파일이 된다.

OS.xlsx -> ProcessWithHeap

프로세스 구조: Stack, HEAP, DATA(BSS, DATA), TEXT(CODE)

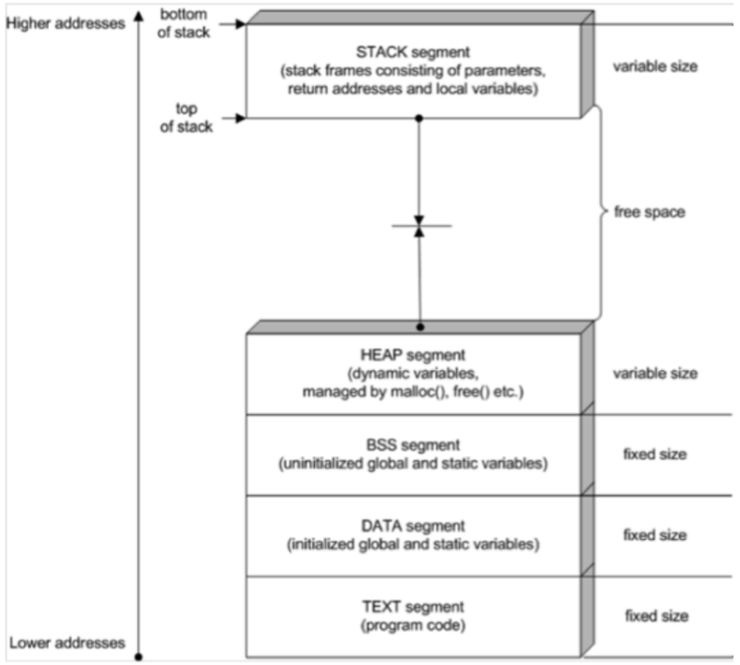
DATA를 BSS와 DATA로 분리

stack

heap

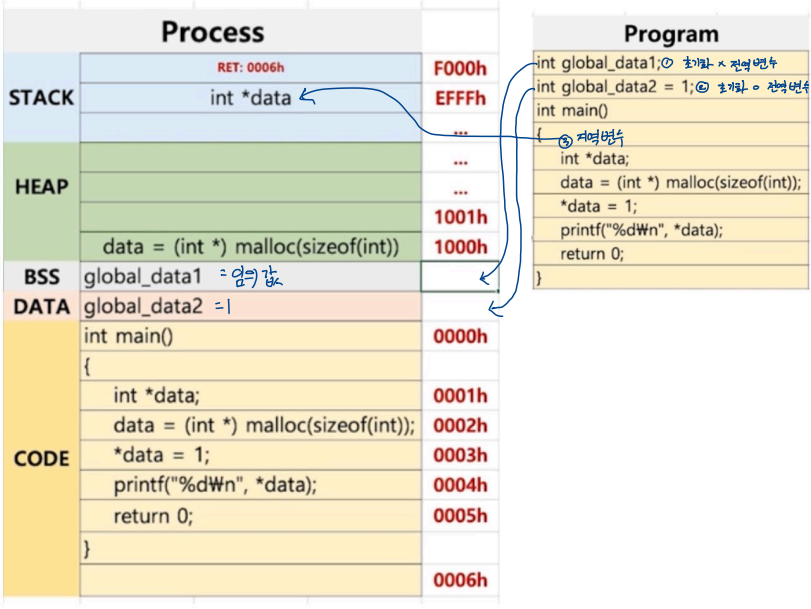
data ↔ BSS → 초기화 되지않은 전역변수
data ↔ DATA → 초기화 된 전역변수

code (text)



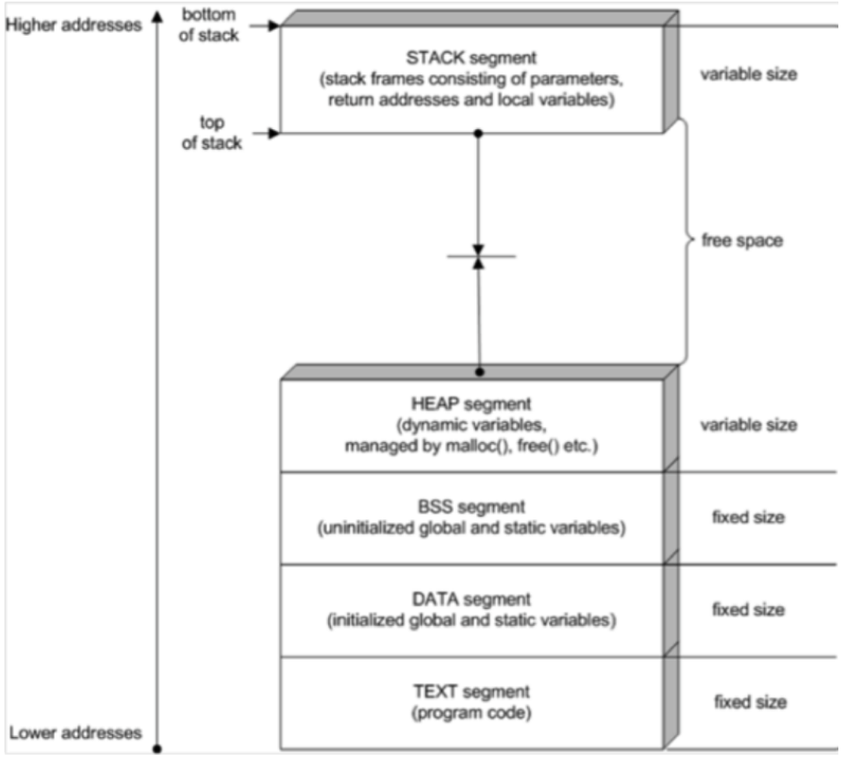
출처: http://www.drdobbs.com/security/anatomy-of-a-stack-smashing-attack-and-h/240001832

OS.xlsx -> ProcessAll

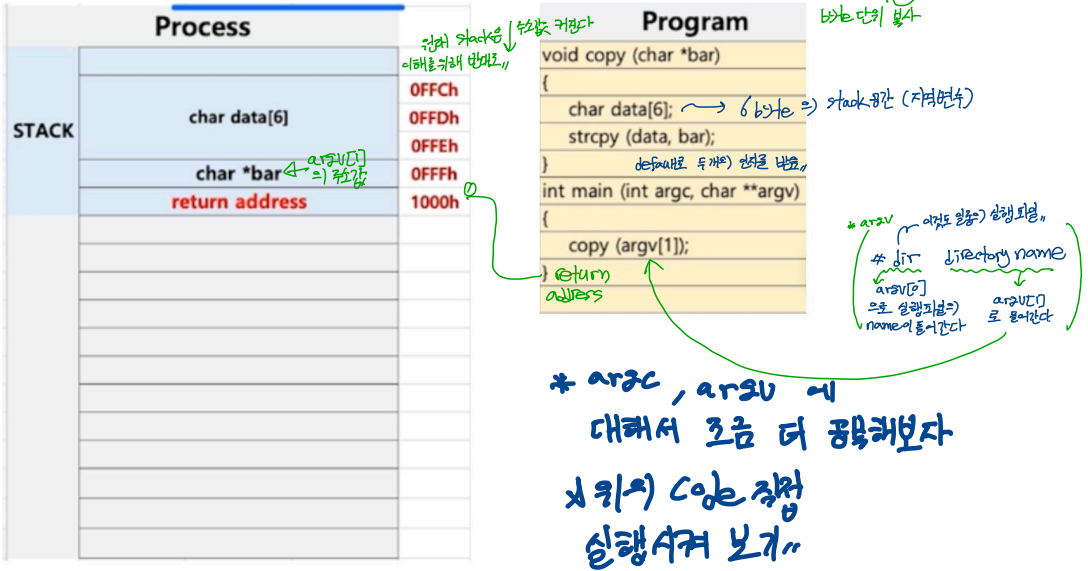

Process
STACK
HEAP
BSS
DATA
CODE
RET: 0006h
int *data
data = (int *) malloc(sizeof(int))
global_data1
= 0의 값
global_data2
= 1
int main()
{
int *data;
data = (int *) malloc(sizeof(int));
*data = 1;
printf("%d₩n", *data);
return 0;
}
F000h
EFFFh
...
...
...
1001h
1000h
0000h
0001h
0002h
0003h
0004h
0005h
0006h
Program
int global_data1;① 초기화 x 전역변수
int global_data2 = 1;② 초기화 o 전역변수
int main()
{
③ 지역변수
int *data;
data = (int *) malloc(sizeof(int));
*data = 1;
printf("%d₩n", *data);
return 0;
}

가볍게 듣기

- 스택 오버플로우: 주로 해커들의 공격에 활용되었음

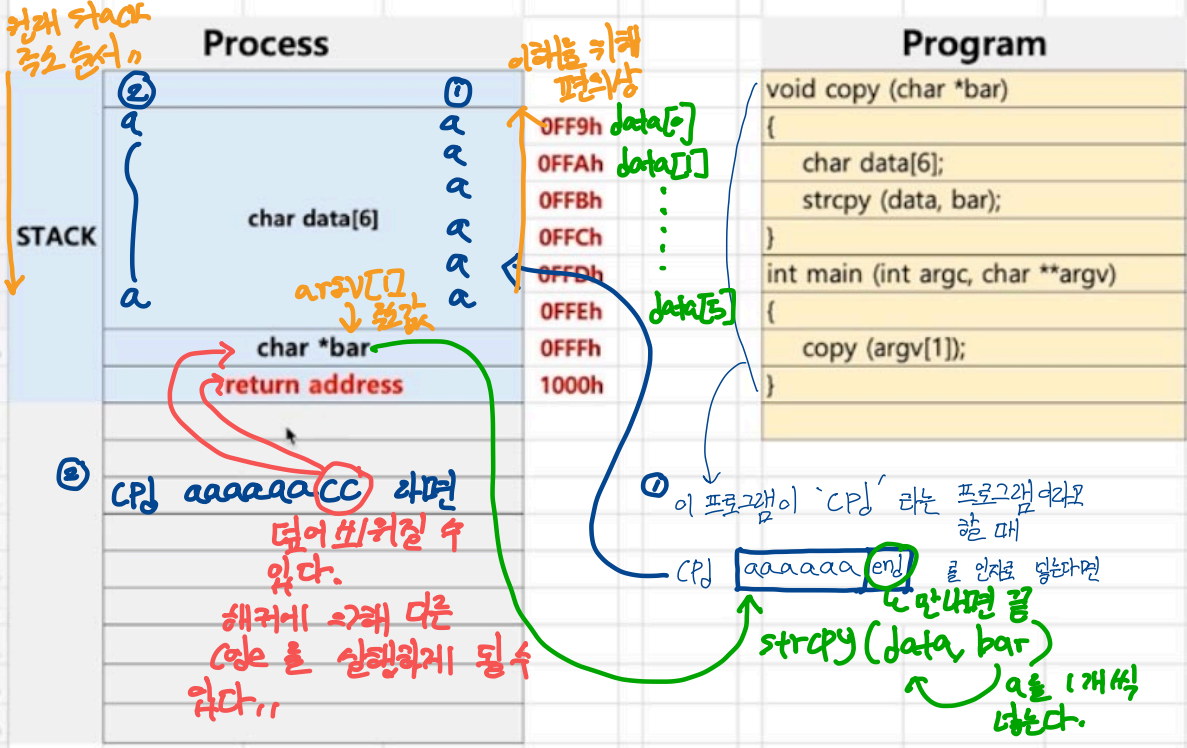


> 출처: http://www.drdobbs.com/security/anatomy-of-a-stack-smashing-attack-and-h/240001832



* argc, argv 에 대해서 조금 더 공부해보자

x키의 code 작성 실행시켜 보기


컴퓨터 stack 주소 순서
Process
Program
STACK
char data[6]
char *bar
return address
argv[1] 의 값
0FF9h
0FFAh
0FFBh
0FFCh
0FFDh
0FFEh
0FFFh
1000h
data[0]
data[1]
data[5]
void copy (char *bar)
{
char data[6];
strcpy (data, bar);
}
int main (int argc, char **argv)
{
copy (argv[1]);
}
CP
aaaaaa
end
strcpy (data, bar)
a를 1개씩 넣는다.
CP aaaaaaCC 라면
덮어씌워질 수 있다.
해커에 의해 다른 명령 등 실행하게 될 수 있다.

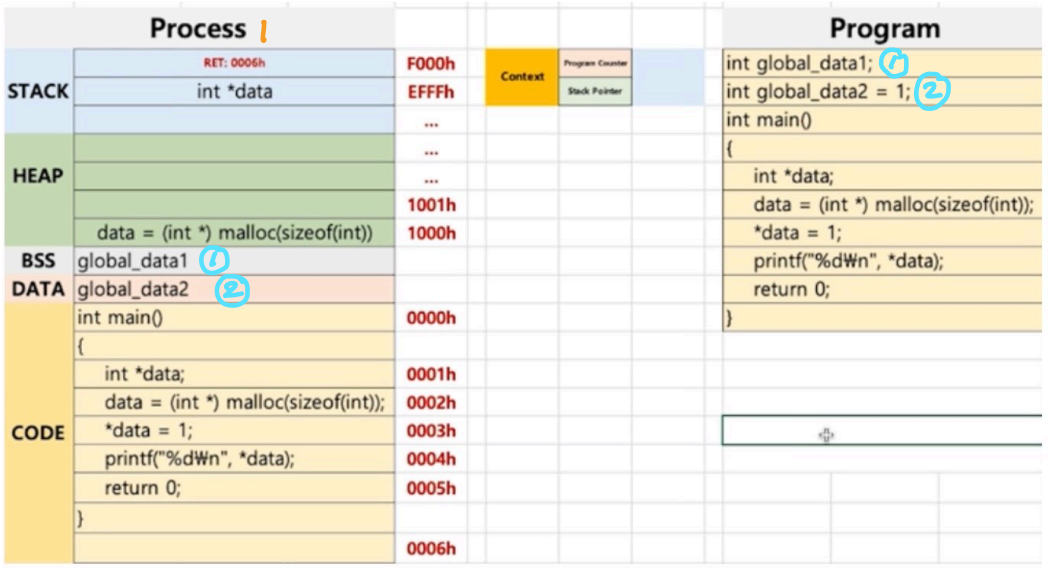
프로세스와 컨텍스트 스위칭

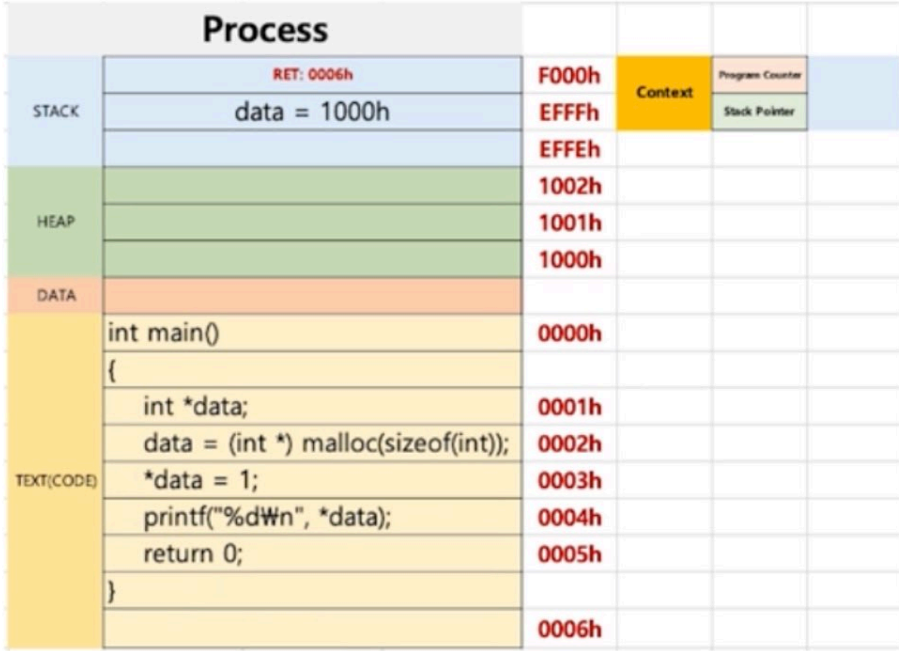
다시 PC, SP에 주목하자.

- PC(Program Counter) + SP(Stack Pointer)

Stack, HEAP, DATA(BSS, DATA), TEXT(CODE)

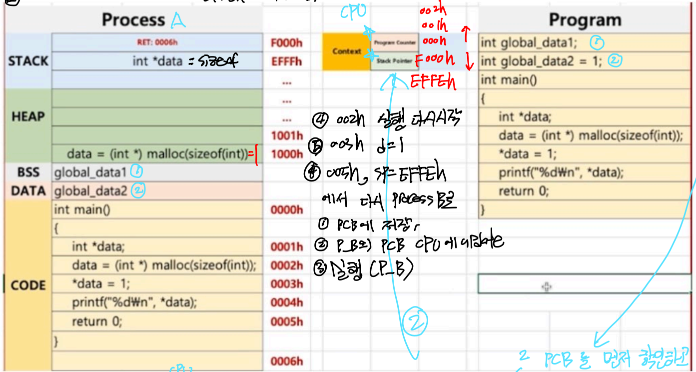
OS.xlsx -> ProcessAllWithCS


Process 1
Program
STACK
HEAP
BSS
DATA
CODE
RET: 0006h
int *data
data = (int *) malloc(sizeof(int))
global_data1
global_data2
int main()
{
int *data;
data = (int *) malloc(sizeof(int));
*data = 1;
printf("%d₩n", *data);
return 0;
}
F000h
EFFFh
1001h
1000h
0000h
0001h
0002h
0003h
0004h
0005h
0006h
Context
Program Counter
Stack Pointer
int global_data1;
int global_data2 = 1;
int main()
{
int *data;
data = (int *) malloc(sizeof(int));
*data = 1;
printf("%d₩n", *data);
return 0;
}

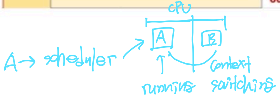

Process
STACK
RET: 0006h
data = 1000h
HEAP
DATA
TEXT(CODE)
int main()
{
int *data;
data = (int *) malloc(sizeof(int));
*data = 1;
printf("%d₩n", *data);
return 0;
}
F000h
EFFFh
EFFEh
1002h
1001h
1000h
0000h
0001h
0002h
0003h
0004h
0005h
0006h
Context
Program Counter
Stack Pointer

① 전역변수는 Data 영역에 저장
② PC=000h, SP=F000h
③ PC=001h, SP=EFFFh (현재는 그 값) 공간 Stack 을 가리켜서 EFFEh 가 되어야 함)
④

③ PC=002h, SP=EFFEh 에서
Process가 Process 2로 변경을 하려며
PCB라는 저장공간에 Context를
저장하고 이 Process는 ready or block 상태로 전환, → B



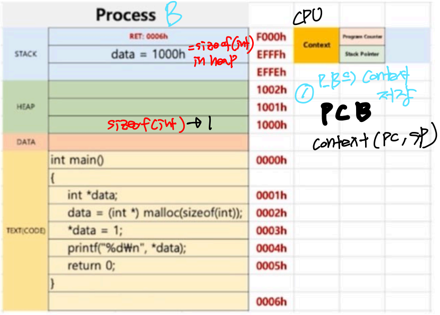
① PC=000h → PC=001h 함수이므로 Return address
stack에 들어감, 지역변수 들어가고
SP는 아무것도 들어가지 않은 EFFEh를 가리키게 된다.
② PC=002h → heap에 sizeof(int) 할당
후 주소를 data 값에 할당 (주소값)
③ PC=003h heap 공간에 1 할당
④ PC=004h, SP EFFEh 인 상태에서 Process A로 이동하려고 함



PCB OS가 관리하는 별도 메모리,
Context (PC, SP) 저장

③ 실행

2 PCB를 먼저 확인하고
CPU의 register에
덮어씌운다.



PCB 프로세스의 상태를 저장하는 데이터 구조로 OS에서 관리하고있다.

> PC, SP는 어디에 저장하나?

Process Control Block (PCB) 에 저장!

> Process Context Block 이라고도 함

1. Porcess ID
2. Register 값 (PC, SP 등)
3. Scheduling Info (Porcess State) ready, block, running
4. Memory Info (메모리 사이즈 limit)

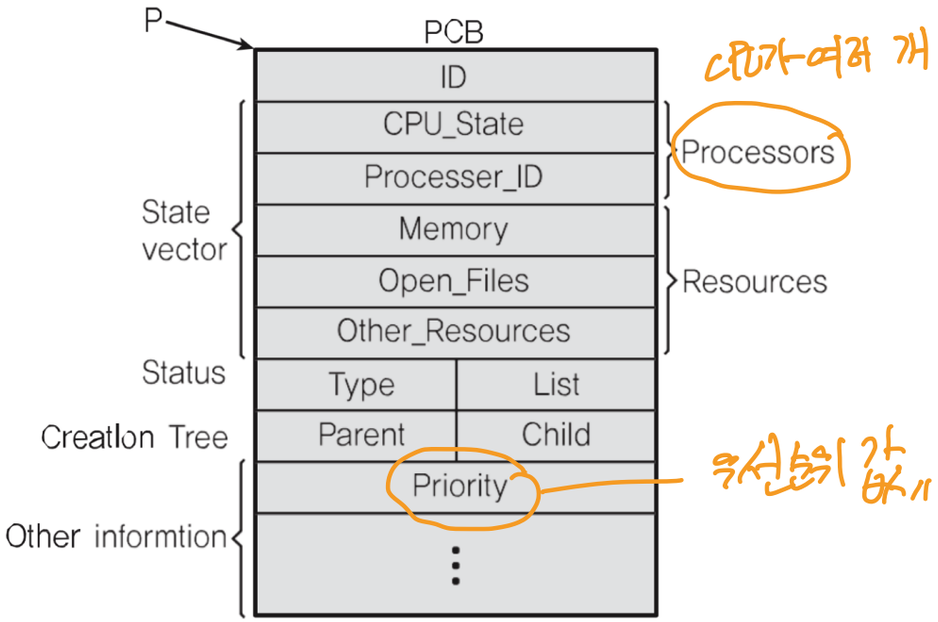
   ...

> PCB: 프로세스가 실행중인 상태를 캡쳐/구조화해서 저장

PCB: 리눅스 예

1. Porcess ID
2. Register (PC, SP 등)
3. Scheduling Info (Porcess State)
4. Memory Info (메모리 사이즈 limit)

C struct 로 관리되고 있음



정리

- 프로세스 구조
  - Stack, HEAP, DATA(BSS, DATA), TEXT(CODE)
- PCB
  - 프로세스 상태 정보 - PC, SP, 메모리, 스케쥴링 정보등