42

막간: 파일과 디렉터리

지금까지 운영체제를 구성하는 두 개의 핵심 개념을 살펴보았다. CPU를 가상화한 "프로세스"와 메모리를 가상화한 "주소 공간"이다. 이 개념들은 서로 협력하여 응용 프로그램들이 서로 독립된 세계에서 실행될 수 있도록 해준다. 각자의 프로세서(또는 프로세서들)를 갖고 있는 것처럼 해주며 각자의 메모리가 있는 것처럼 만들어 준다. 이러한 환상이 시스템을 다루는 프로그램 개발을 더욱 쉽게해준다. 데스크탑과 서버에서 뿐만 아니라 모바일폰을 포함한 프로그래밍이 가능한 모든 플랫폼 등에서 광범위하게 사용되고 있다.

이번 장에서는 **영속 저장 장치**(**persistent storage**)라고 하는 또 하나의 핵심적인 가상화의 퍼즐 조각을 추가한다. **하드 디스크 드라이브** 또는 좀 더 최근의 **솔리드스테이트 드라이브**(**Solid-state storage**, **SSD**)와 같은 저장 장치는 영구적으로 정보를 저장한다(또는 최소한 아주 오랜 기간 동안). 전원 공급이 차단되면 내용이 사라지는 메모리와 다르게 영속 저장 장치는 그러한 상황에서도 그대로 데이터를 보존한다. 운영체제는 그런 장치들을 좀 더 신중하게 다루어야 한다. 사용자가 정말 소중하게 생각하는 데이터를 보관하는 곳이기 때문이다.

핵심 질문: 어떻게 영속 장치를 관리하는가
운영체제가 영속 장치를 어떻게 관리해야 할까? API들은 어떤 것이 있는가? 구현의 중요한 측면은 무엇인가?

앞으로 살펴볼 장들에서는 영속 데이터를 관리하는 핵심 기술들을 살펴볼 것이며 성능과 신뢰성을 향상시키는 기법들을 중점으로 다루겠다. 그 전에 이장에서는 API에 대한 개론부터 시작한다. UNIX 파일 시스템을 사용할 때 만날 수 있는 인터페이스들이다.

42.1 파일과 디렉터리

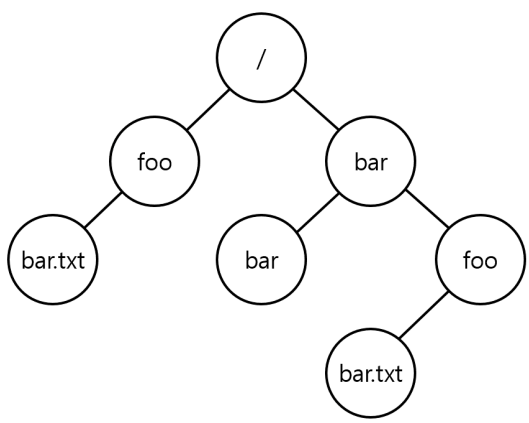
저장 장치의 가상화에 대한 두 가지 주요 개념이 개발되었다. 첫 번째는 **파일**이다. 파일은 단순히 읽거나 쓸 수 있는 순차적인 바이트의 배열이다. 각 파일은 **저수준의 이름**(**low-level name**)을 갖고 있으며 보통은 숫자로 표현되지만 사용자는 그 이름에 대해서 알지

못한다(앞으로 보게 될 것이다). 역사적인 이유로 인해서 이 저수준의 이름을 **아이노드 번호(inode number)**라고 부른다. 앞으로 살펴볼 장에서 이 아이노드에 대해서 더 자세히 살펴볼 것이지만 지금 각 파일은 아이노드 번호와 연결되어 있다고 이해하자.

대부분 시스템에서 운영체제는 파일의 구조를 모른다(예를 들어 어떤 파일이 그림인지 문서인지 C 코드인지 모른다.). 파일 시스템의 역할은 그러한 데이터를 디스크에 안전히 저장하고, 데이터가 요청되면 처음 저장했던 데이터를 돌려주는 것이다. 이렇게 하는 것이 보기보다 쉽지 않다!

두 번째 개념은 **디렉터리**이다. 파일과 마찬가지로 디렉터리도 저수준의 이름(예, 아이노드 번호)을 갖는다. 하지만 파일과는 달리 디렉터리의 내용은 구체적으로 정해져 있다. 디렉터리는 〈사용자가 읽을 수 있는 이름, 저수준의 이름〉 쌍으로 이루어진 목록을 갖고 있다. 저수준 이름으로 "10"을 갖고 있는 파일이 있는데, 그 파일은 사용자가 알아볼 수 있는 "foo"라는 이름을 갖고 있다고 해 보자. "foo"가 들어 있는 디렉터리에는 ("foo", "10")이라는 항목이 있어서 사용자가 읽을 수 있는 이름과 저수준의 이름을 연결하고 있다. 디렉터리의 각 항목은 파일 또는 다른 디렉터리를 가리킨다. 디렉터리 내에 다른 디렉터리를 포함함으로써 사용자는 모든 파일들과 디렉터리들이 저장되어 있는 임의의 **디렉터리 트리(directory tree**, 또는 **디렉터리 계층(directory hierarchy))**을 구성할 수 있다.

디렉터리 계층은 **루트 디렉터리(root directory)**부터 시작하며(UNIX 기반의 시스템에서 루트 디렉터리는 `/`으로 표현된다), 원하는 파일이나 디렉터리의 이름을 표현할 때까지 **구분자(separator)**를 사용하여 **하위 디렉터리**를 명시할 수 있다. 예를 들어서 사용자가 `foo`라는 디렉터리를 루트 디렉터리 `/` 아래에 생성했다고 해 보자. 그리고 `foo` 디렉터리 안에 `bar.txt`라는 파일을 생성하였다면, 그 파일들의 **절대 경로명(absolute pathname)**은 `/foo/bar.txt`로 표현된다. 그림 42.1에 나타난 좀 더 복잡한 디렉터리 트리를 살펴보자. 그림에서 유효한 디렉터리들은 `/`, `/foo`, `/bar`, `/bar/bar`, `/bar/foo`이고 유효한 파일들은 `/foo/bar.txt`와 `/bar/foo/bar.txt`이다. 디렉터리들과 파일들은 파일 시스템 트리 안에서 서로 다른 위치에 있는 경우, 동일한 이



〈그림 42.1〉 **디렉터리 트리의 예제**

팁: 이름을 주의해서 짓자

컴퓨터 시스템에서 이름 짓기는 매우 중요한 부분이다 [SK09]. UNIX 시스템의 경우 당신이 생각할 수 있는 거의 대부분의 것이 파일 시스템을 통해 명명된다. 평범하고 오래된 파일 시스템 같아 보이는 것에서도 단순 파일들뿐만 아니라 장치와 파이프 그리고 프로세스들을 만날 수 있다 [Kil84]. 이름을 일관성 있게 관리하면 시스템의 개념적 모델을 간단하게 유지하며, 시스템을 단순하고 모듈화가 잘되도록 만들어 준다. 그러므로 어떤 시스템이나 인터페이스를 만들때마다 어떤 이름으로 만들 것인지에 대해 신중해야 한다.

름을 가질 수 있다(예, 그림에서 `bar.txt`라는 파일 이름이 두 번 나타나는데, 하나는 `/foo/bar.txt`이고, 다른 하나는 `/bar/foo/bar.txt`이다).

이 예제에서 파일 이름이 두 부분으로 구성되어 있다는 것을 알 수 있다. `bar`와 `txt`가 마침표로 분리되어 있다. 첫 번째 부분은 임의의 이름인 반면에 두 번째 부분은 대체적으로 파일의 **종류**를 나타내기 위해 사용된다. C 코드의 경우 `.c`로 되어 있으며 이미지의 경우는 `.jpg` 또는 음악 파일이면 `.mp3`를 갖고 있다. 하지만 대부분 **관용적**(**convention**)일 뿐이다. 파일 이름이 `main.c` 라고 해서 내용이 반드시 C 소스 코드일 필요는 없다.

파일 시스템이 제공하는 훌륭한 기능 하나를 보았다. 파일들을 효율적으로 명명할 수 있는 기능이다. 어떤 자원을 접근하는 가장 첫 단계는 그 대상의 **이름**을 아는 것이기 때문에 시스템에서 이름짓기 기능은 매우 중요하다. UNIX 시스템상에서 파일 시스템은 디스크, USB 메모리, CD-ROM등 다양한 장치에 존재하는 파일들을 통합된 방법으로 접근할 수 있도록 한다. 모든 파일들은 하나의 디렉터리 트리 상에 위치한다.

42.2 파일 시스템 인터페이스

이번에는 파일 시스템 인터페이스를 좀 더 상세하게 논의해 보자. 파일의 생성과 접근 그리고 삭제 등의 기본부터 시작해 보자. 매우 단순하다고 생각될 수도 있지만, 파일 삭제를 담당하는 `unlink()`라는 살짝 혼란스런 시스템 콜도 다룰 것이다. 이 장이 끝날 때는 더이상 혼란스럽지 않기 바란다.

42.3 파일의 생성

아주 기본적인 연산인 파일의 생성부터 시작해 보자. `open` 시스템 콜을 사용하여 파일을 생성할 수 있다. `open()`을 호출하면서 `O_CREAT` 플래그를 전달하면 프로그램은 새로운 파일을 만들 수 있다. 다음은 현재의 디렉터리에 "foo"라는 파일을 만드는 코드이다.

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
```

`open()`은 다수의 플래그를 받는다. 이 예제에서 프로그램은 `O_CREAT`로 파일을 생성하고, 파일이 열렸을 때 쓰기만 가능하도록 `O_WRONLY` 플래그를 사용하였다. 그리고

여담: `creat()` 시스템 콜

파일을 생성하는 방법은 `creat()`를 다음과 같이 호출하는 것이다

```
int fd = creat("foo");
```

`creat()`는 `O_CREAT | O_WRONLY | O_TRUNC` 플래그를 사용하는 `open()`이라고 생각할 수 있다. `open()`이 파일을 생성할 수 있기 때문에 `creat()`의 사용이 인기를 잃어가고 있다(사실 `open()`을 호출하는 라이브러리 함수로 만들 수 있다). 하지만 이것은 UNIX에서 역사적으로 특별한 위치를 갖고 있다. Ken Tompson에게 UNIX에서 재설계를 하고 싶은 부분이 있다면 어떤 것이 있느냐라는 질문을 했을 때 그는 이렇게 대답했다. "creat의 철자에 e를 더하겠다."

`O_TRUNC` 플래그를 사용하여 파일이 이미 존재할 때는 파일의 크기를 0 byte로 줄여서 기존 내용을 모두 삭제한다.

`open()`의 중요한 항목은 리턴값이다: **파일 디스크립터(file descriptor)**. 파일 디스크립터는 프로세스마다 존재하는 정수로서 UNIX 시스템에서 파일을 접근하는 데 사용된다. open된 파일을 읽고 쓰는 데 사용된다. 물론 해당 파일에 대한 권한을 갖고 있어야 한다. 이러한 측면에서 파일 디스크립터는 **capability(capability)**이다 [Lev84]. 특정 동작에 대한 수행 자격을 부여하는 핸들이다. 파일 디스크립터를 파일 객체를 가리키는 포인터로 볼 수도 있다. 그러한 객체를 생성하면, `read()` 또는 `write()`와 같은 다른 "메소드"로 파일에 접근할 수 있다. 곧 파일 디스크립터가 어떻게 사용되는지 살펴볼 것이다.

42.4 파일의 읽기와 쓰기

파일이 있으면, 그 파일들을 당연히 읽거나 쓰고 싶을 것이다. 이미 존재하고 있는 파일을 읽는 것부터 시작하자. 커맨드 라인을 사용 중이라면 `cat`이라는 프로그램을 사용하여 파일의 내용을 화면에 덤프할 수 있다.

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
```

이 코드에서는 `echo`의 출력을 파일 `foo`로 전송(redirect)하여 그 파일에 "hello"를 저장하도록 하였다. 그런 후에 `cat` 명령어로 파일 내용을 확인하였다. `cat` 프로그램은 어떻게 파일 `foo`에 접근할까?

이것을 알아보기 위해서 프로그램이 호출하는 시스템 콜을 추적하는 도구를 사용한다. Linux에는 **strace**라는 도구가 있고, 다른 시스템도 유사한 도구들이 있다(Mac OS X에서는 **dtruss**를 살펴보고, 오래된 UNIX의 변종들에서는 **truss**를 살펴보자). `strace`가 하는 일은 프로그램이 실행되는 동안에 호출된 모든 시스템 콜을 추적하고, 그 결과를 화면에 보여준다.

팁: strace(및 그 유사한 툴들)를 사용하자

strace 도구는 프로그램이 무엇을 하는지 볼 수 있도록 해주는 어마어마한 도구이다. strace를 실행하면 프로그램이 호출하는 시스템 콜들은 무엇이며 사용하는 인자와 리턴 코드는 무엇인지를 추적할 수 있으며, 따라서 프로그램이 무엇을 하고 있는지에 대한 상당히 깊은 이해가 가능하다.

이 도구는 상당히 유용한 인자들이 있다. 예를 들면 -f를 사용하면 fork된 자식 프로세스도 추적할 수 있다. -t를 사용하면 호출 시각을 알려준다. 그리고 -e trace=open,close,read,write 라고 하면 다른 시스템 콜은 무시하고 해당 시스템 콜들만 추적한다. 다른 강력한 플래그들이 더 있으니 man 페이지를 잘 읽어보고 이 놀라운 도구를 활용하는 법을 알아보라.

다음의 예를 통해 strace를 사용하여 cat이 어떤 동작을 하는지 알아보자(가독성을 위해서 몇몇 호출들은 삭제함).

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE) = 3
read(3, "hello\n", 4096) = 6
write(1, "hello\n", 6) = 6
hello
read(3, "", 4096) = 0
close(3) = 0
...
prompt>
```

cat이 가장 먼저 하는 것은 파일을 읽기 위해서 여는 것이다. 몇 가지 짚고 넘어갈 사항이 있다. 파일은 O_RDONLY라는 플래그가 나타내는 것처럼 읽을 수만 있도록 열렸다(쓸 수는 없음). 두 번째는 O_LARGEFILE 플래그를 사용하여 64 bit 오프셋이 사용되도록 설정하였다. 세 번째는 open()이 성공한 후에 3이라는 값을 파일 디스크립터로 리턴하였다.

어째서 첫 번째 open()임에도 불구하고 예상과 달리 0 또는 1이 아닌 3을 리턴하였을까? 프로세스가 이미 세 개의 파일을 열어 놓았기 때문이다. 이미 열려진 세 개의 파일은 표준 입력과 표준 출력, 그리고 오류 메시지를 기록할 수 있는 표준 에러이다. 각각의 파일 디스크립터는 0, 1 그리고 2로 표현된다. 다른 파일을 처음으로 열게 되면(cat이 하는 듯이), 거의 확실하게 파일 디스크립터는 3일 것이다.

파일 열기가 성공하면 cat은 read() 시스템 콜을 사용하여 파일에서 몇 바이트씩 반복적으로 읽는다. read()의 첫 번째 인자는 파일 디스크립터로서 파일 시스템에 어떤 파일을 읽을 것인지 알려준다. 프로세스는 동시에 여러 파일을 열 수 있기 때문에, 디스크립터는 운영체제가 read 명령이 읽어야 할 파일을 알 수 있게 한다. 두 번째 인자는 read() 결과를 저장할 버퍼를 가리킨다. 위의 시스템 콜 추적 예제에서 strace는 읽은 결과인 "hello"를 두 번째 인자 위치에 표시하였다. 세 번째 인자는 버퍼의 크기로서

여기서는 4 KB이다. `read()`가 성공적으로 리턴하며 읽은 바이트 수를 반환한다("hello"의 5개의 문자와 줄의 끝을 표시하는 문자 하나가 있기 때문에 6을 반환함).

이 시점에서 `strace`의 또 다른 흥미로운 점이 있다. `write()` 시스템 콜이 결과를 쓰는 대상 파일로 파일 디스크립터 1번을 사용하는 것이다. 앞서 설명했듯이 이 디스크립터는 표준 출력(STDOUT) 으로서 "hello"라는 단어를 화면에 나타내기 위해 사용되고, `cat`이 하기로 되어 있는 작업이다. `cat` 프로그램이 `write()`를 직접 호출하는 것일까? (만약 상당히 최적화가 되었다면) 그럴지도 모른다. 그렇지 않다면 `cat`은 라이브러리 루틴인 `printf()` 를 호출했을 것이다. 내부적으로 `printf()`는 전달 받은 문자열에 적절한 포멧을 적용한 후 결과를 표준 출력을 대상으로 `write()`를 호출하여 화면에 출력한다.

출력한 이후 `cat` 프로그램은 파일의 내용을 더 읽으려고 시도하고, 파일에 남은 바이트가 없기 때문에 `read()`는 0을 리턴한다. 프로그램은 리턴 값으로 파일을 끝까지 다 읽었음을 알게 된다. 그런 후 프로그램은 해당 파일 디스크립터를 인자로 `close()`를 호출하여 "foo"라는 파일에서 할 일이 끝났음을 표시한다. 이제 파일은 닫혔으며 읽기 작업은 완료된다.

파일에 쓰는 것도 비슷한 단계를 거친다. 먼저 파일을 쓰기 위해 열고 `write()` 시스템을 호출한다. 파일이 큰 경우 `write()`시스템 콜을 반복적으로 호출할 수 있다. 그 후에 `close()`가 호출된다. `strace`로 파일 쓰기 흐름을 수집해 보자. 당신 스스로가 작성한 프로그램을 사용할 수도 있고 `dd` 도구를 사용할 수도 있다. `dd if=foo of=bar`, 와 같이 실행할 수 있다.

42.5 비 순차적 읽기와 쓰기

지금까지 파일을 읽고 쓰는 과정을 논의하였는데, 모든 접근은 **순차적**이었다. 즉, 처음부터 파일을 끝까지 읽었고, 쓸 때도 처음부터 끝까지 기록하였다.

그렇지만 때로는 파일의 특정 오프셋부터 읽거나 쓰는 것이 유용할 때가 있다. 예를 들어 문서의 인덱스를 만들고 특정 단어를 찾는다고 해 보자. 그러한 경우 문서 내의 **임의**의 오프셋에서 읽기를 수행해야 할 것이다. 이것을 위해서 `lseek()`라는 시스템 콜을 사용한다. 여기에 함수의 프로토타입이 있다.

```
off_t lseek(int fildes, off_t offset, int whence);
```

첫 번째 인자는 파일 디스크립터다. 두 번째 인자는 `offset`으로 파일의 특정 위치(**file offset**)를 가리킨다. 세 번째 인자는 역사적인 이유로 `whence`라고 부르며 탐색 방식을 결정한다. `man` 페이지에는 다음과 같이 나와 있다.

```
whence가 SEEK_SET이면 오프셋은 offset 바이트로 설정된다.
만약 whece가 SEEK_CUR이면 오프셋은 현재 위치에 offset 바이트를 더한 값으로 설정된다.
만약 whence가 SEEK_END이면 오프셋은 파일의 크기에 offset 바이트를 더한 값으로 설정된다.
```

이 설명에서 알 수 있듯이 프로세스가 `open()`한 각 파일에 대해 운영체제는 "현재" 오프셋을 추적하여 다음 읽기 또는 쓰기 위치를 결정한다. 열린 파일의 **개념**에는 현재

여담: lseek()를 호출한다고 디스크 탐색을 하는 것은 아니다

잘못 지어진 lseek()는 디스크와 그 위의 파일 시스템의 동작을 이해하려는 많은 학생들을 혼란에 빠트린다. 그 둘을 혼동하지 말자! 다음 번의 읽기 또는 쓰기의 시작 위치를 변경하기 위해서 lseek()는 단순히 특정 프로세스를 위해서 운영체제가 관리하는 메모리 내의 변수를 변경한다. 디스크 탐색은 디스크에 요청된 읽기 또는 쓰기가 직전에 수행했던 읽기 또는 쓰기와 같은 트랙이 아닌 경우에만 수행된다. 그래서 디스크 탐색은 헤드의 이동을 수반한다. 이 개념을 더 혼란스럽게 만드는 것은 임의의 위치를 읽거나 쓰기 위해서 lseek()를 호출하는 경우이다. 파일의 임의의 부분을 읽고 쓸 때에는 실제로 많은 디스크 탐색이 발생한다. lseek()를 호출하는 것은 분명히 다음 번의 읽기와 쓰기를 위한 탐색을 유도하기는 하지만 그렇다고 해서 실제로 I/O가 발생하는 것은 절대로 아니다.

오프셋이 포함된다. 오프셋은 두 가지 중 하나의 방법으로 갱신된다. 첫째 N 바이트를 읽거나 쓸 때 현재 오프셋에 N이 더해진다. 따라서 각 읽기 또는 쓰기는 *암묵적*으로 오프셋을 갱신한다. 둘째, 앞서 본 것처럼 lseek로 *명시적*으로 오프셋을 변경하는 것이다.

lseek()는 디스크 암을 이동시키는 디스크의 **탐색**(**seek**) 작업과 아무 관계도 없다는 것에 유의해야 한다. lseek() 호출은 커널 내부에 있는 변수의 값을 변경한다. I/O를 처리할 때 디스크 헤드가 어디에 있는지에 따라서 요청을 처리하기 위해 실제 디스크 암을 이동하는 탐색 과정을 수행할 수도 있고 하지 않을 수도 있다.

42.6 fsync()를 이용한 즉시 기록

write() 호출의 목적은 대부분 해당 데이터를 가까운 미래에 영속 저장 장치에 기록해 달라고 파일 시스템에게 요청하는 것이다. 성능상 이유로 파일 시스템은 쓰기들을 일정 시간(예를 들어 5초 또는 30초)동안 메모리에 모은다(**버퍼링**). 일정 간격으로 쓰기요청(들)이 저장 장치에 전달된다. 응용 프로그램의 입장에서는 write() 호출 즉시 쓰기가 완료된 것처럼 보인다. 드물게 (예, write()를 호출하였지만 디스크에 쓰기 직전에 기계가 크래시한 경우) 데이터가 유실되는 경우가 발생한다.

어떤 프로그램은 쓰기에 있어서 좀 더 강력한 보장을 필요로 한다. 예를 들어 DBMS의 복원 모듈은 때때로 강제적으로 즉시 디스크에 기록할 수 있는 기능이 필요하다.

이러한 류의 응용 프로그램을 지원하기 위해서 대부분의 파일 시스템들은 추가적인 제어 API들을 제공한다. UNIX에서 응용 프로그램에게 제공되는 인터페이스는 fsync(int fd)다. 프로세스가 특정 파일 디스크립터에 대해서 fsync()를 호출하면 파일 시스템은 지정된 파일의 모든 **더티**(**dirty** 즉, 갱신된) 데이터를 디스크로 강제로 내려보낸다. 모든 쓰기들이 처리되면 fsync() 루틴은 리턴한다.

fsync()를 사용하는 방법에 관한 간단한 예제를 살펴보자. 코드에서 foo라는 파일을 열어서 데이터를 하나 쓴다. 그리고 나서 fsync()를 호출하여 해당 블럭을

즉시 디스크에 강제적으로 기록한다. `fsync()`가 리턴하면 응용 프로그램은 데이터가 영속성을 갖게 되었다는 것을 보장받기 때문에, 안전하게 다음으로 진행할 수 있다 (`fsync()`가 제대로 구현이 되어 있어야 한다).

```
1 int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
2 assert(fd > -1);
3 int rc = write(fd, buffer, size);
4 assert(rc == size);
5 rc = fsync(fd);
6 assert(rc == 0);
```

물론, 이러한 코드가 예상하는 모든 것을 완벽히 보장해 주지는 못한다. 어떤 경우 파일 foo가 존재하는 디렉터리도 `fsync()` 해주어야 한다. 디렉터리를 함께 `fsync()` 함으로써, 파일 자체와 이 파일이 존재하는 디렉터리 모두 안전하게 디스크에 저장하는 것이 보장된다. 파일이 새로이 생성된 경우, 디렉터리를 반드시 `fsync()` 해주어야 한다. 매우 중요한 사항임에도 불구하고 이런 세세한 사항들은 자주 간과되고 있으며 많은 응용 프로그램 수준의 버그를 만들어낸다 [Pil+13].

42.7 파일 이름 변경

때로는 파일의 이름을 변경하는 것이 매우 유용하다. 명령 행에서 mv 명령으로 파일명을 변경할 수 있다. `foo`를 `bar`라는 새로운 이름으로 바꾸는 명령어는 다음과 같다.

```
prompt> mv foo bar
```

`strace`를 사용하면 `mv`가 `rename(char *old, char *new)`이라는 두 개의 인자를 갖는 시스템 콜을 호출하는 것을 볼 수 있다. 각 인자는 원래의 파일 이름(`old`)과 새로운 이름(`new`)을 나타낸다.

`rename()`은 한 가지 흥미로운 특성을 보장한다. 이 명령어는 (대체적으로) 시스템 크래시에 대해 **원자적**으로 구현되었다. 이름 변경 중 시스템 크래시가 발생했을 때, 파일 이름은 원래의 이름이나 새로운 이름, 둘 중의 하나를 갖게 되며 그 사이의 중간 상태는 발생하지 않는다. 파일의 상태를 원자적으로 갱신해야 하는 응용 프로그램에 있어서 `rename()`은 매우 중요하다.

좀 더 구체적으로 살펴보자. 파일 편집기(예, emacs)로 파일 중간에 한 줄을 삽입한다고 해 보자. 파일의 이름은 `foo.txt`이다. 새로운 파일이 원래의 내용과 추가된 줄을 모두 포함한다는 것이 보장되도록 갱신하는 방법은 다음과 같을 것이다(간단하기 위해서 에러 검사는 무시한다).

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC);
write(fd, buffer, size); // 파일의 새로운 버전을 쓰기
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

이 예제에서 편집기가 하는 일은 간단하다. 새로운 버전의 파일을 임의 이름(`foot.txt.tmp`)으로 쓰고 `fsync()`로 디스크에 기록한다. 그 후에 새로운 파일의 메타데이터와 내용이 디스크에 기록되었다는 것을 확인하면, 임시 파일 이름을 원래

파일 이름으로 변경한다. 이 마지막 단계에서 이전 버전의 파일을 삭제하고 동시에 새 파일로 교체하는 작업이 원자적으로 이루어진다.

42.8 파일 정보 추출

파일 시스템은 각 파일에 대한 정보를 보관한다. 파일에 대한 정보를 **메타데이터(metadata)**라고 부른다. 어떤 파일의 메타데이터를 보려면 `stat()`이나 `fstat()` 시스템 콜을 사용한다. 이 호출들은 파일에 대한 경로명(또는 파일 디스크립터)을 입력으로 받는다. `stat`의 구조는 다음과 같다.

```
struct stat {
    dev_t st_dev;          /* ID of device containing file */
    ino_t st_ino;          /* inode number */
    mode_t st_mode;        /* protection */
    nlink_t st_nlink;      /* number of hard links */
    uid_t st_uid;          /* user ID of owner */
    gid_t st_gid;          /* group ID of owner */
    dev_t st_rdev;         /* device ID (if special file) */
    off_t st_size;         /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t st_blocks;    /* number of blocks allocated */
    time_t st_atime;       /* time of last access */
    time_t st_mtime;       /* time of last modification */
    time_t st_ctime;       /* time of last status change */
};
```

각 파일에 관한 많은 정보가 있다는 것을 알 수 있다. 파일의 크기(바이트 단위), 저수준 이름(아이노드 번호), 소유권, 파일이 접근되고 변경된 시간, 그 외에도 많은 정보가 있다. 정보를 확인하기 위해서 `stat`를 사용한다.

```
prompt> echo hello > file
prompt> stat file
   File: `file'
   Size: 6 Blocks: 8 IO Block: 4096 regular file
 Device: 811h/2065d Inode: 67158084 Links: 1
 Access: (0640/-rw-r-----) Uid: (30686/ remzi) Gid: (30686/ remzi)
 Access: 2011-05-03 15:50:20.157594748 -0500
 Modify: 2011-05-03 15:50:20.157594748 -0500
 Change: 2011-05-03 15:50:20.157594748 -0500
```

일반적으로 파일 시스템은 **아이노드**[1]에 이 정보를 보관한다. 파일 시스템 구현 부분에서 아이노드에 대해서 더 자세히 배우게 될 것이다. 여기서는 앞의 내용과 같은 정보를 저장하는 디스크 자료구조가 아이노드라고 이해하고 넘어가자.

42.9 파일 삭제

현재까지 파일의 생성과 접근에 대해 학습하였다. 그러면 파일은 어떻게 삭제할까? UNIX를 사용해 보았다면 안다고 생각할지도 모른다. `rm`이라는 프로그램을 실행하기만 하면 된다. 그러나 `rm`은 어떤 시스템 콜을 사용하여 파일을 지울까?

1) 어떤 파일 시스템들은 이 자료 구조의 이름을 dnode라 칭한다. 기본 개념은 유사하다.

우리의 오랜 친구인 `strace`를 사용하여 알아보자. 여기서 성가신 이번에는 "foo" 파일을 삭제해 보자.

```
prompt> strace rm foo
...
unlink("foo")  =  0
...
```

트레이스에서 관련 없는 내용들은 제거하니, 명칭으로는 무엇을 하는지 알 수 없는 `unlink()`라는 시스템 콜만 남았다. 보는 바와 같이 `unlink()`는 지워야 하는 파일 이름을 인자로 받은 후에 성공하면 0을 리턴한다. 하지만 왜 시스템 콜의 이름이 "unlink(연결을 끊다)"일까? "remove(제거)" 또는 "delete(삭제)"라고 했으면 안 되었을까? 답을 이해하기 위해서는 파일뿐만 아니라 디렉터리에 대해서도 이해해야 한다.

42.10 디렉터리 생성

디렉터리 관련 시스템 콜들은 디렉터리를 생성하고, 읽고, 삭제한다. 단, 디렉터리에는 절대로 직접 쓸 수 없다. 디렉터리는 파일 시스템의 메타데이터로 분류되며, 항상 간접적으로만 변경된다. 예를 들면 파일이나 디렉터리 또는 다른 종류의 객체들을 생성함으로써 디렉터리를 변경할 수 있다. 파일 시스템은 이런 식으로 디렉터리의 내용이 항상 예상과 일치하도록 보장한다.

디렉터리 생성을 위한 시스템 콜로 `mkdir()`이 있다. 같은 이름의 `mkdir` 프로그램을 사용하여 `mkdir` 프로그램이 실행될 때 무슨 일이 벌어지는지 살펴보자. `foo`라는 디렉터리를 생성해 보자.

```
prompt> strace mkdir foo
...
mkdir("foo", 0777) = 0
...
prompt>
```

처음 디렉터리가 생성되면 빈 상태이지만, 사실은 아주 기본적인 내용이 들어 있기는 하다. 빈 디렉터리에는 실제로 두 개의 항목이 존재한다. 하나의 항목은 디렉터리 자신을 나타내기 위한 것이고, 다른 항목은 자신의 부모 디렉터리를 가리키기 위한 것이다. 전자는 "."(dot) 디렉터리라고 하며 후자는 ".."(dot-dot)이라고 한다. 이를 확인하기 위해서는 `ls` 명령에 `-a` 플래그를 전달하면 된다.

```
prompt> ls -a
./ ../
prompt> ls -al
total 8
drwxr-x---  2 remzi remzi    6 Apr 30 16:17 ./
drwxr-x--- 26 remzi remzi 4096 Apr 30 16:17 ../
```

팁 : 강력한 명령어들은 주의하자

rm 프로그램은 강력한 명령의 아주 좋은 예로서 너무 큰 힘이 때로는 오히려 더 안 좋을 수도 있다는 것을 보여 준다. 예를 들어 여러 파일들을 한 번에 지운다고 해 보자. 그 경우 다음과 같이 쓰면 된다.

```
prompt> rm *
```

*는 현재 디렉터리 내의 모든 파일들을 가리킨다. 경우에 따라서는 디렉터리들도 같이 지우고 싶을 때도 있으며 모든 내용을 다 지우고 싶을 때도 있다. 그런 경우에 rm이 재귀적으로 하위 디렉터리를 타고 들어가서 그 안의 모든 내용을 다 지우도록 할 수도 있다.

```
prompt> rm -rf *
```

이와 같이 적은 수의 문자로 이루어진 명령어를 실수로 파일 시스템의 루트 디렉터리에서 실행하면 문제가 될 수 있다. 그러면 모든 디렉터리와 파일들을 다 지운다. 이런!

그러므로 강력한 명령들은 양날의 검이란 것을 기억하자. 적은 수의 키 입력만으로 많은 일을 할 수 있는 힘을 갖고 있지만 그 명령들은 빠르고 간단하게 엄청난 피해를 입힐 수도 있다.

42.11 디렉터리 읽기

이제 디렉터리를 생성했으니 디렉터리를 읽어 보자. 사실 ls 프로그램이 하는 일이 바로 그 일이다. ls와 유사한 도구를 직접 만들어 어떻게 동작하는지 알아보자.

디렉터리의 open은 파일을 open하는 것과는 다른 새로운 시스템 콜을 사용한다. 아래 예제 프로그램은 디렉터리의 내용을 출력한다. 이 프로그램은 opendir(), readdir(), 및 closedir()를 사용한다. 인터페이스가 무척 단순하다. 간단한 반복문을 사용하여 디렉터리 항목을 하나씩 읽은 후에 디렉터리의 각 파일의 이름과 아이노드 번호를 출력한다.

```
1  int main(int argc, char *argv[]) {
2      DIR *dp = opendir(".");
3      assert(dp != NULL);
4      struct dirent *d;
5      while ((d = readdir(dp)) != NULL) {
6          printf("%d %s\n", (int) d->d_ino, d->d_name);
7      }
8      closedir(dp);
9      return 0;
10 }
```

아래에 보이는 선언은 `struct dirent` 자료 구조 형태의 각 디렉터리 항목에 저장된 정보를 보여 준다.

```
1 struct dirent {
2     char d_name[256]; /* filename */
3     ino_t d_ino;      /* inode number */
4     off_t d_off;      /* offset to the next dirent */
5     unsigned short d_reclen;  /* length of this record */
6     unsigned char d_type;     /* type of file */
7 };
```

디렉터리에는 많은 정보가 있지 않기 때문에(단순하게 이름과 아이노드 번호를 매핑하는 것 이외에 몇 가지만 제공함) 프로그램은 각 파일에 `stat()`을 호출하여 파일 크기와 같은 구체적인 정보를 얻는다. `ls`가 `-l` 플래그를 전달받았을 때, 추가 정보를 얻기 위해 `stat()`을 호출한다. `strace`를 사용하여 `ls`에 `-l` 플래그를 적용했을 때와 하지 않았을 때를 직접 비교해 보라.

42.12 디렉터리 삭제하기

마지막으로 `rmdir()` 시스템 콜을 사용하여 디렉터리를 삭제할 수 있다(`rmdir`이라는 같은 이름의 프로그램이 이 시스템 콜을 사용한다). 파일 삭제와 다른 점은 디렉터리 삭제는 하나의 명령으로 아주 많은 양의 데이터를 지울 수 있기 때문에 좀 더 위험하다는 것이다. 때문에 `rmdir()`은 디렉터리를 지우기 전에 디렉터리가 비어 있어야 한다는 조건이 붙는다(즉, "."와 ".."외에는 없어야 한다). 비어있지 않은 디렉터리에 대해서 `rmdir()`을 호출하면 실패한다.

42.13 하드 링크

파일 삭제 시 왜 `unlink()`를 사용하는지를 이해하기 위해서 이제 파일 시스템 트리에 항목을 추가하는 새로운 시스템 콜 `link()`를 알아보자. `link()` 시스템 콜은 두 개의 인자를 받는데, 하나는 원래의 경로명이고, 다른 하나는 새로운 경로명이다. 원래 파일 이름에 새로운 이름을 "link(연결)"하면 동일한 파일을 접근할 수 있는 새로운 방법을 만들게 된다. 명령행 프로그램 `ln`이 그 일을 하며 아래에 예제를 보인다.

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

"hello"라는 단어가 저장된 파일을 생성하고 이름을 `file`[2]이라고 지었다. `ln` 프로그램을 사용하여 이 파일의 하드 링크를 생성한다. 이후부터는 이 파일을 보려면 `file` 또는 `file2`를 열면 된다.

2) 이 책의 저자들이 얼마나 창의적인지 보라. 우리는 "Cat"이라고 부르던 고양이도 길렀었는데(진실임) 지금은 죽었다. 대신 "Haamy"라는 햄스터를 기르고 있다.

`link`는 새로이 링크하려는 이름 항목을 디렉터리에 생성하고, 원래 파일과 같은 아이노드 번호를 가리키도록 한다. 파일은 복사되지 않는다. 대신 같은 파일을 가리키는 두 개의 이름(`file`과 `file2`)이 생성된다. 각 파일의 아이노드 번호를 출력하여 직접 확인해 보자.

```
prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
```

`ls`에 `-i` 플래그를 사용하면 각 파일의 아이노드 번호를(파일 이름과 함께) 출력한다. 이로써 link가 정확히 무엇을 하였는지 알 수 있다. 동일한 아이노드 번호(이 예제에서는 67158084)에 대한 새로운 링크를 생성했다.

`unlink()`가 왜 `unlink()`가 되었는지 이제 이해되기 시작했을 것이다. 파일을 생성할 때 사실은 *두 가지* 작업을 하게 된다. 하나는 파일 관련 거의 모든 정보를 관리하는 자료 구조(아이노드)를 만드는 것이다. 파일 크기와 디스크 블럭의 위치 등이 포함된다. 두 번째는 해당 파일에 사람이 읽을 수 있는 이름을 *연결*하고 그 연결 정보를 디렉터리에 생성하는 것이다.

파일 시스템에 파일의 하드 링크를 생성한 후에는 원래의 파일 이름(`file`)과 새로 생성된 파일 이름(`file2`) 간에는 차이가 없다. 사실 그 두 개의 파일 이름은 아이노드 번호 67158084에서 찾을 수 있는 파일의 메타데이터에 대한 연결일 뿐이다.

파일 삭제 시 `unlink()`를 호출한다. 위의 예제에서 파일 이름 `file`을 제거한다고 하더라도 여전히 해당 파일을 어려움 없이 접근할 수 있다.

```
prompt> rm file
removed `file'
prompt> cat file2
hello
```

파일을 `unlink`하면 아이노드 번호의 **참조 횟수**(**reference count**)를 검사한다. 이 참조 횟수(때로는 **연결 횟수**(**link count**)라고도 불린다)가 특정 아이노드에 대해 다른 이름이 몇 개나 연결되어 있는지 관리한다. `unlink()`가 호출되면 이름과 해당 아이노드 번호 간의 "연결"을 끊고 참조 횟수를 하나 줄인다. 참조 횟수가 0에 도달하면 파일 시스템은 비로소 아이노드와 관련된 데이터 블럭을 해제하여 파일을 진정으로 "삭제"한다.

파일의 참조 횟수는 `stat()`을 사용하여 확인할 수 있다. 파일에 대한 하드 링크를 생성과 삭제할 때 참조 횟수가 얼마인지 확인해 보자. 이 예제에서는 같은 파일에 대해 세 개의 연결을 만들고 나서 지워보도록 하겠다. 연결 횟수를 주의해서 보자.

```
prompt> echo hello > file
prompt> stat file
... Inode: 67158084          Links: 1 ...

prompt> ln file file2
prompt> stat file
... Inode: 67158084          Links: 2 ...

prompt> stat file2
... Inode: 67158084          Links: 2 ...
```

```
prompt> ln file2 file3
prompt> stat file
... Inode: 67158084          Links: 3 ...

prompt> rm file
prompt> stat file2
... Inode: 67158084          Links: 2 ...

prompt> rm file2
prompt> stat file3
... Inode: 67158084          Links: 1 ...

prompt> rm file3
```

42.14 심볼릭 링크

또 다른 아주 유용한 종류의 링크가 있는데, **심볼릭 링크**(**symbolic link**)라 하고 때로는 **소프트 링크**(**soft link**)라고 부른다. 하드 링크는 제한이 많은 편이다. 디렉터리에 대해서는 하드 링크를 만들 수 없으며(디렉터리 트리에 순환 구조를 만들까 우려하여), 다른 디스크 파티션에 있는 파일에 대해서도 하드 링크를 걸 수 없다(아이노드 번호는 하나의 파일 시스템 내에서만 유일하다)는 등의 제한이 있다. 그렇게 하여 새로운 종류의 심볼릭 링크가 만들어지게 되었다.

심볼릭 링크를 만들기 위해서 동일한 `ln` 프로그램을 사용할 수 있다. 대신 `-s` 플래그를 전달해야 한다. 예를 살펴보자.

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
```

보는 바와 같이 소프트 링크를 만드는 방법은 비슷해 보인다. 원래의 파일은 기존의 이름인 `file` 뿐만 아니라 이제 `file2`라는 심볼릭 링크 이름으로도 접근할 수 있다.

표면적으로는 하드 링크와 유사하지만 소프트 링크는 사실 매우 다르다. 첫 번째 차이는 심볼릭 링크는 다른 형식의 독립된 파일이라는 것이다. 우리는 파일과 디렉터리를 다루었다. 심볼릭 링크는 파일 시스템에 존재하는 세 번째 종류의 유형이다. 심볼릭 링크에 대해 `stat()`을 해 보면 이를 알 수 있다.

```
prompt> stat file
... regular file ...

prompt> stat file2
... symbolic link ...
```

`ls`를 실행시켜도 그 사실을 알 수 있다. `ls`의 긴 형식의 출력의 첫 글자를 자세히 살펴보면 가장 왼쪽 열의 글자가 일반 파일에 대해서는 `-`로 표기되고 디렉터리는 `d`로 그리고 소프트 링크에 대해서는 `l`로 표시되고 있는 것을 알 수 있다. 또한, 심볼릭 링크의 크기(이 경우에는 4 바이트)를 확인할 수 있을 뿐만 아니라 연결의 대상도 보여주고 있다(`file`이라는 이름을 갖는 파일).

```
prompt> ls -al
```

```
drwxr-x---  2 remzi remzi   29 May  3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May  3 15:14 ../
-rw-r-----  1 remzi remzi    6 May  3 19:10 file
lrwxrwxrwx  1 remzi remzi    4 May  3 19:10 file2 -> file
```

`file2`의 크기가 4 바이트인 이유는 심볼릭 링크는 연결하는 파일의 경로명을 저장하기 때문이다. `file`이라는 파일에 연결을 했기 때문에 연결된 파일인 `file2`의 크기는 작다(4 바이트). 좀 더 긴 경로명에 연결한다면 심볼릭 링크 파일은 좀 더 커지게 될 것이다.

```
prompt> echo hello > alongerfilename
prompt> ln -s alongerfilename file3
prompt> ls -al alongerfilename file3
-rw-r----- 1 remzi remzi  6 May  3 19:17 alongerfilename
lrwxrwxrwx 1 remzi remzi 15 May  3 19:17 file3 -> alongerfilename
```

마지막으로 심볼릭 링크가 만들어진 방식 때문에 **dangling reference**라는 문제가 발생할 수도 있다.

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

예제에서 보는 바와 같이 하드 링크와 매우 다르게 원래의 파일인 `file`을 삭제하면 심볼릭 링크가 가리키는 실제 파일은 더 이상 존재하지 않게 된다.

42.15 파일 시스템 생성과 마운트

파일과 디렉터리 그리고 몇 가지 종류의 링크를 다루는 기본 인터페이스를 살펴보았다. 아직 논의해야 하는 한 가지가 남아 있다. 다수의 파일 시스템들이 존재할 때 이들을 묶어서 어떻게 하나의 큰 디렉터리 트리를 구성할까? 유닉스(혹은 리눅스)를 써본 사람은 알겠지만 여러 개의 파일 시스템 파티션들이 모여서 하나의 큰 디렉터리를 구성한다. 각각의 파일 시스템을 생성하고, 이들을 "마운트"함으로써 단일 디렉터리 트리를 구성한다.

대부분의 파일 시스템에서 파일 시스템을 생성하는 `mkfs`("make fs"라고 발음함)라는 도구를 제공한다. 개념은 다음과 같다. 장치명(디스크 파티션, 예, `/dev/sda1`)과 파일 시스템 타입(예, EXT3)을 전달하면, 이 프로그램은 해당 파티션에 전달된 파일 시스템 (예, EXT3) 형식으로 구성된 빈 파일 시스템을 생성한다. 생성된 파일 시스템은 자체적인 디렉터리 구조로 구성되어 있다. 비어있다. 즉, 루트 디렉터리만 존재할 것이다. 빈 파일 시스템을 생성하는 구체적인 과정은 다음 장에서 다루게 된다. 조금만 기다리기 바란다. 그리고 `mkfs`가 말하기를, 파일 시스템이 있을지어다!

새로이 생성된 파일 시스템은 현재 디스크에 존재한다. 새로이 생성된 파일 시스템을 루트 디렉터리에서 시작하는 기존의 디렉터리 구성을 통해 접근할 수 있도록 해주어야 한다. 매우 중요한 작업이다. 이 작업을 마운트라 한다. 그 작업은 `mount` 프로그램을

사용한다(내부적으로 `mount()` 시스템 콜을 사용하여 실제 작업을 처리한다). 하는 일은 간단하다. 기존의 디렉터리 중 하나를 **마운트 지점**(**mount point**)으로 지정한다. 그리고 나서 마운트 지점에 생성된 파일 시스템을 "붙여 넣는다".

예를 들겠다. 파티션 `/dev/sda1`에 EXT3 파일 시스템이 존재한다고 가정하자. 아직 마운트되지 않았다. 이 파일 시스템은 그 자체적인 디렉터리 구성을 갖고 있다. 루트 디렉터리에서 시작하고 그 아래에 `a`와 `b`라는 두 개의 하위 디렉터리가 있다. 그리고 각 디렉터리에는 `foo`라는 파일이 하나 들어 있다. 이 파일 시스템을 `/home/users`라는 위치에 마운트한다고 해 보자. 다음과 같이 작성하면 된다.

```
prompt> mount -t ext3 /dev/sda1 /home/users
```

마운트 작업이 성공하면, 새로운 파일 시스템을 기존의 디렉터리 경로를 통해 접근할 수 있게된다. 이때 어떻게 새로운 파일 시스템에 접근할 수 있는지 주의해서 보자. 루트 디렉터리의 내용을 살펴보기 위해서 다음과 같이 `ls`를 실행해 보자.

```
prompt> ls /home/users/
a b
```

보는 바와 같이 경로명 `/home/users/`는 이제 새롭게 마운트된 디렉터리의 루트를 가리킨다. 마찬가지로 `a`와 `b`를 경로명 `/home/users/a`와 `/home/users/b`로 접근할 수 있다. 마지막으로 `foo`라는 이름의 파일은 `/home/users/a/foo`와 `/home/users/b/foo`로 접근할 수 있다. 이것이 마운트의 묘미이다. 여러 개의 개별적인 파일 시스템을 갖는 대신에 마운트는 모든 파일 시스템들을 하나의 트리 아래에 통합시킨다. 이로써 객체의 호칭을 한결같고 편리하게 하게 된다[3].

시스템에 어떤 파티션들이 어디에 마운트되었는지 알려면 `mount` 프로그램을 실행한다. 다음과 같은 결과가 출력된다.

```
/dev/sda1 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
/dev/sda5 on /tmp type ext3 (rw)
/dev/sda7 on /var/vice/cache type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
AFS on /afs type afs (rw)
```

내용을 살펴보자. 총 7개의 파티션이 있다. 이들 각각의 파티션은 ext3 파일 시스템, proc 파일 시스템(현재 프로세스에 대한 정보를 접근하기 위한 파일 시스템), tmpfs(임시 파일들만을 위한 파일 시스템), 그리고 AFS(분산 파일 시스템)을 사용하고 있다. 7개의 파일 시스템 파티션들이 단일 파일 시스템 트리로 결합되어 있다. `/dev/sda1` 파티션은 / 디렉터리에 연결되어 있다. `/dev/sda7` 파티션은 EXT3 파일 시스템으로 구성되어 있으며 `/var/vice/cache`에 마운트되어 있다.

3) 역자 주: 유닉스와 윈도우 운영체제의 가장 큰 차이점중의 하나가 바로 마운트이다. 유닉스에서 파일 시스템 파티션은 윈도우에서의 드라이브와 같다. 유닉스는 부팅시 마운트 작업을 통해 여러 개의 파일 시스템 파티션을 단일 디렉터리 구조서 묶는다. 윈도우는 묶지 않는다. 모든 파티션이 따로 존재한다. C:, D: 드라이브라는 이름으로 존재한다.

42.16 요약

운영체제를 공부하는 데 있어서, UNIX 파일 시스템 인터페이스에 대한 지식은 기본이다. 인터페이스를 완전히 익히기 위해서는 상당히 많은 내용에 대한 이해가 필요하다. 이들을 이해하는 가장 좋은 방법은 무조건 많이 사용해보는 것이다. 열심히 하자. 그리고, 당연히, 많이 읽어야 한다. Stevens [SR05]가 좋은 시작점이다.

유닉스 파일 시스템의 기본 인터페이스들을 살펴보았다. 조금 이해가 되었기 바란다. 진짜 중요한 것은 API를 어떻게 구현할 것인가이다. 다음 장부터 자세히 다룰 것이다.

참고 문헌

[Kil84] **"Processes as Files"**
Tom J. Killian
USENIX, June 1984
Pseudo(의사) 파일 시스템 내에서 파일 다루듯이 각 프로세스를 다룰 수 있는 /proc 파일 시스템을 소개하는 논문이다. 현대의 Unix 시스템에서 지금도 사용되는 영리한 개념이다.

[Lev84] **"Capability-Based Computer Systems"**
Henry M. Levy
URL: `http://homes.cs.washington.edu/~levy/capabook`
초창기 지역-기반 시스템에 대한 훌륭한 개론이다.

[Pil+13] **"Towards Efficient, Portable Application-Level Consistency"**
Thanumalayan S. Pillai, Vijay Chidambaram, Joo-Young Hwang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
HotDep '13, November 2013
데이터를 디스크에 기록하는 데 응용 프로그램이 얼마나 쉽게 실수할 수 있는지 보인 우리의 연구이다. 구체적으로 말하자면, 응용 프로그램 안에 파일 시스템에 대한 가정이 스며들어서 특정 파일 시스템에서 실행될 때만 응용 프로그램이 정상적으로 동작하도록 만든다는 것을 보였다.

[SK09] **"Principles of Computer System Design"**
Jerome H. Saltzer and M. Frans Kaashoek
Morgan-Kaufmann, 2009
이 시스템 분야의 역작은 이 분야에 관심이 있는 사람이라면 꼭 읽어야 한다. MIT에서 시스템을 가르칠 때 사용하는 것이다. 한 번 읽고 나서 여러 번 더 읽어서 완전히 흡수하라.

[SR05] **"Advanced Programming in the Unix Environment"**
W. Richard Stevens and Stephen A. Rago
Addison-Wesley, 2005
아마도 이 책을 수십만 번은 참고하였을 것이다. 훌륭한 시스템 개발자가 되기 원하는 이들에게 그만큼 유용한 책이다.

숙제

이 숙제에서 이 장에서 설명한 API들의 동작 방법에 대해서 익숙해져 보자. 이를 위해 여러 Unix 명령들을 기반으로 하는 몇 개의 다른 프로그램들을 작성하게 될 것이다.

문제

1. `stat`: `stat` 명령에 해당하는 프로그램을 작성하라. 주어진 파일이나 디렉터리에 대해 `stat()` 시스템 콜을 호출한다. 파일 크기, 할당된 블럭의 수, 참조(연결) 횟수 등을 출력해 보자. 디렉터리 내의 항목의 수가 변함에 따라 디렉터리의 연결 횟수는 어떻게 바뀌는가? 유용한 인터페이스는 `stat()`이다.
2. 파일들의 목록: 어떤 디렉터리의 파일들의 목록을 출력하는 프로그램을 작성하라. 인자가 없이 호출이 되면 파일 이름만을 출력한다. `-l` 플래그를 사용하여 호출되면 각 파일에 대한 정보를 출력해야 한다. 그 정보에는 소유권, 그룹, 권한과 더불어서 `stat` 시스템 콜에 포함되어 있는 정보가 담겨야 한다. 프로그램은 어떤 디렉터리를 읽어야 하는지를 명시할 수 있도록 하나의 인자를 더 받아야 한다. 예, `myls -l directory`. 만약 디렉터리가 주어지지 않으면 현재의 디렉터리를 사용해야 한다. 유용한 인터페이스는 `stat()`, `opendir()`, `readdir()`, `getcwd()`이다.
3. Tail: 파일의 마지막 몇 줄을 출력하는 프로그램을 작성하라. 프로그램은 파일의 거의 끝을 탐색한 후에 데이터의 블럭을 읽고, 명시한 줄 수만큼 거꾸로 올라가도록 하는 식으로 효율적이어야 한다. 원하는 줄을 찾으면 명시한 줄 수에 해당하는 라인들의 시작부터 파일의 끝까지 출력해야 한다. 프로그램을 실행하려면 `mytail -n file` 이라고 적어야 하며, 이때 `n`은 파일의 끝에서부터 출력해야 할 줄 수를 나타낸다. 유용한 인터페이스는 `stat()`, `lseek()`, `open()`, `read()`, `close()`이다.
4. 재귀적 검색: 트리 내에 주어진 어떤 지점에서 시작하여 파일 시스템 내의 디렉터리와 그 안의 각 파일들의 이름을 출력하는 프로그램을 작성하라. 인자 없이 실행시키면 프로그램은 현재 작업 디렉터리에서 시작한다. 현재 디렉터리 내의 모든 내용을 출력하는 것은 물론이고 하위 디렉터리의 내용과 그 이하의 내용들도 모두 출력하도록 한다. 현재 작업 디렉터리(CWD)를 루트로 하여 전체 트리를 다 출력할 때까지 반복한다. 어떤 디렉터리 명을 하나의 인자로 받는다면 그 위치를 루트로 대신 사용한다. 강력한 `find` 명령행 프로그램이 사용하는 옵션들처럼 재미있는 옵션을 추가하여 재귀적 탐색을 개선시켜 보자. 유용한 인터페이스는 직접 찾아보라.