4. 메모리 구조와 할당

4.1 리눅스의 메모리 레이아웃 (text/data/heap/stack)

1. 개요

리눅스에서 프로세스는 실행 시 **고유한 가상 메모리 공간**을 할당받는다.
이 메모리 공간은 여러 영역으로 나뉘며, 각 영역은 서로 다른 목적과 특성을 가진다.

일반적으로 다음과 같은 레이아웃을 가진다 (낮은 주소 → 높은 주소 순):

```
+------------------+ ← 높은 주소
| Stack            |   함수 호출 시 자동 할당되는 메모리
| (grows downward) |
+------------------+
| Heap             |   동적 메모리 할당 영역 (malloc/free)
| (grows upward)   |
+------------------+
| BSS Segment      |   초기화되지 않은 전역/정적 변수
+------------------+
| Data Segment     |   초기화된 전역/정적 변수
+------------------+
| Text Segment     |   실행 코드 (명령어 영역, read-only)
+------------------+ ← 낮은 주소
```

---

2. 주요 영역 설명

📌 Text Segment (코드 영역)

- 프로그램의 **기계어 코드(명령어)**가 저장된다.
- 보통 **읽기 전용(Read-Only)** 속성을 가진다 → 코드 수정 방지
- 실행 파일에서 `.text` 섹션에 해당

예:

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

→ `main()` 함수의 기계어 코드는 **Text Segment**에 위치.

📌 Data Segment (초기화된 데이터 영역)

- **초기값이 있는 전역 변수, 정적 변수**가 저장된다.
- 프로그램 시작 시 **실행 파일에 기록된 초기값**으로 초기화됨.

예:

```
int global_var = 42;  // Data Segment에 저장됨
static int static_var = 10;
```

- 읽기/쓰기 가능

📌 BSS Segment (초기화되지 않은 데이터 영역)

- **초기값이 없는 전역 변수, 정적 변수**가 저장된다.
- 프로그램 시작 시 **0으로 자동 초기화**됨 (Zero-fill).
- 실행 파일 크기를 줄이기 위해 실제 초기화 데이터는 포함하지 않음.

예:

```
int global_uninit;      // BSS Segment
static int static_uninit;
```

📌 Heap (힙 영역)

- **동적 메모리 할당**에 사용된다 (`malloc()`, `calloc()`, `realloc()` 등).
- 런타임 중에 크기가 변할 수 있으며 **위쪽 방향으로 성장**함.

예:

```
int *p = malloc(sizeof(int) * 100);  // Heap 영역에 메모리 할당
```

- 프로그래머가 직접 **해제(free)** 해야 한다.

📌 Stack (스택 영역)

- **함수 호출 시 자동 할당되는 메모리**를 저장한다.
- **지역 변수, 함수 호출 프레임(리턴 주소, 파라미터 등)**이 저장됨.
- 함수가 호출될 때마다 스택 프레임이 **아래쪽 방향으로 성장**함.

예:

```
void func() {
    int local_var = 123;  // Stack 영역
}
```

- 함수 종료 시 스택 프레임 자동 해제.

---

3. 메모리 레이아웃 개념도

```
+---------------------+  ← 높은 주소
| Stack               |
|                     |
| (grows downward)    |
+---------------------+
| Heap                |
| (grows upward)      |
+---------------------+
| BSS Segment         |
+---------------------+
| Data Segment        |
+---------------------+
| Text Segment        |
+---------------------+  ← 낮은 주소
```

---

4. 실제 확인 방법

리눅스에서 **프로세스 메모리 맵** 확인 가능:

```
cat /proc/$(pidof your_program)/maps
```

→ 결과 예시:

```
00400000-0040b000 r-xp 00000000 fd:01 123456 /path/to/executable   → Text
0060a000-0060b000 rw-p 0000a000 fd:01 123456 /path/to/executable   → Data
0060b000-0062c000 rw-p 00000000 00:00 0                            → BSS/Heap
7ffcc2b4a000-7ffcc2b6b000 rw-p 00000000 00:00 0                    → Stack
```

---

5. 결론 🚀

- **Text Segment** → 읽기 전용 실행 코드
- **Data Segment** → 초기값 있는 전역/정적 변수
- **BSS Segment** → 초기값 없는 전역/정적 변수
- **Heap** → 동적 메모리 할당 영역
- **Stack** → 함수 호출 시 자동 할당되는 메모리

이러한 메모리 레이아웃을 이해하면 **메모리 최적화, 디버깅, 보안(스택 오버플로우 방지 등)**에 매우 유용하다.

4.2 `malloc()`, `free()`의 내부 동작

1. 개요

리눅스에서 `malloc()`와 `free()`는 **힙(Heap) 영역**의 메모리를 동적으로 할당/해제하는 함수이다.
이들은 C 언어의 표준 라이브러리 함수지만, 내부적으로는 **커널의 메모리 관리 기능과 긴밀히 연계**되어 동작한다. 🛠️

이번 절에서는:

- `malloc()` 호출 시 어떤 일이 벌어지는지
- `free()`가 메모리를 어떻게 관리하는지
- 내부에서 사용되는 커널 시스템콜
- 동적 메모리 관리 전략

까지 정리해보자.

---

2. 함수 원형

```c
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
```

---

3. 힙(Heap) 영역에서의 메모리 관리

- 프로그램 시작 시 힙 영역은 **초기 크기만큼만 확보**된다.
- `malloc()`가 요청할 때 필요한 만큼 **힙을 확장**할 수 있다.
- 확장은 커널 시스템 콜 `brk()` **또는** `mmap()`을 통해 이루어진다.

힙의 성장 방향

```
+---------------------+ ← 높은 주소
| Stack               |
+---------------------+
| Heap (grows upward) | ← malloc(), free() 사용 영역
+---------------------+
```

---

4. `malloc()`의 내부 동작 흐름

기본 흐름

```
malloc(size) 호출 →
  1️⃣ 내부 메모리 풀(Free list) 확인 →
    → 있으면 바로 할당
    → 없으면 brk() 또는 mmap() 호출로 힙 확장 →
  2️⃣ 메타데이터 기록 (크기 등) →
```

```
6     3️⃣ 사용자에게 유효 영역 포인터 반환
```

사용되는 커널 시스템콜

시스템콜	역할
`brk()`	프로세스 힙 영역 확장/축소 (전통적 방법)
`mmap()`	페이지 단위로 메모리 맵핑 (대용량 할당에 주로 사용)

---

5. 메타데이터의 존재

- `malloc()` 는 단순히 **N 바이트만 반환하지 않는다**.
- 반환한 포인터 **앞 또는 뒤에 메타데이터 영역**이 존재하여 다음 정보를 저장한다:

정보	용도
블록 크기	할당된 메모리 크기
사용 여부	현재 사용 중인지 여부
연결 포인터	Free list에서 링크 유지용 (next/prev)

→ 이 메타데이터는 `free()` **시 사용된다**.

---

6. `free()` 의 내부 동작 흐름

```
free(ptr) 호출 →
  1️⃣ ptr 앞에 위치한 메타데이터 확인 →
  2️⃣ 블록을 Free list에 추가 →
  3️⃣ 인접한 Free 블록이 있으면 병합(coalescing) →
  4️⃣ 필요 시 brk() 또는 munmap() 호출로 메모리 반납
```

중요한 점

- `free()` 호출 후에도 메모리는 **커널에 즉시 반환되지 않을 수 있음**.
- 대신 Free list에 남아 있다가 다음 `malloc()` 요청 시 재사용된다 (성능 최적화 목적).

---

7. 예제 코드 관찰하기

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

int main() {
    char *p1 = malloc(100);
    char *p2 = malloc(200);
```

```
    strcpy(p1, "Hello malloc!");
    strcpy(p2, "Another malloc!");

    printf("%s\n", p1);
    printf("%s\n", p2);

    free(p1);
    free(p2);

    return 0;
}
```

예상 내부 흐름

```
malloc(100) → Free list 확인 → brk()/mmap() 필요시 호출 → 포인터 p1 반환
malloc(200) → Free list 확인 → brk()/mmap() 필요시 호출 → 포인터 p2 반환
free(p1)    → 메타데이터 읽고 Free list에 p1 추가
free(p2)    → 메타데이터 읽고 Free list에 p2 추가
```

---

8. `malloc()`와 `mmap()` 사용 시점 차이

할당 크기	내부 전략
작거나 중간 크기 (수 kB)	`brk()` 기반 힙 확장
매우 큰 할당 (수 MB 이상)	`mmap()`으로 직접 페이지 단위 매핑

확인 방법

```
cat /proc/$(pidof your_program)/maps
```

→ `heap` 영역 또는 **Anonymous mapping** 영역에 mmap 기반 할당 확인 가능.

---

9. 동적 메모리 관리 알고리즘

일반적인 malloc 구현체는 **빠르고 메모리 단편화를 줄이는 전략**을 쓴다:

- **Free list** 관리 (미리 할당된 블록 재사용)
- **First-fit**, **Best-fit**, **Segregated free lists** 등 다양한 전략 존재
- **병합(Coalescing)** 로 인접한 Free 블록 통합

glibc는 기본적으로 **ptmalloc2** 기반으로 동작하며 매우 복잡한 최적화가 적용되어 있다.

---

10. 결론 🚀

함수	내부 동작 요약
`malloc(size)`	Free list 확인 → 없으면 brk()/mmap() 호출 → 메타데이터 기록 후 포인터 반환
`free(ptr)`	메타데이터 확인 → Free list에 추가 → 병합 처리 후 필요 시 메모리 반환

malloc/free는 힙 영역을 효율적으로 활용하기 위한 핵심 도구이며, 내부적으로는 상당히 정교한 메모리 관리 전략을 적용하고 있다.
이해하면 **메모리 사용 최적화, 단편화 분석, 디버깅**에 큰 도움이 된다. 🛠️

4.3 `brk()`, `sbrk()` 저수준 메모리 제어

1. 개요

리눅스에서 **힙(Heap) 영역의 크기**를 직접 제어하는 고전적인 방법으로
`brk()` 와 `sbrk()` 시스템 호출이 존재한다.

- `brk()` : 프로세스 힙의 **끝 위치(Program Break)**를 지정.
- `sbrk()` : 현재 Program Break를 기준으로 **상대적으로 이동**.

오늘날에는 대부분의 `malloc()` 구현체가 `brk()` **와** `mmap()` **을 혼용**하지만, `brk()`는 **저수준에서 힙을 직접 확장/축소**하는 전통적인 방법이다.

---

2. 힙 영역과 Program Break

프로세스의 메모리 레이아웃에서 **힙 영역은 brk 포인터에 의해 경계가 정의**된다:

```
+------------------------+ ← 높은 주소
| Stack                  |
+------------------------+
| Heap (grows upward)    |
| ↑ Program Break        |
+------------------------+
| BSS Segment            |
+------------------------+
| Data Segment           |
+------------------------+
| Text Segment           |
+------------------------+ ← 낮은 주소
```

Program Break → 힙 영역의 **맨 끝 주소**.

---

3. 함수 원형

```
#include <unistd.h>

void *sbrk(intptr_t increment);

#include <unistd.h>
#include <sys/types.h>

int brk(void *end_data_segment);
```

함수	기능
`sbrk(increment)`	Program Break를 **상대적으로 이동** (`increment`만큼 증/감)
`brk(addr)`	Program Break를 **절대 주소로 설정**

반환값

- 성공 시 **새로운 Program Break 위치** 반환
- 실패 시 `-1` 반환 (errno 설정)

---

4. `sbrk()` 사용 예제

아래 예제는 `sbrk()`로 힙 영역을 확장한 후, 동적 메모리처럼 사용하는 예제이다. 🚀

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

int main() {
    void *init_brk = sbrk(0);  // 현재 Program Break 위치
    printf("Initial Program Break: %p\n", init_brk);

    // 힙 영역 4096 bytes(1 page) 확장
    void *new_brk = sbrk(4096);
    printf("Program Break after sbrk(4096): %p\n", sbrk(0));

    // 사용 예시
    char *buf = (char *)new_brk;
    buf[0] = 'H';
    buf[1] = 'i';
    buf[2] = '\0';
    printf("Buffer content: %s\n", buf);

    // 다시 4096 bytes 줄임 (원상복구)
    sbrk(-4096);
    printf("Program Break after sbrk(-4096): %p\n", sbrk(0));

    return 0;
```

```
}
```

5. 실행 예시

```
Initial Program Break: 0x555555756000
Program Break after sbrk(4096): 0x555555757000
Buffer content: Hi
Program Break after sbrk(-4096): 0x555555756000
```

6. `brk()` 사용 예제

brk()는 절대 주소 설정이므로 보통 아래와 같이 활용된다:

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

int main() {
    void *init_brk = sbrk(0);
    printf("Initial Program Break: %p\n", init_brk);

    void *target_brk = init_brk + 4096;
    if (brk(target_brk) == 0) {
        printf("Program Break moved to: %p\n", sbrk(0));
    } else {
        perror("brk");
    }

    // 다시 원위치
    brk(init_brk);
    printf("Program Break restored to: %p\n", sbrk(0));

    return 0;
}
```

7. 차이점: `sbrk()` vs `brk()`

비교 항목	`sbrk()`	`brk()`
동작 방식	Program Break **상대적 이동**	Program Break **절대 주소 지정**
반환값	이동 전 Program Break	성공 시 0, 실패 시 -1
용도	malloc() 내부 구현에서 자주 사용	저수준 메모리 조작 시 직접 사용 가능

8. 현대적 관점

- 오늘날 glibc의 `malloc()`는 **작은 할당 → brk() 기반**, **큰 할당 → mmap() 기반** 혼용 사용.
- `sbrk()` / `brk()`는 더 이상 권장되지 않고 주로 **호환성 유지, 연구용, 테스트용**으로 사용됨.
- 새 코드에서는 일반적으로 **mmap() + munmap()** 기반 메모리 관리가 더 유연하고 안전하다.

---

9. 결론 🚀

개념	설명
Program Break	힙 영역의 끝 위치
`sbrk()`	Program Break를 상대적으로 이동
`brk()`	Program Break를 절대 주소로 설정
용도	힙 확장, 메모리 할당 구현(malloc 내부), 고전적인 메모리 제어

이러한 저수준 제어를 이해하면 **메모리 구조 분석, malloc 내부 이해, 디버깅** 등에 큰 도움이 된다.
운영체제의 **가상 메모리 모델과 사용자 공간 힙 관리 원리**를 파악하는 데 필수적인 기초 지식이다. 🛠️

4.4 `mmap()`을 통한 메모리 매핑

1. 개요

`mmap()`은 리눅스에서 제공하는 매우 강력한 시스템 호출로, **가상 메모리 공간에 파일 또는 익명 메모리 영역을 직접 매핑**할 수 있게 해준다.

- 고성능 서버나 DB, 멀티미디어 프로세싱 시스템 등에서 **파일 I/O 최적화** 및 **메모리 직접 관리**에 널리 사용됨.
- 현대 `malloc()` 구현에서도 큰 블록 할당 시 `mmap()` 사용.

활용 사례

용도	예시
파일의 내용을 메모리에 직접 매핑	고속 파일 읽기/쓰기
익명 메모리 매핑(Heap 확장용)	고성능 동적 메모리
프로세스 간 메모리 공유	IPC(Shared Memory)
메모리 보호 테스트	read-only 영역 생성

---

2. 함수 원형

```c
#include <sys/mman.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

인자 설명

인자	의미
`addr`	매핑할 메모리 주소(보통 `NULL` 사용하여 자동 선택)
`length`	매핑할 크기(바이트 단위)
`prot`	보호 속성 (`PROT_READ`, `PROT_WRITE`, `PROT_EXEC`, `PROT_NONE`)
`flags`	매핑 속성 (`MAP_SHARED`, `MAP_PRIVATE`, `MAP_ANONYMOUS` 등)
`fd`	파일 디스크립터 (익명 매핑 시 `-1`)
`offset`	파일의 어느 위치에서부터 매핑할지 지정

반환값

- 성공 시 **매핑된 메모리 영역의 시작 주소** 반환
- 실패 시 `MAP_FAILED` (`(void *) -1`) 반환

---

3. 파일 매핑 예제

목표

- 텍스트 파일을 메모리에 직접 매핑
- `printf()` 로 파일 내용을 출력

코드

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main() {
    int fd = open("example.txt", O_RDONLY);
    if (fd == -1) {
```

```
        perror("open");
        return 1;
    }

    struct stat st;
    if (fstat(fd, &st) == -1) {
        perror("fstat");
        close(fd);
        return 1;
    }

    size_t size = st.st_size;
    char *map = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
    if (map == MAP_FAILED) {
        perror("mmap");
        close(fd);
        return 1;
    }

    // 메모리에서 직접 출력
    write(STDOUT_FILENO, map, size);

    munmap(map, size);
    close(fd);

    return 0;
}
```

결과

- 파일 전체가 **메모리로 직접 매핑**됨
- `read()` / `write()` 없이도 파일 내용을 메모리에서 즉시 접근 가능

---

4. 익명 메모리 매핑 예제

- **파일이 아닌 순수 메모리 할당** 가능 → Heap처럼 사용

코드

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <unistd.h>

int main() {
    size_t size = 4096;  // 1 page

    char *map = mmap(NULL, size, PROT_READ | PROT_WRITE,
                     MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (map == MAP_FAILED) {
        perror("mmap");
```

```
        return 1;
    }

    // 메모리 사용
    sprintf(map, "Hello from mmap!\n");
    printf("%s", map);

    munmap(map, size);

    return 0;
}
```

결과

```
Hello from mmap!
```

→ **mmap을 malloc 대체처럼 사용 가능**
→ 고성능 서버에서 대용량 메모리 할당 시 유용

---

5. 주요 `flags` 설명

플래그	설명
`MAP_SHARED`	매핑 영역을 다른 프로세스와 공유 (파일 내용도 수정됨)
`MAP_PRIVATE`	매핑 영역을 프로세스 내에서만 사용, 수정 시 복사 발생 (Copy-on-Write)
`MAP_ANONYMOUS`	파일과 무관한 메모리 할당 (`fd` 는 -1)

---

6. mmap vs brk vs malloc

비교 항목	mmap	brk	malloc
기본 목적	고성능 메모리 매핑 / 대용량 메모리	힙 영역 확장	동적 메모리 할당
관리 주체	커널	커널	glibc 내부 구현 (brk + mmap 혼용)
대용량 할당	효율적	비효율적	mmap 사용
공유 메모리 지원	가능 (MAP_SHARED)	불가	불가

---

7. 결론 🚀

`mmap()` 은 리눅스에서 **메모리 관리의 핵심 도구**로 매우 강력하다.

- **파일 직접 매핑** → 고속 I/O
- **익명 메모리 매핑** → 동적 메모리 고성능 처리

- **프로세스 간 공유 메모리 구현 가능**

`malloc()`는 내부적으로 작은 블록은 `brk()` 기반, 큰 블록은 `mmap()` 기반으로 동작하는 경우가 많다.
따라서 **mmap에 대한 이해는 고성능 메모리 관리 및 최적화의 필수 전제**이다. 🛠️

4.5 `/proc/[pid]/maps` 해석법

1. 개요

리눅스에서는 `/proc` 가상 파일 시스템을 통해 **커널 내부 정보를 사용자 공간에서 확인**할 수 있다.

그중 `/proc/[pid]/maps` 파일은:

- 프로세스의 **가상 메모리 맵**을 보여준다.
- 어떤 메모리 영역이 어떤 목적으로 사용되고 있는지 파악할 수 있다.
- **디버깅, 메모리 분석, 보안 연구** 등에 매우 유용하다. 🛠️

---

2. 기본 형식

```
주소 범위              권한     오프셋    디바이스    inode    경로
--------              ----    ------   --------   -----    ----
00400000-00452000  r-xp    00000000 08:01      123456   /bin/bash
00651000-00652000  r--p    00051000 08:01      123456   /bin/bash
00652000-0065b000  rw-p    00052000 08:01      123456   /bin/bash
...
7fffa2b3d000-7fffa2b5e000 rw-p    00000000 00:00      0            [stack]
```

---

3. 각 필드 의미

📌 1️⃣ 주소 범위

```
00400000-00452000
```

- **가상 메모리 주소**의 시작과 끝
- 이 범위 내에서만 접근 가능

📌 2️⃣ 권한

```
r-xp
```

위치	의미
1번째	r : 읽기 가능(Readable)
2번째	w : 쓰기 가능(Writable)
3번째	x : 실행 가능(Executable)

위치	의미
4번째	p : Private (Copy-on-Write), s : Shared

📌 3️⃣ 오프셋

```
00000000
```

- 파일과 매핑된 경우 → 파일의 어느 위치에서부터 매핑이 시작되었는지
- 익명 메모리의 경우 0

📌 4️⃣ 디바이스

```
08:01
```

- 장치 번호 (주 번호:부 번호)

📌 5️⃣ inode

```
123456
```

- 매핑된 파일의 inode 번호
- **0** → 익명 메모리 영역 (Heap, Stack 등)

📌 6️⃣ 경로

```
/bin/bash
[heap]
[stack]
[vvar], [vdso], [vsyscall] 등
```

- 매핑된 영역이 어떤 것인지 설명
- 익명 메모리 영역은 대괄호로 표시

---

4. 주요 영역 해석

영역명	용도
Text (.text)	실행 코드 (보통 r-xp)
Data (.data)	초기화된 전역/정적 변수 (rw-p)
BSS (.bss)	초기화되지 않은 전역/정적 변수 (rw-p)
Heap ([heap])	malloc 등 동적 할당 영역
Stack ([stack])	함수 호출 시 사용하는 스택

영역명	용도
Shared libraries	동적 라이브러리 영역 (libc 등)
mmap anonymous	mmap 기반 익명 메모리
vdso	Virtual Dynamic Shared Object (가상 영역)

---

5. 실습 예시

프로세스 PID 확인

```
pidof bash
```

maps 출력

```
cat /proc/$(pidof bash)/maps
```

샘플 출력

```
00400000-00452000 r-xp 00000000 08:01 123456 /bin/bash
00651000-00652000 r--p 00051000 08:01 123456 /bin/bash
00652000-0065b000 rw-p 00052000 08:01 123456 /bin/bash
00e3f000-00e60000 rw-p 00000000 00:00 0      [heap]
7fffa2b3d000-7fffa2b5e000 rw-p 00000000 00:00 0      [stack]
7ffff7ffa000-7ffff7ffd000 r--p 00000000 00:00 0      [vvar]
7ffff7ffd000-7ffff7fff000 r-xp 00000000 00:00 0      [vdso]
```

---

6. 참고: Stack과 Heap 영역의 변화 관찰

Stack

- 프로그램 실행 중 **스택 프레임 추가/제거**에 따라 Stack 영역 주소가 변할 수 있음

Heap

- `malloc()` 호출 시 Heap 영역 위쪽으로 성장

관찰 예

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

int main() {
    printf("Program Start\n");

    char *p1 = malloc(1000);
    char *p2 = malloc(1000);

```

```
    printf("Allocated p1: %p\n", p1);
    printf("Allocated p2: %p\n", p2);

    getchar();  // pause → maps 파일 직접 관찰 가능

    free(p1);
    free(p2);

    return 0;
}
```

→ `/proc/[pid]/maps`에서 Heap 영역 크기 변화를 관찰할 수 있다.

---

7. 결론 🚀

요소	의미
`/proc/[pid]/maps`	프로세스 가상 메모리 구조 확인
Text/Data/BSS/Heap/Stack	프로세스 메모리 영역 구분
권한 필드 (rwxp)	메모리 영역 보호 속성 확인
Heap/Stack 성장	런타임 시 변화 가능

/proc/[pid]/maps는 **디버깅, 메모리 최적화, 보안 분석**에서 매우 유용한 도구이다.
프로그램의 **실행 시 메모리 구조를 이해**하는 중요한 기초가 된다. 🛠️

실습

`malloc` 없이 `sbrk()`로 메모리 풀 구현

1. 개요

`malloc()`는 내부적으로 힙 영역을 관리하지만, 우리는 **직접 `sbrk()`를 사용해서 메모리를 관리하는 간단한 메모리 풀 (memory pool)**을 구현할 수 있다.

기본 아이디어

- `sbrk()`를 호출해서 일정 크기의 메모리 영역을 확보
- 메모리 풀에서 원하는 크기만큼 **간단하게 잘라서 할당** (`malloc`처럼 작동)
- `free`는 이번 예제에서는 **단순 버전**으로 구현하고, 고급 버전에서는 free list를 따로 구현 가능

---

2. `sbrk()` 복습

```
void *sbrk(intptr_t increment);
```

- **현재 Program Break를 `increment` 만큼 이동** → 메모리 확보
- 반환값: 이전 Program Break

---

3. 메모리 풀 설계 (단순 버전)

구성	설명
초기화	`POOL_SIZE` 만큼 `sbrk()`로 메모리 확보
메타데이터	현재 사용 중인 오프셋 저장
할당	요청 시 메타데이터 업데이트 후 포인터 반환
해제	단순 버전에서는 미구현 (고급 버전에서 free list 필요)

---

4. 코드 예제: sbrk 기반 메모리 풀 구현

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

#define POOL_SIZE (1024 * 1024)  // 1MB

static void *pool_start = NULL;
static void *pool_end = NULL;
static void *pool_curr = NULL;

// 초기화 함수
void init_pool() {
    pool_start = sbrk(0);
    if (sbrk(POOL_SIZE) == (void *) -1) {
        perror("sbrk");
        return;
    }
    pool_end = pool_start + POOL_SIZE;
    pool_curr = pool_start;
    printf("Memory pool initialized: %p - %p\n", pool_start, pool_end);
}

// 메모리 할당 함수
void *my_malloc(size_t size) {
    // 간단하게 8바이트 단위 정렬
    size = (size + 7) & ~7;

```

```
    if (pool_curr + size > pool_end) {
        printf("Out of memory!\n");
        return NULL;
    }

    void *ptr = pool_curr;
    pool_curr += size;
    return ptr;
}

// "해제" 함수 (단순 버전에서는 동작하지 않음)
void my_free(void *ptr) {
    // 구현 안 함 (pool 전체 초기화 필요 시 reset 구현 가능)
}

int main() {
    init_pool();

    char *p1 = (char *)my_malloc(100);
    char *p2 = (char *)my_malloc(200);
    int *arr = (int *)my_malloc(sizeof(int) * 50);

    strcpy(p1, "Hello from my_malloc!");
    printf("%s\n", p1);

    for (int i = 0; i < 50; i++) {
        arr[i] = i * i;
    }

    printf("arr[10] = %d\n", arr[10]);

    // my_free()는 이번 단순 버전에서는 사용 안 함

    return 0;
}
```

5. 결과 예시

```
Memory pool initialized: 0x55e84b9ba000 - 0x55e84babaa00
Hello from my_malloc!
arr[10] = 100
```

6. 한계 및 확장

항목	현재 버전	고급 버전 가능
free 구현	❌	free list 구현 가능
블록 병합	❌	가능

항목	현재 버전	고급 버전 가능
메타데이터	없음	블록 크기, 사용 여부 기록 가능
동시성 처리	❌	lock 추가 가능

👉 **고급 버전**에서는 free list와 블록 메타데이터(헤더)를 붙이고 **malloc/free** 호환 인터페이스로도 구현할 수 있다.

---

7. 결론 🚀

- `sbrk()`로 **직접 메모리 풀을 초기화하고 관리** 가능
- 간단한 `my_malloc()` 구현만으로도 기본 할당 기능을 대체 가능
- `free()`와 재사용 기능을 추가하면 **커스텀 메모리 관리기** 구현도 가능

이런 구조는:

- **게임 엔진**
- **실시간 시스템**
- **고성능 서버** 등에서 **고정 크기 메모리 풀 관리기** 구현 시 많이 사용된다.

`mmap()`을 사용한 파일 메모리 맵핑

1. 개요

`mmap()` 시스템 호출을 사용하면 **파일을 가상 메모리 주소 공간에 직접 매핑**할 수 있다.
이렇게 하면 파일 내용을 디스크 I/O 없이 **메모리처럼 읽고 쓰는 것처럼** 사용할 수 있다.

장점

- 파일 내용을 **빠르게 접근** (페이지 단위 I/O)
- **I/O 시스템콜(read/write) 호출 최소화** → 커널이 페이지 캐시 관리
- 메모리 접근처럼 동작 → 포인터 연산 가능
- **큰 파일도 필요 시 일부만 접근 가능**

---

2. 함수 원형

```
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

주요 인자

인자	의미
addr	매핑할 가상주소 (NULL 사용 권장 → 자동 할당)
length	매핑할 크기 (bytes)
prot	보호 속성 (`PROT_READ` , `PROT_WRITE` , `PROT_EXEC` , `PROT_NONE`)
flags	매핑 특성 (`MAP_SHARED` , `MAP_PRIVATE` , `MAP_ANONYMOUS` 등)
fd	파일 디스크립터 (익명 매핑 시 -1)
offset	파일의 어느 위치에서부터 매핑할지 지정 (페이지 크기 단위 정렬 필요)

반환값

- 성공 시 **매핑된 메모리 영역 시작 주소**
- 실패 시 `(void *) -1` (`MAP_FAILED`) 반환

---

3. 예제: 파일 읽기용 메모리 매핑

목표

- 텍스트 파일 `example.txt` 를 메모리로 매핑
- 메모리에서 직접 **내용 출력**

코드

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main() {
    int fd = open("example.txt", O_RDONLY);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    struct stat st;
    if (fstat(fd, &st) == -1) {
        perror("fstat");
        close(fd);
        return 1;
    }

```

```
    size_t size = st.st_size;

    char *map = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
    if (map == MAP_FAILED) {
        perror("mmap");
        close(fd);
        return 1;
    }

    // 메모리에서 직접 파일 내용 출력
    write(STDOUT_FILENO, map, size);

    munmap(map, size);
    close(fd);

    return 0;
}
```

---

4. 실행 결과 예시

```
gcc mmap_file.c -o mmap_file
./mmap_file
```

```
[example.txt 파일 내용 출력됨]
```

---

5. 주요 포인트

동작 단계	설명
`open()`	파일 디스크립터 얻기
`fstat()`	파일 크기 확인
`mmap()`	파일 내용을 가상 메모리 영역에 매핑
`write()`	메모리 내용을 STDOUT으로 출력
`munmap()`	매핑 해제
`close()`	파일 디스크립터 닫기

---

6. `MAP_SHARED` vs `MAP_PRIVATE`

옵션	의미
`MAP_SHARED`	파일과 메모리 매핑이 공유됨 → 메모리에서 변경 시 파일에도 반영됨

옵션	의미
`MAP_PRIVATE`	Copy-on-Write → 메모리에서 변경해도 파일에는 반영 안 됨

---

7. 예제: 파일 쓰기용 매핑

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>

int main() {
    int fd = open("example.txt", O_RDWR);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    struct stat st;
    if (fstat(fd, &st) == -1) {
        perror("fstat");
        close(fd);
        return 1;
    }

    size_t size = st.st_size;

    char *map = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (map == MAP_FAILED) {
        perror("mmap");
        close(fd);
        return 1;
    }

    // 파일 맨 앞부분 수정
    strcpy(map, "HELLO mmap!\n");

    // 변경 내용 반영 보장 (선택적 → OS에 따라 자동 동기화 되기도 함)
    msync(map, size, MS_SYNC);

    munmap(map, size);
    close(fd);

    return 0;
}
```

→ 실행 후 `example.txt` 파일 내용 수정됨! 🚀

---

8. 결론 🚀

기능	`mmap()` 활용 시 효과
빠른 파일 읽기	커널 페이지 캐시를 활용하여 빠른 메모리 접근
대용량 파일 처리	부분 매핑으로 효율적 I/O 가능
메모리처럼 파일 수정	파일 내용을 메모리에서 다루듯이 처리 가능
IPC (Shared Memory)	프로세스 간 공유 메모리 구현 가능 (`MAP_SHARED` + `mmap`)

`mmap()` 은 **파일 I/O 최적화**, **메모리 관리**, **프로세스 간 통신** 등 다양한 시스템 프로그래밍에서 매우 강력한 도구이다.

힙 사이즈 변경 관찰 실험

1. 실험 목적

- `malloc()` **호출 → 힙 영역 확장 여부 확인**
- `free()` **호출 → 힙 영역 축소 여부 확인** (`brk()` 기반일 때는 일부만 축소, `mmap()` 기반 블록은 `munmap()` 호출로 바로 해제됨)
- `/proc/[pid]/maps` 를 통해 직접 메모리 매핑 변화 관찰 🚀

---

2. 실험용 코드

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

#define PAGE_SIZE (4096)

void pause_for_maps(const char *msg) {
    printf("\n=== %s ===\n", msg);
    printf("PID: %d → Check /proc/%d/maps\n", getpid(), getpid());
    printf("Press Enter to continue...\n");
    getchar();
}

int main() {
    pause_for_maps("Initial state (before malloc)");

    void *p1 = malloc(PAGE_SIZE * 10);  // 약 10 페이지 할당 (약 40KB)
    pause_for_maps("After malloc(p1)");

    void *p2 = malloc(PAGE_SIZE * 100);  // 약 100 페이지 할당 (약 400KB)
    pause_for_maps("After malloc(p2)");

    free(p1);
    pause_for_maps("After free(p1)");

```

```
    free(p2);
    pause_for_maps("After free(p2)");

    return 0;
}
```

---

3. 실행 방법

컴파일

```
gcc heap_test.c -o heap_test
```

실행

```
./heap_test
```

별도 터미널에서 관찰

```
cat /proc/$(pidof heap_test)/maps | grep heap
```

또는 전체 보기:

```
cat /proc/$(pidof heap_test)/maps
```

---

4. 예상 결과

초기 상태 (before malloc)

```
00400000-0040b000 r-xp ... /path/to/heap_test
...
<no heap segment or small heap segment>
```

After malloc(p1)

```
...
00e25000-00e46000 rw-p 00000000 00:00 0 [heap]   ← heap 영역 확장됨
```

After malloc(p2)

```
...
00e25000-00ea6000 rw-p 00000000 00:00 0 [heap]   ← heap 영역 더 확장됨
```

After free(p1) / free(p2)

- **glibc의 malloc/free 구현(ptmalloc2)**는 free된 블록을 내부 **free list에 남겨 재사용**할 뿐,
  `brk()`**를 줄여 heap 영역을 바로 축소하진 않음** (→ 다시 malloc 시 reuse).

→ 결과적으로 heap 영역 크기가 줄어들지 않고 그대로 유지되는 경우가 많음.

→ 반면에 **큰 블록 할당 시** `mmap()` **기반 할당이 발생**하면, 해당 블록은 free 시 `munmap()`으로 바로 해제됨 (maps에서 사라짐).

---

5. 참고: mmap 기반 malloc 확인법

- **큰 malloc 요청(수 MB 이상)** → `mmap()` 사용
- `/proc/[pid]/maps`에서 **anonymous mapping** 영역으로 확인 가능:

```
7f65cd04c000-7f65cd24c000 rw-p 00000000 00:00 0
```

→ free 호출 시 해당 영역이 없어짐.

---

6. 결론 🚀

동작	힙 변화
작은 malloc 호출	heap 영역 확장 (brk() 기반)
작은 free 호출	heap 영역 축소 거의 없음 (free list 재사용 목적)
큰 malloc 호출	mmap() 기반 anonymous 영역 생김
큰 free 호출	해당 mmap 영역 바로 해제됨 (`munmap`)

glibc malloc은 매우 최적화되어 있어, 단순 `free()` 호출만으로 heap 영역이 눈에 띄게 줄어들지는 않음.
→ **메모리 reuse를 위한 성능 최적화 설계** 때문이다.