UNIX File System Call

역사

- **초기(1950~1960년대)**: **파일 시스템 호출 개념 부재**. 물리적 장치의 **직접 관리 필요**.
  - **특이점**: 파일 시스템이 없어, 프로그램이 하드웨어를 직접 관리. 파일의 추상화가 없고, 물리적 위치를 직접 다뤄야 했음.
- **유닉스 도입(1970년대)**: **파일 작업 추상화** 및 **일관된 인터페이스 제공**. `open()`, `read()`, `write()` 와 같은 표준 파일 시스템 호출 도입.
  - **특이점**: **파일 디스크립터 개념** 도입. 장치 독립적 파일 관리 가능. 계층적 파일 시스템 구조 확립.
- **POSIX 표준화(1980년대)**: **운영 체제 간 호환성 확보**를 위한 **표준 API 제공**. POSIX 표준을 통한 시스템 호출 통일.
  - **특이점**: 다양한 운영 체제에서의 **호환성 강화**. `stat()`, `fstat()`, `unlink()` 등의 호출 표준화.
- **VFS 도입(1990년대)**: **가상 파일 시스템 계층 도입**. 여러 파일 시스템을 동시에 지원하는 추상화 계층 구현.
  - **특이점**: **다양한 파일 시스템 지원 가능**. ext2, FAT, NTFS, NFS 등 여러 파일 시스템을 동일한 방식으로 접근 가능.
- **비동기 파일 작업 도입(2000년대 이후)**: **비동기 파일 작업 처리**. 비동기 시스템 호출(`aio_read()`, `aio_write()`)을 통한 성능 최적화.
  - **특이점**: **비동기 I/O 처리**로 대량의 파일 작업에서 성능 향상. 서버와 데이터베이스에서의 동시 작업 처리 최적화.
- **현대(2010년대 이후)**: **보안 및 확장성 강화**. 클라우드 및 분산 파일 시스템 환경에 적합한 **파일 시스템 호출** 제공.
  - **특이점**: **보안 강화**(SELinux, 접근 제어), **클라우드와 분산 시스템**에서의 파일 관리 효율성 증대.

VFS (Virtual File System)

리눅스와 같은 유닉스 계열 운영 체제에서 다양한 파일 시스템을 지원하기 위해 설계된 **추상화 계층**

- 여러 가지 파일 시스템을 일관된 인터페이스로 추상화
- 다양한 파일 시스템(ext4, FAT, NTFS, NFS 등)을 동일한 방식으로 처리할 수 있도록 지원
- 파일 시스템에 대한 확장성과 유연성 강화

주요 구성 요소

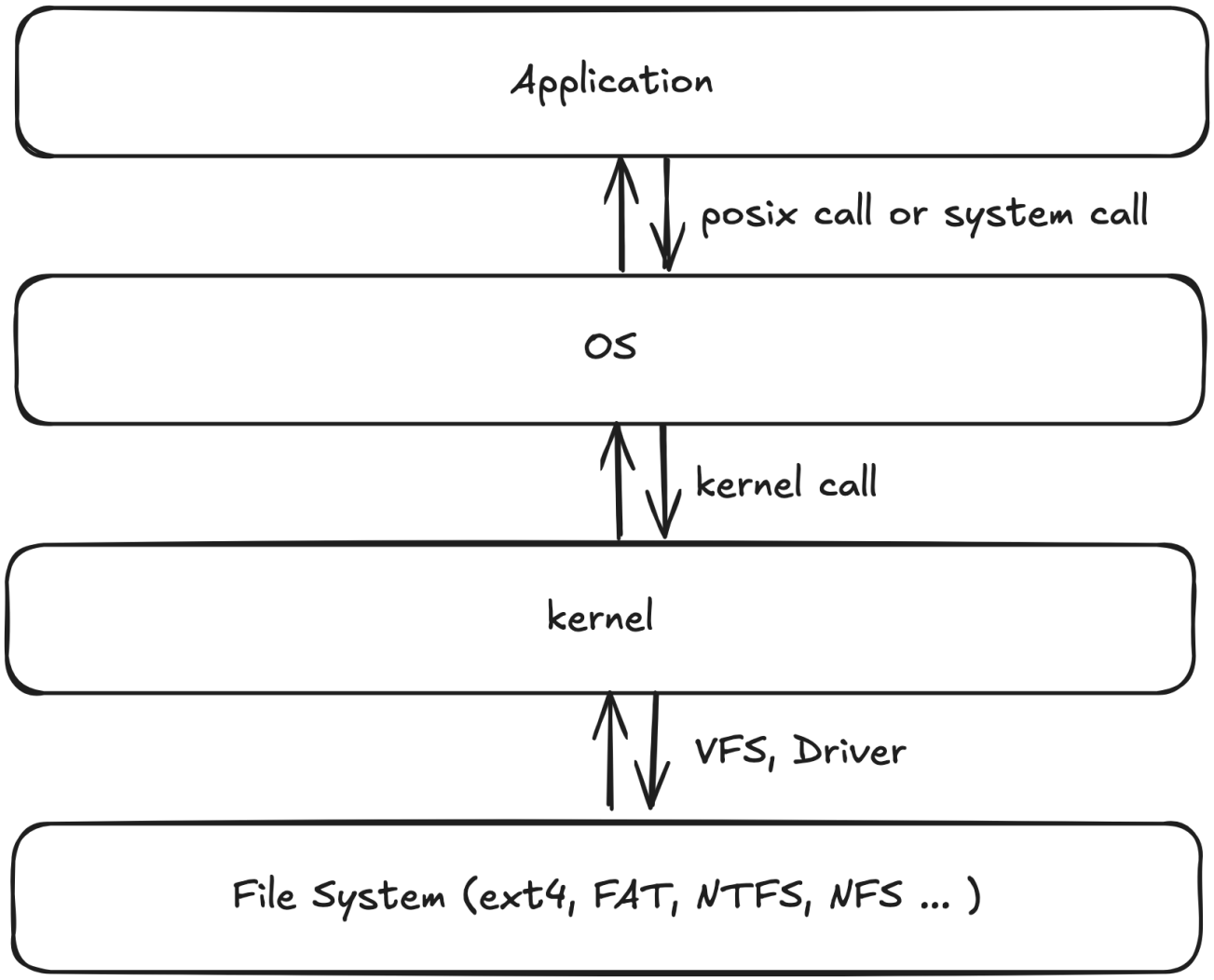
1. **슈퍼블록(Superblock)**:
    - 파일 시스템의 메타데이터를 나타내는 데이터 구조
    - 각 파일 시스템은 자신만의 슈퍼블록을 가지며, 파일 시스템의 크기, 사용 가능한 블록 수, 파일 시스템 형식(ext4, FAT 등)에 대한 정보를 가짐
2. **인덱스 노드(Inode)**:
    - 각 파일 또는 디렉터리에 대한 메타데이터(파일 크기, 파일 소유자, 파일 권한, 파일의 생성 및 수정 시간 등)를 저장하는 데이터 구조
    - 파일의 실제 데이터 위치를 추적하며, 파일에 대한 모든 중요한 정보가 이 구조에 저장
3. **디렉터리 항목(Directory Entry, Dentry)**:
    - 파일 이름과 그 파일의 인덱스 노드(inode)를 연결하는 역할
    - 파일 시스템은 파일 이름을 디렉터리 항목을 통해 찾아내고, 해당 파일에 대한 작업을 수행할 수 있습니다.
4. **파일 객체(File Object)**:
    - 열린 파일에 대한 정보를 저장하는 구조체로, 파일 디스크립터와 연결됨
    - 파일 포인터, 파일 접근 모드, 파일 시스템 내의 위치 등 파일 작업에 필요한 정보를 가짐

VFS의 동작 과정

1. **파일 열기** (`open()`):
    - 사용자가 파일을 열기 위해 `open()` 시스템 호출
    - VFS는 해당 파일의 디렉터리 항목(dentry)을 찾아 파일의 인덱스 노드를 확인
    - 파일이 존재하면 해당 파일의 인덱스 노드를 반환
    - 새로운 파일 객체를 생성하여 이를 연결
2. **파일 읽기/쓰기** (`read()`, `write()`):
    - 파일을 읽거나 쓸 때, VFS는 파일 객체를 통해 인덱스 노드와 파일 시스템을 확인
    - 파일의 데이터를 읽거나 쓰는 작업을 처리
        - VFS는 특정 파일 시스템에 따라 처리 방식을 결정, 각 파일 시스템 드라이버에 필요한 작업을 위임
3. **파일 닫기** (`close()`):
    - 작업이 끝난 파일은 `close()` 시스템 호출을 통해 파일 디스크립터와 연결된 파일 객체가 해제됨
    - VFS는 파일 객체를 해제하고, 필요 시 파일 시스템에 변경 사항을 반영합니다.
4. 파일 삭제 (`unlink()`, `remove()`)
    1. **시스템 호출 발생**: 사용자의 `unlink()` 또는 `remove()` 파일 삭제 API 호출.
    2. **VFS 호출**: 시스템 호출의 커널 전달 및 **`sys_unlink()`** 함수 호출 후 **`vfs_unlink()`** 함수 호출.
    3. **경로 탐색**: VFS의 파일 경로 확인 및 삭제할 파일의 디렉터리 항목(dentry)과 인덱스 노드(inode) 탐색.

4. **참조 카운트 확인**: 파일 사용 여부 확인 및 참조 카운트(reference count) 0 여부 확인.
5. **파일 시스템 드라이버 호출**: VFS의 파일 시스템 드라이버(예: ext4의 `ext4_unlink()`) 호출 및 파일 삭제 작업 수행.
6. **인덱스 노드 및 디스크 블록 해제**: 파일 시스템 드라이버의 파일 메타데이터(inode)와 데이터 블록 해제 및 파일 실제 삭제.
7. **결과 반환**: 파일 삭제 완료 후 작업 성공 여부 확인 및 시스템 호출을 통한 사용자에게 결과 반환.

동작 흐름



주요 API 및 System/Kernel Call

1. 파일 열기 (`fopen`, `open`)

POSIX API: `fopen()`

- `fopen()` 은 파일을 열기 위한 고수준의 POSIX API입니다.
- `fopen(const char *filename, const char *mode)` 형태로 파일을 열고, 파일 포인터를 반환합니다.
- 내부적으로는 시스템 호출 `open()` 을 호출하여 커널에서 파일을 엽니다.

시스템 호출: `open()`

- **`open()`** 은 파일을 열기 위한 저수준의 시스템 호출입니다.
- 정의: `int open(const char *pathname, int flags, mode_t mode)`
- 이 호출은 파일을 열거나, 필요하면 새로 파일을 생성할 수 있습니다. 호출 후 파일 디스크립터가 반환됩니다.

비동기 호출: `aio_open()` (일부 시스템에서 비동기 파일 열기 지원)

- 일부 시스템에서는 비동기적으로 파일을 열 수 있는 비동기 시스템 호출(`aio_open()`)이 구현되기도 합니다. 하지만 파일을 여는 비동기 호출은 흔치 않으며, 주로 비동기 읽기/쓰기에 초점이 맞춰져 있습니다.

커널 호출: `do_filp_open()`

- **`do_filp_open()`** 은 커널 내부에서 파일을 여는 역할을 하는 커널 함수입니다.
- `open()` 시스템 호출이 호출되면, 커널에서 파일 시스템의 파일을 열기 위해 이 함수가 호출됩니다.
- 이 함수는 가상 파일 시스템(VFS)에서 파일 객체를 생성하고, 파일의 메타데이터를 불러오는 역할을 합니다.

---

2. 파일 읽기 (`fread` , `read`)

POSIX API: `fread()`

- `fread()` 는 파일에서 데이터를 읽는 고수준의 API입니다.
- `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)` 형태로 사용되며, 내부적으로 시스템 호출 `read()` 를 호출합니다.

시스템 호출: `read()`

- **`read()`** 는 파일에서 데이터를 읽기 위한 시스템 호출입니다.

- 정의: `ssize_t read(int fd, void *buf, size_t count)`
- 파일 디스크립터 `fd`를 사용하여 파일에서 데이터를 읽고, 지정된 버퍼 `buf`에 데이터를 저장합니다.

비동기 호출: `aio_read()`

- **`aio_read()`**는 비동기적으로 파일을 읽기 위한 시스템 호출입니다. 요청 후 파일 데이터를 읽는 작업이 백그라운드에서 실행되며, 완료된 후 결과를 알립니다.

커널 호출: `vfs_read()`

- **`vfs_read()`**는 커널의 가상 파일 시스템(VFS) 레이어에서 파일 읽기를 처리하는 커널 함수입니다.
- `read()` 시스템 호출이 실행되면, 커널 내부에서 파일 시스템에 접근하여 데이터를 읽기 위해 이 함수가 호출됩니다. 실제로는 각 파일 시스템(ex. ext4, FAT)의 읽기 함수로 이어집니다.

---

3. 파일 쓰기 (`fwrite`, `write`)

POSIX API: `fwrite()`

- `fwrite()`는 파일에 데이터를 쓰는 고수준의 API입니다.
- `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)` 형태로 사용되며, 내부적으로 시스템 호출 `write()`를 호출합니다.

시스템 호출: `write()`

- **`write()`**는 파일에 데이터를 쓰기 위한 시스템 호출입니다.
- 정의: `ssize_t write(int fd, const void *buf, size_t count)`
- 파일 디스크립터 `fd`를 사용하여 버퍼 `buf`의 데이터를 파일에 기록합니다.

비동기 호출: `aio_write()`

- **`aio_write()`**는 비동기적으로 파일에 데이터를 쓰기 위한 시스템 호출입니다. 요청한 후, 데이터 쓰기 작업이 완료될 때까지 기다릴 필요 없이 다른 작업을 수행할 수 있습니다.

커널 호출: `vfs_write()`

- **`vfs_write()`** 는 커널의 가상 파일 시스템(VFS) 레이어에서 파일 쓰기를 처리하는 커널 함수입니다.
- `write()` 시스템 호출이 실행되면, 커널 내부에서 파일 시스템에 접근하여 데이터를 기록하기 위해 이 함수가 호출됩니다.

---

4. 파일 닫기 (`fclose`, `close`)

POSIX API: `fclose()`

- `fclose()` 는 파일을 닫는 고수준의 API입니다.
- `int fclose(FILE *stream)` 형태로 사용되며, 내부적으로 시스템 호출 `close()` 를 호출합니다.

시스템 호출: `close()`

- **`close()`** 는 열린 파일 디스크립터를 닫기 위한 시스템 호출입니다.
- 정의: `int close(int fd)`
- 파일 디스크립터 `fd` 를 사용하여 파일을 닫고, 시스템 자원을 해제합니다.

커널 호출: `filp_close()`

- **`filp_close()`** 는 커널 내부에서 파일 디스크립터를 닫고 자원을 해제하는 커널 함수입니다.
- `close()` 시스템 호출이 커널로 전달되면, 커널은 `filp_close()` 를 호출하여 파일 시스템에서 열린 파일 객체를 해제하고 파일을 닫습니다.

---

5. 파일 상태 조회 (`stat`, `fstat`, `lstat`)

POSIX API: `stat()`, `fstat()`, `lstat()`

- **`stat()`** : 파일의 상태 정보를 조회하는 API입니다. 경로를 기준으로 파일 정보를 확인합니다.
  - `int stat(const char *pathname, struct stat *buf)`
- **`fstat()`** : 열린 파일 디스크립터를 기준으로 파일 상태 정보를 조회합니다.
  - `int fstat(int fd, struct stat *buf)`
- **`lstat()`** : 심볼릭 링크 자체의 정보를 조회하는 API입니다.
  - `int lstat(const char *pathname, struct stat *buf)`

시스템 호출: `stat()`, `fstat()`, `lstat()`

- 각 API는 대응되는 시스템 호출과 연결됩니다. 이 호출들은 파일의 메타데이터(크기, 권한, 소유자, 생성 시간 등)를 가져옵니다.

커널 호출: `vfs_stat()`

- `vfs_stat()` 는 커널에서 파일의 상태 정보를 불러오는 함수입니다. 파일 시스템의 정보를 가져와 사용자에게 반환하는 역할을 합니다.

---

6. 파일 삭제 (`unlink`, `remove`)

POSIX API: `unlink()`, `remove()`

- `unlink()` : 파일을 삭제하는 API입니다. 경로명을 통해 파일을 삭제합니다.
  - `int unlink(const char *pathname)`
- `remove()` : 파일이나 디렉터리를 삭제할 수 있는 API입니다. 파일이든 디렉터리든 경로에 해당하는 파일을 삭제합니다.
  - `int remove(const char *pathname)`

시스템 호출: `unlink()`

- `unlink()` 는 파일을 삭제하는 시스템 호출입니다. 파일 시스템에서 파일에 대한 참조를 제거하고, 실제 파일 삭제 작업이 수행됩니다.

커널 호출: `vfs_unlink()`

- `vfs_unlink()` 는 가상 파일 시스템(VFS) 레이어에서 파일을 삭제하는 커널 함수입니다. 파일 시스템의 타입에 맞는 삭제 함수를 호출하여 실제 파일 삭제를 수행합니다.

POSIX AIO

POSIX AIO(Asynchronous Input/Output)는 POSIX 표준에서 정의한 **비동기 파일 입출력** 방식으로, 파일 I/O 작업이 완료될 때까지 **대기하지 않고** 작업을 요청한 후, 제어권을 즉시 반환하는 비동기적인 I/O 처

리 방법입니다. 이는 동기적인 파일 I/O 방식과 달리, 애플리케이션이 I/O 작업을 요청한 후 다른 작업을 계속 진행할 수 있게 해줍니다.

POSIX AIO는 대규모 파일 처리, 고성능 서버, 네트워크 프로그래밍 등에서 사용되어, I/O 작업을 병렬로 처리하거나, 작업 완료와는 상관없이 애플리케이션이 즉시 응답할 수 있도록 설계되었습니다.

POSIX AIO의 주요 개념

1. **비동기 I/O 요청**: POSIX AIO에서는 파일을 읽거나 쓰는 작업을 요청할 때 I/O 작업이 완료될 때까지 기다리지 않고, 제어권을 즉시 반환합니다.
2. **I/O 작업 완료 확인**: 작업이 완료되었는지 여부는 나중에 **비동기적으로 확인**할 수 있습니다.
3. **I/O 작업 취소**: 필요한 경우, I/O 작업을 취소하거나 완료 상태를 조회할 수 있습니다.
4. **비동기 알림**: I/O 작업이 완료되면 시그널이나 다른 방식으로 알림을 받을 수 있습니다.

주요 POSIX AIO 함수

1. `aio_read()` - 비동기 파일 읽기

```
int aio_read(struct aiocb *aiocbp);
```

- `aio_read()` 는 파일에서 데이터를 비동기적으로 읽는 함수입니다. 이 함수는 호출된 후 즉시 반환되며, 실제 I/O 작업은 백그라운드에서 비동기적으로 수행됩니다.
- `aiocbp` 는 I/O 작업에 대한 세부 정보를 담고 있는 `aiocb` 구조체의 포인터입니다.

2. `aio_write()` - 비동기 파일 쓰기

```
int aio_write(struct aiocb *aiocbp);
```

- `aio_write()` 는 파일에 데이터를 비동기적으로 쓰는 함수입니다. 이 함수 역시 호출 후 바로 반환되며, I/O 작업이 백그라운드에서 비동기적으로 진행됩니다.

3. `aio_error()` - I/O 작업 상태 확인

```
int aio_error(const struct aiocb *aiocbp);
```

- `aio_error()` 는 비동기 I/O 작업의 현재 상태를 확인하는 함수입니다. I/O 작업이 완료되지 않았다면 `EINPROGRESS` 가 반환됩니다.

4. `aio_return()` - 완료된 작업의 결과 반환

```
ssize_t aio_return(struct aiocb *aiocbp);
```

- `aio_return()` 은 비동기 I/O 작업이 완료된 후, 그 작업의 결과를 반환하는 함수입니다. 예를 들어, `aio_read()` 작업이 완료되면 읽은 바이트 수를 반환합니다.

5. `aio_suspend()` - 여러 I/O 작업의 완료를 기다림

```
int aio_suspend(const struct aiocb *const list[], int nent, const struct
timespec *timeout);
```

- `aio_suspend()` 는 여러 개의 비동기 I/O 작업이 완료될 때까지 대기하는 함수입니다. 비동기 작업이 완료되면 대기에서 빠져나오거나, 타임아웃이 발생할 수 있습니다.

6. `aio_cancel()` - 비동기 I/O 작업 취소

```
int aio_cancel(int fd, struct aiocb *aiocbp);
```

- `aio_cancel()` 은 특정 파일 디스크립터에 대한 비동기 I/O 작업을 취소하는 함수입니다. 작업이 아직 완료되지 않은 경우에만 작업을 취소할 수 있습니다.

`aiocb` 구조체 (Asynchronous I/O Control Block)

모든 POSIX AIO 함수는 `aiocb` 라는 구조체를 사용해 I/O 작업의 세부 정보를 저장합니다. 이 구조체는 비동기 I/O 작업을 정의하고 제어하는 데 중요한 역할을 합니다.

```
struct aiocb {
    int          aio_fildes;    // 파일 디스크립터
    off_t        aio_offset;    // 파일 내에서 I/O 작업을 시작할 위치
    volatile void *aio_buf;     // 데이터를 읽거나 쓸 버퍼
    size_t       aio_nbytes;    // I/O 작업에 사용할 데이터 크기
    int          aio_lio_opcode; // 연속적인 I/O 작업 수행 시 사용할 연산 코드
    struct sigevent aio_sigevent; // I/O 작업 완료 시 알림 방법 (시그널 등)
```

```
    int         aio_reqprio;  // I/O 작업의 우선순위
};
```

- `aio_fildes` : I/O 작업을 수행할 파일 디스크립터.
- `aio_offset` : 파일 내에서 읽기나 쓰기를 시작할 위치.
- `aio_buf` : 읽기 또는 쓰기 작업에 사용할 데이터 버퍼.
- `aio_nbytes` : 읽거나 쓸 데이터의 크기.
- `aio_sigevent` : I/O 작업 완료 시 알림을 위한 시그널을 지정하는 구조체.
- `aio_reqprio` : I/O 작업의 우선순위를 설정.

POSIX AIO 동작 흐름

1. **비동기 I/O 요청**:
   - 프로그램은 `aio_read()` 또는 `aio_write()` 함수로 비동기 I/O 작업을 요청합니다.
   - 요청된 작업은 **즉시 반환**되며, 제어권은 다시 호출자에게 넘어갑니다. 이때 커널은 요청된 I/O 작업을 백그라운드에서 처리합니다.
2. **I/O 작업 진행**:
   - 커널은 비동기적으로 요청된 작업을 처리하며, 이를 위해 **aiocb** 구조체를 통해 작업을 관리합니다.
3. **작업 완료 상태 확인**:
   - 프로그램은 `aio_error()` 를 통해 작업이 완료되었는지 확인할 수 있으며, `aio_return()` 을 호출해 결과(예: 읽은 바이트 수)를 얻을 수 있습니다.
4. **작업 완료 시 알림**:
   - I/O 작업이 완료되면 시그널을 통해 프로그램에 알리거나, `aio_suspend()` 를 사용해 특정 I/O 작업이 끝날 때까지 대기할 수 있습니다.

POSIX AIO의 장점

1. **비동기성**: CPU가 I/O 작업의 완료를 기다리지 않고 다른 작업을 계속할 수 있기 때문에, 비동기성을 제공하여 시스템의 성능을 높입니다.
2. **병렬성**: 여러 비동기 I/O 작업을 동시에 요청하고 처리할 수 있습니다. 이는 대규모 파일 작업을 수행할 때 매우 유용합니다.
3. **높은 성능**: 서버 애플리케이션이나 대용량 데이터 처리 시스템에서 특히 유용한데, I/O 작업이 끝나기를 기다리지 않고 다른 작업을 수행함으로써, 전체 시스템의 처리 성능을 극대화할 수 있습니다.

POSIX AIO의 사용 사례

1. **파일 서버**: 파일 서버는 다수의 클라이언트 요청을 처리해야 하기 때문에, 비동기 I/O를 통해 요청을 병렬로 처리하고 응답 시간을 줄일 수 있습니다.
2. **데이터베이스 시스템**: 대규모 데이터베이스 시스템에서 대량의 데이터 읽기/쓰기 작업을 비동기적으로 처리하여, 동시 작업 성능을 극대화할 수 있습니다.
3. **네트워크 프로그래밍**: 비동기 I/O는 네트워크 소켓 프로그래밍에서도 많이 사용되며, 네트워크 데이터 송수신을 대기하지 않고 다른 작업을 처리할 수 있게 합니다.

POSIX AIO의 단점 및 한계

1. **복잡한 오류 처리**: 비동기 방식은 동기 방식에 비해 코드가 복잡하며, I/O 작업의 성공/실패 여부를 확인하는 방식도 복잡해질 수 있습니다.
2. **낮은 호환성**: 일부 운영 체제나 파일 시스템에서 POSIX AIO를 완전히 지원하지 않거나, 특정 조건에서만 사용할 수 있습니다.
3. **과도한 시그널 처리**: 비동기 작업이 완료될 때 시그널을 통해 알림을 받을 수 있지만, 과도한 시그널 처리로 인해 시스템이 부담을 느낄 수 있습니다.
