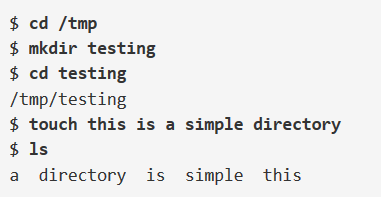
**[Directories](https://digital-forensics.sans.org/blog/2017/06/07/understanding-ext4-part-6-directories" \o "Understanding EXT4 (Part 6): Directories)**

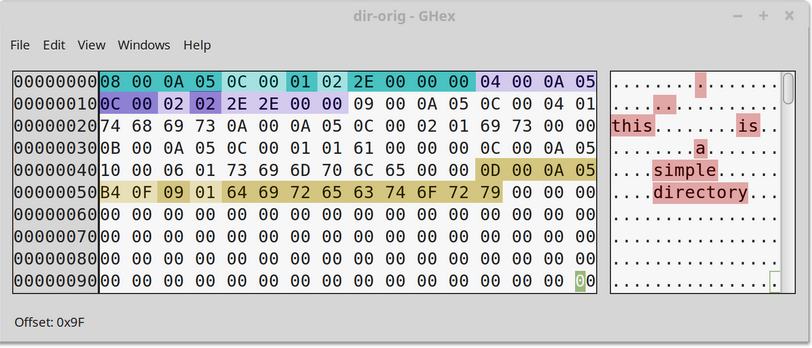
Ext inode 는 파일이름을 저장하지 않은다. 디렉토리는 파일 이름정보가 유지되는 기존 유닉스 파일 시스템의 유일한 위치이다. 즉, 디렉터리는 파일이름을 inode 번호와 연결하는 특수한 파일이다.

또 가장 간단한 경우 ext 디렉터로니는 파일 항목의 순차목록이다. 대부분의 디렉토리 항목은 파일이 디렉토리에 작성되는 순서대로 디렉토리 파일에 추가된다.

작은 디렉토리로 예를 들어보면



Ls 로 디렉토리를 나열 하면 알파벳 순서대로 나열되지만 16진수 편집기에서 보면 실제 디렉토리 순서를 알수 있다.

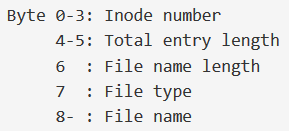


오른쪽에는 ASCII 파일 이름을 하이라이트 표시했다.

보면 touch 명령으로 생성한 파일 순서대로 나타나는 것을 알수 있음.

모든 디렉토리는 “.” 나 “..” 으로 시작해야된다.

각 디렉토리의 앤트리에는 5개의 필드가 있다.

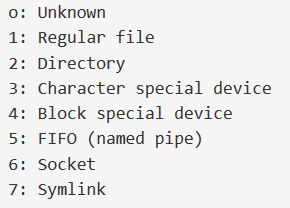


디렉토리 앤트리은 가변 길이이며 두번째 필드는 항목의 전체길이를 의미한다. 앤트리는 4바이트의 경계에 정렬되고 “.’ 같은 경우 앤트리가 9바이트에 맞더라도 총 앤트리 길이는 12바이트이다.

Unix 파일 시스템에서 파일 이름 길이 필드는 2바이트이지만 리눅스에서는 255자를 초과하는 파일 이름을 허용하지 않기 때문에 ext 개발자는 파일이름의 길이로 1바이트만 사용하고 파일 유형을 보유하기 위해 2번째 바이트를 사용하기로 했다.

파일 유형은 inode 에도 저장되지만 디렉토리 항목에 정보를 갖는 것이 이 파일 이름과 함께 파일 유형이 표시되는 ls-F 와 같은 조작에 더 효율적이다.

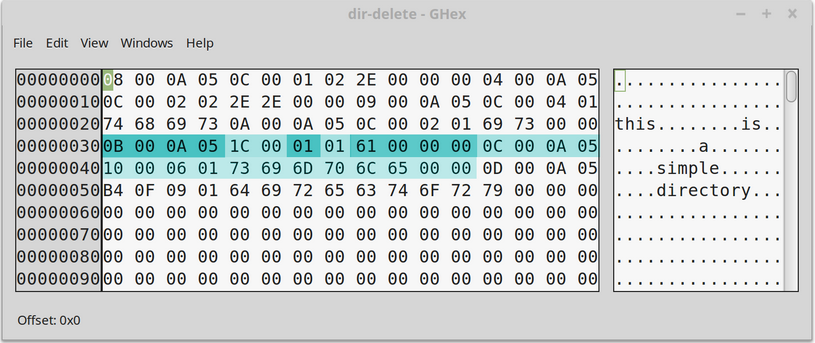
파일 유형은 다음과 같이 정의된 숫자 필드이다.



“.” 이거나 “..” 일경우 디렉토리(2) 이고 이것은 유닉스 파일시스템에서 유일하게 hard link 를 허용하는 경우이다.

디렉토리에서 entry size 는 0x0FB4, 이거나 4020 바이트로 이것은 블록의 끝까지 남아있는 모든 바이트를 소비한다. 마지막 디렉토리 entry는 항상 블록의 끝과 정렬된다. 디렉토리 엔트리는 블록의 경계를 넘을 수 없다.

예제 디렉토레에서 간단한 파일을 삭제하면 어떻게 되는지보자.

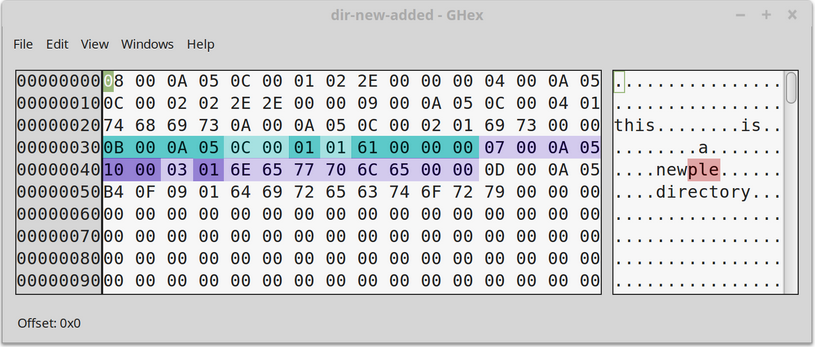


이전에 12바이트였던 파일 a 의 항목 길이는 이제 28 바이트(0x1C) 이며 삭제된 파일 simple 의 엔트리를 효과적으로 소비한다. “a” 파일 앤트리의 파일 이름 길이는 여전히 1바이트이지만 항목의 나머지 부분은 “slack” 공간으로 삭제된 파일의 이전 항목을 보유하고 있는것을 확인 할 수있다.

위의 동작은 Unix 파일 시스템에서 파일이 삭제될 때 표준동작으로 삭제된 파일 앞의 앤트리는 삭제된 앤트리를 사용하기 위해 커진다.

삭제된 디렉토리 앤트리에서 사용되지 않은 slack 공간은 새 파일이 디렉토리에 사용될 때 재사용될 수 있다.

예를들어 new 라는 파일을 추가해보면



“a” 파일의 앤트리 길이는 다시 12 바이트가 되고 16 바이트 길이의 new 파일에 대한 새 파일 앤트리가 생겼다.

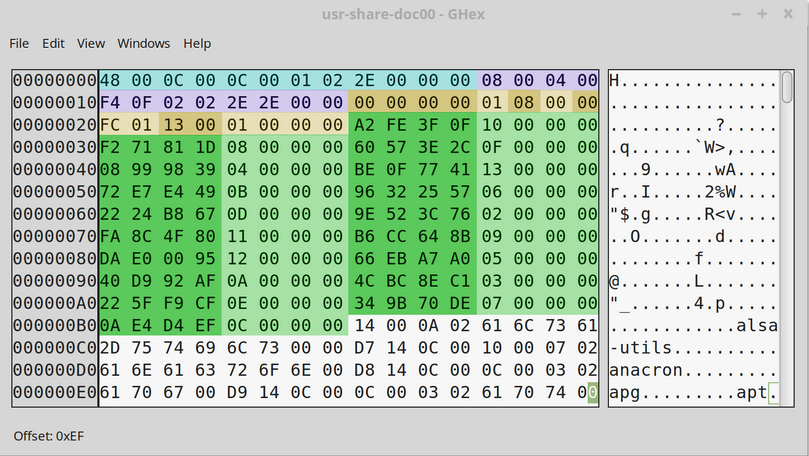
위를 보면 ext 는 분명히 새로운 디렉토리 앤트리에서 여분의 공간을 채우는 null 을 신경쓰지 않는 것을 알수 있다.

커다란 디렉토리

전통적인 유닉스 파일 시스템에 대한 일관적인 비판중 하나는 디렉토리가 커질 수록 성능이 저하된다는 점이 있다. 디렉토리가 정렬되지 않은 파일 목록에 지나지 않은 경우 많은 수의 디렉토리 항목을 순차적으로 검색해야되기 때문에 디렉토리가 커질 수록 평균 검색 시간이 선형적으로 증가한다.

그래서 현대적인 파일 시스템에서는 일종의 검색 가능한 데이터 구조로 디렉토리 항목을 구성하여 문제를 해결한다. Ext3 에서부터는 개발자는 디렉토리 항목 구성을 위해 htree 라는 해시 트리 시스템을 만들었고 ext4 에서는 이것은 표준으로 디렉토리가 단일 블록보다 커질때 볼 수 있다.

Here's the first block of my /usr/share/doc directory:

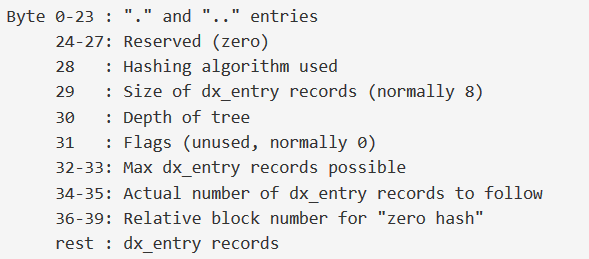


“.” “..” 엔트리는 여전히 블록의 시작부분에 나타나는데 이것은 이전버전과의 호환성을 위한것이다.

“..” 엔트리의 길이는 0x0FF4 또는 4080 바이트이며 나머지 블록을 소비한다.

이는 이전 버전과의 호환성 기능으로 후속 htree 데이터가 디렉토리를 간단한 파일 엔트리 시퀀스로 구문 분석하려는 모든 이전 코드에서 숨겨진다.

“.”나 “..”를 사용한뒤 나머지 블록은 htree의 루트를 정의하는 dx\_root 구조에서 사용됩니다. 기술적으로 ‘.’ ‘..’ 항목도 dx\_root 의 일부분으로 데이터 구조는 전체 첫번째 블록을 소비합니다.



오프셋 24-27에서 4개의 널바이트 이후에 단일 바이트는 htree에서 사용하는 해시 알고리즘을 지정한다. 바이트 코드는 여기에 설명되어있지만 0x01 은 MD4 기반 해싱 알고리즘인 표준으로 보인다.

그다음은 dx\_entry 레코드의 크기로 htree 에서 다양한 블록을 인덱싱하는데 사용된다. 이 레코드 길이는 항상 8 바이트이다.

바이트 32-33은 dx\_root 구조의 초기 필드 다음에 이 블록에 채워질 수 있는 최대 dx\_entry 레코드 수를 설명한다. Dx\_entry 레코드는 블록에서 40 바이트로 시작하므로 이 최대 값은 4096바이트의 블록 크기에서 dx\_root 필드의 40 바이트를 dx\_entry 레코드의 8 바이트 크기로 나눈 것으로 가정 할 수 있다.

바이트 36-39의 zero hash 항목이 추가 dx\_entry 레코드로 계산되므로 실제 값은 0x01FC 또는 508 이다.