`GNU as` como ensamblador de propósito general

Raúl Núñez de Arenas Coronado*
dervishd@linuxfreak.com

11 de noviembre de 2000

Esta ponencia forma parte del III Congreso Hispalinux[1] virtual, celebrado en Noviembre de 2000. Puedes participar en él en la sección de eventos de BarraPunto[2]

Resumen

Este seminario tratará de explicar cómo usar el `GNU as` en plataformas `i386` como ensamblador de modo real, modo protegido y modo V86, a pesar de estar diseñado para ser un ensamblador de 32 bits en modo protegido. Va dirigido a personas que ya tienen conocimientos de `GNU as` y de ensamble en general.

1 Introducción: ¿por qué lenguaje de ensamble?

Antes que nada, seamos correctos: ensamble y no ensamblador. El ensamblador es el programa que se encarga de traducir el lenguaje de ensamble a código binario, que puede ya ser ejecutado por el microprocesador. El lenguaje en sí se denomina lenguaje de ensamble o de ensamblado. En el resto del seminario se asumirá Linux cuando se hable de Sistema Operativo (SO) y de `i386` cuando nos refiramos a la arquitectura o a la plataforma de desarrollo, así que el término `i386` identificará tanto al `80386` como al `80486`, el `Pentium` en todas sus variedades, la familia `AMD K5/K6/Athlon`, los `Cyrix`, etc...

Sigamos ahora con la cuestión más importante en este asunto del ensamble: ¿por qué usar lenguaje de ensamble en Linux si tenemos tal variedad de lenguajes de alto nivel?. Pues bien, a este respecto siempre he defendido la postura de **no** programar jamás en ensamble en Linux. Ni en ningún otro Sistema Operativo, y menos aún con los microprocesadores actuales, que con sus arquitecturas superescalares, sus penalizaciones por el vaciado de sus *pipelines* y demás lindezas hacen que ese código que con tanto primor intentábamos optimizar en ensamble corre más lento que el generado por el compilador. En micros con arquitectura load-store el ensamble resulta incluso contraproducente.

Ciertamente no es fácil para una persona que acaba de introducirse en el mundo del lenguaje de ensamblado escribir código para un `8086`, aunque tras un tiempo no presente absolutamente ninguna dificultad. Otro asunto bien distinto es escribir código en 32 bits, dónde los direccionamientos de memoria son mucho más flexibles y dónde nos encontramos ya con los modos de funcionamiento del `i386`: el modo real, el modo protegido y el modo V86. En mi opinión, sólo se debería programar en ensamble en Linux en los siguientes casos:

- Partes del código cuya velocidad sea **realmente** crítica y que nosotros podamos hacer mejor que el compilador...
- Partes del código cuyo tamaño sea **realmente** crítico. Aquí **sí** se puede hacer un mejor trabajo que con el compilador.
- Desarrollos en modo real y en modo virtual86, como puede ser un *MBR Master Boot Record*, código *BIOS*, controladores de dispositivo *POS Pre-Operating System* o *PXE Pre-Execution Environment*
- Cuando sea necesario acceder a recursos de la máquina que de otra forma no se pueden usar desde los servicios del S.O.

[1] `http://congreso.hispalinux.es`

[2] `http://barrapunto.com/eventos`

Así y todo, teniendo en cuenta que el `GNU cc` soporta lenguaje de ensamble empotrado, no es realmente necesario desarrollar directamente en ensamble la mayoría de las veces, sino que se puede usar, por ejemplo, C.

Por fin, cuando uno ya domina las idiosincrasias de cada modo, el funcionamiento del endiablado microprocesador, la confusa sintaxis de códigos de operación y otras lindezas inherentes a la programación en ensamble, entonces surge la gran pregunta: ¿qué hago ahora en Linux?

2 El lenguaje de ensamblado en Linux

Porque claro, en otros sistemas operativos disponíamos de varios ensambladores, todos más o menos compatibles entre sí o cuando menos con sintaxis parecidas; en otros sistemas simplemente se dificulta la programación en ensamble hasta límites insospechados para evitar que se haga, y no habiendo posibilidades, no hay problemas...

Pero llegamos al Sistema Operativo de nuestros amores, Linux, y descubrimos que, de partida, no tiene sintaxis `Intel`, sino `AT&T`... No entraré en una discusión de qué sintaxis es mejor. En cualquier caso, y como opinión personal, prefiero la sintaxis `AT&T`. Es la sintaxis tradicional de Unix, es más *portable* (en el sentido de que para programar para otra arquitectura sólo implica saber los nuevos códigos de operación y los nuevos operandos con nombre –registros, etc...–, pero no nueva sintaxis.) y, una vez que se acostumbra uno, es más fácil detectar errores por el propio ensamblador.

2.1 La sintáxis de ensamblado de `AT&T`

La sintáxis de `AT&T` es poco más o menos como la de Intel, en el sentido de que no cambia los nombres de los códigos de operación ni de los registros, pero tiene algunas diferencias chocantes a primera vista:

- Los nombres de los códigos de operación (*opcodes*) llevan una letra al final que indica el tamaño de los operandos. Aunque algunas veces se puede inferir el tamaño de los operandos de ellos mismos, resulta una buena práctica añadir esta letra al final. El sufijo (la letra en cuestión) es *b* si queremos indicar operandos tamaño *byte*, *w* para tipo *word*, etc...

- En las instrucciones tales como *mov*, que llevan un operando origen y otro destino, el orden en el que se colocan tras el código de operación es el inverso a la sintáxis `Intel`. En ésta, primero se coloca el destino y luego el origen. En la sintáxis `AT&T` se coloca primero el operando de origen y luego el de destino. Las instrucciones que tienen más de un operando de origen *no tienen los operandos invertidos*, al menos en arquitectura `i386`. Esto también ocurre con las instrucciones de punto flotante debido a como las genera el compilador de C.

- Los operandos inmediatos van precedidos por el símbolo *$* en `AT&T`, mientras que no llevan delimitadores en la sintáxis `Intel`.

- Los nombres de los registros van precedidos por el símbolo *%*, mientras que no van delimitados en la sintáxis `Intel`.

- Los operandos de los saltos absolutos cortos van precedidos por el símbolo emph*, mientras que no van delimitados en la sintáxis `Intel`.

- Los saltos largos se forman indicando *section,offset*, mientras que en la sintáxis `Intel` se forman indicando *section :offset*. Además, las instrucciones para salto largo se llaman *ljmp y lcall*, y la de retorno largo se llama *lret*, en lugar de sus correspondientes en sintáxis `Intel`, que son *call far, jmp far y ret far*.

- Por último, en principio se espera que un programa escrito en lenguaje de ensamble en Unix tenga una sola sección (una sección viene a ser como un segmento. Aunque no sea exacto, se puede considerar así.).

2.2 El programa ensamblador en Linux

Miramos el nombre de nuestro ensamblador, y reza `GNU as`... Un poco de investigación nos lleva a saber que es el ensamblador por defecto, el que usa nuestro bienamado `GNU cc` y todos los demás lenguajes de alto nivel.

En seguida dedicamos nuestros esfuerzos a buscar otro ensamblador en lugar del `GNU as`, que tenga sintaxis `Intel`, que tenga un manual más pequeño, o que no tenga manual llegado el caso (ya sabemos lo tedioso que resulta leer...). Para nuestra satisfacción encontramos al menos tres alternativas:

- Realmode Assembler (`as86`).
- Netwide Assembler (`NASM`).
- Table driven Assembler (`tdasm`).

El primero de ellos, el `as86` nos llena de felicidad: no tiene manual. Además, y presumiblemente, sólo sirve para usar en modo real, pero no importa, porque ¿quién es el loco que iba a programar en modo protegido?. Desgraciadamente vemos al poco que la falta de documentación que tanto nos tranquilizó en un principio es un grave problema a la hora de usarlo. Además, parece haber más de una versión del `as86`: una *oficial* y otra que según parece sólo se usa para ensamblar una pequeña parte de kernel.

El `as86` existe practicamente sólo para eso: ensamblar una parte del código de inicio del kernel. Personalmente no lo aconsejaría para usarse como ensamblador de propósito general por cuanto carece de documentación. Así y todo, se puede usar en modo protegido (el kernel de hecho lo hace) pero no está pensado para eso... Y que quede claro: es un pequeño gran ensamblador.

El `NASM` y el `tdasm` son ensambladores algo más serios. Se han pensado para desarrollar completamente en ensamblador y tienen sintaxis `Intel`. No he usado personalmente el `tdasm`, pero sí he usado el `NASM`, y sin ser santo de mi devoción he de reconocer que es un buen ensamblador con un buen soporte de macros.

Todos estos ensambladore tienen dos cosas en común: la sintaxis `Intel` y el código que generan, que es binario puro, no tiene formato. No es más que los códigos de operación uno detrás de otro, sin cabeceras, etc...

Pero resulta que nosotros no vamos a usar ninguno de estos ensambladores, y ahora veremos por qué.

3 ¿Por qué `GNU as`?

Seamos realistas: quizá el `NASM` sea más cómodo de utilizar con su sintaxis `Intel` que el `GNU as`. Quizá su soporte de macros pudiere ser mejor, quizá su nombre es más bonito... Pero si desarrollamos sólo parte de nuestro código en ensamble, mezclándolo con algún lenguaje de alto nivel, entonces usar un ensamblador como ese es practicamente inviable. En Linux los ejecutables tienen formato `ELF`, así que si generamos código objeto con el `NASM` o similares, tendremos que construirnos un objeto en formato `ELF` con el objeto generado por el `NASM`. Parece que en versiones recientes de `NASM` ya se pueden generar directamente objetos `ELF`.

Esto no es una tarea trivial, y obliga a conocer la estructura de un `ELF` para poder hacerlo. Afortunadamente en Linux contamos con `binutils`, que nos permite hacer este tipo de operaciones. De todas formas no es sencillo.

Además, resulta que nuestro compilador `GNU cc` soporta ensamble empotrado, así que decidimos que para algunos códigos muy pequeños no resulta rentable desarrollar el código con `NASM` y preferimos empotrarlo: mala idea; el ensamble empotrado del `GNU cc` tiene sintaxis `AT&T`...

Llegados a este punto tenemos varias alternativas:

- Desarrollar completamente en ensamble con `NASM` y luego integrar el código en el resto del programa con `binutils` y mucha maña.
- Desarrollar completamente con ensamble empotrado, olvidar la sintaxis `Intel` y pasarse a la `AT&T`
- No desarrollar y dedicarnos a la pastelería, profesión que ofrece perspectivas más que interesantes a un programador.

Me temo que si no se nos dan bien las artes culinarias debemos escoger entre las dos primeras opciones. Sin querer hacer publicidad engañosa creo que la más inteligente es la segunda: si vamos a usar ensamble junto a lenguajes de alto nivel pronto descubriremos que resulta más práctico empotrarlo, porque rara vez nos surgirá la necesidad de hacerlo de otra forma. Es más, puesto que ya aprendemos sintaxis `AT&T` nos deja la puerta abierta para usar el `GNU as`.

El `GNU as` genera código en distintos formatos, según esté configurado. En ese aspecto es igual que el resto de herramientas de desarrollo de GNU. Esto lo hace particularmente útil, ya que el código objeto que genera puede ser usado directamente: ya estará en el mismo formato que el resto de los códigos objeto.

Pero esta ventaja se convierte en una contrariedad cuando queremos generar con el `GNU as` código binario sin formato. Veremos que esto resulta infinitamente más simple que hacer lo contrario (coger código generado por `NASM` y convertirlo a formato `ELF`, por ejemplo).

Otra ventaja que podemos destacar del `GNU as` es que genera código en 16 y 32 bits (hablamos de bits de direccionamiento de memoria), en varios modos y en multitud de arquitecturas. Es practicamente el único ensamblador capaz de esto, y es sin duda un punto a su favor, pese a que al final el lenguaje de ensamble es practicamente no portable.

En resumen: todas las opciones tienen sus ventajas e inconvenientes, y hay que valorar éstos a la hora de elegir una alternativa. Por lo general, si desarrollamos mucho código que deba ejecutarse en modo real o virtual86 es mejor usar una herramienta como el `NASM`, aunque ya veremos que también el `GNU as` puede ser usado para ello. Por el contrario, si lo que hacemos de forma habitual es escribir en un lenguaje de alto nivel con partes escritas en ensamble, lo nuestro es la sintaxis `AT&T`, el ensamble empotrado y el `GNU as`.

4 `GNU as` como ensamblador de propósito general

Como ya se apuntó anteriormente, el `GNU as` tiene una ventaja importante a la hora de mezclar código escrito en un lenguaje de alto nivel con código escrito en lenguaje de ensamblado: genera el mismo formato que los demás compiladores (esto, por otra parte, resulta obvio: casi todos los compiladores lo que hacen es generar código en lenguaje de ensamble y ensamblar luego con el `GNU as`). Como añadidura, podremos usar todas nuestras herramientas tradicionales de desarrollo con el código generado por el `GNU as`. Qué maravilla... y qué limitación...

Pues sí, porque en principio las herramientas de desarrollo servirán sólo para cuando el `GNU as` se utilice en modo nativo, y no cuando lo utilicemos para algún propósito particular. Explicaré esto en más detalle algo más adelante.

Pero vayamos al asunto que nos ocupa: usar `GNU as` como ensamblador de propósito general en una plataforma `i386`.

Por lo pronto, vamos a suponer que tenemos `GNU as` configurado para generar código para el objetivo `elf-i386`, que suele ser lo usual en plataformas PC y que es por otro lado el objetivo que nos interesa precisamente. Pues bien, esto significa que sin hacer nada nuestro `GNU as` ya genera código `elf-i386` por defecto, o lo que viene a ser lo mismo en Linux/PC, que genera código en 32 bits para `80386` o superiores, en modo protegido. Esto no quiere decir en absoluto que el modo protegido sea 'de 32 bits'. El modo protegido implica, entre otras muchas cosas, separación entre tareas en código y en datos, pero no tamaño de operandos ni espacio de direccionamiento. El `80286` tenía un modo protegido con direccionamiento máximo de 24 bits, no de 32, por ejemplo. Bien, ya tenemos uno de los tres modos de la plataforma `i386` cubierto. Veamos qué pasa con los otros.

4.1 `GNU as` como ensamblador para modo real

Empecemos con el modo real. En modo real, un `i386` es como un `8086` sólo que mucho más rápido y con ciertas extensiones:

- Tiene registros de propósito general de 32 bits en lugar de 16 bits.
- Tiene dos registros de segmento más (*fs* y *gs*).
- El direccionamiento relativo de memoria es más flexible, ya que permite más combinaciones de registro base+registro índice que un `8086`. Así y todo, hay que hacer notar que sólo se permite direccionar con 16 bits, como en el `8086`... Hay que tener cuidado con los desplazamientos.
- Tiene instrucciones añadidas, y algunas instrucciones que ya existen en el `8086` permiten más combinaciones de operandos que en un `8086`.

Hay alguna más, pero las que nos interesan son esas. En particular, la limitación de direccionamiento de memoria de 16 bits y no 32 como cabría esperar. En realidad se pueden usar las mismas combinaciones de operandos de direccionamiento en las instrucciones que en modo protegido, sólo que

Hay formas de direccionar en 32 bits con el microprocesador en modo real, pero se sale del tema de este seminario.

Pese a que tengamos un `i386` estará limitado, pero no hay que olvidar que sigue siendo un `i386`, con sus instrucciones extendidas, etc... La única limitación que nos importa es esa que implica el término *16 bits*.

Desgraciadamente, poco o nada sabe nuestro `GNU as` de lo limitado de un `i386` en modo real. Para él, el `i386` está siempre en modo protegido, es decir, que permite direccionamiento en 32 bits. Esto significa que los operandos de las instrucciones se supondrán de 32 bits y el direccionamiento se supondrá también en 32 bits, es decir, que podremos direccionar nada menos que 4Gb por segmento... Pero esto **no** es cierto en modo real, en dónde sólo podemos direccionar 1Mb+64kb de memoria en total...

¿Qué ocurre entonces? Pues que si escribimos un código con `GNU as`, lo ensamblamos y posteriormente lo ejecutamos en modo real, no se comportará correctamente, ya que se interpretará de otra forma. No entraré en detalles sobre lo que ocurre pero basicamente se trata de que las instrucciones, en modo real, esperan operandos de 16 bits y no de 32, y puesto que el `GNU as` generará operandos de 32 bits, cada instrucción irá con una *cola* de 16 bits que se interpretará como otra instrucción. Oh, fatalidad...

Para nuestra fortuna, en las últimas versiones de `binutils` (siento no poder precisar la versión exacta a partir de la cual ocurre), el `GNU as` entiende una directiva que le dice que olvide eso de generar código en 32 bits y que lo genere en 16 bits. Estupendo, eso parece solucionar todos nuestros problemas. Esa directiva es la *.code16*. Incluso podemos insertar pedazos de código ensamblados en 32 bits en segmentos definidos con *.code16*, simplemente escribiendo *.code32*, el código 32 y *.code16* para volver al código 16.

En cuanto a los operandos, por defecto serán de 8/16 bits mientras que los sufijos de las instrucciones sean *b* o *w*. Si el sufijo fuese *l* se generará el *size prefix* y después la instrucción, con lo que el `i386` en modo real sabrá que la instrucción usa operandos en 32 bits y la interpretará correctamente. Los problemas no van a venir por ahí...

Y ¿qué pasa con la memoria?. Pues que el `GNU as`, en su implacable exactitud, no nos dejará usar direccionamiento extendido, por mucho que nosotros aseguremos que los desplazamientos son de 16 bits. Este direccionamiento, como en el `8086` consiste en tres partes: un registro base, un registro índice y un desplazamiento (un número...). La diferencia es que en el `8086` están bastante restringidas las combinaciones de estos tres elementos y no todos los registros pueden actuar como base, no todos los registros índice se pueden usar con todos los base, etc... Ciertas combinaciones de registro base+registro índice, y ciertos registros base son inválidos en código puro de 16 bits, es decir, son inválidos en un `8086`. Pero resulta que en nuestro `i386` ¡son perféctamente válidos!. ¿Por qué, entonces, el `GNU as` no quiere que los usemos?[3]. Pues mucho me temo que es una incongruencia del `GNU as`, que en 16 bits no quiere usar direccionamientos inválidos para el `8086` pero sí que nos deja usar códigos de operación extendidos del `i386`...

En cualquier caso, no es problema, ya que se puede preceder cada direccionamiento de memoria que queramos hacer como si fuese en 32 bits de la directiva *.addr32*. Esta directiva obliga al `GNU as` a generar, en segmentos *.code16* (segmentos con direccionamiento de 16 bits) el llamado 'address prefix', que le dice al `i386` que interprete esa instrucción con direccionamiento de 32 bits. Asimismo, existe la directiva *.addr16* para lo inverso.

Llegados a este punto, hemos escrito nuestro código de tal forma que está en un segmento (o sección) *.code16*, hemos intentado no abusar de los operandos de 32 bits y del direccionamiento en 32 bits (ambos añaden un prefijo de 1 byte y en determinados casos esto puede ser un desperdicio de espacio), y, en fin, hemos hecho un programa. ¿Qué viene ahora? Muy simple, ensamblarlo. Asumiremos que nuestro programa se llama *test.s* y que queremos generar *test.o*:

```
as --fatal-warnings -o test.o test.s
```

Pues bien, ya tenemos un código objeto en formato `elf-i386`... que nos es totalmente inútil. Lo que nosotros queremos es generar un programa que se ejecute en modo real, y ciertamente *test.o* no lo es... Pero no desesperemos. Aquí llega para ayudar el programa `objcopy`. Este programa es capaz de copiar código objeto de un formato a otro, y lo único necesario para que nos permita convertir nuestro `elf-i386` a código en modo real es que nuestro `objcopy` soporte el objetivo *binary*. Haremos lo siguiente:

```
objcopy -v -j .data -O binary test.o
```

Ahora *test.o* debería contener exclusivamente el código para ejecutar en modo real. Pero ¿qué es eso de `-j .data`?, preguntará el avispado lector. Muy simple: `binutils` trabaja con *secciones*, que son, a todos los efectos, como segmentos de código. Si bien es posible crear un programa con `GNU as` que tenga múltiples secciones, no he tenido oportunidad de probarlo, y no es en absoluto necesario. Nosotros vamos a asumir que nuestro programa sólo tiene una sección, y va a ser la sección *.data*. ¿Por qué? pues muy simple: así no será tratada como código por ninguna utilidad que pudiere no entender el objetivo *binary*, sino que será considerada como datos y no se modificará. Hay que decirle pues a `objcopy` que copie sólo la sección *.data*. Se dirige al lector al manual del `binutils` para entender el funcionamiento y aplicación de las secciones.

Ya tenemos un código listo para ejecutarse perfectamente en modo real. Existe un Master Boot Record que se ha desarrollado siguiendo esta técnica y que será publicado en breve bajo licencia GPL.

[3]Parece, en cualquier caso que esta incongruencia se ha solucionado ya en la versión 2.10 del `GNU as`.

4.2 GNU as como ensamblador para modo virtual86

Cuando usamos el modo virtual86 el microprocesador está realmente en modo protegido, pero emula un 8086 virtual, lo cual quiere decir que cuando usemos el GNU as tendremos que tener **todas** las precauciones que ya teníamos en modo real. Basicamente el modo virtual86 es un modo real dentro del modo protegido, o mejor aún, un i386 que direcciona como un 8086.

Al igual que en el modo real, podemos usar direccionamiento con sintaxis de 32 bits, es decir, con las combinaciones extendidas de registro base+registro índice del i386, pero con cuidado de que la suma total del desplazamiento no exceda 64kB por segmento, o provocaremos una excepción.

Por otro lado, la generación del binario es como en modo real. (ver sec 4.1)

4.3 Interacción con binutils

Para terminar, hemos mencionado anteriormente que se puede usar binutils con los códigos generados por el GNU as usado como ensamblador de propósito general, y esto no es cierto al 100%. Por lo general, debieran funcionar perfectamente, pero por lo menos en la versión 2.9.5 de binutils, el comando objdump y otros no entienden los códigos de operación que se generan en *.code16*, ya que ellos no disponen de directivas que les indiquen el tamaño por defecto de los operandos y de los direccionamientos.

Esto, por suerte o por desgracia (según quien opine) no es un error, sino el comportamiento obvio de binutils. Esperemos, aún así, que en futuras versiones se amplíe este comportamiento.

5 Notas finales

Creo que queda demostrada la utilidad del GNU as como ensamblador de propósito general, si bien se manifiestan ciertas dificultades que, si bien son salvables, pueden dar lugar a dudas sobre el GNU as. Como dije anteriormente, no se trata de 'vender' el GNU as, sino de hacer ver que para ciertos casos es la solución óptima (cuando se trata de mezclar código ensamble con lenguajes de alto nivel), y que para el resto es una solución más que digna y sin nada que envidiar a ensambladores como el NASM.

Además, este ensamblador está hecho por y para GNU, que si bien para algunos puede significar que tiene una dudosa calidad, para el que escribe supone una garantía que va más allá del precio.

Espero, en cualquier caso, haber sido de ayuda para aquellos que aún codifican en ensamble parte o todo su trabajo.

Se recomienda leer la documentación de binutils en general, y de GNU as en particular para aprender a manejar este ensamblador y aprender más en general de la programación en ensamble en Linux.

Asímismo, existe un *HOWTO* sobre programación en ensamble bajo Linux, llamado *Assembly-HOWTO*, que puede encontrarse en los mirrors del Linux Documentation Project[4] en diversos formatos e idiomas.

Un ejemplo fehaciente del uso del GNU as como ensamblador en modo real se puede encontrar en el Linux Native MBR, que puede encontrarse en mi página WEB[5]. Funciona ;)

Información sobre el i386 se puede encontrar en Intel Corporation[6], y el manual de referencia para la programación del i386, que ya no está disponible en las páginas de Intel se puede encontrar en esta dirección http://www.it.dtu.dk/~jsm/49275/386/toc.htm en formato HTML.

No podemos olvidar, obviamente, a la Asociación Hispalinux[7].

Si se quiere una lista de distribución en Internet (pero con espíritu FidoNet[8]) para hablar sobre el Sistema Operativo Linux[9], hay que dirigirse a FidoLinux[10].

6 Agradecimientos

Este seminario no se hubiese redactado jamás si no se hubiese dado una combinación de culpables. Para empezar y como cosa básica, si Linus Torvalds no hubiese creado el kernel de Linux; para seguir, si un montón enorme de gente (son demasiados para citarlos) no hubiesen creado todo el software que acompaña a este

[4] http://www.linuxdoc.org
[5] http://rauln.ivworlds.org
[6] http://www.intel.com/
[7] http://www.hispalinux.es/
[8] http://www.fidonet.org
[9] http://www.linux.org
[10] http://www.elistas.net/foro/fidolinux

kernel y que constituye el S.O. Linux. Entre este software se cuenta el `GNU as` y todas las demás herramientas de desarrollo.

Pero toda esta gente nada sabe de este seminario, ya que los que de verdad le han dado vida son (en ningún orden en particular):

- La gente de FidoLinux en general, que son estupendos y que se han leído la versión preliminar de este panfleto infame.
- Javi Polo (*DrSlump*), por animarme en primer lugar a hacer esto y por ser ese macarra melenudo que toda lista de Linux necesita. Gracias ;)
- Jose Antonio Fernández Luengo (*Skaven*), por animarme también a tirarme a la piscina con esto y por ser la rata asquerosa que guarda celosamente nuestra lista. Gracias ;)
- Jesus Rodriguez (*Piyayo*), por los ánimos y las peticiones de ejemplos que al final no he puesto X'D. Gracias ;)
- Daniel Payno, por maquetar esto y hacer que el LaTeX produjese algo legible a pesar de todos mis esfuerzos por lo contrario. Gracias :)
- Antonio Tejada Lacaci, por estar ahí al quite corrigiendo todos mis errores sobre el `i386` y por atrapar enormes gazapos en el texto. Gracias :)
- Neil Turton, por su inestimable (e incansable) ayuda a la hora de explicarme la idiosincrasia del `GNU as` y por ser el coautor de mi proyecto de Master Boot Record. Thanks again, Neil :)
- *Wizord*, por el recordatorio del modo protegido del 286, el gran olvidado en todas las fiestas de Sistemas Operativos. Gracias :)))
- Paco Brufal, por resistirse a la tentación de introducir alguna broma de las suyas en el texto de este seminario vía dominación mental XD.
- Esther, por ser mi novia, aunque sepa muy poco de Linux y aunque no le interese ni lo más mínimo el contenido de este seminario XD.
- A todos aquellos que sin duda estoy olvidando debido a mi innata estulticia y a los que pido perdón por esta y otras marranadas que voy a hacerles en el futuro por el simple hecho de conocerme...