FAT32: timestamps

-Como faltaba 1 bit más para los segundos (para cubrir de 0 a 59), se decidió siempre multiplicar por 2 (hacer un corrimiento a izquierda) los 5 bits, asi aparece un 6to bit en 0. Lo que provoca esto es que los segundos siempre serán pares (por ejemplo, 31s se representará como 30s creo, y bueno el 30 como 30). Solo estan los segundos exactos en la fecha de creación (y no en la de última modificación).

PREGUNTAR bien a que se refiere cuando comenta que se deben leer todos los directory entry de un directorio (en la diapositiva de 'Resumen').

MBR

PREGUNTAR qué significado tiene el área entre la MBR y el comienzo de la 1ra partición (ya que tiene como 1MB de 0x00) → **es** más bien por cuestiones "éticas", **para marcar bien la división entre la MBR y las particiones**, porque si de casualidad los sectores del disco en vez de ser de 512 bytes son de más (por ejemplo 4k), podríamos llegar a leer la MBR y parte de la primera partición, y es "preferible" que sean lecturas separadas (pero como poder se podría tener la 1ra partición pegadita a la MBR).

PREGUNTAR si la MBR es compatible con varios tipos de filesystem (por el código de partición: https://es.wikipedia.org/wiki/C%C3%B3digo_de_tipo_de_partici%C3%B3n). **Sí, muchos**.

Haruspex (*librería de Python que parsea filesystems y tablas de particiones*)

PREGUNTAR bien qué es la MBR y qué función cumple en haruspex cuando hace *'haruspex.mbr.Table()*' (**fijarse que al final de** http://www.tolaemon.com/docs/fat32.htm **habla sobre la MBR :D**). Y si es que maneja las particiones, cuando un disco tiene más de una partición, pero con un editor hexa abrimos una partición individual en vez de todo el disco, ¿también está la MBR al comienzo? ¿y también nos dice la información de la otra partición aunque no la tengamos abierta? **No está, solo vemos la MBR si abrimos la totalidad del dispositivo, si abrimos solo una partición, primero veremos la cabecera del filesystem que tenga**.

PREGUNTAR bien lo del parámetro 'base_address' al instanciar FAT32. (y en que se basan esas direcciones, qué son realmente, ¿a qué se refiere con OFFSET? -> **cuando abrimos un archivo es como que se estructura en un arreglo de bytes (realmente es como un stream creo), entonces al momento de leer o escribir datos, lo haremos con BYTES. Y lo mismo pasa si hacemos un 'seek',**

para posicionarse en un lugar del archivo se le pasa cierto BYTE, nos posicionaremos en el byte 'x' del archivo. Entonces el offset es eso, un determinado n° (ENTERO) de byte.)

EJ:

Representación hexadecimal de los bytes que tiene/compone_a un archivo:	0x**AB** 0x**17** 0x**FF**
OFFSET:	**0** **1** **2**

➔ Dentro del módulo **FAT32.py**

La clase **FileRecord** representa a un **Directory Entry**, a una entrada de 32 bytes de un directorio (los metadatos asociados a un archivo o directorio dentro del directorio actual).

O sea, a esta clase se la instancia con datos en crudo (los binarios: 32 bytes) y hace el parseo/traducción a lo que representan esos bytes (según la estructura de FAT32).

Dentro de la funcion '_parse', se hace uso de las 'property' para setear los valores de atributos. O sea, le pasa el valor al setter y luego en la definicion de la funcion setter se hacen verificaciones sobre ese valor recibido (como en el setter de 'cluster'), o directamente ahi se hace el parseo en sí (como en el setter de 'name', aunque recien en __str__ convierte los 'bytes' a string), o es un miti-miti como con los timestamps, que en una funcion 'global' se hace el parseo, retornando un objeto 'datetime' y luego en el setter se hacen unas verificaciones antes de asignarlo.

La clase **Directory** representa a los **directorios** de FAT32. Principalmente contiene una lista con los archivos y directorios que habría dentro de un directorio. En FAT32 cada directorio tiene asignado uno o más clusters del disco, donde se almacenan los metadatos de cada archivo que contiene. En esencia, se almacenan los Directory Entry, una entrada de 32 bytes por cada archivo. Entonces en este caso el parseo consiste en pedirle al filesystem el n° de cluster donde empieza el directorio (esto realmente sale del directory entry del directorio que contiene a este directorio), y a partir de ahi empezar a leer de a 32 bytes e ir creando instancias de FileRecord (los directory entry).

PREGUNTAR porque en el parser, al final usa un generator, o sea cual es la justificación en vez de por ejemplo poner una lista por comprensión.

La clase **FileHandle**... (pedirle a Bruno que me explique un poco cómo funca) (¿puede leer el contenido de un archivo? **SÍ** ¿nos devuelve el binario o el ascii? **BINARIO**)

La clase **FAT32** es la que representa al filesystem en sí y hará uso de las clases anteriores. Busca emular el comportamiento del sistema de gestión de archivos (modulo del sistema operativo) ante la presencia de un dispositivo de almacenamiento con un filesystem FAT32 (obviamente como esto que hacemos es a bajo nivel, podremos ver más cosas que la que nos muestra a nosotros el sistema operativo). (**PREGUNTAR** *BIEN ESTE CONCEPTO A BRUNO*).

Para inicializarla, hay que pasarle la ruta del dispositivo de almacenamiento que queramos abrir (para saber esto, hay que ver cuál es el "**PhysicalDrive**" del disco: https://stackoverflow.com/questions/6522644/how-to-open-disks-in-windows-and-read-data-at-low-level); y también hay que pasarle el offset donde comienza la partición que queremos 'explorar' del dispositivo (que tenga FAT32 como sistema de archivos). Recordemos que un dispositivo puede tener varias particiones con distintos filesystems. El que sabe en qué parte del disco comienzan esas particiones es el MBR (o GPT), que se ubica en los primeros 512 bytes del dispositivo (primer sector). Entonces se usa el módulo de MBR para saber el offset.

Y bueno internamente primero lee el encabezado de FAT32 (el primer sector, lo que sería el 'volume id' o 'boot area') para parsearlo y luego hace algunos cálculos de direcciones. Luego lee las 2 FATs y las guarda en listas de 4 bytes por campo/posición (**PREGUNTAR** *SI DE VERDAD CUANDO CONECTAMOS UN DISPOSITIVO CON FAT32, QUEDA CARGADO SIEMPRE EN MEMORIA LAS FATS* -> **en general sí, al menos una parte de ella**). Y por último lee el directorio raíz.

Nota: Recordemos que este módulo solo lee un filesystem FAT32, no es que construya uno, y no puede por ejemplo agregar archivos o directorios. No escribe. Y por supuesto el dispositivo ya tiene que estar formateado en FAT32.

Después también están definidas 2 funciones: _read_cluster (que justamente lee un cluster del filesystem y nos devuelve los datos en crudo (los bytes)) y open (que abre un archivo o directorio pasado como ruta absoluta, leyendo la cadena de clusters hasta encontrarlo: esto lo podemos ver en la clase Directory -> parse).

(**PREGUNTAR** *SI DE VERDAD CUANDO ABRIMOS UNA CARPETA O ARCHIVO, QUEDA CARGADO SIEMPRE EN MEMORIA LOS BINARIOS DE LA CADENA DE CLUSTERS HASTA QUE LO CERREMOS, o por lo menos los directory entry que contiene*)