Week 4 Assignment

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.**
   A. exec의 suffix로 붙는 문자의 의미를 반드시 설명해 주세요.

(답안)

기능 설명

exec() 계열 함수는 현재 프로세스 이미지를 새로운 프로세스 이미지로 교체함. 기본적으로 첫 인자로 실행하고자 하는 파일의 이름을 받음.

- int execl(const char *pathname, const char *arg, ..., (char *) NULL);
  실행하고자 하는 파일의 경로를 첫 번째 인자로 받음(pathname). 해당 실행파일에 전달할 인자를 List로 받으며 List의 각 원소는 NULL로 끝나는 문자열인 null-terminated string임. List의 종료는 NULL을 인자로 넘겨 표시함.
- int execlp(const char *file, const char *arg, ..., (char *) NULL);
  실행하고자 하는 파일의 이름을 첫 번째 인자로 받음(file). execl()과 마찬가지로 해당 실행파일에 전달할 인자는 List로 넘겨받음. 그러나 execl()과 달리 실행하고자 하는 파일이 '/'로 시작하지 않는 경우 쉘 환경과 동일하게 환경변수 PATH에 명시된 디렉토리에서 찾음.
- int execle(const char *pathname, const char *arg, ..., (char *) NULL, char *const envp[]);
  실행하고자 하는 파일의 경로를 첫 번째 인자로 받음(pathname). execl()과 마찬가지로 해당 실행파일에 전달할 인자는 List로 넘겨받음. 환경변수를 null-terminated string에 대한 Array의 포인터로 전달받음. Array의 끝은 마지막 포인터를 NULL 포인터를 사용해 나타냄.
- int execv(const char *pathname, char *const argv[]);
  실행하고자 하는 파일의 경로를 첫 번째 인자로 받음(pathname). 실행파일의 인자를 null-terminated string의 array 포인터로 전달함(argv). Array의 끝은 마지막 포인터를 NULL 포인터로 설정하여 표시함. 해당 array의 첫번째 원소는 실행파일의 이름을 사용해야함.
- int execvp(const char *file, char *const argv[]);
  실행하고자 하는 파일의 이름을 첫 번째 인자로 받음(file). execv()와 동일한 방식으로 실행파일에 전달한 인자를 명시함(argv).
- int execvpe(const char *file, char *const argv[], char *const envp[]);
  execv()와 동일한 방식으로 실행파일에 전달한 인자를 명시함(argv). 실행파일이 '/'로 시작되지 않을 때 환경변수 PATH에 명시된 경로에서 파일을 탐색하고 환경변수를 설정할 수 있음.

exec() 계열 함수의 Suffix 규칙.

각 함수의 exec를 제외하고 끝 글자들로부터 해당 함수의 기능을 알 수 있음. e, l, p, v 총 4가지 문자를 사용하며 각 기능은 아래와 같음.

- e: 새로운 프로세스 이미지가 사용할 환경변수를 설정함. 환경변수는 null-terminated string의 array로 표현되며 array 포인터를 인자로 전달함. Array의 끝은 NULL 포인터를 사용하여 표시.
- l: 실행파일에 전달할 인자를 List로 전달함. 함수에 가변 인자로 전달되며 그 끝은 (char *) NULL을 마지막으로 전달해 표시.
- p: 실행할 파일이 (file 파라메터)가 '/'로 시작하지 않는 경우 PATH 환경변수에 명시된 경로에서 실행파일을 찾음.
- v: 실행파일에 전달할 인자를 vector로 전달함. Vector는 null-terminated string의 array의 포인터로 전달. Array의 끝은 NULL 포인터를 사용하여 표시.

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

(답안)

Pipe

선입선출 (First-in First-out, FIFO) 방식의 IPC(Inter-process communication) 방법. 단방향 통신만을 지원함. 부모 프로세스가 Pipe 생성 후 자식 프로세스가 이를 상속 받아 사용함. Linux에서는 pipe가 두개의 file descriptor (fd)를 생성하고 하나를 쓰기, 하나를 읽기에 사용하며 File API를 통해 통신함.

Shared Memory

메모리 공간을 공유하여 통신하는 IPC 방법. 메모리 공간을 공유하여 다수의 프로세스 들이 모든 방향으로 통신이 가능함. 동시에 접근이 가능하며 이에따른 동기화 문제가 발생할 수 있음. 이를 해결하기 위하여 Semaphore등의 동기화 기법을 적용할 수 있음.

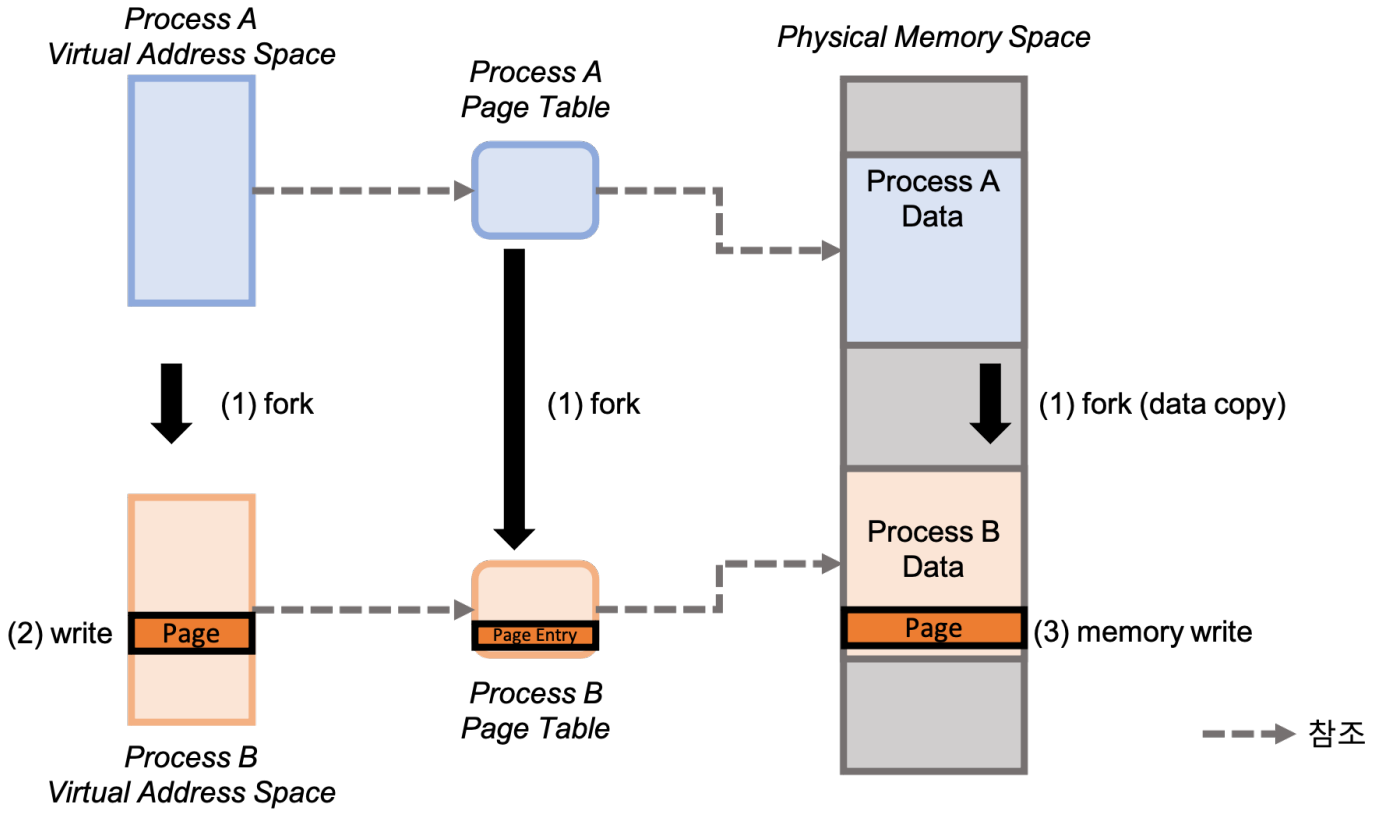
Signal

해당 프로세스에 Signal을 전달하여 통신하는 IPC 기법. 위에 서술한 방식들과 달리 데이터를 명시적으로 전송하지 않음. 한 프로세스가 Signal을 발생시키면 OS로부터 이를 전달받은 프로세스가 Signal Handler를 사용하여 이를 알아차리고 반응할 수 있음. 시스템에서 사용자가 마음대로 사용할 수 있도록 예약해 둔 SIGNAL을 사용하거나 Signal Handler를 등록해야 하는 등 다른 기법에 비해 자유도가 떨어짐.

3. **Fork를 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, B가 사용하고 있는 메모리 공간에 Read와 Write 수행하면 생기는 변화를 각 Process의 메모리 공간, 각 Process의 Page Table, 그리고 Physical Memory Space를 사용해 그림으로 나타내세요.**
   A. 각 케이스에 대해 동작 순서를 표시하는 번호와 설명이 필요합니다.
   B. 총 네가지 경우에 대해 설명해야합니다.
      i. Copy-on-write가 없이 Process A가 Process B를 fork로 생성하고 B가 Write 한 경우.
      ii. Copy-on-write가 없이 Process A가 Process B를 fork로 생성하고 B가 Read 한 경우.
      iii. Copy-on-write가 있는 상황에서 Process A가 Process B를 fork로 생성하고 B가 Write 한 경우.
      iv. Copy-on-write가 있는 상황에서 Process A가 Process B를 fork로 생성하고 B가 Read 한 경우.

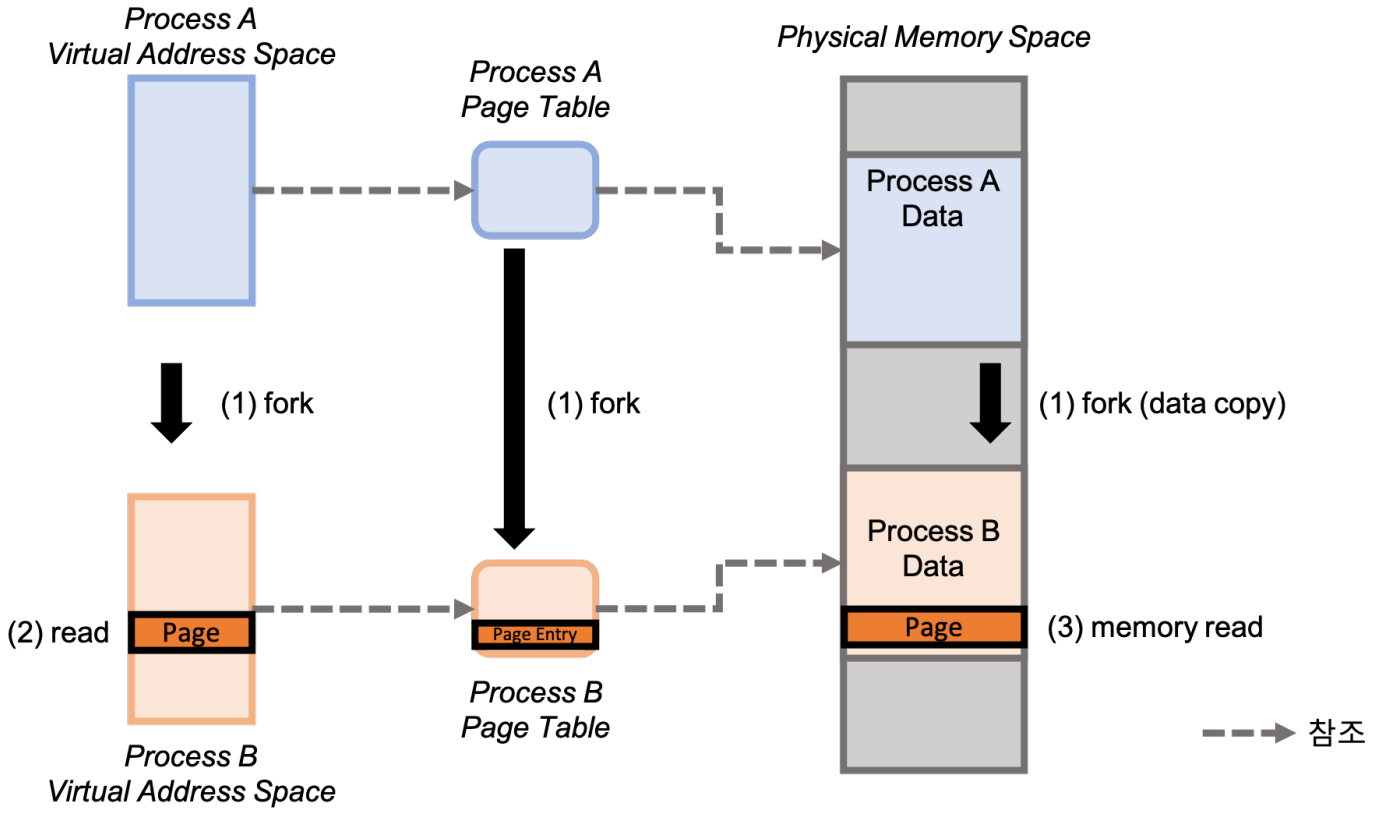
(답안)

(1) Copy-on-write가 없이 Process A가 Process B를 fork로 생성하고 B가 Write 한 경우.



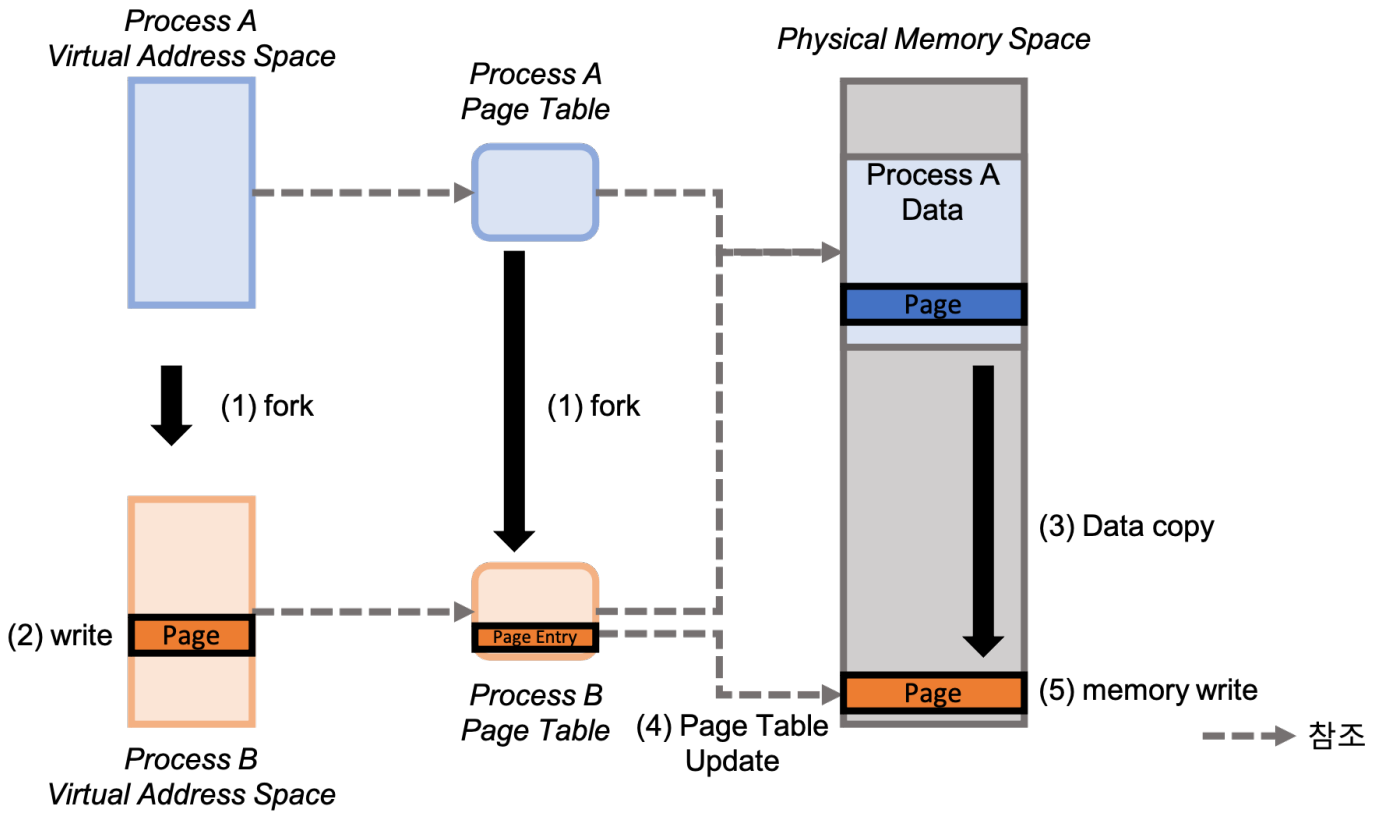
해당 케이스의 경우, copy-on-write가 없으므로 fork 수행 시 physical memory의 데이터가 실제로 복사됨 (그림에서 (1)). 이후 프로세스 B가 write를 행하는 경우, Page Table Translation을 거쳐 복사된 데이터에 쓰기를 수행함. (그림에서 (2), (3)).

(2) Copy-on-write가 없이 Process A가 Process B를 fork로 생성하고 B가 Read 한 경우.



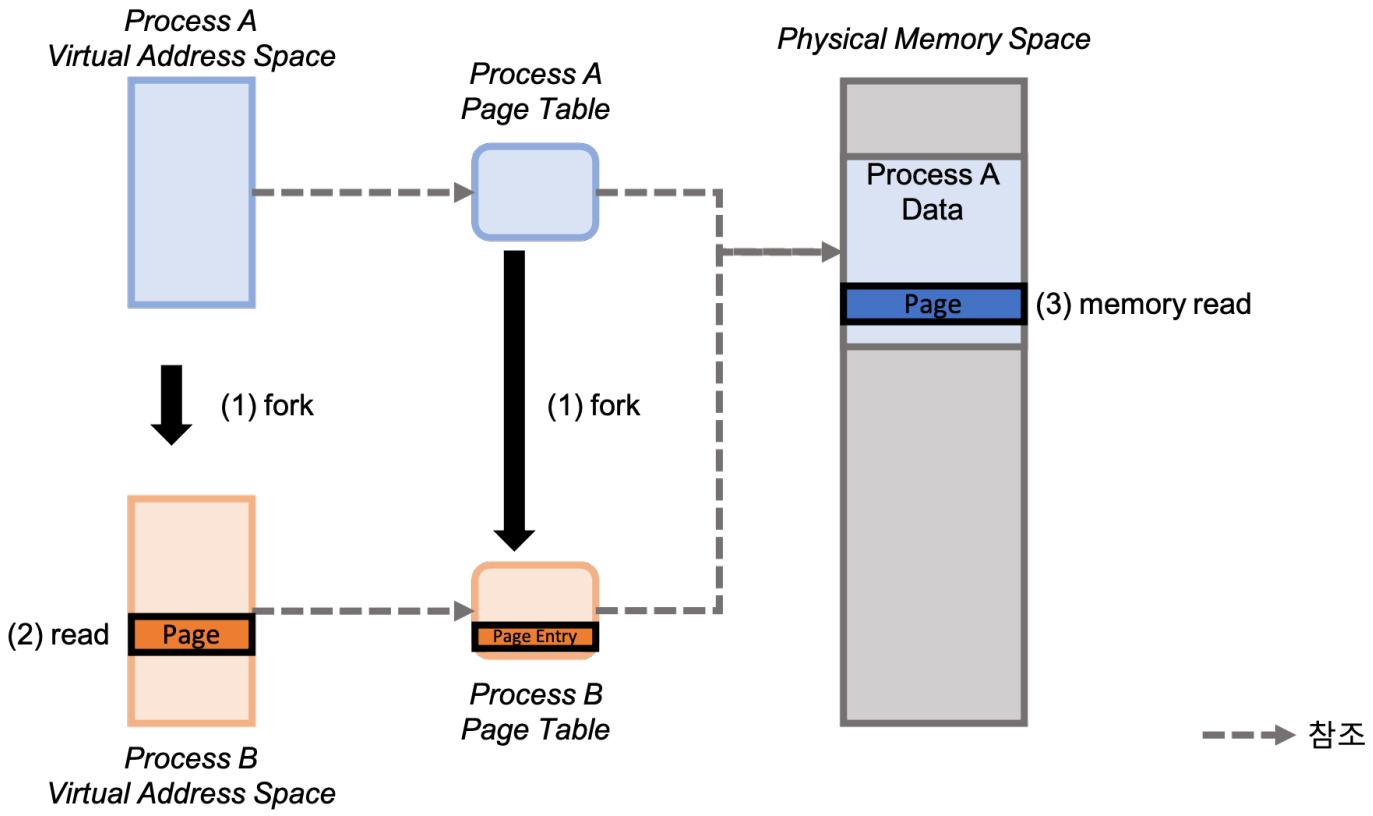
해당 케이스의 경우, copy-on-write가 없으므로 fork 수행 시 physical memory의 데이터가 실제로 복사됨 (그림에서 (1)). 이후 프로세스 B가 read를 행하는 경우, Page Table Translation을 거쳐 복사된 데이터를 읽음. (그림에서 (2), (3)).

(3) Copy-on-write가 있는 상황에서 Process A가 Process B를 fork로 생성하고 B가 Write 한 경우.



해당 케이스의 경우 copy-on-write 기법으로 인해 초기 프로세스 B의 Page Table은 프로세스 A의 데이터들을 가리킴 (그림에서 (1). 이후 프로세스 B가 write를 수행할 경우 해당 Physical page만을 복사함 (그림에서 (2), (3)). 이에 따라 프로세스 B의 Page Table Entry가 변경되고 실제 쓰기는 복사된 데이터에 수행됨 (그림에서 (4), (5)).

(4) Copy-on-write가 있는 상황에서 Process A가 Process B를 fork로 생성하고 B가 Read 한 경우.



해당 케이스의 경우 copy-on-write 기법으로 인해 초기 프로세스 B의 Page Table은 프로세스 A의 데이터들을 가리킴 (그림에서 (1). 이후 Process B가 read를 수행하면 기존 Process A의 데이터를 읽음 (그림에서 (2), (3))

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

(답안)

Read()/write() File API

Read 및 write 등의 File API를 사용하는 방식은 명시적으로 system call을 부르게 됨. System call은 pagecache 혹은 저장 장치로 부터 데이터를 가져와 User space의 프로세스에게 넘겨줌. 이 때 Kernel space에 있는 데이터를 User space로 복사해야 하므로 데이터 복사 비용이 발생함. 또한 잦은 File API 호출은 system call인한 User space와 Kernel space간의 전환을 발생시켜 성능 저하를 유발함. 그러나 유저가 원하는 파일의 offset과 양을 명시하기 때문에 필요한 만큼의 버퍼만을 준비할 수 있음.

Mmap() memory mapped file

Mmap은 파일의 일부 영역을 프로세스의 가상 메모리 주소 공간에 맵핑함. 이를 통해 파일을 메모리 접근과 동일한 방식으로 읽고 쓸 수 있으며 파일의 중간을 읽더라도 해당 메모리 주소의 offset만 계산하면 바로 접근할 수 있음. Mmap을 통한 파일 접근은 해당 주소를 접근했을 때 파일 데이터가 준비되지 않은 경우 page fault 처리를 통해 저장 장치의 데이터에 접근해 가져옴. Page fault가 발생하지 않은 경우, 즉 데이터가 이미 메모리에 적재되어 사용 가능한 경우에는 system call을 이용한 File API와 달리 User to Kernel mode 전환이 일어나지 않아 성능 이점이 있음. 또한 가상 메모리에 의존해 동작하여 사용자가 버퍼를 직접 준비하고 관리할 필요 없음. 그러나 이용할 데이터의 양을 명시적으로 지정하지 않아 File API를 사용하여 한번의 system call로 읽을 수 있는 데이터를 연속된 page fualt를 통해 처리하는 경우가 발생할 수 있으며 이때에는 성능 저하가 일어날 수 있음.