

운영체제 및 실습 - Memory -

1

한예진, 김수창, 이승준
Dankook University

Linux Memory Layout

2

1. Linux Memory Layout

2. Stack

3. Heap

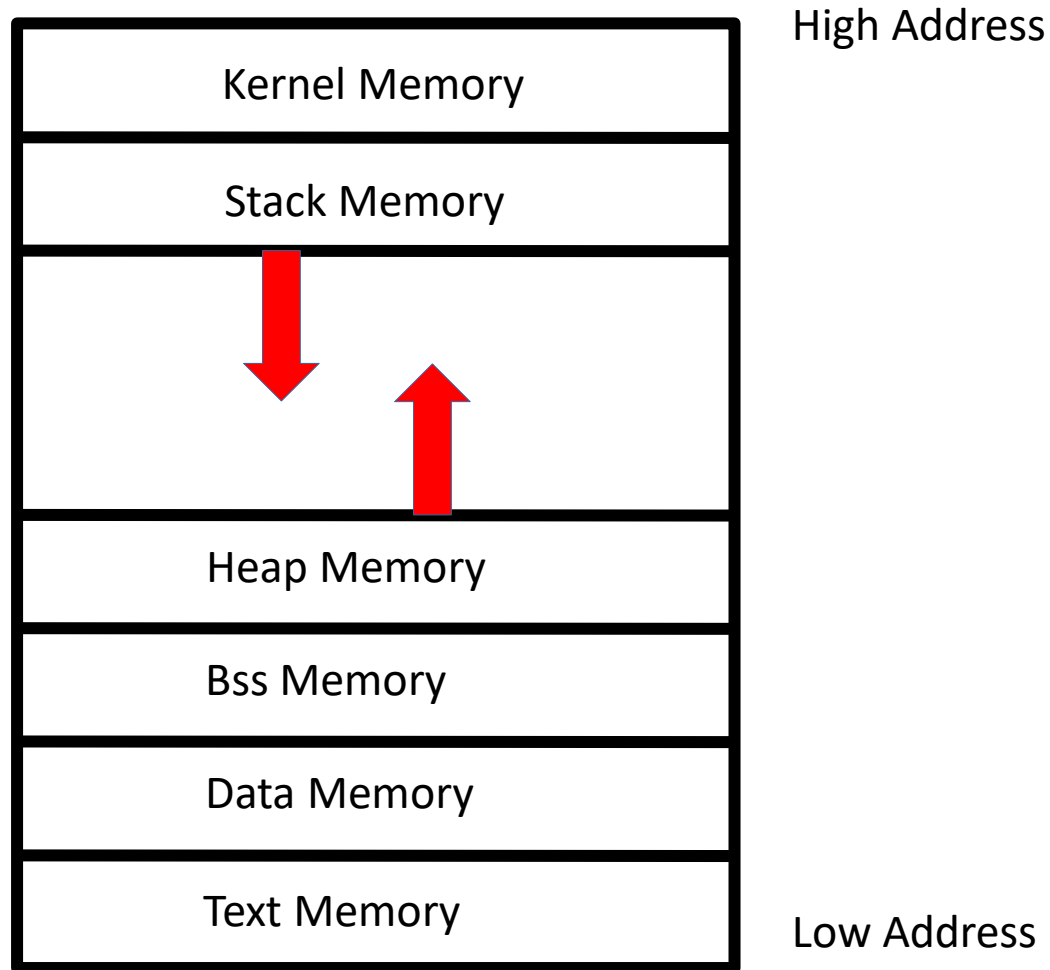
4. malloc? free, realloc?

5. Homework

Linux Memory Layout

3

1. Stack



1. Stack

1. 프로그램에서 지역변수들이 할당
2. 높은 주소에서 낮은 주소로 증가
3. 함수마다 자신의 Stack Frame을 가지며 함수의 종료 시 Stack Frame 또한 사라진다.

주의) Stack Frame이 사라진다고 해서 저장된 데이터가 삭제되는 것은 아님

1. Stack

Install tools

```
yum update
```

```
yum upgrade
```

```
yum install dnf
```

```
dnf install gcc gdb python3 wget git
```

```
wget https://github.com/pwndbg/pwndbg/releases/download/2024.02.14/pwndbg-2024.02.14-1.x86\_64.rpm
```

```
dnf install pwndbg-2024.02.14-1.x86_64.rpm
```

Linux Memory Layout

6

1. Stack

```
#include <stdio.h>
#include <stdint.h>

int main(void)
{
    int64_t apple = 3;
    int64_t orange = 7;
    int64_t name[4] = {1,2,3,4};

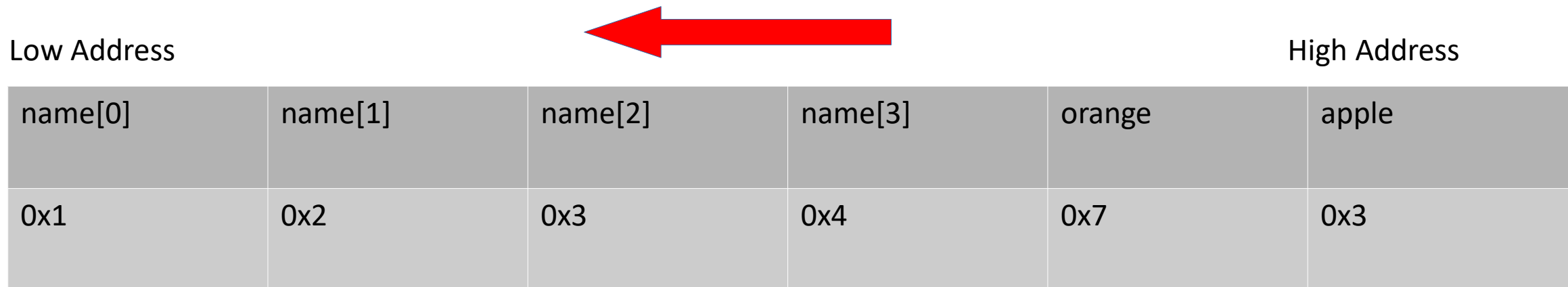
    printf("apple address : %p\n", &apple);
    printf("orange address: %p\n", &orange);
    printf("name address : %p\n", name);
    return 0;
}
```

Linux Memory Layout

1. Stack

```
[root@c5a679270c57 ~]# gcc test.c
[root@c5a679270c57 ~]# ./a.out
apple address : 0x7ffc870d7ee8
orange address: 0x7ffc870d7ee0
name address : 0x7ffc870d7ec0
[root@c5a679270c57 ~]#
```

변수를 생성하는 과정에서 높은 주소에서 낮은 주소로 할당되는 것을 볼 수 있다.

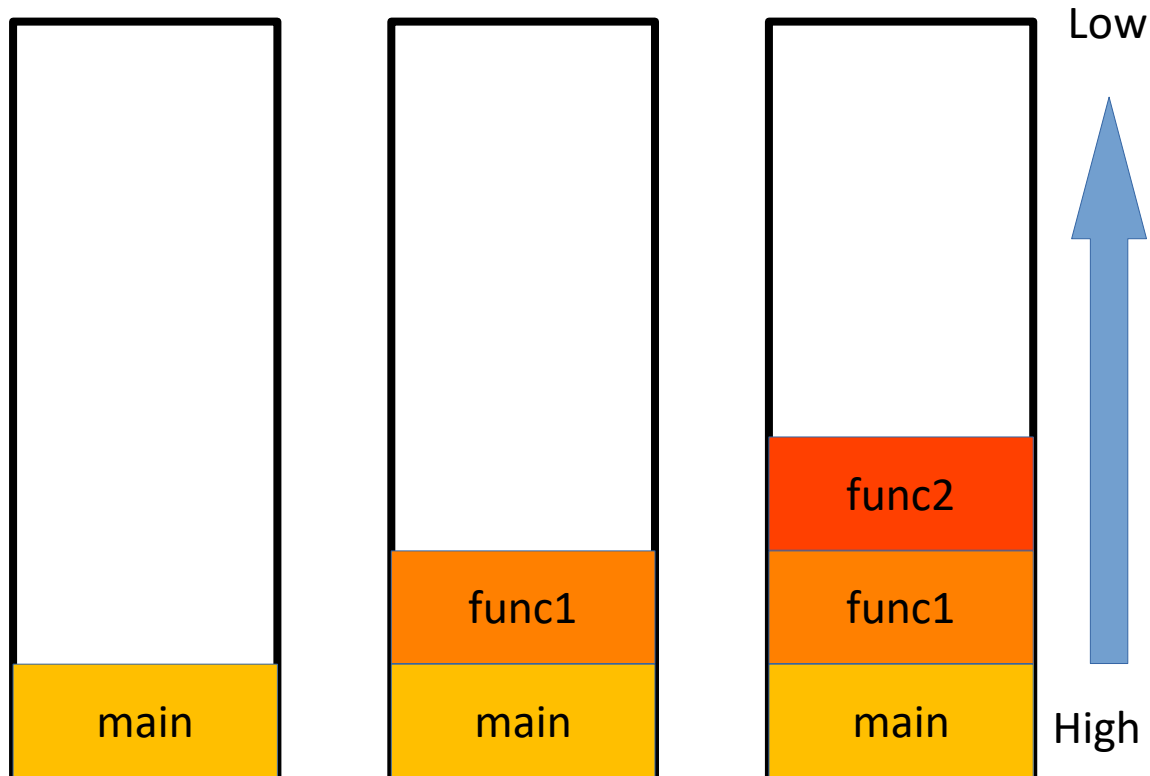


Linux Memory Layout

1. Stack

Stack Frame?

함수를 위한 스택 공간(ex. 지역변수, 매개변수 할당)



```
#include <stdio.h>

void func2(){
    printf("Hello func2\n");
}

void func1(){
    printf("Hello func1\n");
    func2();
}

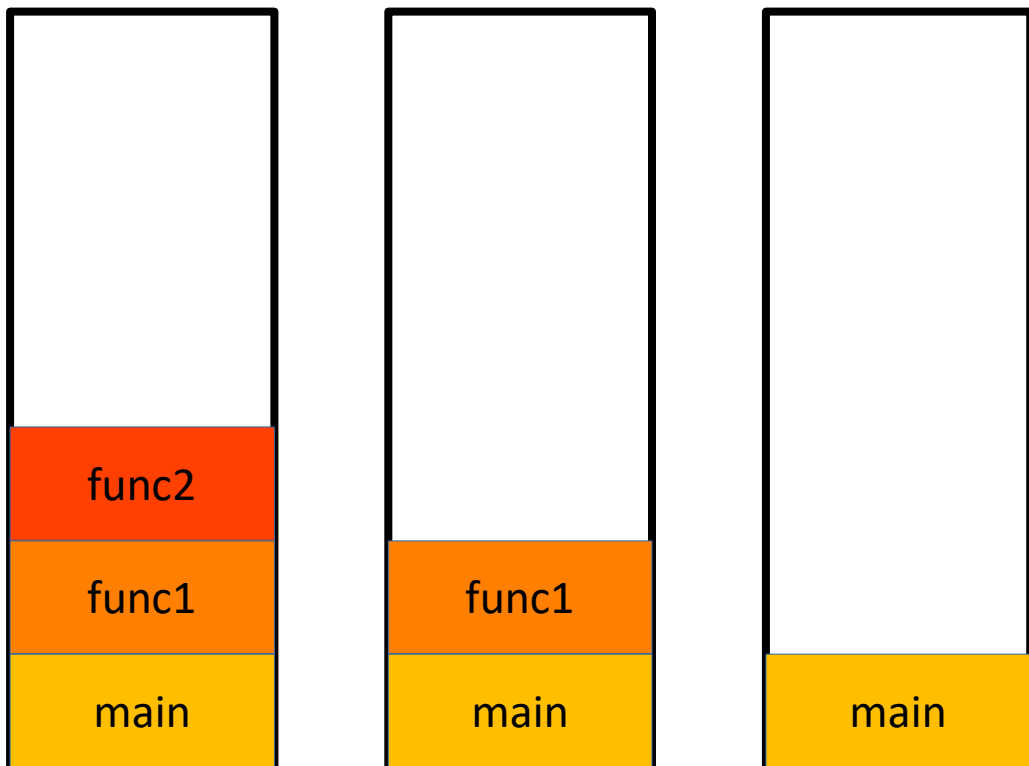
int main(void){
    func1();
    return 0;
}
```


Linux Memory Layout

1. Stack

func1(), func2()에 저장된 지역변수들이 해당 함수를 벗어나서 사용하지 못하는 이유!

함수 호출이 끝나는 과정?



그럼 func1(), func2()함수들이 종료되고 나서
해당 함수들에 선언했던 변수들의 값은 무엇으로 초기화?

1. Stack

```
#include <stdio.h>

int *func2_ptr;

void func2(){
    int func2_data = 120;
    func2_ptr = &func2_data;
}

void func1(){
    func2();
}

int main(void){
    func1();
    printf("func2 use after data: %d\n", *func2_ptr);
    return 0;
}
```

```
[root@c5a679270c57 ~]# gcc frame.c
[root@c5a679270c57 ~]# ./a.out
func2 use after data: 120
[root@c5a679270c57 ~]#
```

사실 데이터는 남아있음!

1. Stack

그렇다면 왜 사용하지 못하는가?

func3()함수를 호출하고나서

*func2_ptr의 값은 무엇이 될까?

```
#include <stdio.h>

int *func2_ptr;

void func2(){
    int func2_data = 120;
    func2_ptr = &func2_data;
}

void func1(){
    func2();
}

void func3(){
    char buffer[200] = {0};
    int i=0;
    for(;i<200;i++) {
        buffer[i] = 7;
    }
}

int main(void){
    func1();
    printf("func2 use after data: %d\n", *func2_ptr);
    func3();
    printf("after fucntion data: %d\n", *func2_ptr);
    return 0;
}
```

1. Stack

```
[root@c5a679270c57 ~]# gcc frame.c
[root@c5a679270c57 ~]# ./a.out
func2 use after data: 120
after fuction data: 117901063
[root@c5a679270c57 ~]#
```

쓰레기값으로 덮어짐!

즉, 특정 함수에서 선언된 지역변수는 특정 함수의 호출이 끝난 뒤에는 어떠한 값으로 덮어질지 모름

1. Heap

그럼 특정 함수에서 선언된 값을 함수의 호출이 끝난 뒤에도 사용하고 싶다면?

Heap Memory!

1. Heap

프로그램에서 동적으로 할당된 공간을 의미!
(일반적으로 malloc함수를 이용하여 Heap에 공간을 할당)

1. Heap 메모리에 할당된 공간은 프로그램이 종료될 때 까지 유지
2. 낮은 주소에서 높은 주소로 할당

스택은 자동으로 정리되지만 Heap은 그렇지 않기에 장점이자 **곧 메모리 누수라는 큰 단점** 발생 가능

1. Heap

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

int *heap(){
    int *a = (int*)malloc(4);
    *a = 3;
    return a;
}

int main(void){
    int *a = heap();
    printf("A(%p) : %d\n",a, *a);
}
```

```
[root@c5a679270c57 ~]# gcc ma.c
```

```
[root@c5a679270c57 ~]# ./a.out
```

```
A(0x939010) : 3
```

```
[root@c5a679270c57 ~]#
```

1. Heap

Heap공간은 Stack공간과 달리 **미리 할당되지 않은 영역이다.**

그렇기에 Heap 메모리 할당 요청을 하면 Heap공간에서 **매번 할당**을 진행하여야 하는데 어떤 syscall이 호출?

```
BRK(2)                                Linux Programmer's Manual

NAME
    brk, sbrk - change data segment size

SYNOPSIS
    #include <unistd.h>

    int brk(void *addr);

    void *sbrk(intptr_t increment);
```


1. Heap

DESCRIPTION

`brk()` and `sbrk()` change the location of the program break, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory.

`brk()` sets the end of the data segment to the value specified by addr, when that value is reasonable, the system has enough memory, and the process does not exceed its maximum data size (see `setrlimit(2)`).

`sbrk()` increments the program's data space by increment bytes. Calling `sbrk()` with an increment of 0 can be used to find the current location of the program break.

RETURN VALUE

On success, `brk()` returns zero. On error, -1 is returned, and errno is set to `ENOMEM`.

On success, `sbrk()` returns the previous program break. (If the break was increased, then this value is a pointer to the start of the newly allocated memory). On error, (void *) -1 is returned, and errno is set to `ENOMEM`.

CONFORMING TO

4.3BSD; SUSv1, marked LEGACY in SUSv2, removed in POSIX.1-2001.

NOTES

Avoid using `brk()` and `sbrk()`: the `malloc(3)` memory allocation package is the portable and comfortable way of allocating memory.

Various systems use various types for the argument of `sbrk()`. Common are int, ssize_t, ptrdiff_t, intptr_t.

1. Heap

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

void *mymalloc(int size){
    if(brk(0) == -1){
        printf("Failed Alloc in Heap\n");
        return NULL;
    }

    void *heap_memory = sbrk(size);
    if(heap_memory == (void*)-1){
        printf("Failed Alloc in Heap\n");
        return NULL;
    }

    return heap_memory;
}

int main(void){
    int i=0;
    int *heap = (int*)mymalloc(4);
    *heap = 32;
    printf("Heap(%p): %d\n", heap, *heap);
    return 0;
}
```

```
[root@c5a679270c57 ~]# gcc sbrk.c
[root@c5a679270c57 ~]# ./a.out
Heap(0xe3d000): 32
[root@c5a679270c57 ~]#
```

Heap 영역에 메모리 공간을 할당하고 값을 넣는데 성공함!

1. Heap

그렇다면 malloc함수는 brk(), sbrk()에 비해 어떠한 장점을 지니는가?

1. Heap

```
int main(void){
    int i=0;
    int *heap, *ma;
    clock_t start, end;
    double cpu_time_used;

    start = clock();
    for(;i<10000000;i++){
        heap = (int*)mymalloc(sizeof(int));
    }
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Time taken: %f seconds\n", cpu_time_used);

    start = clock();
    for(i=0;i<10000000;i++){
        ma = (int*)malloc(sizeof(int));
    }
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Time taken: %f seconds\n", cpu_time_used);

    return 0;
}
```

1. Heap

```
[root@c5a679270c57 ~]# ./a.out  
Time taken: 6.710000 seconds  
Time taken: 0.350000 seconds  
[root@c5a679270c57 ~]#
```

malloc()함수를 사용하는 것이 brk-sbrk syscall을 사용하는 것보다 속도가 현저히 빠름

그렇다면 malloc함수는 내부적으로 brk-sbrk를 사용하지 않는 것인가?

1. Heap

```
# strace ./a.out
```

```
brk(NULL)           = 0x55de178a3000  
brk(NULL)           = 0x55de178a3000  
brk(0x55de178c4000) = 0x55de178c4000
```

내부적으로 brk-sbrk syscall 사용!

1. Heap

어떤 알고리즘이 malloc함수가 빠르게 동작하게 하는 것일까

즉, malloc함수는 어떻게 Heap 메모리를 관리할까

1. Heap

malloc()함수는 아래 구조체를 통해서 Heap을 관리

```
struct malloc_chunk {  
    INTERNAL_SIZE_T  prev_size; /* Size of previous chunk (if free). */  
    INTERNAL_SIZE_T  size;      /* Size in bytes, including overhead. */  
    struct malloc_chunk* fd;     /* double links -- used only if free. */  
    struct malloc_chunk* bk;  
    /* 위의 fd와 bk는 오직 해제된 상태에서 나타나며 해제되지 않을 때는 userdata로 사용된다. */  
    /* Only used for large blocks: pointer to next larger size. */  
    struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */  
    struct malloc_chunk* bk_nextsize;  
};
```

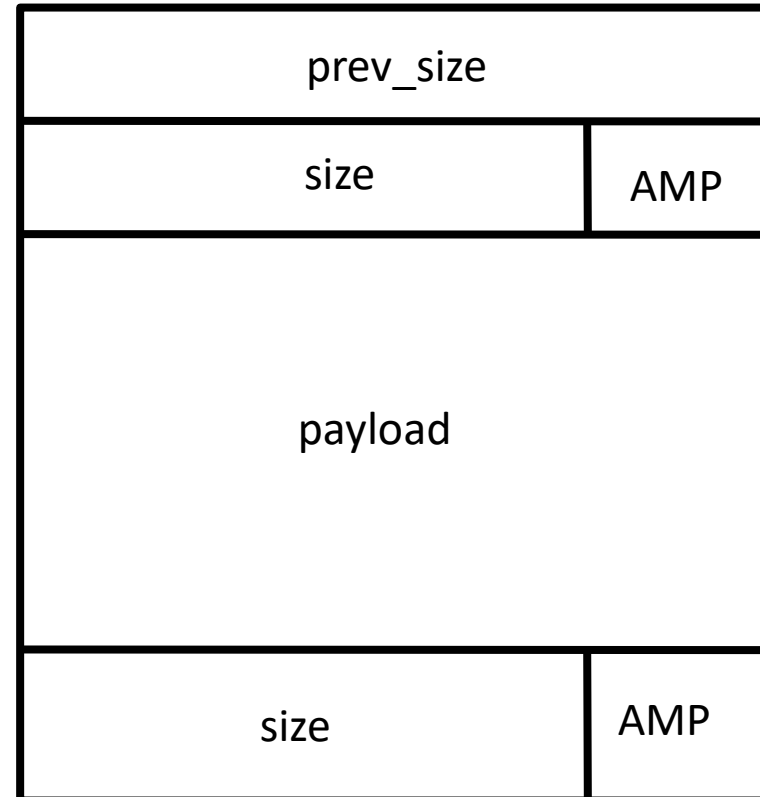

1. Heap

When Chunk alloc

Chunk address



Returned by malloc



AMP

A : 할당된 Arena

M : mmap할당 유무

P : 이전 chunk 사용 유무

P=1

1. Heap

Malloc 함수 알고리즘 (초기 dmalloc 기준) (**최신버전 기준 아님**)

malloc함수는 사용자가 요청할 때마다 Heap 메모리공간을 brk-sbrk syscall로 요청하여 할당하는 것이 아닌 미리 한번에 크게 할당받고 **Top Chunk**에서 조금 조금씩 분할

사용자가 요청하면 Top Chunk에서 분할

만약 메모리를 Free한다면?

해당 메모리를 OS에 반환하는 것이 아닌 Fastbin, Smallbin, Largebin, Unsortedbin과 같은 자료구조에 저장

사용자가 추후 요청한다면 자료구조에 저장된 청크를 꺼내서 재사용!

1. Heap

malloc함수는 수 많은 세월동안 좋은 알고리즘들로 구성되어 있기에 엄밀히 따지자면 이전에 언급한 방식은 틀렸다고 봐도 됨.

하지만 기본적인 방식은 저것을 기반,

동작과정을 매우 정확하게 따지자면 tcache, thread arena, treebin, mmap, chunk 분할, 합병, ...

동작과정만 살펴봐도 TABA 수업기간 끝남

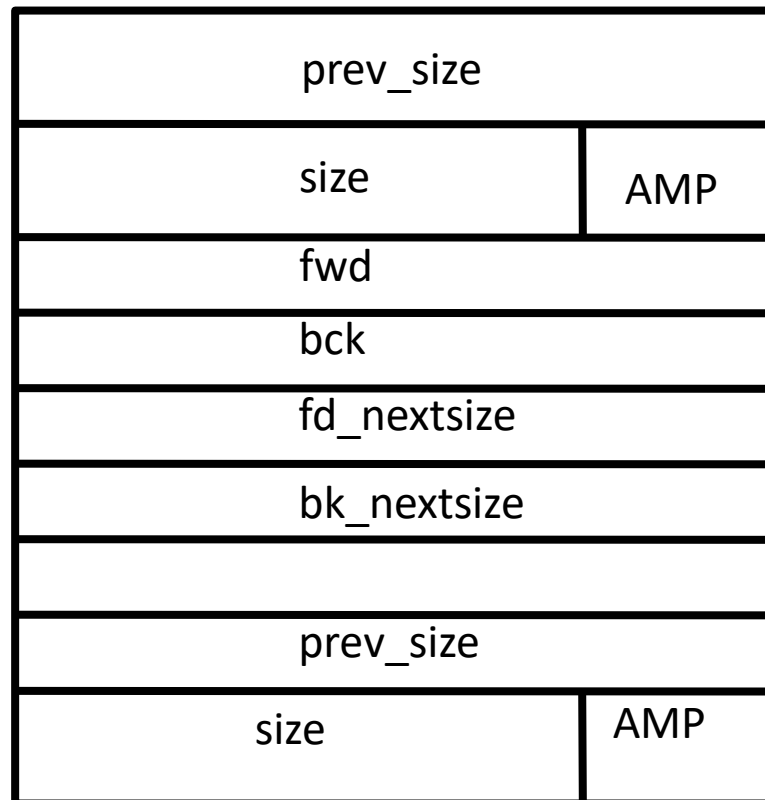
1. Heap

When Chunk Free

Chunk address



Returned by malloc



Fwd : 다음 연결된 청크의 주소

Bck : 이전 연결된 청크의 주소

Chunk들은 Single or double linked list로 구성

P=0

1. Heap – Example

4바이트의 메모리를 해제한다면?

작은 크기의 바이트이기때문에 Fastbin 자료구조에 저장
Fastbin은 Single Linked List형태로 동작하며 LIFO형태를 지님

그리고 4바이트를 요청할 때 Fastbin에서 목록 탐색

```
[root@c5a679270c57 ~]# gcc heap.c
[root@c5a679270c57 ~]# ./a.out
Hello A: 0x1cf0010
Hello B: 0x1cf0030
Hello C: 0x1cf0030
[root@c5a679270c57 ~]#
```

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

int main(void){
    int *a = (int*)malloc(sizeof(int));
    int *b = (int*)malloc(sizeof(int));
    printf("Hello A: %p\n", a);
    printf("Hello B: %p\n", b);
    free(a);
    free(b);

    int *c = (int*)malloc(sizeof(int));
    printf("Hello C: %p\n", c);

    return 0;
}
```

1. Heap – Example

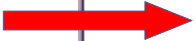
```
# gcc heap.c -g
# pwndbg a.out
# b* main
# r
```

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

int main(void){
    int *a = (int*)malloc(sizeof(int));
    int *b = (int*)malloc(sizeof(int));
    printf("Hello A: %p\n", a);
    printf("Hello B: %p\n", b);
    free(a);
    free(b);

    int *c = (int*)malloc(sizeof(int));
    printf("Hello C: %p\n", c);

    return 0;
}
```



```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0xbc1000
Size: 0x20 (with flag bits: 0x21)

Allocated chunk | PREV_INUSE
Addr: 0xbc1020
Size: 0x20 (with flag bits: 0x21)

Top chunk | PREV_INUSE
Addr: 0xbc1040
Size: 0x20fc0 (with flag bits: 0x20fc1)

pwndbg> 
```

1. Heap – Example

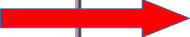
```
# gcc heap.c -g
# pwndbg a.out
# b* main
# r
```

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

int main(void){
    int *a = (int*)malloc(sizeof(int));
    int *b = (int*)malloc(sizeof(int));
    printf("Hello A: %p\n", a);
    printf("Hello B: %p\n", b);
    free(a);
    free(b);

    int *c = (int*)malloc(sizeof(int));
    printf("Hello C: %p\n", c);

    return 0;
}
```



```
pwndbg> heap
Free chunk (fastbins) | PREV_INUSE
Addr: 0xbc1000
Size: 0x20 (with flag bits: 0x21)
fd: 0x00

Free chunk (fastbins) | PREV_INUSE
Addr: 0xbc1020
Size: 0x20 (with flag bits: 0x21)
fd: 0xbc1000

Top chunk | PREV_INUSE
Addr: 0xbc1040
Size: 0x20fc0 (with flag bits: 0x20fc1)

pwndbg> 

```



```
pwndbg> fastbin
fastbins
0x20: 0xbc1020 → 0xbc1000 ← 0x0
pwndbg> 
```

1. Heap – Example


```
# gcc heap.c -g
# pwndbg a.out
# b* main
# r
```

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

int main(void){
    int *a = (int*)malloc(sizeof(int));
    int *b = (int*)malloc(sizeof(int));
    printf("Hello A: %p\n", a);
    printf("Hello B: %p\n", b);
    free(a);
    free(b);

    int *c = (int*)malloc(sizeof(int));
    printf("Hello C: %p\n", c);

    return 0;
}
```



```
pwndbg> heap
Free chunk (fastbins) | PREV_INUSE
Addr: 0xbc1000
Size: 0x20 (with flag bits: 0x21)
fd: 0x00

Allocated chunk | PREV_INUSE
Addr: 0xbc1020
Size: 0x20 (with flag bits: 0x21)

Top chunk | PREV_INUSE
Addr: 0xbc1040
Size: 0x20fc0 (with flag bits: 0x20fc1)

pwndbg> fastbin
fastbins
0x20: 0xbc1000 ← 0x0
pwndbg> 
```


1. Heap – Example

A chunk와 B Chunk를 할당받고 A와 B순서대로 Free하면

Fastbin자료 구조는 아래와 같음

A Chunk Free :: $A \rightarrow \text{NULL}$

B Chunk Free :: $B \rightarrow A \rightarrow \text{NULL}$

C Chunk Alloc :: $A \rightarrow \text{NULL}$

즉, 재할당시 LIFO로 할당

1. Heap Exploit

만약 c 할당 이후 한번 더 할당을 하게 된다면 B Chunk가 가리켰던 A Chunk를 재할당 받게 된다.

만약! B Chunk의 FD값을 A Chunk가 아닌 다른 공간으로 Overwrite한다면?

Heap Exploit!

1. Heap Exploit – Example

시나리오1

1. A chunk와 B chunk를 8바이트씩 할당받는다.
2. free(A); free(B); free(A) 순서대로 메모리를 해제한다. - Double Free Bug
3. 그럼 Fastbin에는 $A \rightarrow B \rightarrow A \rightarrow \text{NULL}$ 로 구성된다. (Single Linked List)
4. `int *a = malloc(8);` A를 재할당받는다.
5. *a의 payload부분에 BSS 영역의 주소를 쓴다.
6. 그럼 $B \rightarrow A \rightarrow \text{NULL}$ 에서 $B \rightarrow A \rightarrow [\text{BSS영역의주소}]$ 로 구성된다.
7. `malloc(8), malloc(8)` 할당하여 Fastbin에 $[\text{BSS영역의 주소}] \rightarrow \text{NULL}$ 로만 남게한다.
8. `int *bss = malloc(8)` 호출시 $[\text{BSS영역의 주소}]$ 가 반환됨
9. **We Can Exploit!**

1. Homework

1. 자신만의 malloc 함수를 구현

- 1.1 사용자가 맨처음 할당 요청할 때는 brk-sbrk syscall 사용
- 1.2 그 이후 요청시 사용자 정의 자료구조에서 chunk 재사용 및 재할당

2. 자신만의 free 함수를 구현

- 2.1 특정 메모리 해제시 자료구조를 만들어서 저장
- 2.2 같은 메모리 중복 해제시 alert

제출 : 32180767@dankook.ac.kr

제출 필수유무 : 필수아님 malloc 내부구조에 대해 탐구하고싶다면 구현 및 제출

Question