10장 메모리 관리

10.1 변수와 메모리의 이해

프로세스

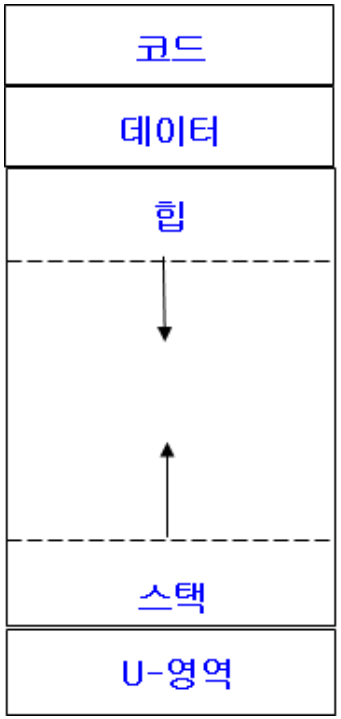
- 프로세스는 실행중인 프로그램이다.

- 프로그램 실행을 위해서는
  - 프로그램의 코드, 데이터,
  - 스택, 힙,
  - U-영역(meta data for process, such as struct user) 등이 필요하다.

- 프로세스 이미지(구조)는 메모리 내의 프로세스 레이아웃

프로세스 구조

- 프로세스 구조



스택은 엄격한 LIFO 구조이나,
힙은 요구 순서에 일정한 규칙이 없다.

- 코드 세그먼트(code segment)
  - 기계어 명령어
  - 프로그램 코드 및 리터럴 상수 저장

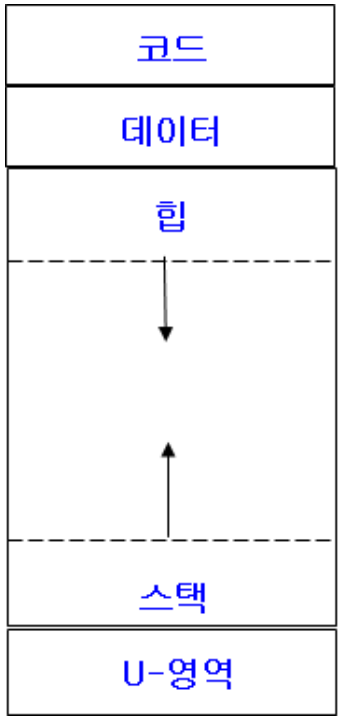
- 데이터 세그먼트(data segment)
  - 프로그램의 시작~종료까지 유지되는 데이터 저장
  - 전역 변수, 정적 변수

  - 심볼릭 상수, 매크로 상수, 문자열 상수 저장
  - **rodata** 세그먼트 공간
  - Symbolic constant (const double PI=3.14;)
  - Macro constant (#define MAX 100)
  - string constant (char* p = “hello”;)

  - 초기화 되지 않은 전역 변수 공간
  - **bss** 영역 : 프로그램이 시작되면 0으로 초기화
  - E.g. int maxcount = 99; (initialized)
  - E.g. long sum[1000]; (uninitialized)

프로세스 구조

- 프로세스 구조



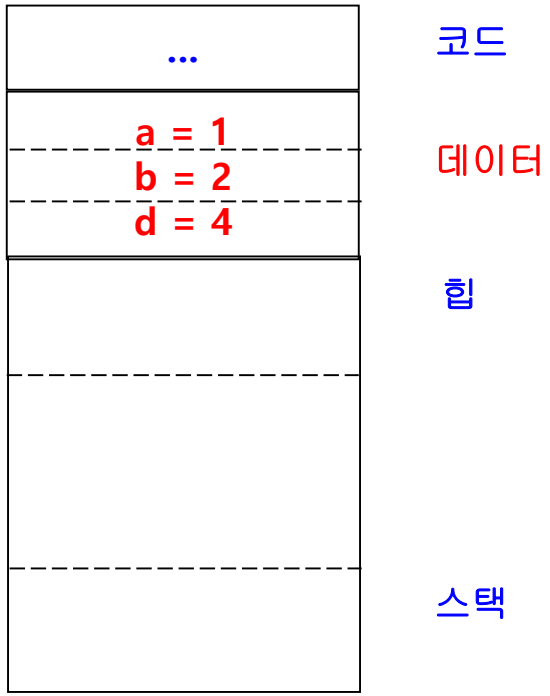
스택은 엄격한 LIFO 구조이나,
힙은 요구 순서에 일정한 규칙이 없다.

- 스택(stack) //시스템이 할당 해제 관리
  - 실행 시간 스택으로 함수가 호출될 때마다 생성.
  - 활성 레코드(activation record) 혹은 스택 프레임(stack frame)이라 함
  - 지역 변수, 매개 변수, 반환주소, 반환값 등 저장
  - 운영체제에 의해서 관리

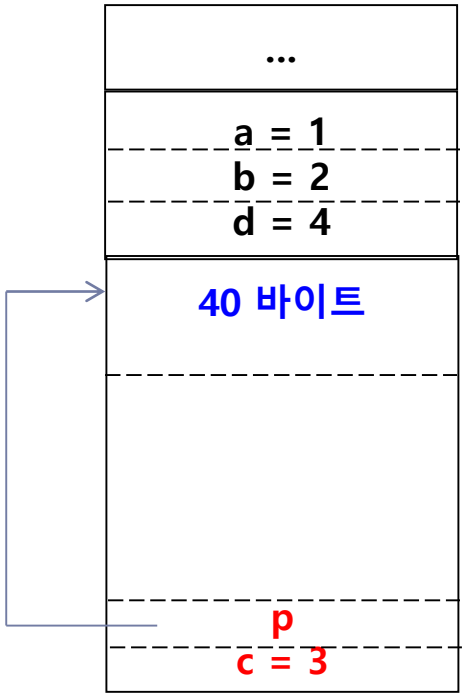
- 힙(heap) //개발자가 할당 해제 관리
  - 동적 메모리 할당
  - Malloc() in C
  - 사용자에 의해서 관리

프로그램 시작 시점의 메모리

```
#include <stdio.h>
#include <stdlib.h>
int a = 1;
static int b = 2;

int main() {
    int c = 3;
    static int d = 4;
    char *p;

    p = (char *) malloc(40);
    fun(5);
}

void fun(int n)
{
    int m = 6;
     ...
}
```



main() 함수 실행할 때 메모리 영역



Main 함수도 하나의 함수로 취급하여 스택에 저장

코드

데이터

Heap : malloc() 함수 사용으로 힙 영역 할당
동적 변수 (dynamic variable)이라 함

→ 동적 메모리 할당의 이유 : 필요시 사용하고 해제, 메모리 절약

Stack :
main() 함수의 인자 및 지역 변수 저장 공간 (int c, char* p)
func() 함수가 추가로 실행되면 매개변수(인자) n, 지역변수 m 이 stack에 추가적으로 생성되고 실행 종료 시, 자동 삭제

→ 이와 같이 함수 호출/복귀에 따라 자동으로 할당/해제 되는 변수를 C 언어에서 자동 변수(automatic variable)라 함

DATA 영역의 이해 (문자열 상수, 배열 vs. 포인터)

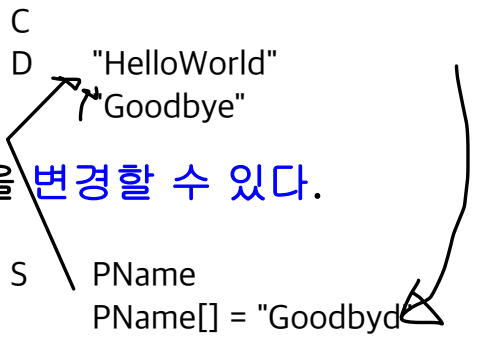
배열 : 스택에 정적으로 할당
포인터 : 읽기 전용 메모리 영억에 저장
고정된 메모리 공간

```
char* pName1 = "HelloWorld";
strcpy(pName1, "Goodbye");
```

이와같이 포인터 변수를 통하여 문자열 상수를 변경하려고 하면,
컴파일 오류는 발생하지 않지만 운영체제에서 오류가 발생하여 프로그램의 실행이 중지된다.
메모리 관리에서 오류

```
char* pName1 = "HelloWorld";
pName1 = "Goodbye";
```

데이터 세그먼트끼리
값 변경 가능
포인터(읽기전용) -> stack 수정 X



포인터 변수 p는 데이터 세그먼트에 있으므로, 포인터 변수의 값(주소)을 변경할 수 있다.
따라서 문자열 상수의 주소를 p에 저장할 수 있다.

stack끼리 수정 불가능
stack(읽기전용) ->포인터 수정 X

```
char pName[] = "HelloWorld";
pname = "Goodbye";
```

배열 이름은 포인터(주소) 상수이다. 주소값을 새로운 문자열(주소)로 변경할 수 없다.

```
char pName[] = "HelloWorld";
strcpy(pName, "Goodbye");
```

pName은 배열이 되고, 데이터 세그먼트에 배열이 저장된다.
따라서 문자 배열은 우리가 얼마든지 내용을 변경할 수 있다.
(슬라이드 15와 동일 사례)

```
char buf[6];
buf = "apple";    //불가능 (배열은 포인터 상수)
```

상수 포인터 : 이름 자체가 첫 번재 요소의 주소를 나타냄

```
char str[4] = "abc"; //가능
```

할당 방법에 따른 변수들의 분류

	변수 구분	변수 종류
데이터 세그먼트	정적 변수	전역변수, static 변수
stack	자동 변수	지역변수, 매개변수
heap	동적 변수	힙 할당 변수

왜 지역 변수와 매개변수를 자동 변수라 하는가?
스택의 관점에서 자동으로 할당되고 해제되기 때문

10.2 동적 메모리 할당

동적 메모리 할당

- 동적 할당을 사용하는 이유

```
int num;
printf("학생수:");
scanf("%d", &num);
int score[num];
```

  - 컴파일 에러 발생
    - 변수를 배열의 크기로 사용하면 컴파일 오류
    - 컴파일 타임에 메모리 결정 불가 (배열 메모리 공간은 정적 할당)

  - 윗 문장 대신 아래의 문장으로 수정해야 함.
    - malloc(num); //실행 시간에 동적으로 크기를 입력 받음

    - 이 경우 num의 타입(크기)을 모름, 따라서 아래와 같이 크기 지정
    - malloc(num*sizeof(int));

동적 메모리 할당

- 동적 할당을 사용하는 이유
  - malloc()은 할당된 메모리의 주소를 반환
    - 할당된 메모리 주소를 받을 포인터 변수 선언 필요 (int* score)
    - 이 때 할당된 메모리의 데이터 타입과 동일한 포인터 변수
    - int* score = malloc(num*sizeof(int));
    - 포인터==배열, 따라서, score[0], score[1]... 등으로 참조 가능
    - 동적 배열 크기 가능
    - 즉, 배열 크기가 실행 시간에 결정 by num 입력 → 동적 메모리 할당

동적 메모리 할당

- 동적 할당을 사용하는 이유
  - malloc()은 주소를 반환하는데, 다양한 타입의 메모리 할당 가능
    - char, int, double 등
  - 따라서, malloc()의 포인터 타입은 void*
    - 포인터의 데이터 타입(가리키는 값의 타입)이 미리 정해져 있지 않음
    - 장점: 어떠한(모든) 데이터 타입의 변수에 대한 주소값도 가질 수 있음
    - 제약: 타입에 대한 정보가 없어서 역참조가 불가능
      해당 주소에서 몇 바이트(타입)를 읽어야 하는지 모르기 때문
    - int* score = (int*) malloc(num*sizeof(int)); → 형 변환 필수! 및 최종 형태
      (참고) 동적 할당은 완료된 상태, 하지만 malloc은 초기화를 하지는 않음

동적 메모리 할당

- 사용 후, 처리 과정
  - 메모리의 동적 할당은 힙 영역을 활용하여 생성 됨
  - 운영체제가 자체적으로 해제 못함
    - 메모리 누수(system slow) 발생, 책임 없는 개발자
  - 포인터가 가리키는 메모리가 해제
    - free(score); //free(주소); 해당 주소 영역 해제
  - C++의 new & delete

동적 메모리 할당

- 동적 메모리 할당
  - 필요할 때 필요한 만큼만 메모리를 요청해서 사용하여 메모리를 절약한다.
  - 사용한 후 더 이상 필요가 없을 경우, 해당 영역을 반납한다.

- 관련 함수
  - malloc( )
  - calloc( )
  - realloc( )

  - free( )

메모리 할당 (C 언어의 library 함수)

```
#include <stdlib.h>
void *malloc(size_t size);
```
size 바이트만큼의 메모리 공간을 할당하며 그 시작주소를 void* 형으로 반환한다.

```
void free(void *ptr);
```
포인터 p가 가리키는 메모리 공간을 해제한다.

- 힙에 동적 메모리 할당

- malloc() 함수는 메모리를 할당할 때 사용하고, free()는 할당한 메모리를 해제할 때 사용한다.
  - 반드시 malloc()으로 할당 받은 영역을 free() 해야 하며, 일부 영역만 free() 할 수는 없다.

메모리 할당 예 (40바이트 할당)

- char *ptr;
- ptr = (char *) malloc(40);

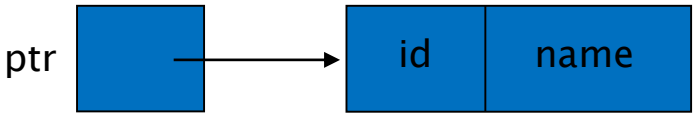
할당된 공간의 크기는 같지만, 사용은 다름

가령, i번째 변수 접근시 *(ptr+i) or ptr[i]로 접근
이때 실제 메모리 주소는 포인터 증감 연산에 따라 +1 or +4 씩 변환

- int *ptr;
- ptr = (int *) malloc(10 * sizeof(int));

구조체를 위한 메모리 할당 예

```
struct student {
        int id;
        char name[10];
};
struct student *ptr;
ptr = (struct student *) malloc(sizeof(struct student));
```



구조체 하나의 개체를 위한 메모리 할당
N개의 구조체 개체에 대한 메모리 할당 사례 → Next Slide

구조체 배열을 위한 메모리 할당 예

```
struct student *ptr;
ptr = (struct student *) malloc(n * sizeof(struct student));
```

특정한 구조체 변수의 크기 영역을 N개 할당하는 것도 가능

stud1.c

```
#include <stdio.h>
#include <stdlib.h>
struct student {
    int id;
    char name[20];
};

/* 학생 수를 입력받고 이어서 학생 정보를 입력받은 후,
이들 학생 정보를 역순으로 출력하는 프로그램 */
int main()
{
    struct student *ptr; // 동적 할당된 블록을 가리킬 포인터
    int n, i;
    printf("몇 명의 학생을 입력하겠습니까? ");    //n 명을 입력하겠다.
    scanf("%d", &n);
    if (n <= 0) {
        fprintf(stderr, "오류: 학생 수를 잘못 입력했습니다.₩n");
        fprintf(stderr, "프로그램을 종료합니다.₩n");
        exit(1);
    }
```

stud1.c

```
    ptr = (struct student *) malloc(n * sizeof(struct student));    // n명 공간 확보
    if (ptr == NULL) {
        perror("malloc");
        exit(2);
    }

    printf("%d 명의 학번과 이름을 입력하세요.₩n", n);
    for (i = 0; i < n; i++)
        scanf("%d %s₩n", &ptr[i].id, ptr[i].name);    //ptr[i]로 i번째 접근

    printf("₩n* 학생 정보(역순) *₩n");
    for (i = n-1; i >= 0; i--)
        printf("%d %s₩n", ptr[i].id, ptr[i].name);

    printf("₩n");
    exit(0);
}
```

배열 할당

- 같은 크기의 메모리를 여러 개를 할당할 경우

```
#include <stdlib.h>
void *calloc(size_t n, size_t size);
```
크기가 size인 메모리 공간을 n개 할당한다. 값을 모두 0으로 초기화 한다(malloc은 초기화 하지 않음). 실패하면 NULL를 반환한다.

  - 자료형의 크기와 개수를 따로 지정하는 것이 가능

$$\text{malloc}\ (n * \cdots) == \text{calloc}\ (n, \cdots)$$

  - 모든 바이트를 0으로 초기화
    - malloc + memset 효과

calloc() 예

```
int *p,*q;

p = malloc(10*sizeof(int));
if (p == NULL)
        perror("malloc");


q = calloc(10, sizeof(int));        // 크기를 인자로 지정하고 0으로 초기화
if (q == NULL)
        perror("calloc");
```

배열 할당

- 이미 할당된 메모리의 크기 변경

ptr = 0
->malloc와 같은 동작

newsize =0
-> free와 같은 동작

```
#include <stdlib.h>
void *realloc(void *ptr, size_t newsize);
```

ptr이 가리키는 이미 할당된 메모리의 크기를 newsize로 변경한다.
→ newsize가 0이면 free()와 같은 효과
→ ptr이 NULL이면 (즉 기존 메모리가 없으면), malloc()을 새로 수행하는것과 같은 효과

- 이미 존재하는 메모리를 새로운 크기로 변경해 줌
- 새롭게 할당된 크기의 메모리는 할당이 허용되는 한, 기존 데이터도 그대로 유지

- 주의 : 추가 크기만큼 허용 불가 시, 널 포인터 반환, 기존 메모리 영역 주소 소실 (메모리 누수)

기본 원칙:

- **메모리 크기 변경이 가능한 경우**:
  - 기존 메모리 블록의 **주소를 유지**하며, 크기를 조정합니다.
- **메모리 크기 변경이 불가능한 경우**:
  - **새로운 메모리 블록을 할당**하고, 기존 데이터를 복사한 후, 기존 메모리 블록을 해제합니다.

배열 할당

```
int * mem = malloc();
mem = realloc(); //실패 시 mem에는 null값 존재
```

```
int * mem = malloc();
/*
   process - realloc 필요
*/
// 기존 메모리 주소 백업
int * mem_temp = mem;

// 메모리 재할당
mem = realloc();

// 복구 과정
if( mem == null ) {
    mem = mem_temp;
}
```

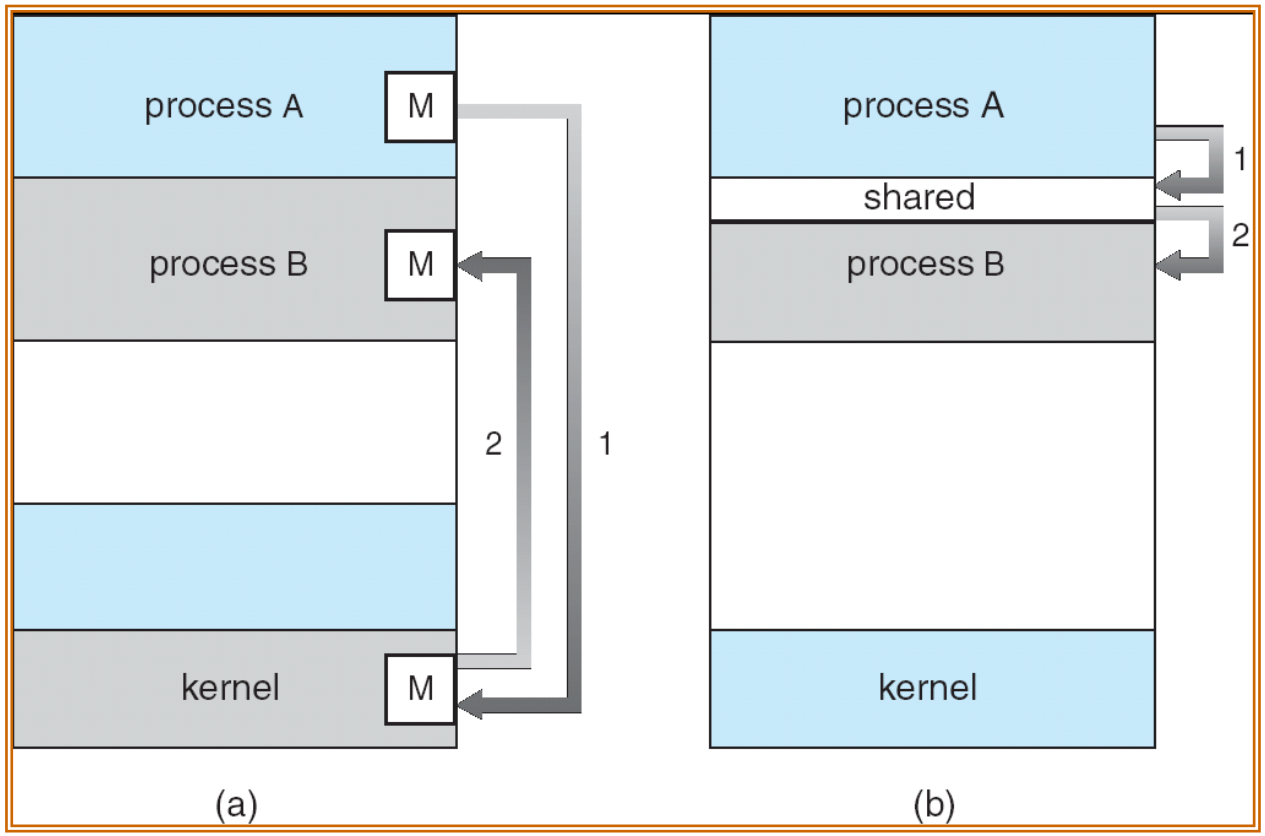
realloc() 함수를 수행하기 이전
기존 메모리 공간을 백업 해 두어야 함

10.4 공유 메모리

공유 메모리의 필요성

- 메모리 영역 보호
  - 다수의 프로세스가 동시에 실행되는 환경에서의 메모리 영역의 보호는 필수적이다.
  - 각 프로세스의 데이터, 스택, 힙 등의 영역은 개별 영역
  - 메모리 영역의 침범? SIGSEGV (segment violation)
- 프로세스 사이의 데이터 공유
  - 통신을 이용한 방법 (파이프, 소켓 등)
  - 메모리 영역의 공유(shared memory)를 통한 데이터 공유

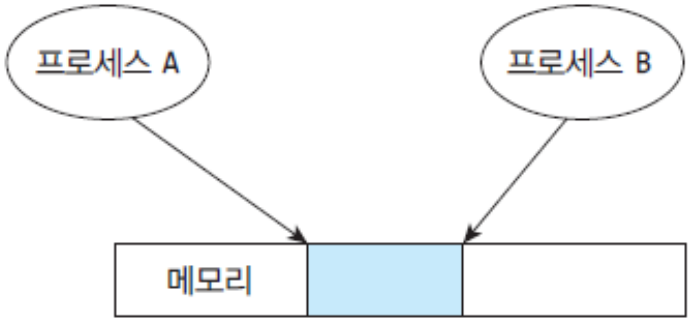
Message Passing vs. Shared Memory



send (*P, message*) – send a message to process P
receive(*Q, message*) – receive a message from process Q

공유 메모리의 필요성

- 공유 메모리
  - 프로세스 사이에 메모리 영역을 공유해서 사용할 수 있도록 해준다.



공유 메모리 관련 함수

- 공유 메모리 **생성** shmget()


```
#include <sys/shm.h>
#include <sys/ipc.h>
int shmget(key_t key, size_t size, int shmflg);
```

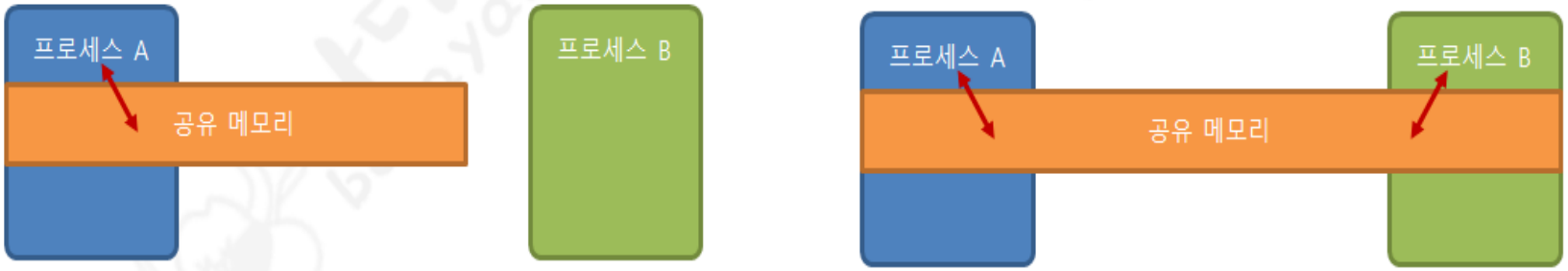
key를 사용하여 key가 가리키는 새로운 공유메모리를 size 크기만큼 생성하고, 공유 메모리의 ID를 반환한다.

공유 메모리 관련 함수

- 공유 메모리 생성 shmget()의 인자
  - key
    - IPC_PRIVATE (항상 새로운 공유 메모리 생성) 사용
    - 혹은 key 생성 함수 ftok() 함수로 생성한 키 사용
  - size : 공유 메모리의 크기
  - shmflg
    - IPC_CREATE (공유 메모리 생성 및 |0644등의 권한 추가 지정)
    - IPC_EXCL (IPC_CREATE와 함께 사용될 경우, 기존 공유 메모리가 존재하면 실패 값 리턴)
    - 0의 경우 새롭게 생성하지 않고, 기존에 생성된 공유 메모리를 얻어 옴 (e.g., shm2.c)
  - (예시) shmid = shmget (key, 1024, IPC_CREATE|0644)

공유 메모리 관련 함수

- 공유 메모리 **연결** shmat()




```
#include <sys/shm.h>
#include <sys/ipc.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

shmid 공유 메모리를 이 프로세스의 메모리 위치 shmaddr에 연결하고 그 주소를 반환한다.

공유 메모리 관련 함수

- 공유 메모리 연결 shmat()의 인자
  - shmaddr
    - NULL일 경우, 커널이 적절한 주소를 선정하여 반환
  - shmflg
    - 공유 메모리에 권한 지정
    - SHM_RDONLY : 공유 메모리를 읽기 전용으로 설정
    - 해당 flag가 없으면 (i.e., 0의 경우), 읽기/쓰기 모드 (쓰기 전용은 없음)
    - SHM_RND : shmaddr이 NULL 이 아닌 경우 사용되며, shmaddr을 반올림 하여 메모리 페이지 경계를 맞추는 역할
  - 공유 메모리의 자료형을 모르기 때문에 이 함수의 반환값은 void*
    - 아래 예시에서처럼 type casting 필요
  - (예시) shmaddr = (char *) shmat (shmid, NULL, 0)

공유 메모리 관련 함수

- 공유 메모리 연결 **해제** shmdt()

```
#include <sys/shm.h>
#include <sys/ipc.h>
int shmdt(const void *shmaddr);
```

공유 메모리에 대한 연결 주소 shmaddr를 연결해제 한다.

성공 시 0, 실패 시 -1을 반환한다.

  - shmat()에서 받은 공유 메모리 연결 포인터 shmaddr를 전달받아 공유 메모리 연결을 해제 함

공유 메모리 관련 함수

- 공유 메모리 **제어** shmctl()

```
#include <sys/shm.h>
#include <sys/ipc.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

shmid 공유메모리를 cmd 명령어에 따라 제어한다.

  - 공유 메모리를 제어하기 위해 사용
    - 공유 메모리에서 정보를 얻거나,
    - 어떤 값을 쓰거나,
    - 공유 메모리를 삭제하는 등의 조작

      참고) 삭제와 해제(분리, detach)는 서로 다른 연산임

공유 메모리 관련 함수

- 공유 메모리 제어 shmctl()

```
#include <sys/shm.h>
#include <sys/ipc.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

shmid 공유메모리를 cmd 명령어에 따라 제어한다.

- cmd
  - IPC_RMID: 공유 메모리 제거 (remove id)
  - IPC_SET: 공유 메모리 정보(사용자 ID, 그룹 ID, 접근권한)를 buf에서 지정된 값으로 설정
  - IPC_STAT: 현재 공유 메모리 정보를 buf에 저장
  - SHM_LOCK: 공유 메모리 잠금
  - SHM_UNLOCK: 공유 메모리 잠금 해제

공유 메모리 관련 함수

- 공유 메모리 제어 shmctl()
- buf : shmid_ds 구조체에 대한 포인터, <sys/shm.h>에서 정의

```
struct shmid_ds {
    struct ipc_perm shm_perm;    /* Ownership and permissions */
    size_t          shm_segsz;   /* Size of segment (bytes) */
    time_t          shm_atime;   /* Last attach time */
    time_t          shm_dtime;   /* Last detach time */
    time_t          shm_ctime;   /* Creation time/time of last
                                    modification via shmctl() */
    pid_t           shm_cpid;    /* PID of creator */
    pid_t           shm_lpid;    /* PID of last shmat(2)/shmdt(2) */
    shmatt_t        shm_nattch;  /* No. of current attaches */
    ...
};
```

공유 메모리: shm1.c

공유 메모리를 생성하고 이를 사용하여
다른 프로세스에게 메시지를 보내는 프로그램

```
 1 #include <sys/ipc.h>
 ...
 7 int main()
 8 {
 9     int shmid;
10     key_t key;
11     char *shmaddr;
12
13     key = ftok("helloshm", 1);  //공유 메모리 키 생성
14     shmid = shmget(key, 1024, IPC_CREAT|0644);   //공유 메모리 생성
15     if (shmid == -1) {
16         perror("shmget");
17         exit(1);
18     }
19
20     printf("shmid : %d", shmid);
21     shmaddr = (char *) shmat(shmid, NULL, 0);   //공유 메모리 연결
22     strcpy(shmaddr, "hello shared memory");    //연결된 주소에 메시지를
                                                  // 복사하여 공유메모리로 보냄
23     return(0);
24 }
```

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

실행 결과 및 공유 메모리 정보 확인

$shm1

shmid : 17

$ ipcs –m // 프로세스의 ipc 정보 확인 (-m 공유 메모리 세그먼트 확인 옵션)

------ Shared Memory Segments --------

key shmid owner perms bytes nattch status

0x00000000 4 chang 600 16384 1 dest

0xffffffff 17 chang 644 1024 0

…

shm1의
strcpy(shmaddr, "hello shared memory");

해당 문장은 콘솔로 특별한 출력이 제공되지 않음
shm2를 통해 확인 해 보자.

공유 메모리: shm2.c

공유 메모리를 통해서 다른 프로세스로부터 메시지를 받는 프로세스

```
 1 #include <sys/ipc.h>
...
 7 int main()
 8 {
 9     int shmid;
10     key_t key;
11     char *shmaddr;
12
13     key = ftok("helloshm", 1);  //공유 메모리 키 생성
14     shmid = shmget(key, 1024, 0);  //key가 가리키는 공유메모리 획득 (생성 X)
15     if (shmid == -1) {
16         perror("shmget");
17         exit(1);
18     }
19
20     printf("shmid : %d₩n", shmid);  //공유 메모리 id 출력
21     shmaddr = (char *) shmat(shmid, NULL, 0);  //공유 메모리 연결
22     printf("%s₩n", shmaddr);          //연결된 주소를 사용하여 공유메모리에 있는
                                          메시지 출력
24     return(0);
25 }
```

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
```

실행 결과

실행 결과

shmid : 17

hello shared memory

shm1.c에서와 동일한 id 사용

해당 프로그램에서 보낸 메시지를 읽어서 출력

부모-자식 프로세스 사이의 메모리 공유: shm3.c

```
 1 #include <sys/ipc.h>
 2 #include <sys/shm.h>
 3 #include <sys/types.h>
 4 #include <sys/wait.h>
 5 #include <unistd.h>
 6 #include <stdlib.h>
 7 #include <stdio.h>
 8
 9  int main()
10 {
11    int shmid;
12    char *shmp1, *shmp2;
13
14    shmid = shmget(IPC_PRIVATE,
          10*sizeof(char),IPC_CREAT|0666);
15    if (shmid == -1) {
16       printf("shmget failed₩n");
17       exit(0);
18    }
20  if (fork() == 0) {
21     shmp1 = (char *) shmat(shmid, NULL, 0);
22     for (int i=0; i<10; i++)
23        shmp1[i] = i*10; //연결주소에 데이터 전송
24        shmdt(shmp1);
25     exit(0);
26  } else {
27     wait(NULL);
28     shmp2 = (char *) shmat(shmid, NULL, 0);
29     for (int i=0; i<10; i++) //공유 메모리 데이터 출력
30        printf("%d ", shmp2[i]);
31     shmdt(shmp2);   //공유 메모리 연결해제
32     if (shmctl(shmid,IPC_RMID,NULL)==-1)
33        printf("shmctl failed₩n");  //공유 메모리 제거
34   }
35   return 0;
36 }
```

서로 다른 변수가 같은 메모리 영역 참조
shmp1 vs. shmp2

공유 메모리를 사용하여
자식 프로세스에서 부모 프로세스로
메시지를 보내는 프로그램

실행 결과

실행 결과

0 10 20 30 40 50 60 70 80 90

공유 메모리를 사용한 부모-자식 프로세스의 통신

공유 메모리를 사용하여 자식 프로세스에서 부모 프로세스로 메시지를 보내는 프로그램

자식 프로세스가 공유메모리를 통해 보낸 값을 부모 프로세스가 사용하는 것을 확인 함

10.5 메모리 관리 함수

C 언어에서의 메모리 관리 함수

include <string.h>

void *memset(void *s, int c, size_t n);

s에서 시작하여 n 바이트만큼 바이트 c로 설정한 다음에 s를 반환한다.

int memcmp(const void *s1, const void *s2, size_t n);

s1과 s2에서 첫 n 바이트를 비교해서, 메모리 블록 내용이 동일하면 0을 반환하고 s1이 s2보다 작으면 음수를 반환하고, s1이 s2보다 크다면 양수를 반환한다.

void *memchr(const void *s, int c, size_t n);

s가 가리키는 메모리의 n 바이트 범위에서 문자 c를 탐색한다. c와 일치하는 첫 바이트에 대한 포인터를 반환하거나,c를 찾지 못하면 NULL을 반환한다.

void *memmove(void *dst, const void *src, size_t n);

src에서 dst로 n 바이트를 복사하고, dst를 반환한다.

void *memcpy(void *dst, const void *src, size_t n);

src에서 dst로 n 바이트를 복사한다. 두 메모리 영역은 겹쳐지지 않는다. 만일 메모리 영역을 겹쳐서 쓰길 원한다면 memmove() 함수를 사용해라. dst를 반환한다.

메모리 관리 함수

- 메모리 관련 함수는 문자열 처리 함수와 유사
  - strcpy() vs. memcpy()
  - strcmp() vs. memcmp

- 차이점
  - 인자와 반환값
    - 문자열 처리함수는 대부분 char* 이지만, 메모리 처리 함수는 void*
  - 길이 확인
    - 문자열 처리 함수는 시작주소만 알려주고 끝은 NULL로 확인하여 길이를 별도로 알려줄 필요가 없음
    - 메모리 관련 함수는 시작 주소와 함께 작업 대상의 크기(길이) n을 알려 주어야 함

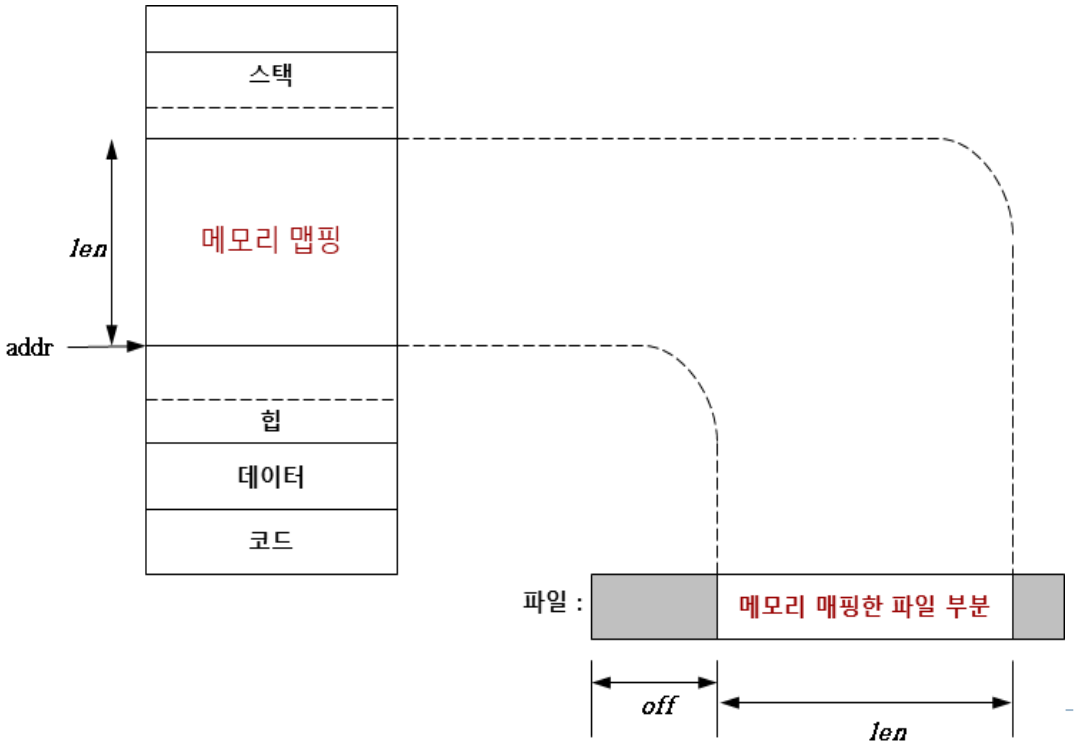
mem.c

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

void main()
{
    char str[32]="Do you like Linux?";
    char *ptr,*p;

    ptr = (char *) malloc(32);
    memcpy(ptr, str, strlen(str));
    puts(ptr);
    memset(ptr+12,'l',1);  //ptr+12(대문자 L)에서 1바이트를 소문자 l로 변경
    puts(ptr);

    p = (char *) memchr(ptr,'l',18);  //탐색
    puts(p);  //첫번째 l부터 출력
    memmove(str+12,str+7,10);      //str+7인 "like linux"를 str+12인 "linux" 자리에
    puts(str);
}
```

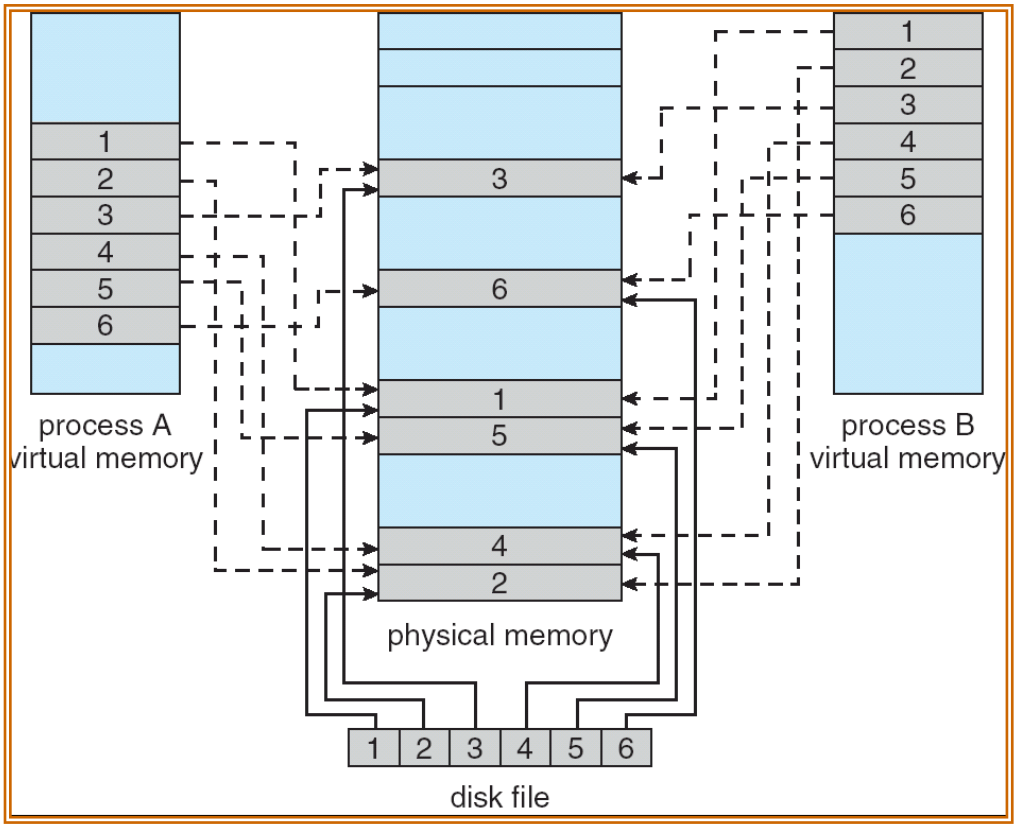
```
$ mem
Do you like Linux?
Do you like linux?
like linux?
Do you like like Linux
```

메모리 맵핑 (Memory Mapped File)

- 메모리 맵핑
  - 파일의 일부 영역에 메모리 주소를 부여할 수 있다.
  - 마치 메모리상의 변수를 사용하는 것처럼 파일을 사용할 수 있다.
  - 메모리 주소를 나타내는 포인터와 배열을 사용하여 파일의 데이터 접근 가능



메모리 맵핑 (Memory Mapped File)



→ Some parts of virtual address space of process A is designated as file area

→ File access is treated as normal page access (File I/O vs. Memory R/W)

메모리 매핑 시스템 호출

```
#include <sys/types.h>
#include <sys/mman.h>
caddr_t mmap(caddr_t addr, size_t len, int prot, int flag, int fd, off_t off);
```

fd가 나타내는 파일의 일부 영역(off부터 len 크기)에 메모리 주소를 부여하고 메모리 맵핑된 영역의 시작 주소(addr)를 반환한다.

- 매개 변수 6가지
  - addr: 메모리 맵핑에 부여할 메모리 시작 주소,
    이 값이 NULL(0)이면 시스템이 적당한 시작 주소를 선택한다.
  - len: 맵핑할 파일 영역의 크기로 메모리 맵핑의 크기와 같다.
  - prot: 매핑된 메모리 영역에 대한 보호 정책을 나타낸다.
    PROT_READ(읽기), PROT_WRITE(쓰기), PROT_EXEC(실행), PROT_NONE(접근 불가)
  - fd: 대상 파일의 파일 디스크립터
  - off: 맵핑할 파일 영역의 시작 위치 (0이면 파일 전체)

메모리 매핑 시스템 호출

```
#include <sys/types.h>
#include <sys/mman.h>
caddr_t mmap(caddr_t addr, size_t len, int prot, int flag, int fd, off_t off);
```

fd가 나타내는 파일의 일부 영역(off부터 len 크기)에 메모리 주소를 부여하고 메모리 맵핑된 영역의 시작 주소(addr)를 반환한다.

- 매개 변수 6가지
  - flag
    - MAP_FIXED : 반환 주소가 addr와 같아야 함
    - MAP_FIXED가 지정되지 않고, addr이 0이 아니면 커널은 addr를 참고만 함
    - MAP_SHARED나 MAP_PRIVATE 중 하나가 반드시 지정되어야 함
    - MAP_SHARED : 부여된 주소에 쓰면 파일에 저장된다.
    - MAP_PRIVATE : 부여된 주소에 쓰면 파일의 복사본이 생성되고 이후로 복사본을 읽고 쓰게 된다.

MAP_FIXED 하지 않고
addr로 고정 주소를 준 경우 임.

메모리 매핑을 사용한 cat 명령어 구현:mmap.c

```
 1 #include <stdio.h>
 ...
 8
 9 int main(int argc, char *argv[])
10 {
11     struct stat sbuf;
12     char *p;
13     int fd;
14
15     if (argc < 2) {
16         fprintf(stderr, "사용법: %s 파일이름₩n", argv[0]);
17         exit(1);
18     }
19
20     fd = open(argv[1], O_RDONLY);
21     if (fd == -1) {
22         perror("open");
23         exit(1);
24     }
```

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

메모리 매핑을 사용한 cat 명령어 구현:mmap.c

```
26      if (fstat(fd, &sbuf) == -1) {  //파일 정보를 sbuf에 저장 (다음 슬라이드 참고)
27          perror("fstat");
28          exit(1);
29      }
30
31      p = mmap(0, sbuf.st_size, PROT_READ, MAP_SHARED, fd, 0);
32      if (p == MAP_FAILED) {     //MAP_FAILED 상수 정의 -1
33          perror("mmap");
34          exit(1);
35      }
36
37      for (long l = 0; l < sbuf.st_size; l++)
38          putchar(p[l]);       //메모리 주소를 참조하여 파일의내용을 메모리 연산을 통해 읽고
                                   이를 모니터로 출력
39
40      close(fd);
41      munmap(p, sbuf.st_size);   // 메모리 매핑 해제  mummap (addr, len)
42      return 0;
43  }
```

파일의 전체 영역을 메모리 맵핑 함
(offset 0부터 파일의 크기까지의 길이)

stat() 함수 : 파일 상태를 출력하는 시스템 호출 함수

- 파일 하나당 하나의 i-노드가 있으며,
  i-노드 내부에 파일에 대한 모든 상태 정보가 저장되어 있다.
- i-노드에 저장되어 있는 상태정보를 가져오는 역할

```
#include <sys/types.h>
#include <sys/stat.h>
int stat (const char *filename, struct stat *buf);  (파일이름으로지정)
int fstat (int fd, struct stat *buf);   (대상 파일을 fd로 지정)
int lstat (const char *filename, struct stat *buf);  (링크 자체의 stat)
```

기본적으로 stat()과 동일, 단 대상 파일이 심볼릭 링크일 경우, 링크가 가리키는 파일이 아니라 링크 자체에 대한 정보를 가져옴

파일의 상태 정보를 가져와서 **stat 구조체 buf에 저장**. 성공하면 0, 실패하면 -1을 리턴한다.

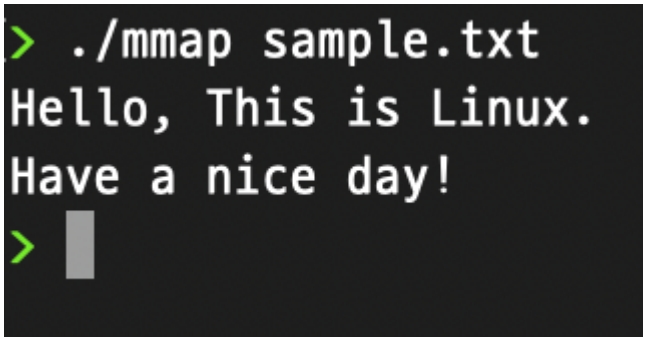
stat 구조체

- stat( ) 시스템 호출은 파일의 정보를 가져와 **stat 구조체**에 저장하는 역할을 수행 함

```
struct stat {
    mode_t st_mode;          // 파일 타입(slide 27~28)과 사용권한(slide 31)
    ino_t st_ino;            // i-노드 번호
    dev_t st_dev;            // 장치 번호
    dev_t st_rdev;           // 특수 파일 장치 번호
    nlink_t st_nlink;        // 링크 수
    uid_t st_uid;            // 소유자의 사용자 ID
    gid_t st_gid;            // 소유자의 그룹 ID
    off_t st_size;           // 파일 크기
    time_t st_atime;         // 최종 접근 시간 (slide 35~37)
    time_t st_mtime;         // 최종 수정 시간 (slide 35~37)
    time_t st_ctime;         // 최종 상태 변경 시간
    long st_blksize;         // 최적 블록 크기
    long st_blocks;          // 파일의 블록 수
};
```

메모리 매핑을 사용한 cat 명령어 구현:mmap.c

- 실행 결과
  - 파일로부터 데이터를 읽는 대신, 메모리 맵핑된 주소를 배열처럼 활용하여 데이터에 접근
    - 파일 접근 연산 vs. 메모리 접근 연산
  - Memory mapping을 사용하여 파일의 내용을 출력하는 cat 명령어



해당 출력 문장은 sample.txt의 내용임
해당 파일의"Hello~~ " 내용을
메모리 맵핑 된 파일을 통해
포인터 연산(메모리 연산)을 활용하여
파일의 내용을 출력한 결과임

핵심개념

- 지역변수와 매개변수에 대한 메모리 공간은 실행시간 스택에 자동적으로 할당되며 동적 변수는 힙에 할당된다.
- 동적 할당을 사용하는 이유는 필요할 때 필요한 만큼만 메모리를 요청해서 사용하여 메모리를 절약하기 위해서이다.
- malloc 함수는 메모리를 할당할 때 사용하고 free는 할당한 메모리를 해제할 때 사용한다.
- malloc() 함수는 메모리 할당에 성공하면 할당된 메모리의 시작주소를 반환하고 실패하면 NULL을 반환한다.
- 공유 메모리는 프로세스간 사이에 메모리 영역을 공유해서 사용할 수 있도록 해준다.
- 메모리 맵핑을 이용하면 파일의 일부 영역에 메모리 주소를 부여할 수 있다.