서술형 과제 **(4**주차**)**

최규형

1. **fork()** 함수와 달리 **exec()** 함수는 다양한 기능들을 제공하는 **exec** 계열 함수들이 존재합니다**. execl, execlp, execle, execv, execv, execvp, execvpe** 함수들의 기능을 설명하고 각 함수들의 차이점을 표로 정리하세요**.**
    - exec의 suffix로 붙는 문자의 의미를 반드시 설명해 주세요.

exec

실행파일을 인자로 받아 현재 프로세스 코드 영역을 덮어씌운 뒤 순차 실행한다.

- exec**l**
    - (함수 인자 영역에) 인자 list를 받는다
- exec**v**
    - (별도 선언된) 인자 variable을 받는다

함수명	함수 인자	**suffix** 의미	기능 **/** 특징
execl	-디렉토리와 파일 이름이 합친 전체 이름 -명령어 인수 리스트(argv) -마지막인자=NULL	**l** (함수 인자 영역에) 인자 **list**를 받아 exec() 실행	argv 리스트에 담긴 실행파일을 인자로 받아, 현재 프로세스 코드 영역을 덮어씌우고 순차 실행
execlp	-파일이름 -명령어 인수 리스트 (argv) -마지막인자=NULL	**p** 디폴트 **PATH** 환경변수 값으로 경로를 지정하여 exec 실행	경로(path)를 별도로 입력하지 않고 디폴트 path를 사용함
execle	-파일이름 -명령어 인수 리스트 (argv) -NULL -별도의 환경변수 string 인자 (USER, PATH 등) >예시: **char *envp[] = {:"USER=dave", "PATH=/bin", (char*)0};**	**e** 환경 변수(**e**nvironment variables)를 별도로 지정하고자 할 때 사용	USER, PATH 등을 담고 있는 환경변수를 별도로 선언한 뒤, 해당 변수를 함수의 인자로 사용함
execv	-디렉토리 -외부 인자 리스트	**v** (별도 선언된) 인자 **variable**을 받아 exec() 실행	외부 변수로 선언된 argv 리스트에 담긴 실행파일을 인자로 받아, 현재 프로세스 코드 영역을 덮어씌우고 순차 실행
execvp	-파일이름 -외부 인자 리스트	**p** 디폴트 **PATH** 환경변수 값으로 경로를 지정하여 exec 실행	경로(path)를 별도로 입력하지 않고 디폴트 path를 사용함
execve	-파일이름 -외부 인자 리스트	**e** 환경 변수(**e**nvironment	USER, PATH 등을 담고 있는 환경변수를 별도로 선언한 뒤,

	-외부 환경변수 리스트	variables)를 별도로 지정하고자 할 때 사용	해당 변수를 함수의 인자로 사용함

2. 프로세스간 커뮤니케이션 **(Inter-process Communication)** 기법 중 **Pipe, Shared Memory**를 포함해 **3**가지 이상 기법을 조사하고 각 기법들의 제약 조건, 사용 방식의 차이점에 대해 작성해주세요.

[1] Pipe

커널 공간의 메모리를 사용함

- 제약조건
  - 기본적으로 단방향 통신 가능
  - 부모 프로세스로부터 자식 프로세스로만 통신 가능
- 사용방식
  - 파이프라인 기능을 위한 변수 선언
    - msg = “Hello Child Process!”;
    - int fd[2], pid, nbytes;
    - if (pipe(fd) < 0) // pipe(fd) 로 파이프 생성
      - exit(1);
  - 파이프라인 정보 쓰기 (부모 프로세스)
    - write(fd[1], msg, MSGSIZE);
  - 파이프라인 정보 읽기 (자식 프로세스)
    - nbytes = read(fd[0], buf, MSGSIZE);
    - printf("%d %s\n", nbytes, buf);

```c
char* msg = "Hello Child Process!";
int main()
{
    char buf[255];
    int fd[2], pid, nbytes;
    if (pipe(fd) < 0)  // pipe(fd) 로 파이프 생성
        exit(1);
    pid = fork(); // 이 함수 실행 다음 코드부터 부모/자식 프로세스로 나뉘어짐
    if (pid > 0) {  // 부모 프로세스에는 자식 프로세스 pid값이 들어감
        write(fd[1], msg, MSGSIZE); //fd[1]에 씁니다.
        exit(0);
    }
    else {  // 자식 프로세스에는 pid값이 0이 됨
        nbytes = read(fd[0], buf, MSGSIZE); // fd[0]으로 읽음
        printf("%d %s\n", nbytes, buf);
        exit(0);
    }
    return 0;
}
```

[2] Message Queue

커널 공간의 메모리를 사용함

- 제약조건
  - 기본적으로 FIFO(First In, First Out) 정책에 따라 데이터 전송
  - 부모-자식 뿐 아니라, 어떠한 프로세스 간에도 양방향 통신 가능
- 사용방식
  1. msgget // 메시지큐 생성 시스템콜
     a. 코드 예시
     b. msqid = msgget(key, msgflg) // key=식별자(정수), msgflg = 옵션
        i. msgflg 설정
        ii. IPC_CREAT: 새로운 key라면 식별자 생성, IPC_CREAT|접근권한
        iii. 예시: IPC_CREAT|0644 -> rw-r--r--
  2. msgsnd // 메시지 send (전송) 시스템콜
     a. 코드 예시
     b. msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT)
        i. 생성한 메시지큐(msqid)에, &sbuf에 들어가 있는 데이터를 buf_length만큼 전송하겠다
        ii. msgflg 설정
            1. 블록 모드 (0) / 비블록 모드 (IPC_NOWAIT)
            2. 블록 모드 = 수신자가 데이터를 읽을 때까지 다음 코드 실행을 중단 (=대기)
            3. 비블록 모드 = 수신자가 데이터를 읽지 않더라도 다음 코드를 실행함
  3. msgrcv // 메시지 receive (수신) 시스템콜
     a. 코드 예시
     b. ssize_t = msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg)
     c. msgrcv(msqld, &rbuf, MSGSZ, 1, 0)
        i. msqid에 선언된 메시지큐로부터, 변수공간 주소 &rbuf에다가 내가 원하는 메시지 사이즈 MSGSZ만큼을 받아오겠다
            1. 이 때, msqid에 할당된 메시지 큐의 key는 메시지를 send한 메시지 큐와 동일한 key를 가져야 함
        ii. msgtyp 설정
            1. (msgtyp = 0) 첫번째 메시지
            2. (msgtyp > 0) 타입이 일치하는 첫번째 메시지
        iii. msgflg 설정
            1. 블록 모드 (0) / 비블록 모드 (IPC_NOWAIT)
            2. 블록 모드 = 수신자가 데이터를 읽을 때까지 다음 코드 실행을 중단 (=대기)
            3. 비블록 모드 = 수신자가 데이터를 읽지 않더라도 다음 코드를 실행함

*참고

메시지큐의 key 생성을 위한 ftok() 함수를 사용할 수 있음

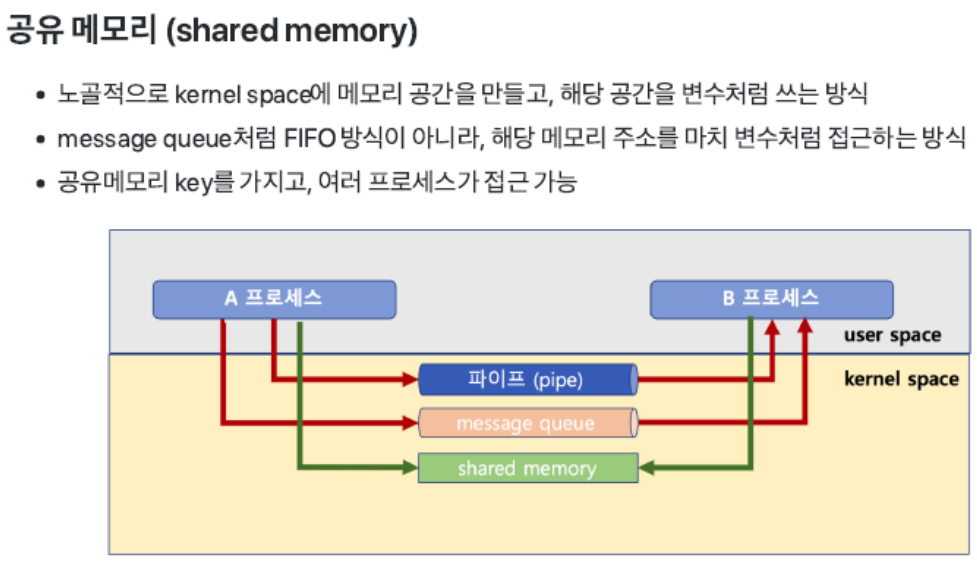
- path 경로명의 inode 값과 그 숫자값(id) 기반으로 key 생성
- 경로 삭제 후 재생성시 inode 값이 달라지므로, 이전과는 다른 key 값이 리턴됨

(ftok 함수 예제)

```
#include <sys/ipc.h>

key_t ftok(const char *path, int id);


// 예

key = ftok(“keyfile”, 1);

id = msgget(key, IPC_CREAT|0640);
```

[3] Shared Memory

커널 공간의 메모리를 사용함



- 제약조건
  - 노골적으로 커널 공간에 메모리를 만든 뒤 변수처럼 공간을 사용함
  - 메모리 주소를 변수처럼 접근하므로 PIPE(단방향), 메시지 큐(FIFO 양방향)와 다르게 자유로운 접근이 가능함
  - 공유메모리 key를 사용하여 여러 프로세스가 접근 가능함
- 사용방식

1. 공유 메모리 생성
2. 공유 메모리 연결
3. 공유 메모리 해제
4. 공유 메모리 읽기
5. 공유 메모리 쓰기

- 마치 포인트 변수처럼 사용, 이하 코드 예제

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

#include <sys/ipc.h>

#include <sys/shm.h>


// ############## 1. 공유 메모리 생성 ##############

// key: 임의 숫자 또는 ftok 함수로 생성한 키 값

// size: 공유 메모리 크기

// shmflg: 공유 메모리 속성


// 리턴 값: 공유 메모리 식별자 리턴

int shmget(key_t key, size_t size, int shmflg);


// 예

shmid = shmget((key_t)1234, SIZE, IPC_CREAT|0666))


// ############## 2. 공유 메모리 연결 ##############

#include <sys/types.h>

#include <sys/shm.h>

// shmid: shmget 함수로 생성한 공유 메모리 식별자

// shmaddr: 공유 메모리 연결 주소 (보통 (char *)NULL으로 설정하면, 알아서 적절한 주소로 연결)
```

// shmflg: 공유 메모리의 읽기/쓰기 권한 (0이면 읽기/쓰기 가능, SHM_RDONLY면 읽기만 가능)

// 리턴 값: 성공시 연결된 공유 메모리의 시작 주소를 리턴

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

// 예

```
shmaddr = (char *)shmat(shmid, (char *)NULL, 0)
```

// ############## 3. 공유 메모리 해제 ##############

```
#include <sys/types.h>
#include <sys/shm.h>
int shmdt(char *shmaddr);
```

// ############## 4. 공유 메모리에서 읽기 ##############

```
printf("%s\n", (char *)shmaddr)
```

// ############## 5. 공유 메모리에 쓰기 ##############

```
strcpy((char *)shmaddr, "Linux Programming")
```

// ############## 참고 - shmctl() ##############

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

// shmid: shmget 함수로 생성한 공유 메모리 식별자

// cmd: 수행할 제어 기능 (예: IPC_RMID - shmid로 지정한 공유 메모리 제거)

// buf: 제어 기능에 사용되는 공유 메모리 구조체의 구조

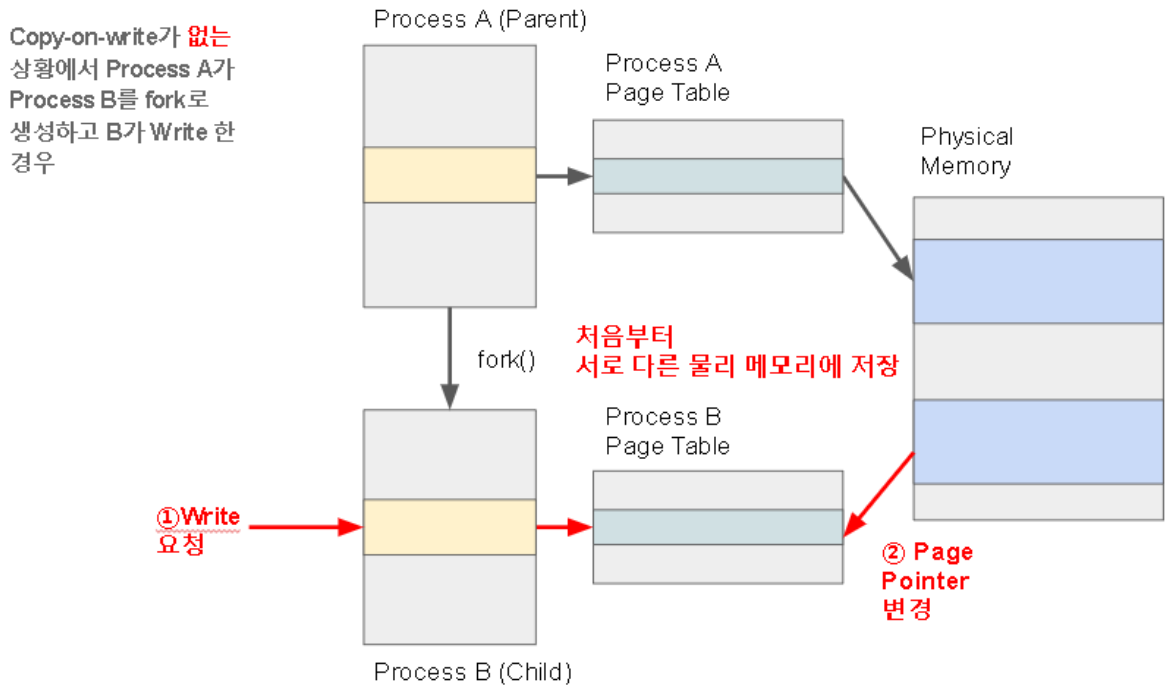
int shmctl(int shmid, int cmd, struct shmid_ds *buf);

// 예

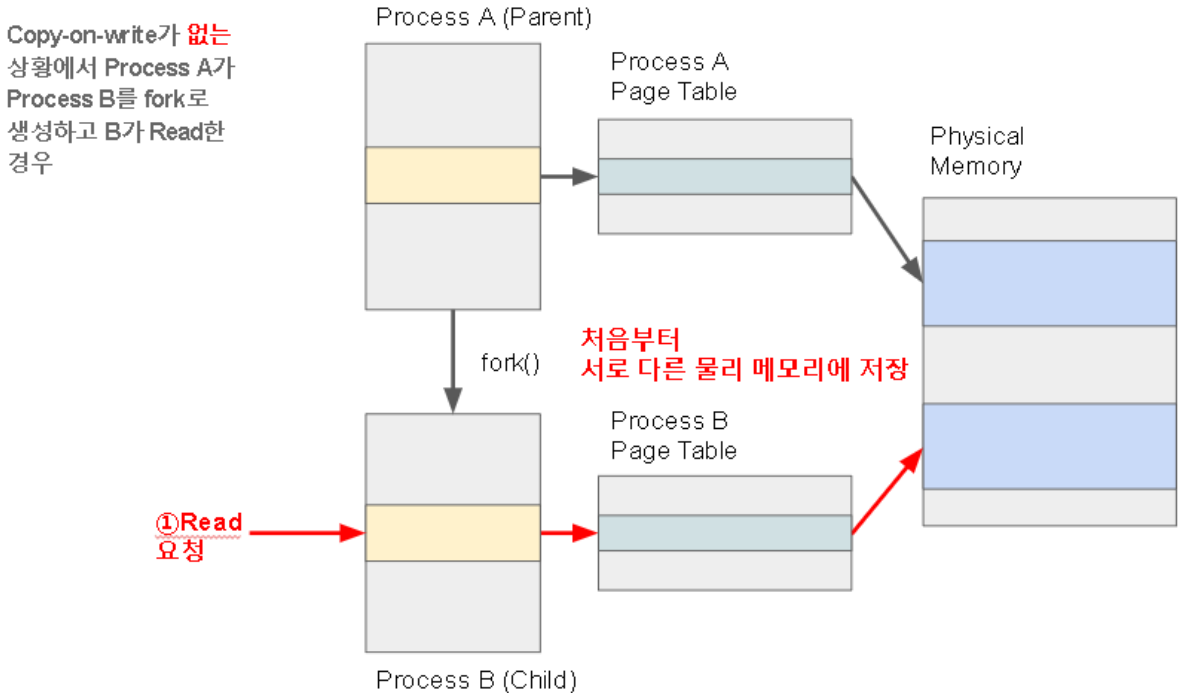
shmctl(shmid, IPC_RMID, (struct shmid_ds *)NULL);

3. **Fork**를 이용한 프로세스 생성시 메모리 복사 비용을 줄이기 위한 기법 중 하나로 **copy-on-write** 기법이 있습니다**. Copy-on-write** 기법이 있을 때와 없을 때 두 경우에 대해서 프로세스 **A, B**가 사용하고 있는 메모리 공간에 **Read**와 **Write** 수행하면 생기는 변화를 각 **Process**의 메모리 공간**,** 각 **Process**의 **Page Table,** 그리고 **Physical Memory Space**를 사용해 그림으로 나타내세요**.**
    - 각 케이스에 대해 동작 순서를 표시하는 번호와 설명이 필요합니다.
    - 총 네가지 경우에 대해 설명해야 합니다.
        1. Copy-on-write가 없이 Process A가 Process B를 fork로 생성하고 B가 Write 한 경우.
        2. Copy-on-write가 없이 Process A가 Process B를 fork로 생성하고 B가 Read 한 경우.
        3. Copy-on-write가 있는 상황에서 Process A가 Process B를 fork로 생성하고 B가 Write 한 경우.
        4. Copy-on-write가 있는 상황에서 Process A가 Process B를 fork로 생성하고 B가 Read 한 경우.

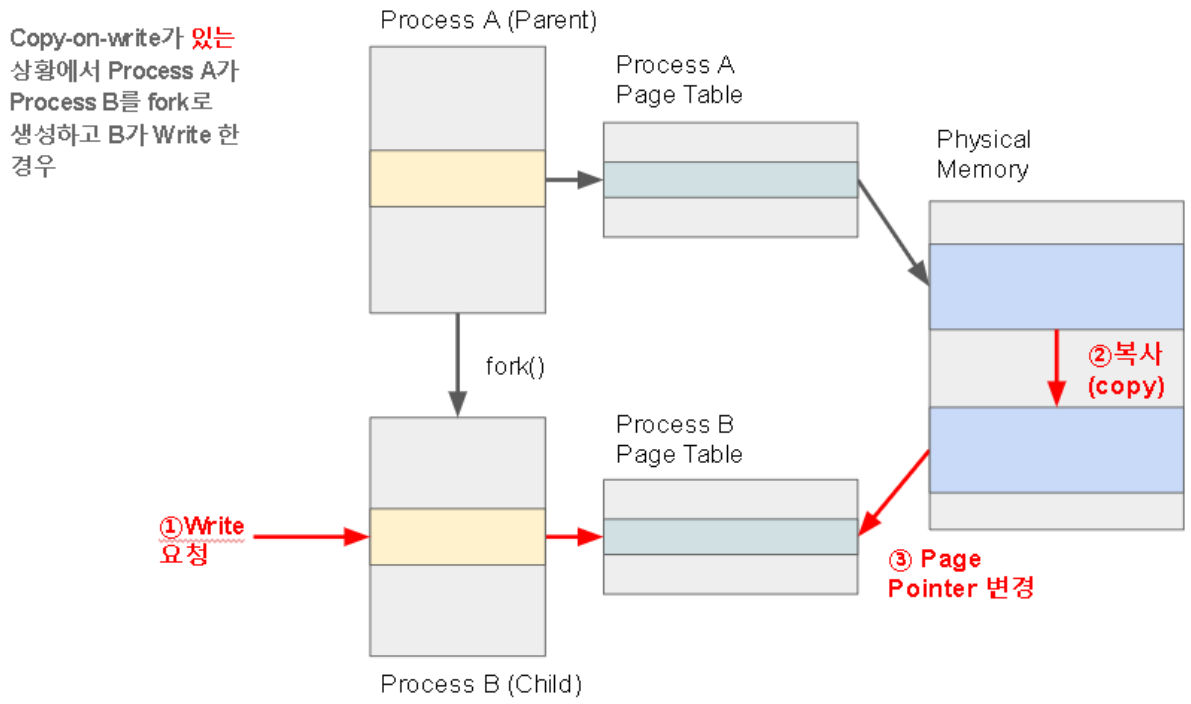
1.


Copy-on-write가 없는 상황에서 Process A가 Process B를 fork로 생성하고 B가 Write 한 경우
Process A (Parent)
Process A
Page Table
Physical
Memory
fork()
처음부터
서로 다른 물리 메모리에 저장
Process B
Page Table
①Write
요청
② Page
Pointer
변경
Process B (Child)

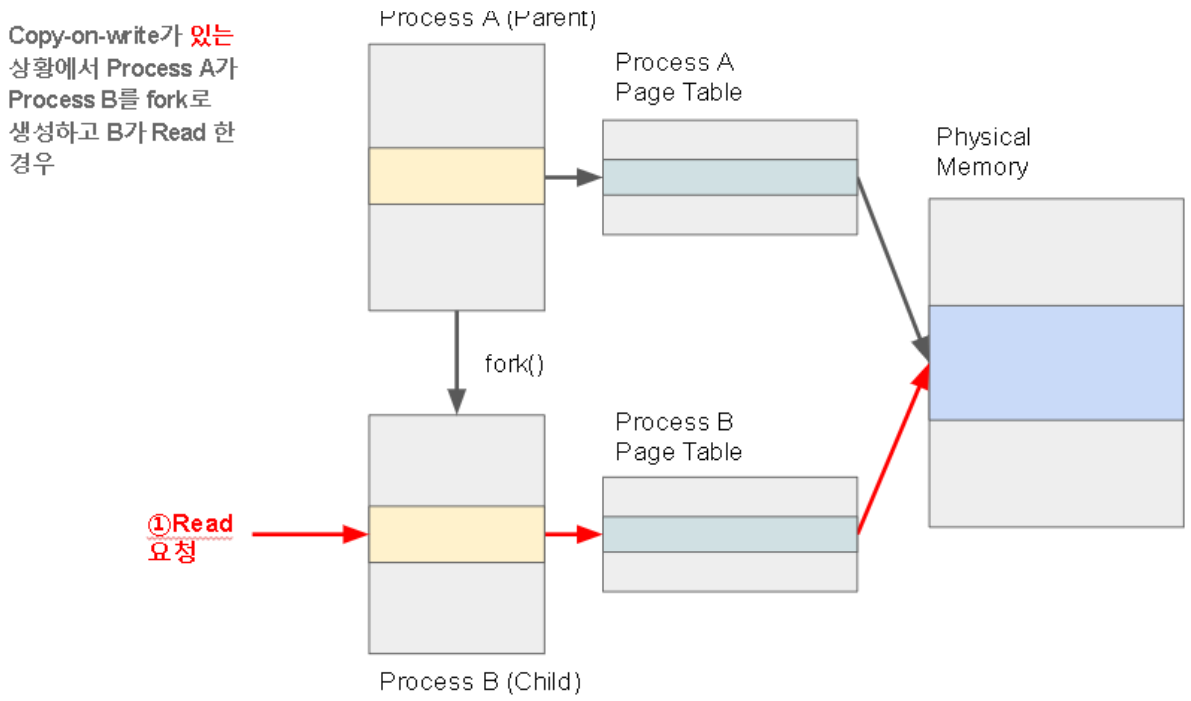
2.


Copy-on-write가 없는 상황에서 Process A가 Process B를 fork로 생성하고 B가 Read한 경우
Process A (Parent)
Process A
Page Table
Physical
Memory
fork()
처음부터
서로 다른 물리 메모리에 저장
Process B
Page Table
①Read
요청
Process B (Child)

3.


Copy-on-write가 있는 상황에서 Process A가 Process B를 fork로 생성하고 B가 Write 한 경우
Process A (Parent)
Process A Page Table
Physical Memory
fork()
②복사 (copy)
Process B Page Table
①Write 요청
③ Page Pointer 변경
Process B (Child)

4.


Copy-on-write가 있는 상황에서 Process A가 Process B를 fork로 생성하고 B가 Read 한 경우
Process A (Parent)
Process A Page Table
Physical Memory
fork()
Process B Page Table
①Read 요청
Process B (Child)

4. 파일에 접근하는 방법으로 **read, write** 함수들을 사용하는 방식과 **mmap**을 사용하는 **Memory Mapped File** 방식이 있습니다**.** 두 방식의 차이점과 장단점을 접근 방식**,** 사용성**, OS** 내부에서 처리 방식 관점에서 서술 하십시오**.**

Read & Write 함수 기반 접근법

- 장점
  - 필요한 만큼만 데이터가 메모리에 저장되어 있으므로 메모리 저장공간을 효율적으로 사용할 수 있음
- 단점
  - lseek()을 사용하여 개별 파일마다 데이터 주소를 탐색해야 하는 번거로움
  - 필요할 때마다 파일에 접근해야 하므로 반복적인 파일 접근에 따른 성능 저하 발생

mmap 기반 **Memory Mapped File** 접근법

- mmap 기반 메모리 동작
  1. mmap 실행 시, 가상 메모리 주소에 file 주소 매핑
  2. 해당 메모리 접근 시 아래 동작 (=on-demand paging / lazy allocation)
     a. 페이지 폴트 인터럽트 발생
     b. OS에서 file data 복사하여 물리 메모리 페이지에 넣어줌
  3. 메모리 read
     a. 해당 물리 페이지 데이터를 읽는다
  4. 메모리 write
     a. 해당 물리 페이지 데이터를 수정 후, 페이지 상태 flg 중 dirty bit을 1로 수정한다
  5. 파일 close
     a. 파일을 닫을 때 물리 페이지 데이터가 file에 업데이트됨
     b. 파일 접근 횟수를 줄일 수 있으므로 성능 개선 효과
- 장점
  - read(),. write() 함수 호출 시 반복적인 파일 접근을 방지하여 성능 개선
  - mapping된 영역은 파일 처리를 위한 lseek()을 사용하지 않고 간단한 포인터 조작으로 탐색 가능함
    - lseek() : 파일에 따른 데이터 주소를 개별적으로 찾아가는 함수
- 단점
  - mmap은 페이지 사이즈 단위로 매핑
    - 페이지 사이즈 단위 정수배가 아닐 경우, 한 페이지 정도의 공간이 추가적으로 할당되며, 남은 공간은 0으로 채워짐 (비효율적)