ChatGPT

오픈북 시험 대비 예상 문제 유형 정리

목차

SCULL 커널 모듈 소스코드 (scull.h, scull_main.c, scull_pipe.c, scull_access.c)

O/X 문제

- **문제:** SCULL 장치는 데이터 저장을 위해 동적으로 메모리를 할당한다. (O/X)
  정답: O. SCULL 드라이버는 `kmalloc` 등을 사용하여 필요한 시점에 메모리를 할당하고, 링크드 리스트 형태로 데이터를 저장한다 [1]. 고정 크기의 버퍼를 미리 할당하는 것이 아니라, 쓰기 동작 시 새로운 메모리 블록(quantum)을 확보하여 유연하게 파일 크기를 확장한다.

- **문제:** `scull_dev` 구조체에는 동시 접근을 제어하기 위한 mutex 변수가 포함되어 있다. (O/X)
  정답: O. `scull_dev` 구조체에는 `struct mutex mutex` 필드가 있으며, SCULL의 `read`/`write` 구현에서는 이 mutex로 임계 구역을 보호한다. 이를 통해 여러 프로세스가 동일 장치를 동시에 접근할 때 데이터 일관성을 유지한다.

- **문제 (오답 유도):** `copy_from_user` 함수는 사용자 버퍼로부터 데이터를 복사한 후, **성공 시 복사한 바이트 수를 반환한다.** (O/X)
  정답: X. `copy_from_user`는 **복사하지 못한 바이트 수**를 반환한다 [2]. 즉, 0을 반환하면 모든 데이터를 정상 복사한 것이고, 0이 아닌 값은 해당 바이트만큼 복사에 실패했음을 의미한다. SCULL의 `write` 구현을 보면, 이 함수의 반환값이 0이 아닐 때 에러로 처리하고 있다 [2].

단답형 문제

- **문제:** SCULL에서 quantum과 qset은 무엇이며, 어떤 역할을 하는가?
  정답: quantum은 SCULL 장치에서 한 개의 메모리 블록의 크기를 의미한다. qset은 이러한 메모리 블록에 대한 포인터들을 묶는 배열의 크기(한 quantum 세트당 포함할 수 있는 quantum 개수)이다. SCULL은 파일에 저장되는 데이터를 여러 quantum으로 나누어 저장하며, `scull_qset` 구조체는 포인터 배열(qset 크기만큼)과 다음 노드를 가리키는 포인터로 이루어진 링크드 리스트로 구현된다. 이러한 구조를 통해 SCULL은 큰 데이터를 연속된 메모리 없이도 관리할 수 있다.

- **문제:** `scull_open` 함수에서 파일을 열 때 수행하는 특별한 작업은 무엇이며, 그 이유는?
  정답: `scull_open`은 주로 장치의 사용 준비를 하고, 필요에 따라 이전에 남은 데이터를 정리한다. 예를 들

어 SCULL의 특별 장치들 중 **쓰기 모드로 열렸을 때 기존 데이터를 삭제**하는 동작이 있다. 실제로 `scull_u_open` 구현에서는 파일이 쓰기 전용으로 열릴 경우 `scull_trim` 함수를 호출하여 장치에 저장된 기존 데이터를 모두 해제한다 [3]. 이렇게 함으로써 이전 내용이 남아 있지 않은 깨끗한 상태에서 새로운 데이터를 기록할 수 있다. 그 외에 `scull_open`은 장치의 `private_data`에 해당 장치 구조체 포인터를 넣어주는 작업도 수행한다 (이를 통해 이후 `read` / `write`에서 장치에 접근) [4].

- **문제:** SCULL의 메모리 해제 함수 `scull_trim`은 어떤 기능을 하며, 언제 호출되는가?
  정답: `scull_trim`은 해당 SCULL 장치가 사용하는 모든 메모리를 해제하고 구조를 초기 상태로 리셋하는 함수이다. 구체적으로, 링크드 리스트로 연결된 모든 `scull_qset` 노드들을 순회하면서 각 노드가 가리키는 데이터 블록(quantum)을 `kfree`로 해제하고, 노드 자체도 해제한다. 그리고 장치 구조체의 `size`를 0으로 초기화하고 `data` 포인터를 NULL로 설정한다. 이 함수는 **장치를 완전히 비울 때** 사용되며, 대표적으로 장치를 쓰기 전용으로 열 때 이전 내용 삭제를 위해 호출하거나, 모듈을 제거(`scull_cleanup_module`)할 때 남은 메모리를 정리하는 경우 등에 호출된다.

서술형 문제

- **문제:** SCULL 장치에서 `read()` 시스템콜이 수행되는 과정을 설명하시오. (파일 포인터 f_pos와 SCULL의 데이터 구조 관점을 포함)
  정답: 사용자 프로세스가 SCULL 장치 파일에 대해 `read()`를 호출하면, 커널은 SCULL 드라이버의 `scull_read` 함수로 제어를 넘긴다. 이 함수에서 우선 장치의 mutex를 획득하여 다른 접근을 막는다. 그런 다음 현재 파일 포인터 f_pos 위치와 요청한 바이트 수를 바탕으로, SCULL의 메모리 구조 내에서 어느 위치부터 얼마나 읽어야 하는지 계산한다. SCULL 장치는 데이터를 링크드 리스트의 **quantum 세트**로 관리하므로, f_pos를 quantum 세트 단위로 나누어 해당 노드(`scull_qset`)와 quantum 인덱스를 구한다. 해당 위치에 데이터가 존재하면, `copy_to_user` 함수를 통해 커널 버퍼의 데이터를 사용자 버퍼로 복사한다. 복사가 성공하면 f_pos를 읽은 바이트만큼 증가시키고, mutex를 풀어준 뒤 복사한 바이트 수를 반환한다. 만약 요청한 위치에 데이터가 없거나(`scull_qset` 노드가 NULL이거나 해당 quantum 포인터가 NULL) EOF에 도달한 경우, 읽지 않고 0을 반환한다. 이 과정에서 `read()`는 데이터가 없더라도 **파일 끝에 도달하지 않았다면 블로킹되지 않고 0을 반환**하는데, SCULL의 경우 메모리상에 데이터가 없는 부분은 EOF처럼 취급하여 `0`을 리턴한다. 마지막으로 mutex를 해제하여 다른 프로세스가 장치에 접근할 수 있도록 한다.

- **문제:** SCULL 모듈에는 `scull_pipe` (파이프 장치)와 `scull_access` (액세스 제어 장치)와 같은 변형이 있습니다. **scull_pipe** 장치와 일반 SCULL 장치(scull0)의 동작상의 차이를 설명하고, **scull_access** 장치들이 제공하는 기능은 무엇인지 설명하시오.
  정답: scull_pipe는 두 프로세스 간의 파이프처럼 동작하도록 만들어진 장치입니다. 일반적인 SCULL 장치가 메모리에 데이터를 저장하고 임의 접근을 지원하는 반면, scull_pipe는 **고정 크기의 버퍼**를 두고 한쪽에서 쓰여진 데이터를 다른 쪽에서 읽어가는 형태로 구현됩니다. 만약 버퍼가 비어 있는데 읽으려고 하면 읽는 프로세스는 데이터를 기다리며 슬립하고, 버퍼가 가득 찬 상태에서 쓰려고 하면 쓰는 프로세스가 블록됩니다. 이는 `wait_queue`를 이용해 구현되며, 생산자-소비자 파이프와 유사한 동작을 합니다. 반면 일반 SCULL 장치는 이러한 블로킹 대신 파일처럼 동작하여 EOF를 바로 반환합니다. 한편 **scull_access** 장치들은 장치에 대한 접근 제한을 보여주는 예제입니다. 예를 들어 **scull_uid** 장치는 **동일 사용자가 여러 번 열 수 있지만 다른 사용자가 이미 열었으면 접근을 거부**하는 장치이고 [5] [6], **scull_priv**(프라이빗 장치)는 **한 번에 오직 하나의 프로세스만 열 수 있는** 장치입니다. 이를 통해 장치 파일의 소유권 개념과 동기화 기법을 보여주며, 내부적으로 spinlock과 `wait_queue`를 사용해 동시 접근을 제어합니다 [7] [8]. 요약하면, scull_pipe는 파이프처럼 블로킹 I/O를 지원하고, scull_access 계열은 UID나 단일 프로세스 기반으로 exclusive access를 구현하여 보안/동기화 개념을 설명합니다.

코드 작성 문제

- **문제:** SCULL 모듈의 메모리 해제 함수인 `scull_trim`을 아래 조건에 맞게 구현하시오. (힌트: 링크드 리스트로 연결된 `scull_qset`을 순회하며 할당된 메모리를 해제)

답안 (예시):

```
int scull_trim(struct scull_dev *dev) {
  struct scull_qset *qset_ptr, *next;
  int i;
  // 현재 장치의 첫 번째 qset부터 순회
  for (qset_ptr = dev->data; qset_ptr != NULL; qset_ptr = next) {
    // 다음 노드 미리 보관
    next = qset_ptr->next;
    // 현재 qset이 가지고 있는 데이터 블록들 해제
    if (qset_ptr->data) {
      for (i = 0; i < dev->qset; i++) {
        if (qset_ptr->data[i])
          kfree(qset_ptr->data[i]);
      }
      kfree(qset_ptr->data);
    }
    kfree(qset_ptr);  // 현재 qset 노드 구조체 자체 해제
  }
  // 장치 포인터와 size 초기화
  dev->data = NULL;
  dev->size = 0;
  return 0;
}
```

해설: 위 코드는 장치에 연결된 모든 `scull_qset` 노드를 순회하면서, 각 노드가 가리키는 메모리 블록(quantum)을 모두 반환(free)하고 노드 구조도 해제한다. 마지막으로 장치의 데이터 포인터를 NULL로, 저장된 크기(size)를 0으로 리셋하여 장치를 초기 상태로 만든다. 이 함수를 호출하면 해당 SCULL 장치에 저장된 모든 데이터가 메모리에서 삭제된다. `scull_trim`은 주로 장치를 처음 열 때 이전 내용을 지우거나 모듈 언로드时에 호출된다.

- **문제:** SCULL 장치를 **동적 Major 번호**로 등록하도록 코드를 수정하려 한다. `register_chrdev()`을 사용하는 현재 코드에서 Major 번호를 동적으로 할당받으려면 어떻게 해야 하는가? 해당 수정 사항을 구현하시오.
  답안 (요지): `register_chrdev`에 현재 상수로 지정된 Major 번호 대신 `0`을 전달해야 한다. 예를 들어 기존 `register_chrdev(SCULL_MAJOR, "scull", &scull_fops)` 호출을 `register_chrdev(0, "scull", &scull_fops)`로 변경한다. 이때 반환값을 정수 변수에 저장해야 하는데, 이 값이 커널이 할당한 **새로운 Major 번호**이다. 아래와 같이 수정할 수 있다:

```
int major;
major = register_chrdev(0, "scull", &scull_fops);
if (major < 0) {
  printk(KERN_WARNING "scull: 등록 실패\n");
  return major;
}
printk(KERN_INFO "scull: 할당된 Major = %d\n", major);
```

또한 `unregister_chrdev()`를 호출할 때도 할당받은 Major 번호를 사용해야 한다 (예: `unregister_chrdev(major, "scull")`). **해설:** Major 번호 0을 지정하면 커널이 비어있는 Major 번

호를 자동으로 할당해주며, 그 번호를 반환한다 [9] [10]. 따라서 동적 할당된 Major를 변수에 저장하여 이후에 사용해야 한다. 이 변경으로 충돌 위험 없이 Major 번호를 확보할 수 있고, `/proc/devices` 등을 통해 할당된 번호를 확인할 수 있다.

시스템프로그램 : Filesystem

O/X 문제

- **문제:** 리눅스 시스템에서 **모든 것은 파일로 표현**되며, 파이프나 소켓도 파일 디스크립터로 다룬다. (O/X)
  정답: O. 리눅스에서는 장치, 파이프, 소켓 등이 모두 파일 디스크립터(File Descriptor) 인터페이스를 통해 접근된다. 예를 들어 표준 입력/출력도 파일 디스크립터 0,1,2로 표현되며, 파이프나 소켓도 `read()/write()` 등의 파일 I/O 호출로 다룰 수 있다. 단, 내부 구현은 각각의 파일 시스템 또는 네트워크 스택에서 별도로 처리되지만, **사용자 입장에서는 동일한 방식**으로 사용한다.

- **문제:** 하나의 디렉터리 내에서는 **동일한 이름의 파일**을 두 개 이상 생성할 수 있다. (O/X)
  정답: X. 같은 디렉터리에서는 파일 이름이 고유해야 한다. 이미 존재하는 이름으로 새 파일을 생성하려 하면 오류가 발생한다. (심볼릭 링크의 대상이나 하드 링크를 통한 같은 파일에 대한 참조는 가능하지만, 그것들도 각각 디렉터리 내에서는 고유한 이름을 가져야 한다.)

- **문제: 심볼릭 링크**(symbolic link)는 원본 파일과 동일한 inode 번호를 갖는다. (O/X)
  정답: X. 심볼릭 링크는 별도의 파일이며 자체 inode를 가진다. 심볼릭 링크 파일은 그 내용으로 원본 경로명을 가지고 있을 뿐이고, 원본 파일의 inode와는 무관하다. 반면 **하드 링크**는 원본과 동일 inode를 가리키는 디렉터리 엔트리이므로 inode 번호가 같다. 따라서 심볼릭 링크와 하드 링크는 동작과 구조 면에서 다르다.

- **문제 (오답 유도):** `open()` 시스템 호출은 읽을 데이터가 없을 경우에도 블로킹되어 기다리다가, 파일에 내용이 생기면 반환된다. (O/X)
  정답: X. `open()` 호출 자체는 **파일을 여는 동작**일 뿐이며, 데이터를 읽는 동작을 수행하지 않는다. 따라서 파일을 여는 단계에서 블로킹/비블로킹은 관련이 없다. 읽을 데이터 유무에 따라 블로킹되는 것은 `read()` 호출에 대한 설명이다. 일반 파일의 경우 `read()`는 읽을 데이터가 없으면 바로 0(EOF)을 반환하고 종료하지만, 파이프나 소켓의 경우 데이터가 들어올 때까지 `read()`가 블록될 수 있다. 혼동하지 말아야 한다.

단답형 문제

- **문제: inode**란 무엇이며, 파일에 대한 어떤 정보를 저장하고 있는가?
  정답: inode는 리눅스 파일 시스템(예: ext4)에서 **파일의 메타데이터**를 담고 있는 자료구조이다. 각 파일마다 고유한 inode가 있으며, 여기에는 파일의 크기, 소유자 UID/GID, 접근 권한(permission), 생성/변경 시간(timestamp) 등의 정보와 함께 해당 파일이 저장된 **데이터 블록의 위치를 가리키는 포인터**들이 들어 있다. 디렉터리는 파일 이름을 inode 번호와 연결하는 역할을 하고, 파일의 실제 내용은 inode가 가리키는 데이터 블록에 저장된다. inode 자체에는 파일 이름이 없고 메타데이터와 데이터 위치 정보만 존재한다.

- **문제: 하드 링크(hard link)**와 **심볼릭 링크(symbolic link)**의 차이를 간단히 설명하시오.
  정답: 하드 링크는 동일한 파일 내용에 대해 **동일한 inode를 가리키는** 디렉터리 엔트리이다. 따라서 하드 링크를 추가하면 링크 수(link count)가 증가하며, 어느 하나의 링크를 지워도 다른 링크가 남아 있다면 파일 내용은 유지된다. 반면 심볼릭 링크는 별도의 파일로서, 그 내용이 원본 파일의 경로를 가리키는 **텍스트**이다. 심볼릭 링크는 원본 파일의 inode와 무관한 자기만의 inode를 가지며, 원본 파일이 삭제되면 심볼릭 링크는 끊어진 링크가 되어(target을 찾지 못하게 되어) 내용에 접근할 수 없게 된다. 요약하면, **하드 링크는 동일한 파일에 대한 또 하나의 이름**이고, **심볼릭 링크는 대상 파일의 경로를 가리키는 특수 파일**이다.

- **문제:** 리눅스에서 파일을 생성하고 삭제할 때 사용되는 시스템 호출은 무엇인지, 각각 쓰시오.
  정답: 파일 생성을 위해서는 보통 `open()` 시스템 호출을 `O_CREAT` **플래그와 함께** 사용한다. 예를 들어 `open("foo.txt", O_RDWR | O_CREAT, 0644)`와 같이 호출하면 지정 경로에 파일이 없을 경우 새 파일을 생성한다. 한편 파일을 삭제(제거)하는 시스템 호출은 `unlink()`이다. `unlink("foo.txt")`를 호출하면 해당 경로의 디렉터리 엔트리가 삭제되며, 해당 파일 inode의 링크 수를 감소시킨다. 링크 수가 0이 되면 파일의 데이터 블록이 해제된다 (즉, 실제 파일 내용이 지워지는 효과).

- **문제: 파일 디스크립터(File Descriptor)**와 **파일 기술자(File Description)**의 차이를 설명하시오.
  정답: 파일 디스크립터는 프로세스별로 관리되는 정수형 식별자이며, 프로세스가 파일을 열면 커널이 해당 파일에 대한 파일 디스크립터를 하나 할당한다. 반면 파일 기술자(file description 또는 open file object)는 커널 내부에서 열린 파일에 대한 구조체로, 파일 offset이나 상태(flags)를 포함한다. 여러 개의 파일 디스크립터가 하나의 파일 기술자를 참조할 수 있는데, 예를 들어 한 프로세스에서 `dup`으로 파일 디스크립터를 복제하거나 `fork`를 통해 자식 프로세스가 부모의 열린 파일을 상속받은 경우 **서로 다른 디스크립터 번호들이 같은 파일 기술자(커널 객체)를 가리키게 된다**. 이 경우 파일 offset 등 상태를 공유하므로, 한 쪽에서 `read()`를 하면 다른 쪽의 offset도 증가한다. 요컨대, 파일 디스크립터는 **프로세스가 가지고 있는 핸들**이고, 파일 기술자는 **커널이 관리하는 열린 파일에 대한 정보**이다.

- **문제 (오답 유도):** `read()` 시스템 호출이 **항상** 파일의 끝에 도달하면 0을 반환하므로, 읽을 데이터가 없는 경우에도 즉시 0을 리턴하여 호출한 프로세스로 제어가 돌아간다. (예/아니오)
  정답: 아니오. `read()` 호출에서 0을 반환하는 것은 **파일의 끝(EOF)**에 도달한 경우이다 [11]. 일반 파일에서 `read()`는 현재 파일 오프셋이 파일 크기보다 작으면 데이터가 존재하므로 0을 반환하지 않고, 데이터가 없다면 0을 반환하며 종료한다(EOF). 그러나 FIFO나 터미널, 소켓 같은 경우 `read()`는 **디폴트로 블로킹 동작**을 한다 [12]. 예를 들어 파이프의 경우 읽을 데이터가 없으면 프로세스를 sleep시켜 데이터가 생길 때까지 기다리게 한다. 따라서 "항상 0을 반환하여 제어가 돌아간다"는 것은 일반 파일의 EOF 상황만을 가리키며, 모든 경우에 적용되지 않는다. (시험에서는 블로킹 모드의 존재를 혼동하지 않도록 유의해야 한다.)

서술형 문제

- **문제: 파일 시스템의 블록 할당 방식**에는 연속 할당(contiguous allocation), 연결 할당(linked allocation), 인덱스 할당(indexed allocation)이 있다. 각 방식의 특징과 장단점을 설명하시오.
  정답:
- **연속 할당:** 파일의 모든 데이터를 디스크 상에서 연속된 물리적 블록들에 배치하는 방식이다. 장점은 연속된 블록에 있기 때문에 **순차 접근 속도가 매우 빠르고** 디스크 시크가 최소화된다. 또한 첫 블록과 크기만 알면 접근이 쉬워 **임의 접근(random access)**도 용이하다. 단점은 **외부 단편화**(fragmentation)가 발생하기 쉽고, 파일 크기를 예측하거나 변경하기 어려운 경우 공간 낭비 또는 확장 곤란 문제가 생긴다. 파일이 커지면 연속 공간을 확보하기 어려워질 수 있다.
- **연결 할당:** 파일을 여러 디스크 블록으로 나누어 저장하되, 각 블록에 다음 블록의 주소를 기록하여 **링크드 리스트**처럼 연결하는 방식이다. 장점은 연속된 공간이 필요 없으므로 **외부 단편화가 발생하지 않고**, 파일이 확장될 때도 용이하게 이어붙일 수 있다. 그러나 단점으로는 **임의 접근이 매우 비효율적**이다. 중간 블록을 찾으려면 처음부터 링크를 따라가야 하므로 접근 시간이 많이 걸린다. 또한 각 블록에 링크 정보를 저장해야 하기 때문에 약간의 공간 오버헤드와 **신뢰성 문제**(링크 손실 시 데이터 손실 위험)가 있다.

- **인덱스 할당:** 파일에 대한 모든 데이터 블록 주소를 모아 놓은 **인덱스 블록(테이블)**을 별도로 두는 방식이다. 이 인덱스 블록에는 해당 파일의 각 블록들이 위치한 디스크 주소들이 순서대로 저장된다. 장점은 링크드 리스트처럼 임의 접근이 어려운 문제를 해결하여, **임의 블록 접근도 인덱스 테이블을 참조하여 바로 접근**할 수 있다는 것이다. 또한 연속할당처럼 큰 연속 공간이 필요 없으므로 **단편화 문제가 적고 파일 확장도 유연**하다. 다만 단점은 별도의 인덱스 블록을 유지해야 하므로 **추가적인 메모리/디스크 오버헤드**가 있고, 큰 파일의 경우 인덱스 블록이 여러 개 필요해질 수 있다(이를 해결하기 위해 다중 간접 블록 등을 사용함).
  이 세 가지 방식은 파일 시스템에 따라 채택하는 방법이 다르며, 예를 들어 FAT 파일 시스템은 연결 할당(FAT 테이블이 링크 역할), ext4 등은 인덱스 할당 방식을 사용한다.

- **문제: 파일 읽기(write)와 쓰기(read)** 시스템콜이 커널 내부에서 어떻게 동작하는지, VFS 계층과 장치 드라이버와의 상호 작용을 포함하여 간략히 설명하시오.
  정답: 사용자 영역에서 `read(fd, buf, count)` 또는 `write(fd, buf, count)`를 호출하면, 우선 커널의 **시스템콜 인터페이스**를 통해 VFS(Virtual File System) 계층의 해당 함수가 호출된다. VFS는 파일 디스크립터 `fd`를 통해 해당 파일의 열린 파일 객체(file description)를 찾아가고, 그 내부에 연결된 파일 시스템의 구체적인 함수 포인터(`struct file_operations`의 `read`/`write`)를 호출한다.

- 만약 대상이 일반 파일이라면, VFS는 해당 파일 시스템 (예: ext4)의 `read`/`write` 구현을 호출한다. 이 구현에서는 페이지 캐시를 거쳐 디스크 블록 I/O를 수행하거나, 경우에 따라 버퍼캐시를 사용하여 효율을 높인다. 디스크 I/O가 필요하면 I/O 스케줄러와 블록 디바이스 드라이버를 거쳐 실제 디스크에서 데이터를 읽거나 쓴다.

- 만약 대상이 문자 디바이스 파일(예: `/dev/tty` 등)이라면, VFS는 그 장치 드라이버가 등록한 `file_operations`의 `read`/`write` 함수를 호출하게 된다. 예를 들어 SCULL 장치의 경우 SCULL 드라이버의 `scull_read`/`scull_write`가 호출되어, 메모리에서 데이터를 복사하거나 하는 동작을 수행한다. 블록 디바이스 파일의 경우에도 유사하게 해당 드라이버의 `read`/`write` (사실상 block device는 VFS 차원에서 페이지 캐시 등을 통해 처리)로 제어가 넘어간다.
  마지막으로 이러한 하위 계층 동작이 완료되어 데이터가 복사되면 시스템콜은 사용자 공간으로 복귀하며, `read`는 읽은 바이트 수를, `write`는 쓴 바이트 수를 리턴한다. 이처럼 VFS는 파일 유형에 상관없이 동일한 인터페이스를 제공하고, 실제 작업은 **해당 파일 시스템 또는 디바이스 드라이버의 구현체**가 수행하도록 매개 역할을 한다.

- **문제: 외부 단편화(External Fragmentation)**와 **내부 단편화(Internal Fragmentation)**의 차이를 파일 시스템 관점에서 설명하시오.
  정답: 외부 단편화는 파일들이 디스크 공간에 할당되고 해제되는 과정에서 **사용하지 못하는 작은 빈 공간들이 영역 곳곳에 흩어지는 현상**을 말한다. 예를 들어 연속 할당 방식에서 파일 크기 확장을 위해 새로운 연속 공간을 찾기 어렵게 되는 문제가 외부 단편화로 인한 것이다. 이로 인해 충분한 총 용량이 있어도, 연속된 큰 공간을 찾지 못하면 큰 파일을 저장하지 못하는 상황이 생길 수 있다.
  내부 단편화는 파일이 할당 단위(예: 디스크 블록 또는 클러스터 크기)에 딱 맞게 채워지지 않고 **일부 빈 공간을 남기는 경우**를 말한다. 예를 들어 디스크 블록 크기가 4KB일 때 5KB 크기의 파일은 2개 블록을 차지하지만 두 번째 블록에 3KB 정도의 빈 공간이 남는다. 이 남은 부분이 내부 단편화이다. 내부 단편화는 할당 단위가 클수록 심해질 수 있고, 저장된 파일들의 평균 크기에 따라 전체 용량의 일부 손실로 이어진다. 요약하면, **외부 단편화는 빈 공간들이 조각나서 생기는 문제**이고 **내부 단편화는 할당 단위 내에 사용되지 않는 공간이 생기는 문제**이다.

코드 작성 문제

- **문제:** C 언어의 저수준 파일 I/O 함수(`open`, `read`, `write`)를 사용하여, 주어진 입력 파일 `"input.txt"`의 내용을 `"output.txt"` 파일로 복사하는 간단한 프로그램의 주요 부분을 작성하시오. (파일 열기, 읽기/쓰기 루프, 예외 처리 포함)
  답안 (예시):

```c
#include <fcntl.h>   // open
#include <unistd.h>  // read, write, close
#include <stdio.h>   // perror
#define BUFSIZE 4096
int main() {
  int fd_in, fd_out;
  char buf[BUFSIZE];
  ssize_t bytes;
```

```
    fd_in = open("input.txt", O_RDONLY);
    if (fd_in < 0) { perror("open input"); return 1; }
    fd_out = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd_out < 0) { perror("open output"); close(fd_in); return 1; }

    // 파일 복사 루프
    while ((bytes = read(fd_in, buf, sizeof(buf))) > 0) {
        if (write(fd_out, buf, bytes) != bytes) {
            perror("write error");
            close(fd_in);
            close(fd_out);
            return 1;
        }
    }
    if (bytes < 0) perror("read error");

    close(fd_in);
    close(fd_out);
    return 0;
}
```

해설: 위 코드는 `open`으로 입력 파일을 읽기 전용(`O_RDONLY`)으로 열고, 출력 파일을 없으면 새로 생성(`O_CREAT`), 이미 존재하면 길이를 0으로 잘라내는(`O_TRUNC`) 쓰기 전용으로 연다. 그 후 `read`로 입력 파일에서 버퍼 크기만큼 데이터를 읽고, `write`로 출력 파일에 쓴다. 이를 읽을 데이터가 없을 때까지 반복한다. 읽기(`read`)의 반환값이 0이면 EOF에 도달한 것이고, 양수이면 해당 바이트만큼 읽은 것이다. 오류 처리로 `read`가 0보다 작은 경우와 `write`가 쓴 바이트 수가 기대와 다를 경우를 체크한다. 마지막으로 열린 파일 디스크립터들을 `close`하여 자원을 해제한다. 이처럼 low-level I/O를 사용하면 버퍼링 없이 커널 시스템 콜을 직접 호출하므로, stdio를 사용하는 방법에 비해 제어는 세밀하지만 구현에 주의가 필요하다.

- **문제:** `stat()` 시스템 호출을 이용하여, 특정 파일의 크기와 마지막 수정 시간을 출력하는 코드를 작성하시오. (힌트: `struct stat`과 `stat()` 활용)
  답안:

```
#include <sys/stat.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "사용법: %s <파일경로>\n", argv[0]);
        exit(1);
    }
    struct stat st;
    if (stat(argv[1], &st) < 0) {
        perror("stat");
        exit(1);
    }
    printf("파일 크기: %lld 바이트\n", (long long)st.st_size);
    printf("마지막 수정 시간: %s", ctime(&st.st_mtime));
```

```
    return 0;
}
```

해설: `stat()` 함수를 호출하면 지정한 파일의 inode 메타데이터 정보를 `struct stat`에 채워준다. 여기서 `st_size` 필드가 파일 크기(바이트 단위)이며, `st_mtime`은 마지막 수정 시간이다. `ctime()` 함수를 이용하여 `st_mtime`을 사람 읽을 수 있는 문자열로 변환하여 출력했다. 이 코드는 파일이 존재하지 않거나 접근할 수 없으면 `stat()` 호출이 -1을 반환하며, 그런 경우 오류를 출력하고 종료한다.

Linux 디바이스 드라이버 기초

O/X 문제

- **문제:** 커널 모듈은 일반 사용자 프로그램과 달리 **커널 공간**(Kernel Space)에서 실행된다. (O/X)
  정답: O. 커널 모듈은 커널에 로드되어 동작하므로, 코드가 커널 공간에서 실행된다. 따라서 사용자 공간과 달리 메모리 보호를 받지 못하며, 잘못된 메모리 접근 시 시스템 패닉이 일어날 수 있다. 또한 커널 공간 코드는 `printf`를 쓸 수 없고 대신 `printk`와 같은 함수를 사용해야 한다.

- **문제:** `insmod` 명령으로 커널 모듈을 로드하면 해당 모듈의 `__init`으로 지정된 초기화 함수가 자동으로 호출된다. (O/X)
  정답: O. `insmod` 또는 `modprobe`로 모듈을 삽입하면 커널이 모듈의 초기화 함수를 실행한다. 이 함수는 보통 `module_init()` 매크로로 등록되어 있으며, 여기서 장치 등록이나 자료 구조 초기화를 수행한다. 반대로 `rmmod`로 모듈을 제거할 때는 등록된 `module_exit()` 함수가 호출되어 정리 작업을 한다.

- **문제:** 모든 문자 디바이스 드라이버는 `ioctl` 함수를 반드시 구현해야 한다. (O/X)
  정답: X. `ioctl`은 장치 제어를 위한 선택적(optional) 함수이다. 구현하지 않으면 디폴트로 커널이 `-ENOTTY` (해당 ioctl 없음) 에러를 반환할 뿐, 드라이버로서는 필수 요소가 아니다. 단, 특정 장치에 특화된 제어 기능이 필요할 때만 구현하면 된다.

- **문제:** 커널 함수 `copy_to_user()`와 `copy_from_user()`는 프로세스의 유저 공간 메모리에 접근하기 위한 함수이다. **인터럽트 핸들러 내에서도 안전하게 호출할 수 있다.** (O/X)
  정답: X. `copy_to_user` / `copy_from_user`는 호출 시 **페이지 폴트가 발생할 수 있고**, 현재 실행 중인 프로세스를 잠시 중단하고 메모리 페이징을 수행할 수 있다. 따라서 이는 sleep이 가능한 문맥(프로세스 컨텍스트)에서만 호출이 안전하며, 인터럽트 컨텍스트나 하드웨어 인터럽트 처리 중에는 사용하면 안 된다. 일반적인 드라이버의 `read` / `write` 구현은 프로세스 컨텍스트에서 호출되므로 위 함수를 써도 되지만, 인터럽트 루틴이나 하드웨어 시점에서는 사용자 메모리에 접근해선 안 된다.

- **문제 (오답 유도):** `register_chrdev()` 함수를 호출하면 커널이 자동으로 **/dev** 디렉터리에 해당 디바이스 파일을 생성해준다. (O/X)
  정답: X. `register_chrdev()`는 커널 내부에 드라이버를 등록할 뿐, **디바이스 노드 파일을 생성하지는 않는다.** 디바이스 파일 생성은 `mknod` 명령으로 수동으로 하거나, `udev`와 같은 장치 관리 데몬이 커널 이벤트를 받아 자동 생성하는 것이다. (신형 방식으로 `alloc_chrdev_region`과 `class_create`, `device_create` 등을 사용하면 udev에 의해 자동으로 `/dev`에 노드가 생기지만, `register_chrdev`만으로는 노드 생성이 이뤄지지 않는다.) 시험에서는 드라이버 등록과 장치 파일 생성 과정을 혼동하지 않도록 주의해야 한다.

단답형 문제

- **문제: Major 번호**와 **Minor 번호**란 무엇이며, 각각 어떤 역할을 하는가?
  정답: Major 번호는 커널이 특정 장치 드라이버를 식별하는 데 사용되는 번호이다. 각 문자/블록 장치 드라이버는 고유한 Major 번호를 갖고, 커널은 디바이스 파일의 Major 번호를 보고 어느 드라이버의 함수들을 호출할지 결정한다 [13]. Minor 번호는 개별 드라이버 내에서 장치 인스턴스를 식별하는 번호이다. 예를 들어 하나의 드라이버가 여러 장치를 관리할 때(예: `ttyS0`, `ttyS1` 등), 모두 동일한 Major를 가지되 Minor가 0,1,...로 달라진다. 드라이버는 전달받은 Minor 번호를 보고 어느 장치를 접근해야 하는지 구분할 수 있다.

- **문제:** `MODULE_LICENSE("GPL")`와 같은 매크로를 커널 모듈에 넣는 이유는 무엇인가?
  정답: 이 매크로는 커널에 해당 모듈의 라이선스 정보를 알려주는 용도이다. "GPL"로 설정하면 모듈이 GPL 호환임을 나타내며, 커널은 이 정보를 보고 비 GPL 모듈 로드 시 경고하거나 특정 심볼(내부 API) 사용을 제한한다. 예를 들어 커널은 라이선스가 부여되지 않은 모듈을 로드하면 "tainted kernel" 상태로 표시하며, 일부 내부 심볼 내보내기(EXPORT_SYMBOL_GPL로 정의된 기능)는 GPL 라이선스 모듈에서만 사용할 수 있다. 따라서 올바른 라이선스 선언은 윤리적/기술적으로 모두 중요하다.

- **문제: 커널 공간**과 **유저 공간**의 차이를 설명하시오.
  정답: 커널 공간은 운영체제 커널이 사용하는 메모리 영역으로, 최고 권한으로 코드가 실행되는 영역이다. 여기에선 하드웨어 자원에 접근하거나 모든 메모리를 다룰 수 있지만, 안정성을 위해 일반 프로세스는 접근하지 못한다. 유저 공간은 일반 응용 프로그램이 동작하는 메모리 영역으로, 각 프로세스마다 격리되어 있다. 유저 공간 코드가 실수로 다른 프로세스나 커널 메모리를 침범하지 못하도록 하드웨어와 커널이 보호하며, 접근 시 **페이지 폴트** 등의 예외가 발생한다. 요컨대, 커널 공간은 운영체제의 핵심이 돌아가는 특권 영역이고, 유저 공간은 응용 프로그램이 돌아가는 제한된 영역이다.

- **문제: 문자 디바이스 드라이버**에서 `file_operations` 구조체는 무엇이며 왜 필요한가?
  정답: `struct file_operations`는 해당 드라이버가 지원하는 파일 관련 연산들을 함수 포인터 형태로 모아놓은 구조체이다 [14]. 예를 들어 여기에 `open`, `read`, `write`, `release` 등의 함수 포인터를 채워 넣으면, 사용자가 장치 파일을 열거나 읽을 때 커널이 이 구조체를 찾아 드라이버의 해당 함수를 호출한다. 요약하면, **파일 연산 테이블**로서 드라이버의 인터페이스를 정의한 것이라고 할 수 있다. 이 구조체를 드라이버 등록 시 커널에 제공해야 커널이 시스템콜을 해당 드라이버 코드와 연결할 수 있다.

- **문제: printk** 함수는 무엇이며, 어느 경우에 사용되는가? 또 `printk(KERN_INFO "...")`처럼 지정하는 `KERN_INFO`의 의미는?
  정답: `printk`는 커널에서 제공하는 로그 출력 함수로, 사용자 프로그램의 `printf`와 유사한 역할을 한다. 커널 공간에서는 표준 입출력이 없기 때문에 디버깅이나 정보 출력을 위해 이 함수를 사용하며, 메시지는 주로 **커널 로그 버퍼**에 기록된다. `KERN_INFO` 등은 **로그 레벨**을 나타내는 매크로로, 메시지의 중요도나 우선순위를 커널에 전달한다. 예를 들어 `KERN_ALERT`, `KERN_ERR`, `KERN_WARNING`, `KERN_INFO`, `KERN_DEBUG` 등이 있으며, 작은 등급일수록 긴급한 메시지이다. `KERN_INFO`는 일반 정보 메시지로서, dmesg나 /var/log/kern.log 등을 통해 확인할 수 있다. 이러한 레벨을 지정함으로써 시스템 관리자가 메시지 필터링을 할 수 있고, 너무 사소한(debug) 메시지는 평소 숨기는 등의 조치가 가능하다.

- **문제 (오답 유도):** `mutex`와 `spinlock`은 모두 동기화용 락인데, **인터럽트 컨텍스트에서도 둘 다 사용할 수 있다.** (예/아니오)
  정답: 아니오. `mutex`는 잠금 시 현재 쓰레드를 sleep시켰다가 락이 풀리면 깨우는 방식으로 구현되기 때문에 **인터럽트 컨텍스트에서는 사용할 수 없다.** 인터럽트 컨텍스트에서는 sleep이 불가능하므로, 대신 `spinlock` 같은 락을 사용해야 한다. `spinlock`은 락이 풀릴 때까지 CPU를 계속 점유하여 기다리는 방식으로, 짧은 기간 임계구역 보호나 인터럽트 내 동기화에 사용된다. 다만 `spinlock`도 인터럽트 컨텍스트에서 사용할 때에는 `spin_lock_irqsave` 같이 인터럽트를 적절히 관리하는 함수를 써야 한다. 요약하면, **인터럽트 컨텍스트에서는 mutex를 사용할 수 없고** (예외적으로 trylock 후 실패시 바로 포기하는 것은 가능하나 일반적이지 않음), 대신 spinlock 등을 사용해야 함을 기억해야 한다.

서술형 문제

- **문제: 커널 모듈**을 개발하고 로드하여 장치를 사용하는 전체 과정을 설명하시오. (모듈 컴파일, 로드, 장치 파일 생성, 테스트, 언로드 단계를 포함)
  정답: 먼저 커널 모듈 소스 코드를 작성하고 Makefile을 통해 커널 빌드 환경에서 컴파일한다. 이때 커널 헤더와 적절한 컴파일 옵션이 필요하다. 컴파일이 성공하면 `.ko` 확장자의 모듈 파일이 생성된다.
  다음으로 `sudo insmod module.ko` 명령으로 모듈을 커널에 로드한다. 로드 시에 모듈의 초기화 함수가 실행되어 (예: `module_init`으로 등록된 함수) 드라이버가 자신을 등록한다. 문자 디바이스의 경우 `register_chrdev_region` 또는 `register_chrdev` 등을 통해 Major/Minor 번호와 `file_operations`를 커널에 등록하게 된다.
  장치 파일을 생성하는 단계는 수동 또는 자동으로 이뤄지는데, 수동이라면 `mknod` 명령을 사용하여 `/dev` 디렉터리 밑에 해당 Major/Minor 번호로 캐릭터 디바이스 노드를 만든다. 예를 들어 Major 240, Minor 0인 장치라면 `mknod /dev/mydev c 240 0` 형태가 된다 [15]. 현대적인 시스템에서는 모듈 로드시 `udev`가 `sysfs` 이벤트를 받아 자동으로 `/dev`에 장치 파일을 생성해주기도 한다 (드라이버가 `class_create` 등을 호출한 경우).
  이렇게 `/dev`에 장치 노드가 준비되면, 사용자 공간에서 일반 파일처럼 `open("/dev/mydev")`으로 장치를 열고 `read`/`write` 등으로 드라이버와 상호작용할 수 있다. 이때 드라이버의 구현에 따라 실제 하드웨어를 제어하거나, 메모리 연산 등을 수행하여 결과를 돌려준다.
  드라이버 테스트가 끝나면 `sudo rmmod module` 명령으로 모듈을 언로드(제거)한다. 이 과정에서 모듈의 종료 함수(예: `module_exit` 등록 함수)가 호출되어 장치 등록을 해제하고(`unregister_chrdev` 등), 할당했던 자원들을 반납한다. 모듈이 언로드되면 `/dev`의 장치 파일은 더 이상 유효하지 않으므로 필요하면 삭제한다.
  전체적으로, **컴파일 → insmod → (필요 시 mknod) → 장치 접근 → rmmod**의 흐름으로 커널 모듈 사용이 이루어진다.

- **문제: 블로킹 I/O**와 **논블로킹 I/O**의 개념을 설명하고, 문자 디바이스 드라이버에서 블로킹 I/O를 구현하기 위한 방법을 예를 들어 설명하시오.
  정답: 블로킹 I/O란 어떤 입출력 동작(read/write 등)이 즉시 완료되지 못할 경우, 해당 호출을 수행한 프로세스를 **대기 상태로 두고** I/O가 가능해질 때까지 기다리게 하는 방식이다. 이렇게 하면 CPU를 효율적으로 사용할 수 있지만, 그 쓰레드는 응답을 기다리는 동안 다른 작업을 못한다. 반면 논블로킹 I/O는 I/O 호출이 즉시 결과를 돌려주는데, 만약 할 수 있는 작업이 없으면 에러코드(`-EAGAIN`) 등을 리턴하여 프로세스가 기다리지 않고 다른 작업을 할 수 있게 한다. 논블로킹 모드는 주로 `fcntl`로 파일 디스크립터를 `O_NONBLOCK` 플래그로 설정해서 이용한다.
  문자 디바이스 드라이버에서 블로킹 I/O를 구현하려면, 커널에서 해당 프로세스를 sleep시켰다가 조건이 충족되면 깨우는 메커니즘이 필요하다. 이를 위해 주로 **wait queue(대기 큐)**를 사용한다. 예를 들어 생산자-소비자 문제를 생각하면, 드라이버 내부 버퍼에 데이터가 없을 때 `read`를 호출한 프로세스를 `wait_event_interruptible(queue, 데이터존재)` 등을 통해 sleep시킨다. 그리고 다른 쓰레드나 인터럽트가 데이터를 넣으면 `wake_up_interruptible(&queue)`를 호출하여 잠든 프로세스를 깨운다. 이때 프로세스의 `read` 호출은 깨어나서 데이터를 복사한 후 반환하게 된다. 반대로 버퍼가 가득 찼을 때 `write` 호출을 블로킹하려면 비슷한 방식으로 작성한다.
  예를 들어 SCULL 파이프 장치를 보면, `read` 쪽은 버퍼에 데이터가 없으면 `wait_event_interruptible`로 데이터를 쓸 때까지 잠들고, `write` 쪽은 버퍼에 빈 공간이 없으면 데이터가 읽혀 빈 공간이 생길 때까지 잠든다. 이처럼 **조건이 만족될 때까지 sleep→깨움**의 흐름으로 블로킹 I/O를 구현한다. 반면 논블로킹 모드(`O_NONBLOCK`)로 열린 경우에는 기다리지 않고 곧바로 -EAGAIN을 리턴하도록 조건 분기를 넣는다. 이러한 매커니즘을 통해 드라이버는 블로킹/논블로킹 동작을 모두 지원할 수 있다.

- **문제: 동시성 제어**가 필요한 이유와, 리눅스 커널에서 제공하는 동기화 기법 두 가지를 설명하시오 (사용 예시 포함).
  정답: 다중 프로세서 환경이나 인터럽트, 멀티쓰레드 상황에서는 동일한 자원에 동시에 접근하여 데이터 불일치나 경합(race condition)이 발생할 수 있다. 이를 막기 위해 동시성 제어(또는 동기화)가 필요하다.

커널에서 제공하는 대표적인 동기화 기법으로 **스핀락(spinlock)**과 **세마포어/뮤텍스(semaphore/mutex)**가 있다.

- **스핀락:** 간단한 플래그 락으로, 락을 얻을 때까지 호출자가 계속 CPU를 점유하며 바쁘게 대기(spin) 하는 방식이다. 락이 비교적 짧은 시간 내에 해제되는 상황에 적합하며, 인터럽트 처리 같은 컨텍스트에서도 사용할 수 있는 것이 장점이다. 예를 들어 다중 CPU 환경에서 전역 변수 접근을 보호할 때 `spin_lock(&my_lock); ... critical section ...; spin_unlock(&my_lock);` 형태로 사용한다. 단, 락이 걸린 동안에는 반복 대기하며 CPU를 양보하지 않으므로, 장기 작업에는 부적합하다. 또한 sleep을 유발하는 코드를 임계구역 내에서 수행하면 안 된다.
- **뮤텍스/세마포어:** 락을 얻지 못하면 호출 프로세스를 sleep 상태로 전환하여 CPU를 양보하는 락이다. **뮤텍스**는 동시에 하나의 쓰레드만 임계영역에 들어갈 수 있게 하는 이진 세마포어이다. `mutex_lock(&my_mutex); ...; mutex_unlock(&my_mutex);` 형태로 쓰며, 락이 이미 잡혀 있으면 현재 쓰레드는 sleep되었다가 락이 풀릴 때 wakeup 되어 이어 실행된다. 이 방식은 긴 임계구역이나 선점 환경에서 효율적이지만, **인터럽트 컨텍스트에서는 사용 불가**하다 (sleep 불가 환경). **세마포어**는 일반화된 형태로, 카운팅 세마포어를 사용하면 임계 리소스 개수가 N개일 때 동시 최대 N개의 쓰레드까지 허용하는 등도 가능하다. 사용 예로는 `down(&sem)` / `up(&sem)`이 있으며, `down` 시 자원이 없으면 sleep했다가 up으로 신호가 오면 깨어난다.
  이 밖에도 커널은 원자적 연산(atomic_t, bit operations), RW락, RCU 등의 여러 동기화 도구를 제공한다. 개발자는 상황에 맞게 올바른 기법을 선택해야 한다. (예: 인터럽트와 보텀하프 간 보호는 spinlock+irqsave, 프로세스 간 긴 임계구역은 mutex 등.)

코드 작성 문제

- **문제:** 커널에서 동적 메모리를 할당하고 해제하는 간단한 예시 코드를 작성하시오. (예: `kmalloc`로 버퍼 할당 후 사용, `kfree`로 해제)
  답안:

```
#include <linux/slab.h>  // kmalloc, kfree 등
...
void example(void) {
  char *buffer;
  // GFP_KERNEL 플래그로 할당 (잠잘 수 있는 컨텍스트에서 호출 가정)
  buffer = kmalloc(100, GFP_KERNEL);
  if (!buffer) {
    printk(KERN_ERR "메모리 할당 실패\n");
    return;
  }
  // 할당된 메모리 사용
  snprintf(buffer, 100, "Hello Kernel World");
  printk(KERN_INFO "buffer: %s\n", buffer);
  // 메모리 해제
  kfree(buffer);
}
```

해설: 이 코드에서는 `kmalloc`을 이용해 100바이트 크기의 메모리를 할당하고 있다. GFP_KERNEL 플래그는 일반적인 컨텍스트에서 사용할 수 있는 메모리 할당 옵션으로, 필요시 프로세스를 sleep시켜 메모리를 확보할 수도 있음을 의미한다. 할당 실패 시 NULL을 반환하므로 체크한다. 할당된 메모리는 일반 커널 힙 영역이며, 사용 후에는 반드시 `kfree`로 해제하여야 메모리 누수가 발생하지 않는다. (실제로 드라이버 코드에서는 init 과정에서 kmalloc한 뒤 exit에서 kfree하는 패턴이 흔히 나타난다.) 이 코드는 커널 모듈의 일부분이라고 가정할 수 있다.

- **문제: 문자 디바이스 드라이버**에서 사용되는 `file_operations` 구조체를 초기화하는 코드를 작성하고, 대표적인 항목 3개를 설명하시오.
  답안:

```
#include <linux/fs.h>
// 드라이버에서 사용할 함수 선언
static int my_open(struct inode *inode, struct file *filp);
static ssize_t my_read(struct file *filp, char __user *buf, size_t count, loff_t *pos);
static ssize_t my_write(struct file *filp, const char __user *buf, size_t count, loff_t *pos);
static int my_release(struct inode *inode, struct file *filp);

static struct file_operations my_fops = {
  .owner   = THIS_MODULE,    // 모듈 소유자 (모듈 사용 중 언로드 방지용)
  .open    = my_open,        // 디바이스 열때 호출될 함수
  .read    = my_read,        // 디바이스에서 읽을 때 호출될 함수
  .write   = my_write,       // 디바이스에 쓸 때 호출될 함수
  .release = my_release,     // 디바이스 닫을 때 호출될 함수
  // .llseek 등 필요한 경우 추가 구현
};
```

해설: 위 코드에서는 문자 디바이스의 동작을 정의하는 `my_fops` 구조체를 설정하였다. `owner` 필드는 보통 `THIS_MODULE`로 지정하여 이 장치가 열려 있는 동안 모듈이 언로드되지 않도록 한다. `open`은 `open()` 시스템콜에 대응하며, 장치 파일이 열릴 때 호출된다. 여기에서 드라이버는 초기화나 자원 할당, `private_data` 설정 등을 할 수 있다. `read`는 사용자 영역에서 `read()`를 호출할 때 실행되는 함수로, 장치에서 데이터를 읽어 사용자 버퍼로 복사하는 역할을 한다. `write`는 `write()` 호출에 대응하여, 사용자로부터 데이터를 받아 장치에 기록하는 작업을 한다. `release`는 파일을 닫을 때 호출되어 자원 정리나 종료 로그 출력 등을 담당한다. 이 외에도 필요에 따라 `llseek`(파일 위치 변경), `ioctl`(장치 제어), `poll`/`compat_ioctl` 등의 필드를 넣을 수 있다. 이 구조체를 커널에 등록하면 커널이 해당 장치의 시스템콜을 처리할 때 이들 함수들을 호출해준다.

- **문제: 인터럽트 핸들러**를 등록하고 해제하는 예제 코드를 작성하시오. (예: IRQ 번호 1에 대한 핸들러 설치 및 제거)
  답안:

```
#include <linux/interrupt.h>
#define MY_IRQ_NUM 1  // 예: 키보드 인터럽트 번호 1
static irqreturn_t my_interrupt_handler(int irq, void *dev_id) {
  // 인터럽트 발생 시 처리할 일 수행
  printk(KERN_INFO "Interrupt %d 발생!\n", irq);
  return IRQ_HANDLED;
}

static int __init mydrv_init(void) {
  if (request_irq(MY_IRQ_NUM, my_interrupt_handler, IRQF_SHARED,
          "my_device", (void*)MY_IRQ_NUM)) {
    printk(KERN_ERR "IRQ %d 등록 실패\n", MY_IRQ_NUM);
    return -1;
  }
```

```
    printk(KERN_INFO "IRQ %d 핸들러 등록 성공\n", MY_IRQ_NUM);
    return 0;
}

static void __exit mydrv_exit(void) {
    free_irq(MY_IRQ_NUM, (void*)MY_IRQ_NUM);
    printk(KERN_INFO "IRQ %d 핸들러 해제\n", MY_IRQ_NUM);
}
```

해설: 이 코드는 키보드 인터럽트(IRQ 1)를 예시로 인터럽트 핸들러를 등록/해제하는 방법을 보여준다. `request_irq` 함수를 통해 커널에 인터럽트 핸들러를 등록하며, 인자로 IRQ 번호, 핸들러 함수 포인터, 플래그, 장치명, 장치 식별자를 넘긴다. 여기서는 IRQ 공유를 허용(`IRQF_SHARED`)하고, 식별자로 IRQ 번호 자체를 포인터로 변환하여 사용했다. 실제 드라이버에서는 보통 `dev_id`로 장치 구조체 포인터 등을 넘겨 각기 장치를 구분한다. 인터럽트가 발생하면 커널이 `my_interrupt_handler`를 호출하며, 이 함수에서 반드시 `IRQ_HANDLED` 또는 `IRQ_NONE` 등을 반환해야 한다. 코드에서는 단순히 로그만 출력하고 IRQ를 처리했다고 알린다. 모듈이 언로드될 때는 `free_irq`로 등록 해제를 해야 한다. 등록 시와 같은 IRQ 번호와 dev_id를 사용하여 매칭된 핸들러를 제거한다. 이 작업을 누락하면 모듈 언로드 후에도 인터럽트가 발생하면 잘못된 주소를 호출하여 시스템이 불안정해진다.

---

1 3 4 5 6 7 8 SCULL 커널 모듈 소스코드 (scull-master).pdf
file://file-TPPQMCtuVRUhPfPp1GaaGU

2 9 10 14 9. linux-driver-formatted.txt
file://file-NWt7se4qa7dcEw75cihZ2R

11 12 13 15 10. Character Device Driver.pdf
file://file-Rzdv6YTJogLgCSC52JMSnP