Tipo de Datos de los Timestamps de ext2 en Memoria

Quiero saber qué tipo de variable se usa en memoria para almacenar timestamps en **ext2** (a_time, c_time, m_time ...). Si es signed o unsigned, porque de eso dependerá cuántos segundos podrá almacenar y cuál será la fecha máxima que podremos alcanzar sin que haya problemas. (básicamente si es unsigned podremos almacenar el doble, ya que los segundos son positivos y no usaríamos la parte negativa del signed [salvo para fechas muy antiguas]).

# signed	# unsigned
>>> (2**32)/2	>>> 2**32
2147483648	4294967296

En disco se graban 1s y 0s y listo, no importa el tipo de dato, eso es propio de la memoria no del disco... En el disco lo que importa es si se escriben los bytes en little endian (le) o big endian (be), pero tengo ganas de saber qué tipo de datos se usa en memoria para obtener el tiempo unix actual y grabar dicha variable en disco.

CONCLUSIÓN

Por todo lo que investigué, al parecer es signed (ya que no se ve que explícitamente diga 'unsigned' o 'u' en el tipo de dato, y por convención si no dice eso, es signed).
Antiguamente un entero con signo de 32 bits, y actualmente de 64 bits.

Pero no sé bien de qué depende eso, ¿de la versión del kernel linux? del encargado de hacer 'writes' en el filesystem (driver)? o de la arquitectura del procesador?

Repuesta corta: depende de la versión que tengamos de Linux, si tenemos la última (v5), estaremos tranquilos que usa tipos de 64 bits. La versión del driver del filesystem ext2 (*que es donde se manejan los timestamps, junto con todo lo referido al fs*), viene de la mano con la versión de Linux, así que estará actualizado (Linux es el todo). Y el procesador también influye, pero creo que más bien por el lado de rendimiento (eso lo explico al final de todo).

Respuesta larga…

El **driver del filesystem ext2** de Linux (https://github.com/torvalds/linux/tree/master/fs/ext2), es decir, el que tienen implementadas las funciones para manejar el filesystem (reads, writes, etc, etc, etc, etc), actualmente usa variables ENTERAS CON SIGNO de 64 BITS para manipular los timestamps, ya sea para leer del disco, o para obtener el tiempo unix actual y asignárselo a un timestamp (a_time, c_time, m_time,...), etc.

Nota: *puede ser que casos puntuales de computadoras que tengan instalada una versión de Linux antigua, o que su arquitectura sea de 32 bits, los **timestamps** se manejen con variables de **32 bits**, ya que **antiguamente** así se manejaban*.

Realmente todo este maneje se hace en un **VFS** (*virtual filesystem*), o sea se trabaja en **memoria** con el VFS y en algún momento se graban los datos en el verdadero FS que está en el disco (dispositivo de almacenamiento). **Lo referido al VFS *general*, lo podemos ver en**: https://github.com/torvalds/linux/blob/master/include/linux/fs.h

Nota: Digo '*general*', porque en por ejemplo ext2/ext2.h o ext4/ext4.h también hay estructuras que se usan en memoria, pero que son una especificación del VFS general: *tienen acceso a las estructuras generales de memoria y las completan en base a sus necesidades puntuales, o agregan nuevas* (más adelante hablo más sobre esto de las 'generalidades').

De todas formas, **en el disco se graban los timestamps en 32 bits en el formato *little endian*.**

```
297  /*
298   * Structure of an inode on the disk
299   */
300  struct ext2_inode {
301          __le16  i_mode;         /* File mode */
302          __le16  i_uid;          /* Low 16 bits of Owner Uid */
303          __le32  i_size;         /* Size in bytes */
304          __le32  i_atime;        /* Access time */
305          __le32  i_ctime;        /* Creation time */
306          __le32  i_mtime;        /* Modification time */
307          __le32  i_dtime;        /* Deletion Time */
308          __le16  i_gid;          /* Low 16 bits of Group Id */
309          __le16  i_links_count;  /* Links count */
310          __le32  i_blocks;       /* Blocks count */
311          __le32  i_flags;        /* File flags */
312          union {
313                  struct {
314                          __le32  l_i_reserved1;
315                  } linux1;
316                  struct {
317                          __le32  h_i_translator;
318                  } hurd1;
```

https://github.com/torvalds/linux/blob/master/fs/ext2/ext2.h#L297

(no se bien cómo se haría esto, grabar 64 bits en 32… Creo que no es que graben 64 bits [*ya que las estructuras del inodo y superbloque de ext2 en disco están definidas con 1 solo campo de 32 bits para cada timestamp, se graban 32 bits y listo*], sino que usando variables enteras signed de 64 bits, ahora podemos abarcar en su parte positiva todos los bits del antiguo entero signed de 32 bits, como si fuera un unsigned int. Entonces cuando leamos del disco esos 32 bits, podremos aprovecharlos completamente y llegar al valor +4.294.967.295, cuando antes solo podíamos llegar al +2.147.483.647. Obviamente ahora con un signed de 64 bits podríamos almacenar un valor mucho más grande, pero al momento de querer escribir en el disco, solo podremos escribir 32 bits, estamos limitados a eso ya que así está definido en ext2. En ext4 se amplía el valor de los timestamps porque se agrega un segundo campo en disco de 32 bits para cada uno.)

	Signo	**Rango de valores**	**Signo**	**Rango de valores**
Entero 32 bits (*int o long*)	signed	$-2^{31} \; a \; 2^{31}-1$	unsigned	$0 \; a \; 2^{32}-1$
Entero 64 bits (*long long*)	signed	$-2^{63} \; a \; 2^{63}-1$	unsigned	$0 \; a \; 2^{64}-1$

La implementación de **ext4**, actualmente también usa enteros con signo de 64 bits (**REVISAR PÁGINA 13** *) para los timestamps, ya que se actualizó para evitar el problema del año 2038*.
Se fue migrando todo a 64 bits. De todas formas, en el disco se sigue grabando de a 32 bits como en ext2*, pero la diferencia es que ahora en el disco hay digamos una parte 'low' del timestamp (la original de 32 bits) y una parte 'high' (32 bits). *Más adelante hablo un poco más sobre esto último*.

Links útiles:
* https://stackoverflow.com/questions/60261913/how-do-ext2-3-4-filesystems-deal-with-64-bit-time-t
* https://searchdatacenter.techtarget.com/tip/The-Unix-year-2038-problem
https://www.zdnet.com/article/linux-is-ready-for-the-end-of-time/

Struct *timespec*

Realmente las **variables** que se usan para los **timestamps** son del **tipo** definido como '**timespec**'. Es un *struct* con 2 campos enteros signed: uno para los segundos (64 bits), y otro para los nanosegundos (32 bits) (este último campo es originario de ext4, pero puede estar en ext2/3 si el tamaño de inodo: *superblock. s_inode_size* es >= 256 bytes. **EXPLICADO EN PÁGINA 15**).
Recordemos que los timestamps en Linux se basan en el Tiempo Unix, por ende almacenan la cantidad de segundos que pasaron desde la Unix Epoch [*01/01/1970 00:00:00 UTC+0*].

32 bits [*antes se usaba*] (https://github.com/torvalds/linux/blob/master/include/vdso/time32.h):

```
5   typedef s32             old_time32_t;
6
7   struct old_timespec32 {
8           old_time32_t    tv_sec;
9           s32             tv_nsec;
10  };
```

64 bits (https://github.com/torvalds/linux/blob/master/include/linux/time64.h):

```
8   typedef __s64 time64_t;
9   typedef __u64 timeu64_t;
10
11  #include <uapi/linux/time.h>
12
13  struct timespec64 {
14          time64_t        tv_sec;                 /* seconds */
15          long            tv_nsec;                /* nanoseconds */
16  };
```

Siendo '*__s64*' un '*long long*' (64 bits):

```
33  typedef __signed__ long long __s64;
34  typedef unsigned long long __u64;
```

https://github.com/torvalds/linux/blob/master/include/uapi/asm-generic/int-ll64.h

y para curiosear, acá se definen todos los tipos enteros que se usan:

```
16  typedef __s8  s8;
17  typedef __u8  u8;
18  typedef __s16 s16;
19  typedef __u16 u16;
20  typedef __s32 s32;
21  typedef __u32 u32;
22  typedef __s64 s64;
23  typedef __u64 u64;
```

https://github.com/torvalds/linux/blob/master/include/asm-generic/int-ll64.h

Podemos ver esto en la definición del ***struct inode*** (usado en memoria, en el VFS):

```
644         dev_t                   i_rdev;
645         loff_t                  i_size;
646         struct timespec64       i_atime;
647         struct timespec64       i_mtime;
648         struct timespec64       i_ctime;
649         spinlock_t              i_lock; /* i_blocks, i_bytes, maybe i_size */
650         unsigned short          i_bytes;
```

https://github.com/torvalds/linux/blob/master/include/linux/fs.h#L611

(*Todo esto haciendo referencia a la versión actual de **Linux: 5.14**. Todas las capturas de código son sacadas del repo oficial de Linux, que está justamente en la versión 5.14 al día de hoy*)

Antiguamente (linux 2.6) se veía así el inodo en memoria (https://elixir.bootlin.com/linux/v2.6.11/source/include/linux/fs.h#L426):

```
dev_t                   i_rdev;
loff_t                  i_size;
struct timespec         i_atime;
struct timespec         i_mtime;
struct timespec         i_ctime;
unsigned int            i_blkbits;
unsigned long           i_blksize;
```

```
10  #ifndef _STRUCT_TIMESPEC
11  #define _STRUCT_TIMESPEC
12  struct timespec {
13          time_t  tv_sec;         /* seconds */
14          long    tv_nsec;        /* nanoseconds */
15  };
16  #endif /* _STRUCT_TIMESPEC */
```

```
77  #ifndef _TIME_T
78  #define _TIME_T
79  typedef __kernel_time_t         time_t;
80  #endif
```

```
21  typedef long            __kernel_time_t;
```

Siendo un *long* de 32 bits.

https://elixir.bootlin.com/linux/v2.6.11/source/include/asm-x86_64/posix_types.h#L21

Revisar si en ext2 está implementado así, en 32 bits. -> Por lo que estuve viendo, la única definición del struct 'inode' (que sería lo que se maneja en memoria, o sea, VFS) está en https://github.com/torvalds/linux/blob/master/include/linux/fs.h#L611, por ende si tenemos la última versión del kernel Linux, **nuestro ext2 tendrá timestamps de 64 bits**; aunque no se bien la diferencia con "ext2_inode_info" (https://github.com/torvalds/linux/blob/master/fs/ext2/ext2.h#L632) que también supuestamente es una estructura de memoria, aunque no incluye los timestamps jej (**ESTO ÚLTIMO SE RESPONDE CON LO DICHO EN PÁGINA 8, Y SE CONCLUYE EN PÁGINA 14**).

Links útiles:
(https://www.halolinux.us/kernel-reference/the-ext2sbinfo-and-ext2inodeinfo-structures.html).
(https://www.ionos.es/digitalguide/servidores/know-how/como-averiguar-la-version-de-linux/)

Investigando el manejo de los timestamps en ext2

Definiciones de ext2 (structs,funcs): https://github.com/torvalds/linux/blob/master/fs/ext2/ext2.h

ext2.h tiene el siguiente include: #include **<linux/fs.h>**.

Definiciones de los filesystem **en general** que son compatibles con Linux (structs, funciones, etc): https://github.com/torvalds/linux/blob/master/include/linux/fs.h

donde se define, entre otras cosas, la estructura '*inode*'.

Ahora si nos vamos a la implementación del inodo de ext2:
https://github.com/torvalds/linux/blob/master/fs/ext2/inode.c

Vemos que prácticamente siempre que es necesario referenciar a un inodo para modificarlo o lo que sea, se hace con un '*struct inode*'. Por ejemplo:

```
1534  static int __ext2_write_inode(struct inode *inode, int do_sync)
1535  {
1536          struct ext2_inode_info *ei = EXT2_I(inode);
1537          struct super_block *sb = inode->i_sb;
1538          ino_t ino = inode->i_ino;
1539          uid_t uid = i_uid_read(inode);
1540          gid_t gid = i_gid_read(inode);
1541          struct buffer_head * bh;
1542          struct ext2_inode * raw_inode = ext2_get_inode(sb, ino, &bh);
1543          int n;
```

Incluso para los timestamps:

```
598             inode->i_ctime = current_time(inode);
```

Por ende, podemos concluir que ext2 hoy en día usa timestamps de tipo *timespec64*, o sea, timestamps de 64 bits con signo ;).

Nota1: ese caso de 'current_time', la función está definida en:
https://github.com/torvalds/linux/blob/master/include/linux/fs.h#L1746

```
1746    extern struct timespec64 current_time(struct inode *inode);
```

La cual la implementa el **inodo general** de todo filesytem compatible con Linux:

```
2299    /**
2300     * current_time - Return FS time
2301     * @inode: inode.
2302     *
2303     * Return the current time truncated to the time granularity supported by
2304     * the fs.
2305     *
2306     * Note that inode and inode->sb cannot be NULL.
2307     * Otherwise, the function warns and returns time without truncation.
2308     */
2309    struct timespec64 current_time(struct inode *inode)
2310    {
2311            struct timespec64 now;
2312
2313            ktime_get_coarse_real_ts64(&now);
2314
2315            if (unlikely(!inode->i_sb)) {
2316                    WARN(1, "current_time() called with uninitialized super_block in the inode");
2317                    return now;
2318            }
2319
2320            return timestamp_truncate(now, inode);
2321    }
2322    EXPORT_SYMBOL(current_time);
```

https://github.com/torvalds/linux/blob/master/fs/inode.c#L2298

Acá también se implementa otra función referida a timestamps:

```
1751    int generic_update_time(struct inode *inode, struct timespec64 *time, int flags)
1752    {
1753            int dirty_flags = 0;
1754
1755            if (flags & (S_ATIME | S_CTIME | S_MTIME)) {
1756                    if (flags & S_ATIME)
1757                            inode->i_atime = *time;
1758                    if (flags & S_CTIME)
1759                            inode->i_ctime = *time;
1760                    if (flags & S_MTIME)
1761                            inode->i_mtime = *time;
1762
1763                    if (inode->i_sb->s_flags & SB_LAZYTIME)
1764                            dirty_flags |= I_DIRTY_TIME;
1765                    else
1766                            dirty_flags |= I_DIRTY_SYNC;
1767            }
1768
1769            if ((flags & S_VERSION) && inode_maybe_inc_iversion(inode, false))
1770                    dirty_flags |= I_DIRTY_SYNC;
1771
1772            __mark_inode_dirty(inode, dirty_flags);
1773            return 0;
1774    }
1775    EXPORT_SYMBOL(generic_update_time);
```

Nota2: tanto include/linux/**fs.h** como fs/**inode.c** son **generales** a cualquier filesystem, o sea, en las implementaciones de los distintos fs (que asumo son los llamados '***drivers***' de cada fs) compatibles con Linux (*ext2/3/4, ntfs, fat32, extFat, etc, etc*), se puede hacer uso de la cabecera general **fs.h** que define por ejemplo la estructura de un inodo general en memoria, funciones generales, etc., e **inode.c** implementa las funciones generales de manejo de inodos, las cuales varias están definidas en fs.h. Ya luego por ejemplo **ext2** y **ext4** tienen algunas **definiciones** de funciones y structs más **específicas** en **ext2.h** y **ext4.h** e **implementaciones** más **específicas** de inodos en ***ext2/inode.c*** y ***ext4/inode.c***.

Ext2: https://github.com/torvalds/linux/tree/master/fs/ext2
Ext4: https://github.com/torvalds/linux/tree/master/fs/ext4

[*Yo creo que por más que cierto fs no use inodos como tal, se aprovecha el concepto de representación de un archivo/directorio para usar ciertas funciones que puedan servir para la representación de archivo concreta del fs en cuestión, por ejemplo en fat32 los metadatos de un archivo están directo en los directory entries, los DE representan a los archivos/directorios, y quizás alguna funcion de fs.h ayude a modificar/escribir/leer algún dato o lo que sea.*
De hecho, acá está el inodo de fat jé: https://github.com/torvalds/linux/blob/master/fs/fat/inode.c o sea por más que en fat no haya inodos, la implementación de Linux del driver para ser compatible con fat, utiliza inodos, los reaprovecha adaptándolos a la organización de fat.]

[Además, encontré esto: "Filesystems without inodes generally store the information as part of the file itself. Some modern filesystems also employ a database to store the file's data. Whatever the case, the inode object is constructed in whatever manner is applicable to the filesystem." http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch12lev1sec6.html]

[Y esto: "Inode exists both as a VFS entity (in memory) and as a disk entity (for UNIX, HFS, NTFS, etc.). The inode in VFS is represented by the structure struct inode. Like the other structures in VFS, struct inode is a generic structure that covers the options for all supported file types, even those that do not have an associated disk entity (such as FAT)."

"The inode structure is the same for all file systems. In general, file systems also have private information. These are referenced through the i_private field of the structure. Conventionally, the structure that keeps that particular information is called <fsname>_inode_info, where fsname represents the file system name. For example, minix and ext4 filesystems store particular information in structures struct minix_inode_info, or struct ext4_inode_info."
https://linux-kernel-labs.github.io/refs/heads/master/labs/filesystems_part2.html]

Nota3: este inodo general no tiene definido el struct de '*timespec64*', sino que lo obtiene de fs.h, ya que hace un #include **<linux/fs.h>** (*al igual que el inodo particular de ext2*). Y fs.h tampoco tiene definido el struct '*timespec64*', pero vemos que hace referencia a él en varias partes del código, por lo que en alguno de sus # include lo incluirá.
(*debería haber un #include **<linux/time.h>***)*.

Nota4: *técnicamente **llegamos al struct '*timespec64*' de la siguiente manera**:

Todo arranca en **time.h**: https://github.com/torvalds/linux/blob/master/include/linux/time.h

102 lines (87 sloc) | 3.14 KB

```c
/* SPDX-License-Identifier: GPL-2.0 */
#ifndef _LINUX_TIME_H
#define _LINUX_TIME_H

# include <linux/cache.h>
# include <linux/math64.h>
# include <linux/time64.h>

extern struct timezone sys_tz;

int get_timespec64(struct timespec64 *ts,
                   const struct __kernel_timespec __user *uts);
int put_timespec64(const struct timespec64 *ts,
                   struct __kernel_timespec __user *uts);
```

Ese #include **<linux/time64.h>** nos lleva a:
https://github.com/torvalds/linux/blob/master/include/linux/time64.h

163 lines (141 sloc) | 4.36 KB

```c
/* SPDX-License-Identifier: GPL-2.0 */
#ifndef _LINUX_TIME64_H
#define _LINUX_TIME64_H

#include <linux/math64.h>
#include <vdso/time64.h>

typedef __s64 time64_t;
typedef __u64 timeu64_t;

#include <uapi/linux/time.h>

struct timespec64 {
        time64_t        tv_sec;                 /* seconds */
        long            tv_nsec;                /* nanoseconds */
};
```

Donde se define el struct '*timespec64*'.

Como algo interesante, en **time.h** también se incluye:

```
60   # include <linux/time32.h>
```

Y si vamos a ese archivo (https://github.com/torvalds/linux/blob/master/include/linux/time32.h):

72 lines (63 sloc) | 1.75 KB

```
1   #ifndef _LINUX_TIME32_H
2   #define _LINUX_TIME32_H
3   /*
4    * These are all interfaces based on the old time_t definition
5    * that overflows in 2038 on 32-bit architectures. New code
6    * should use the replacements based on time64_t and timespec64.
7    *
8    * Any interfaces in here that become unused as we migrate
9    * code to time64_t should get removed.
10   */
11
12  #include <linux/time64.h>
13  #include <linux/timex.h>
14
15  #include <vdso/time32.h>
16
17  struct old_itimerspec32 {
```

Nos encontramos con ese interesante comentario, lo que nos da a entender que ya prácticamente todo respecto a timestamps, se migró/actualizó a 64 bits :).

Pero creo que conservan las estructuras de 32 bits por las dudas… por si alguna computadora las requiere en su Linux, quizás por la arquitectura que tenga o algo. O quizás simplemente de recuerdo las dejaron je.

Y bueno, ese include de 'vdso/time32.h' nos lleva al antiguo '*timespec*', como vimos antes (https://github.com/torvalds/linux/blob/master/include/vdso/time32.h):

17 lines (13 sloc) | 274 Bytes

```
1  /* SPDX-License-Identifier: GPL-2.0 */
2  #ifndef __VDSO_TIME32_H
3  #define __VDSO_TIME32_H
4
5  typedef s32             old_time32_t;
6
7  struct old_timespec32 {
8          old_time32_t    tv_sec;
9          s32             tv_nsec;
10 };
11
12 struct old_timeval32 {
13         old_time32_t    tv_sec;
14         s32             tv_usec;
15 };
16
17 #endif /* __VDSO_TIME32_H */
```

Igual otra cosa interesante es que acá:
https://github.com/torvalds/linux/blob/master/include/uapi/linux/time.h

También se encuentra digamos que el antiguo/original '*timespec*'

```
8  #ifndef __KERNEL__
9  #ifndef _STRUCT_TIMESPEC
10 #define _STRUCT_TIMESPEC
11 struct timespec {
12         __kernel_old_time_t     tv_sec;         /* seconds */
13         long                    tv_nsec;        /* nanoseconds */
14 };
15 #endif
```

Su #include <linux/types.h> nos lleva a*:
https://github.com/torvalds/linux/blob/master/include/linux/types.h

Donde su #include <uapi/linux/types.h> nos lleva a:
https://github.com/torvalds/linux/blob/master/include/uapi/linux/types.h

Donde su #include <linux/posix_types.h> nos lleva a:
https://github.com/torvalds/linux/blob/master/include/uapi/linux/posix_types.h

Donde su #include <asm/posix_types.h> nos lleva a:
https://github.com/torvalds/linux/blob/master/include/uapi/asm-generic/posix_types.h

Donde finalmente se define el tipo '*__kernel_old_time_t*':

```
84  /*
85   * anything below here should be completely generic
86   */
87  typedef __kernel_long_t __kernel_off_t;
88  typedef long long       __kernel_loff_t;
89  typedef __kernel_long_t __kernel_old_time_t;
90  #ifndef __KERNEL__
91  typedef __kernel_long_t __kernel_time_t;
92  #endif
93  typedef long long __kernel_time64_t;
94  typedef __kernel_long_t __kernel_clock_t;
```

Y '*__kernel_long_t*' está definido más arriba como un *long* (32 bits):

```
14  #ifndef __kernel_long_t
15  typedef long            __kernel_long_t;
16  typedef unsigned long   __kernel_ulong_t;
17  #endif
```

* Pero si nos vamos por '#include <linux/time_types.h>':
https://github.com/torvalds/linux/blob/master/include/uapi/linux/time_types.h

```
5   #include <linux/types.h>
6
7   struct __kernel_timespec {
8           __kernel_time64_t       tv_sec;                 /* seconds */
9           long long               tv_nsec;                /* nanoseconds */
10  };
```

Vemos el actual timespec de 64 bits, donde '*__kernel_time64_t*' también está definido en https://github.com/torvalds/linux/blob/master/include/uapi/asm-generic/posix_types.h pero esta vez como un *long long* (64 bits).

```
93  typedef long long __kernel_time64_t;
```

(*antiguamente era así*:
https://elixir.bootlin.com/linux/v2.6.11/source/include/asm-x86_64/posix_types.h#L21)

Extra1: Tipo de datos de timestamps en ext4

* **SÍ, ES ASÍ**, ya que al igual que con ext2, la implementación del inodo de ext4: https://github.com/torvalds/linux/blob/master/fs/ext4/inode.c hace referencias a '*struct inode*', el cual por lo que ya vimos, sus campos de timestamps son de tipo '*timespec64*'.

Por ejemplo:

```
3710  void ext4_set_aops(struct inode *inode)
3711  {
3712          switch (ext4_inode_journal_mode(inode)) {
3713          case EXT4_INODE_ORDERED_DATA_MODE:
3714          case EXT4_INODE_WRITEBACK_DATA_MODE:
3715                  break;
3716          case EXT4_INODE_JOURNAL_DATA_MODE:
3717                  inode->i_mapping->a_ops = &ext4_journalled_aops;
3718                  return;
3719          default:
3720                  BUG();
3721          }
3722          if (IS_DAX(inode))
3723                  inode->i_mapping->a_ops = &ext4_dax_aops;
3724          else if (test_opt(inode->i_sb, DELALLOC))
3725                  inode->i_mapping->a_ops = &ext4_da_aops;
3726          else
3727                  inode->i_mapping->a_ops = &ext4_aops;
3728  }
```

O respecto a timestamps:

```
4125          inode->i_mtime = inode->i_ctime = current_time(inode);
```

Además de que tiene especialmente los include de 'fs.h' y 'time.h':

```
22  #include <linux/fs.h>
23  #include <linux/mount.h>
24  #include <linux/time.h>
25  #include <linux/highuid.h>
```

Y como extra, en ext4.h (https://github.com/torvalds/linux/blob/master/fs/ext4/ext4.h) podemos ver en el inodo particular de ext4 en memoria, el tipo de dato del campo i_crtime (es nuevo acá):

```
1103          /*
1104           * File creation time. Its function is same as that of
1105           * struct timespec64 i_{a,c,m}time in the generic inode.
1106           */
1107          struct timespec64 i_crtime;
```

El resto de los timestamps, como dice ahí, están en el inodo genérico (*por eso tampoco los vemos en el inodo particular de **ext2.h**: "ext2_inode_info" jeje*).

Con ese comentario **confirmamos** que **<fs/inode.c>** es la **implementacion genérica** de un inodo, y después cada fs de ser necesario hace implementaciones para agregarle cosas especificas ;)

Y también podemos ver estas 2 funciones, haciendo uso del struct '*timespec64*':

```
893   static inline __le32 ext4_encode_extra_time(struct timespec64 *time)
894   {
895           u32 extra =((time->tv_sec - (s32)time->tv_sec) >> 32) & EXT4_EPOCH_MASK;
896           return cpu_to_le32(extra | (time->tv_nsec << EXT4_EPOCH_BITS));
897   }
898
899   static inline void ext4_decode_extra_time(struct timespec64 *time,
900                                             __le32 extra)
901   {
902           if (unlikely(extra & cpu_to_le32(EXT4_EPOCH_MASK)))
903                   time->tv_sec += (u64)(le32_to_cpu(extra) & EXT4_EPOCH_MASK) << 32;
904           time->tv_nsec = (le32_to_cpu(extra) & EXT4_NSEC_MASK) >> EXT4_EPOCH_BITS;
905   }
```

Extra2: Uso de los nanosegundos en los timestamps

Los timestamps con precisión de nanosegundos, son originarios de Ext4. Esto quiere decir que los timestamps pueden ser de la forma "*seconds* + *nanosecons**10^-9", por ejemplo 1111111111.123456789. Los nanosegundos de cada timestamp se graban en el disco en un 2do campo de 32 bits asociado a cada uno. Concretamente 30 bits están destinados a los nanosegundos, y los 2 bits restantes se usan para ampliar el rango (de 32 bits) de los segundos; o sea, tendríamos timestamps de 2^34 segundos y 2^30 nanosegundos como máximo.
[https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Inode_Timestamps]
[https://stackoverflow.com/questions/60261913/how-do-ext2-3-4-filesystems-deal-with-64-bit-time-t]

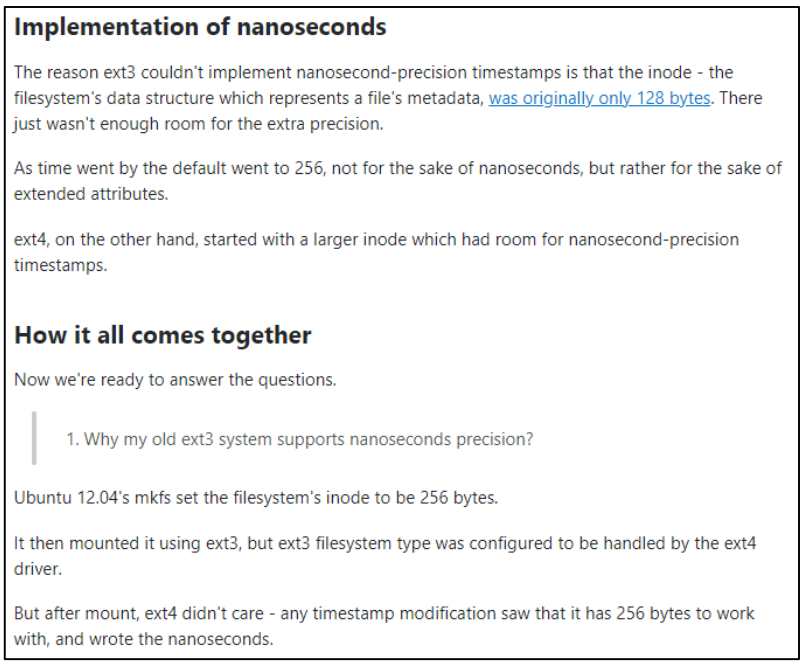
Pero también es posible que aparezcan los nanosegundos en ext2, porque la condición para tenerlos es que el tamaño del inodo sea >= 256 (ya que se necesitan campos extras donde guardar los nanosegundos: https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Inode_Table), y al momento de formatear un dispositivo de almacenamiento en ext2, es posible establecerlo así.

0x80	__le16	i_extra_isize	Size of this inode - 128. Alternately, the size of the extended inode fields beyond the original ext2 inode, including this field.
0x82	__le16	i_checksum_hi	Upper 16-bits of the inode checksum.
0x84	__le32	i_ctime_extra	Extra change time bits. This provides sub-second precision. See Inode Timestamps section.
0x88	__le32	i_mtime_extra	Extra modification time bits. This provides sub-second precision.
0x8C	__le32	i_atime_extra	Extra access time bits. This provides sub-second precision.
0x90	__le32	i_crtime	File creation time, in seconds since the epoch.
0x94	__le32	i_crtime_extra	Extra file creation time bits. This provides sub-second precision.
0x98	__le32	i_version_hi	Upper 32-bits for version number.
0x9C	__le32	i_projid	Project ID.

¿Pero cómo es posible que ext2 sepa cómo grabar y luego manipular los nanosegundos? Bueno por lo que estuve leyendo, realmente el driver que controla a ext2 o ext3, es el de ext4, pero obviamente limitando sus funciones propias de ext4 que sean incompatibles con los demás. O sea, como ext2 es la base de ext4, ext4 puede hacer todo lo que ext2 hace, y por ende sabe manipularlo. Y si se encuentra con un ext2 con inodos de 256 bytes, interpreta que sería como manipular un ext4 y graba los nanosegundos (siempre y cuando tengamos una versión reciente del Kernel Linux).

> The actual support for nanoseconds vs seconds resolution depends on the size of the inode chosen at format time. 128-byte inodes support only second resolution, 256-byte inodes support nanosecond resolution. The ext3 filesystem driver in the kernel is actually running ext4 module for many years, so it is supporting the nanosecond timestamps.

[https://unix.stackexchange.com/questions/574585/file-timestamps-precision-ext3-with-nanoseconds-ext4-with-milliseconds]



Implementation of nanoseconds

The reason ext3 couldn't implement nanosecond-precision timestamps is that the inode - the filesystem's data structure which represents a file's metadata, was originally only 128 bytes. There just wasn't enough room for the extra precision.

As time went by the default went to 256, not for the sake of nanoseconds, but rather for the sake of extended attributes.

ext4, on the other hand, started with a larger inode which had room for nanosecond-precision timestamps.

How it all comes together

Now we're ready to answer the questions.

> 1. Why my old ext3 system supports nanoseconds precision?

Ubuntu 12.04's mkfs set the filesystem's inode to be 256 bytes.

It then mounted it using ext3, but ext3 filesystem type was configured to be handled by the ext4 driver.

But after mount, ext4 didn't care - any timestamp modification saw that it has 256 bytes to work with, and wrote the nanoseconds.

Aclaración: *El tener o no nanosegundos, no tiene que ver con los llamados 'atributos extendidos', eso es otra cosa. Depende del tamaño del inodo.*

https://man7.org/linux/man-pages/man7/xattr.7.html
https://en.wikipedia.org/wiki/Extended_file_attributes

https://stackoverflow.com/questions/20618492/where-are-extended-attributes-stored
https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Extended_Attributes

[https://stackoverflow.com/questions/60841940/file-timestamps-precision-ext3-with-nanoseconds-ext4-with-milliseconds]

[https://man7.org/linux/man-pages/man5/ext4.5.html]

Y como un extra, si vemos la estructura de los timestamps en memoria:

```
13  struct timespec64 {
14          time64_t        tv_sec;                 /* seconds */
15          long            tv_nsec;                /* nanoseconds */
16  };
```

El campo de los nanosegundos se define como '*long*', que serían 32 bits con signo.

Quiere decir que su rango es de [-2^31 a 2^31-1], por lo cual van a entrar bien en el rango positivo (y en el negativo también si es que se usa…), ya que recordemos que se usan 30 bits para los nanosegundos (2^30).

Nota: de todas formas, yo en mi parser solo parseo los 128 bytes de los inodos, ya que ese es el tamaño base de todo inodo, y el único que había originalmente en ext2. En el caso que analice un ext2 con inodos de 256 bytes, mis timestamps seguirán mostrándose sin los nanosegundos. Pero si quiero puedo encontrarlos: leo un inodo en crudo, me posiciono en los bytes donde deberían estar los nanosegundos, y parseo los 30 bits.

CONCLUSIÓN 2

Sobre mi implementación del parser de ext2:

Toda esta investigación la hice para poder entender y ver cómo realmente se manejan los timestamps internamente. Gracias a esto puedo concluir que a mi parser por el momento todo esto no le afecta demasiado. Yo parseo los timestamps asumiendo que directamente son '*unsigned int*' (32 bits), es mi implementación y tomé ese criterio, pero yo lo veo bien, ya que si actualmente el driver de ext2 (implementación oficial) se maneja con variables enteras signed de 64 bits, con nuestro unsigned int abarcamos todo lo que pueda almacenarse en un timestamp, porque recordemos que el límite en ext2 son los 32 bits que se pueden escribir en el disco (*cualquier cosa releer el final de la página 2, donde explico más esto*).

Por ahí si mi parser además hiciera 'writes', quizás sí me pondría a ver si grabó los bytes con variables signed o unsigned, y que luego sea coherente con mi lectura de bytes del disco. Mi implementación lee en unsigned, y si el driver del fs grabó en signed no pasa nada, porque el rango del unsigned abarca todo el rango positivo del signed. Lo único creo que sería si me encuentro con algún archivo de Linux con timestamps de antes del 1/1/1970, que asumo que almacenarán valores negativos (*para eso también son 'signed', segundos negativos representan fechas antiguas*), por ende, mi parser no los interpretará bien.

Sobre las arquitecturas de las computadoras:

Y por otro lado hay que tener presente que si la arquitectura de nuestra computadora es de 32 bits, quiere decir que los registros del procesador sólo pueden almacenar datos de 32 bits creo, entonces creo que por eso se dice que los sistemas de 32 bits llegarían hasta el 2038, y no hasta el 2106, porque creo que en esos casos Linux maneja los timestamps con la implementación de 32 bits (signed int/long) [-2.xxx.xxx.xxx ; 2.xxx.xxx.xxx]. Si quisiéramos alcanzar los 4.xxx.xxx.xxx de lo que sería un unsigned int/long, tendríamos que usar un signed long long (ya que no hay un tipo intermedio), que es de 8 bytes, o sea, 64 bits (sí, nos sobraría para muchos años más que el 2106 jeje). Necesitaríamos un procesador de 64 bits.

(*o quizás sí ande en uno de 32 bits, pero más lento, porque se requieren 2 registros para almacenar un dato de 64 bits*).

Links interesantes:
https://stackoverflow.com/questions/28977587/is-it-ok-to-use-64bit-integers-in-a-32bit-application
https://stackoverflow.com/questions/11901259/how-will-64-bit-variable-be-referenced-in-a-32-bit-process

También hay que estar atento al tamaño de los tipos de datos, porque a veces varían según la arquitectura de nuestra computadora y el compilador del lenguaje.

Por ejemplo, los tipos en C (que es en el lenguaje donde se programó Linux, y por ende, ext2):

> 55
>
> When writing in C or C++, every datatype is architecture and compiler specific. On one system int is 32, but you can find ones where it is 16 or 64; it's not defined, so it's up to compiler.
>
> As for `long` and `int`, it comes from times, where standard integer was 16bit, where `long` was 32 bit integer - and it indeed *was* longer than `int`.

> 48
>
> The specific guarantees are as follows:
>
> - `char` is at least 8 bits (1 byte by definition, however many bits it is)
> - `short` is at least 16 bits
> - `int` is at least 16 bits
> - `long` is at least 32 bits
> - `long long` (in versions of the language that support it) is at least 64 bits
> - Each type in the above list is *at least* as wide as the previous type (but may well be the same).
>
> Thus it makes sense to use `long` if you need a type that's at least 32 bits, `int` if you need a type that's reasonably fast and at least 16 bits.

(https://stackoverflow.com/questions/7456902/long-vs-int-c-c-whats-the-point)

De todas formas, como yo usé Python para programar el parser, no se declaran los tipos y eso, solo tuve en cuenta usar bien el módulo struct con los bytes leídos del disco:

Format	C Type	Python type	Standard size
`x`	pad byte	no value	
`c`	`char`	bytes of length 1	1
`b`	`signed char`	integer	1
`B`	`unsigned char`	integer	1
`?`	`_Bool`	bool	1
`h`	`short`	integer	2
`H`	`unsigned short`	integer	2
`i`	`int`	integer	4
`I`	`unsigned int`	integer	4
`l`	`long`	integer	4
`L`	`unsigned long`	integer	4
`q`	`long long`	integer	8
`Q`	`unsigned long long`	integer	8
`n`	`ssize_t`	integer	

https://docs.python.org/3/library/struct.html

Nota: yo actualmente en mi parser, para todos los timestamps que tenía que leer del disco (32 bits), usé el formato 'I', o sea un entero sin signo. Lo del '*unsigned*' me base del PDF sobre ext2*, pero lo de usar '*int*' en vez de '*long*' fue decisión mía en ese momento. Quizás ahora que sé que los tipos de ext2 son *long* para 32 bits y *long long* para 64 bits, me convendría usar el formato 'L' de struct, para ser más fiel a ext2. De todas formas, con el 'int' me funciona bien*.

Nota2*: además, recordemos cómo son los tipos que se usan al leer directo del disco:

```
297  /*
298   * Structure of an inode on the disk
299   */
300  struct ext2_inode {
301          __le16  i_mode;         /* File mode */
302          __le16  i_uid;          /* Low 16 bits of Owner Uid */
303          __le32  i_size;         /* Size in bytes */
304          __le32  i_atime;        /* Access time */
305          __le32  i_ctime;        /* Creation time */
306          __le32  i_mtime;        /* Modification time */
307          __le32  i_dtime;        /* Deletion Time */
308          __le16  i_gid;          /* Low 16 bits of Group Id */
309          __le16  i_links_count;  /* Links count */
310          __le32  i_blocks;       /* Blocks count */
311          __le32  i_flags;        /* File flags */
312          union {
313                  struct {
```

Donde '*__le32*' se define como '*unsigned int*':

```
29  typedef __u16 __bitwise __le16;
30  typedef __u16 __bitwise __be16;
31  typedef __u32 __bitwise __le32;
32  typedef __u32 __bitwise __be32;
33  typedef __u64 __bitwise __le64;
34  typedef __u64 __bitwise __be64;
```

https://github.com/torvalds/linux/blob/master/include/uapi/linux/types.h

```
26  typedef __signed__ int __s32;
27  typedef unsigned int __u32;
```

https://github.com/torvalds/linux/blob/master/include/uapi/asm-generic/int-ll64.h

* *Viendo esto último, al parecer sí usan el 'int' para 32 bits, así que puedo dejar el parser como está*. Creo que antiguamente había casos que se usaba un '*long*' para representar esos 32 bits, y ahora ya solo '*int*', o seguirá dependiendo de la arquitectura… Igual en el struct de los timestamps (*timespec64*) sí usan un '*long*' para el campo de los nanosegundos, que por lo que sé son 32 bits…

Links interesantes 1:
https://searchdatacenter.techtarget.com/tip/The-Unix-year-2038-problem
https://www.zdnet.com/article/linux-is-ready-for-the-end-of-time/
https://en.wikipedia.org/wiki/Unix_time#:~:text=C%2B%2B20.-,representing%20the%20number,-%5Bedit

Links interesantes 2:
https://giltesa.com/wp-content/uploads/2011/10/tapoDato.png
https://docs.oracle.com/cd/E19253-01/819-6957/chp-typeopexpr-2/index.html
https://stackoverflow.com/questions/13553707/what-does-signed-and-unsigned-values-mean