INIT/main.c

5.1 Arranque del sistema

Este documento pretende explicar qué sucede justo después de encender el ordenador, es decir, cómo la imagen del núcleo de Linux se copia en memoria y se ejecuta. Describiremos qué pasos se llevan a cabo y quiénes intervienen desde el encendido hasta el momento en el que el usuario puede empezar a interactuar con el computador.

El proceso de inicializar un computador es largo y tedioso, ya que inicialmente casi todos los dispositivos, incluyendo la memoria, están en un estado impredecible. Es además, un proceso totalmente dependiente de la arquitectura del computador.

Cuando queremos ejecutar un programa, podemos escribir su nombre en un terminal, o bien podemos abrirlo desde su icono en un entorno de escritorio, como GNOME o KDE; luego, el kernel lo carga y lo ejecuta. Pero algún otro software debe primero cargar el kernel y ejecutarlo; y otro software debe cargar el cargador del kernel y ejecutarlo, y así podríamos estar infinitamente.

Algo tiene que terminar con esta regresión infinita, y ese algo es el hardware. Así, en el nivel más bajo, el primer paso para arrancar el sistema requiere ayuda del hardware. Dicho hardware, normalmente ejecuta alguna rutina corta (de sólo lectura que se guarda en una dirección concreta de memoria para que ésta no requiera de ningún cargador de software) que, a su vez, ejecuta algo más grande y más sofisticado, y éste, a su vez, algo más grande y sofisticado, hasta que se carga el kernel.

Los detalles de cómo se realiza cada cosa varían dependiendo de la arquitectura en la que estemos, pero el procedimiento es el mismo.

En el proceso de arranque podemos encontrar las siguientes etapas:

- Para empezar, cada CPU de nuestro ordenador debe inicializarse a sí mismo, y ejecuta un test a sí misma que dura fracciones de segundo. En un sistema multiprocesador, una de las CPUs es siempre la CPU primaria, y las odemás las secundarias; la CPU primaria se encarga de todo lo que queda del arranque del sistema y, posteriormente el kernel activa la CPU secundaria.

- Después, la CPU ejecuta la instrucción que se encuentra en la dirección 0xffffffff0, una dirección muy cercana a la última dirección posible en una CPU de 32 bits. Esto es así porque, como no todos los ordenadores tienen el privilegio de tener 4GB de memoria RAM, en esta dirección no hay memoria. Para aquellas máquinas que sí tienen 4GB de RAM o más, simplemente pierden esa porción de memoria en la parte alta del espacio de direcciones de la BIOS (en realidad son exactamente 64Kb), una pérdida insignificante cuando tenemos 4GB.

La instrucción que se encuentra en dicha dirección es un salto al comienzo del código de la BIOS (Basic Input/Output System) que se encuentra en la placa base y controla la siguiente fase del arranque. A la CPU ni siquiera le importa que la BIOS existe, y eso es lo que hace posible que procesadores de Intel puedan funcionar en ordenadores con una arquitectura

diferente a la de PC. La CPU simplemente ejecuta la instrucción que se encuentre en esa dirección, y es ya una parte de la arquitectura PC hacer que esa instrucción sea un salto a la BIOS.

- La BIOS empieza eligiendo un dispositivo para el arranque, usando sus propias reglas internas. Con frecuencia, estas reglas se pueden modificar durante el arranque (lo que conocemos como el acceso a la BIOS, que en algunos sistemas se hace presionando la tecla “suprimir”), y navegando por sus opciones.

En este dispositivo, la BIOS lee el primer sector –primeros 512 bytes- que se conoce como MBR (Master Boot Record). Lo que pasa a continuación depende de cómo se haya instalado Linux en nuestro sistema. Lo que suele hacer la BIOS es buscar los llamados “números mágicos” en el MBR, que nos indican que realmente estamos en el MBR de la memoria, y luego busca la localización del sector de arranque. Posteriormente carga este sector, que contiene la dirección donde se encuentra LILO/GRUB, en memoria, y salta a su comienzo.

Master Boot Record (MBR), el cual se divide en 3 zonas:

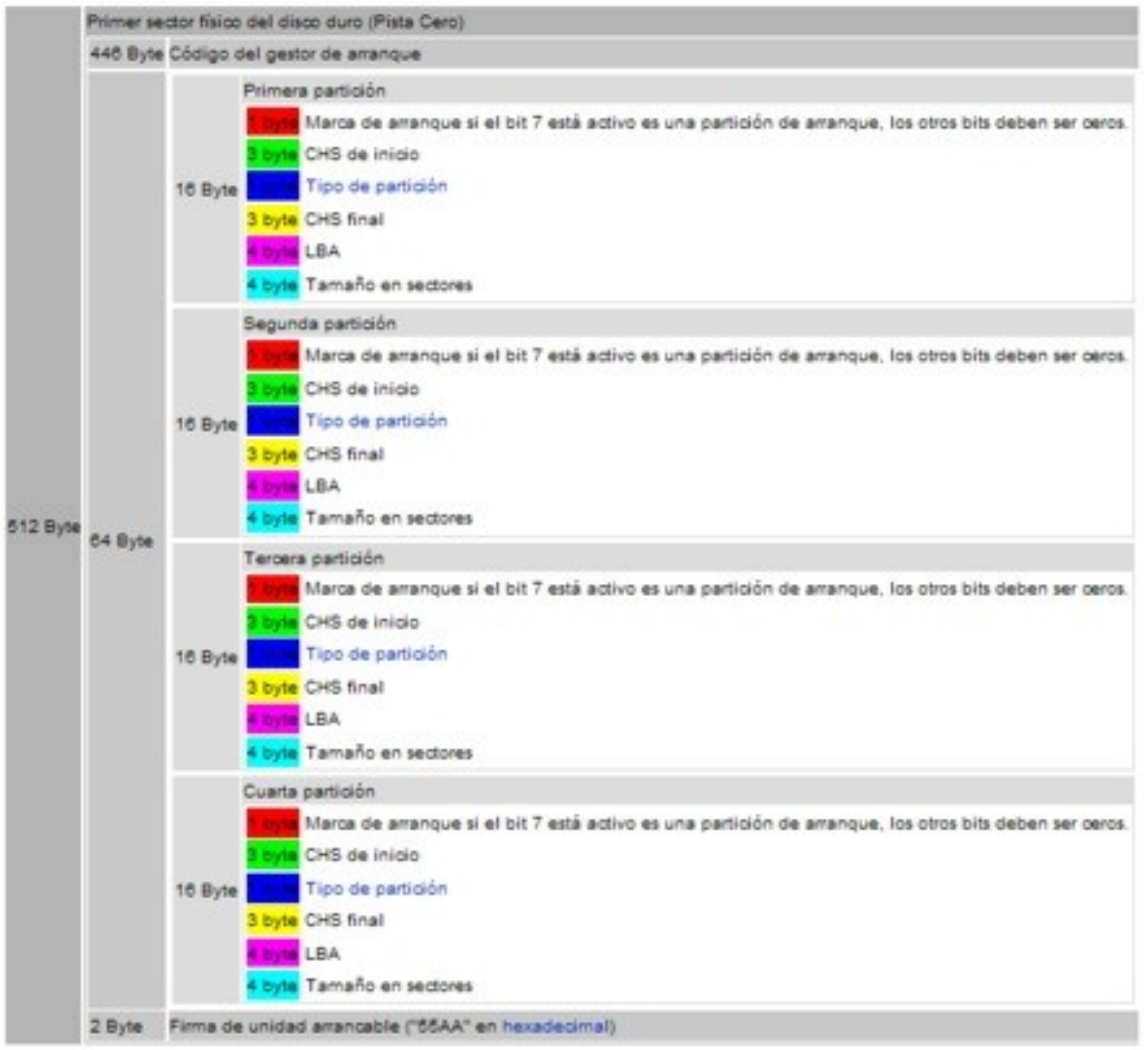
A: 446B	B: 64B	C: 2B

A) Reservados para código de programas.

B) Tabla de particiones de hasta 4 entradas. La tabla de particiones contiene información requerida por el sistema operativo sobre la distribución del disco duro y el tipo de sistema de archivos.

C) Los últimos dos bytes deben contener una “cifra mágica” (AA55): un MBR que tenga otra cifra será tratado como no válido por parte de la BIOS y de todos los sistemas operativos de PC.

Imagen aclarativa de la estructura del Master Boot Record:



- Los sectores de arranque son los primeros que se encuentran en cada partición a excepción de la partición extendida, que es un “contenedor” para otras particiones. En Linux los sectores de arranque de una partición están, en principio, vacíos, incluso después de haber generado el sistema de archivos. Por lo tanto, una partición Linux no es autoarrancable aunque tenga un núcleo y un sistema de archivos raíz válido. Para arrancarlo necesitamos de un gestor de arranque como Lilo o Grub.

Llegados a este punto, hemos realizado ya una transición de lo que era nuestra área de interés al principio, hardware y pequeñas rutinas, a lo que es nuestra área de interés en este momento, el software propiamente dicho.

- En este momento, se está cargando el GRUB. Él mismo termina de cargarse y encuentra sus datos de configuración en disco (grub.conf), que le dice, entre otras cosas, dónde encontrar el kernel y qué opciones pasarle al inicio. A continuación, el GRUB carga el kernel en memoria y salta hacia ella.

- Normalmente, los núcleos se encuentran guardados como archivos comprimidos que contienen las instrucciones sin comprimir básicas al principio, que posteriormente descomprimen al resto. Por tanto, el siguiente paso para el núcleo es descomprimirse a sí mismo, mediante la carga de unas rutinas básicas, que son setup() y startup_32(), y luego saltar al comienzo de la imagen del kernel sin comprimir. En este punto, el kernel ya se ha terminado de cargar.

En resumen, los pasos que hemos descrito son los siguientes:

1.- La CPU se inicializa y ejecuta una instrucción en una localización fija.

2.- La instrucción salta a la BIOS.

3.- La BIOS encuentra un dispositivo de arranque y accede a su MBR, que apunta al GRUB.

4.- La BIOS carga y transfiere el control al GRUB.

5.- GRUB carga el kernel comprimido.

6.- El kernel comprimido se descomprime y transfiere el control al kernel previamente descomprimido.

Después de que el kernel se haya cargado en memoria, y el hardware imprescindible –como la MMU (unidad de manejo de memoria)- haya sido configurada a bajo nivel, el kernel salta a la función “start_kernel”, que se encarga de realizar lo que nos resta de inicialización.

La etiqueta __init le dice al compilador gcc que coloque esta función en una sección especial del kernel. Después de que el kernel haya terminado de inicializarse, él mismo liberará esta sección especial de memoria. Lo mismo pasa con las etiquetas __initfunc e __initdata.

A continuación, se inicializan algunos de los componentes del núcleo, como la memoria, las interrupciones hardware, el paginado, etc. En particular, esto lo hace la función “setup_arch”. También se inicializa la tabla de procesos. Luego lanza la función init para terminar la inicialización del núcleo, inicializa el resto de cpu’s, y posteriormente, ejecuta el programa init, el primer proceso de usuario del sistema (PID=1).

El proceso init inicializa el resto del sistema que no esta en el núcleo, (correo, editores, entorno gráfico, etc). Ejecuta un login para el Terminal, espera a que un usuario se introduzca y lanza un shell para el usuario.

Rutinas en ensamblador

setup ()

setup.S contiene código en ensamblador de 16 bits que se ejecuta en modo real, y que se encarga de reconfigurar todos los dispositivos del sistema según las necesidades de Linux. Los pasos son los siguientes:

1.- Invoca a la BIOS para averiguar cuánta memoria hay en el sistema.

2.- Configura el teclado (velocidad de repetición de las teclas).

3.- Inicializa la tarjeta de vídeo, y el controlador de disco.

4.- Comprueba la existencia de MCA, ratón PS/2 o BIOS APM.

5.- Mueve el núcleo si está en la zona baja de memoria.

6.- Establece una IDT y una GDT provisionales.

7.- Deshabilita casi todas las interrupciones en los controladores de interrupciones.

8.- Pasa a modo protegido (32 bits) y salta a startup32().

startup_32()

Existen dos rutinas startup.

La primera 35 startup_32: se encarga de descomprimir el núcleo si es necesario. Está en /arch/x86/boot/compressed/head_32.S

- El núcleo comprimido está en 0x00001000 o en 0x00100000. Se inicializan los registros de segmentación y se ponen a 0 zonas de datos reservadas del núcleo.

- Se descomprime el nucleo (decompress_kernel()) en 0x00100000

  · Mensajes “Uncompressing Linux...” y “OK, booting the kernel”

  · Si residía en 0x00100000, se descomprime detrás y luego se copia en 0x00100000

- Salta de nuevo a 0x00100000, donde está la verdadera startup32()

  · En caso de un núcleo sin comprimir, habríamos saltado a esta directamente.

La segunda 85 ENTRY(startup_32) establece el entorno para el primer proceso del sistema (proceso 0). Está en /arch/x86/kernel/head_32.S

- Establece los segmentos con sus valores finales

- Activa la paginación

  · Inicialmente, la paginación se establece de forma que las direcciones lineales sean iguales a las físicas en los primeros 8Mb.

- Prepara la pila de modo núcleo

- Inicializa la IDT con manejadores vacíos ignore_int

- Pone los parámetros obtenidos de la BIOS y los pasados al núcleo en el primer marco de página

- Identifica el modelo de CPU y toma medidas al respecto

- Se cargan GDT e IDT para un modelo de memoria plano

  · Linux no emplea la segmentación, pero ésta no se puede desactivar del todo: se hace que todos los segmentos tengan base=0 y límite = 4Gb-1.

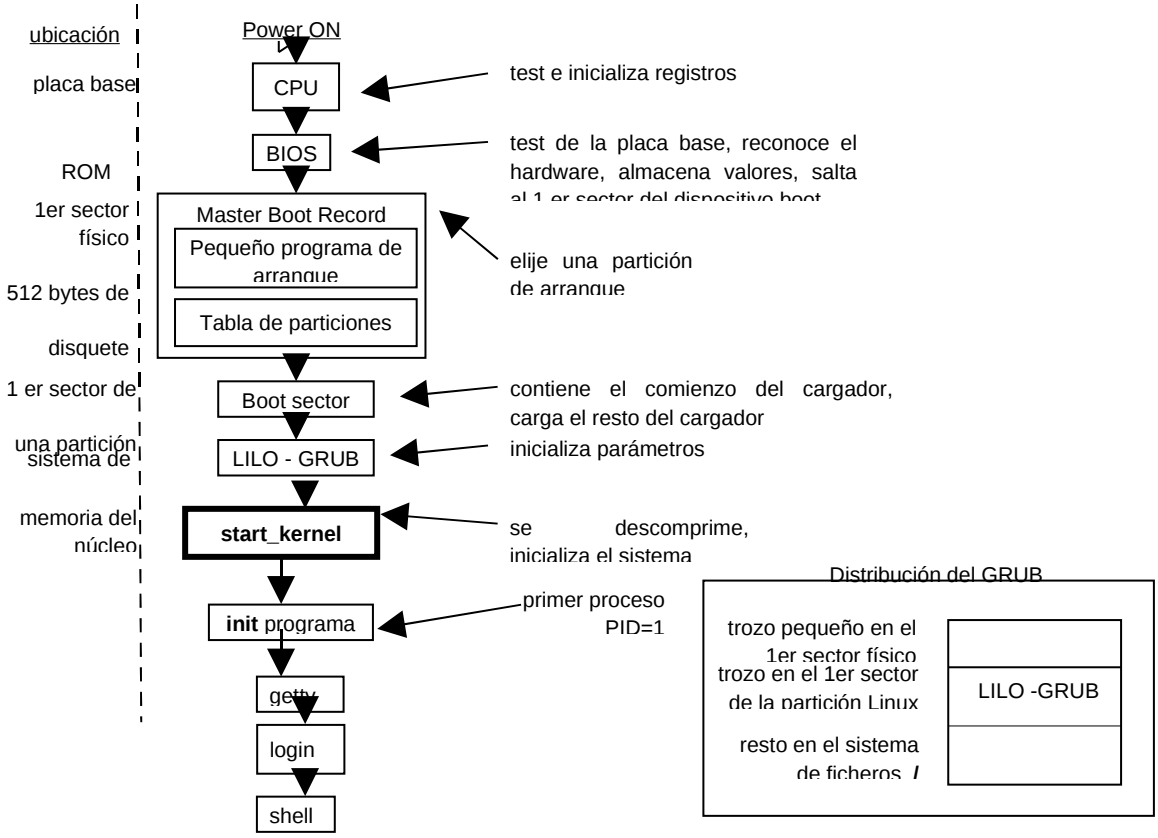
- Se salta a start_kernel()

```
447         cmpb $0,%cl             # the first CPU calls start_kernel
 448         je   1f
 449         movl $(__KERNEL_PERCPU), %eax
 450         movl %eax,%fs           # set this cpu's percpu
 451         movl (stack_start), %esp
 4521:
 453  #endif /* CONFIG_SMP */
 454         jmp *(initial_code)

599   ENTRY(initial_code)
 600         .long i386_start_kernel
```

A partir de aquí, el código es en C.

Esquema gráfico del arranque del sistema.



5.2 START_KERNEL

Tras haber cargado en memoria el núcleo y después de haber hecho las inicializaciones del manejador de memoria y del Hardware, el núcleo llama a ***start_kernel***, que es el punto de entrada al inicio del sistema, realiza todo el proceso de arranque de Linux, llamando a todas las funciones necesarias para poner el núcleo en un estado operacional.

La función init/main.c:start_kernel() está escrita en C y realiza lo siguiente:

1. Realiza un cierre global del núcleo (es necesario para que sólo una CPU realice la inicialización). (566)

2. Muestra el "anuncio" del núcleo Linux conteniendo la versión, el compilador usado para construirlo, en qué máquina se ha compilado, a qué hora, etc. Esto es tomado desde la variable linux_banner definida en init/version.c y es la misma cadena mostrada por cat /proc/version (571)

3. Realiza configuraciones específicas de la arquitectura (análisis de la capa de memoria, copia de la linea de comandos de arranque otra vez, etc.). (572)

4. Inicializa los datos requeridos por el planificador (scheduler). (585)

5. Inicializa las traps (excepciones). (604)

6. Inicializa las irqs (interrupciones hardware). (606)

7. Inicializa la fecha y la hora del sistema. (608)

8. Inicializa el subsistema softirq (interrupciones software). (610)

9. Analiza las opciones del arranque de la línea de comandos.

10. Inicializa la consola. (625)

11. Si el soporte para módulos ha sido compilado en el núcleo, inicializa la facilidad para la carga dinámica de módulos.

12. Si la línea de comandos "profile=" ha sido suministrada, inicializa los perfiles de memoria intermedia.

13. Habilita las interrupciones.

14. Llama a mem_init(), que calcula max_mapnr, totalram_pages y high_memory y muestra la linea "Memory: ...". (652)

15. kmem_cache_init(), inicializa la mayoría del asignador slab. (655)

16. Calcula el valor BogoMips para esta CPU. (662)

17. fork_init(), crea uid_cache, inicializa max_threx_threads basándose en la cantidad de memoria disponible y configura RLIMIT_NPROC para que init_task sea max_threads/2. (672)

18. Realiza "chequeos de fallos" específicos de la arquitectura y, cuando es posible, activa las correcciones para los fallos de procesadores/bus/etc. Comparando varias arquitecturas vemos que "ia64 no tiene fallos" e "ia32 tiene unos pocos". Un buen ejemplo es el "fallo f00f", el cual es sólo chequeado si el núcleo ha sido compilado para menos de un 686 y corregido adecuadamente. (690)

19. rest_init se encarga de desbloquear el núcleo, limpiar los restos que han ido dejando las inicializaciones anteriores y de lanzar un nuevo hilo de ejecución con la función init para que termine el proceso de arranque del sistema. (697)

Se va a un bucle vacío, este es un hilo vacío con pid=0

5.3 INIT

La función *init,* la última función de inicialización del núcleo, prepara el sistema para el lanzamiento del proceso *init*. La función init es la encargada de inicializar los dispositivos, liberar la memoria que se utilizó para las inicializaciones previas, activa el planificador de tareas, establecer los canales de comunicación con la consola y lanzar el proceso init.

Intenta ejecutar un programa inicial especificado por el usuario o uno de los programas /sbin/init, /etc/init, o /bin/init. Si ninguno de estos programas existe, intenta arrancar un shell para que el superusuario pueda reparar el sistema. Si ésto tampoco es posible, el sistema se detiene.

Init configura el sistema y crea procesos siguiendo /etc/inittab

Un RunLevel es:

- Una configuración software de que servicios deben lanzarse.

- Un estado en el que el sistema se puede encontrar.

/etc/inittab define distintos niveles de ejecución.

El proceso init es el primer proceso de usuario que lanza el núcleo, y es el responsable de lanzar el resto de los procesos que se necesitan para inicializar por completo el sistema.

/bin/init Arranque del primer proceso del sistema.

Init es un proceso especial, es el primer proceso de usuario ejecutado por el Núcleo. Tiene PID=1. Es responsable de disparar el resto de procesos necesarios para que el sistema pueda empezar a usarse, por lo tanto, es el padre de todos los procesos de usuario.

En principio, el núcleo de Linux es independiente de la estructura del sistema de archivos que hay por encima, y desconoce completamente el nombre de los programas de usuario. El fichero ejecutable /sbin/init junto con el directorio raiz “/” son las únicas excepciones. El núcleo necesita conocer el nombre de al menos un ejecutable de usuario para poder ponerlo en ejecución.

Este proceso será el encargado de poner en marcha todo el resto de procesos que hacen faltan para mantener el sistema: Servidores de ficheros, demonios de contabilidad, demonios de servicios de red, procesos de login asociados a terminales, etc.

Existen dos sistemas de arranque en las diferentes versiones de unix/Linux actual: BSD y Sistem V, siendo esta última la que parece que este imponiéndose en la actualidad dada su

modularidad y versatilidad. Por dicho motivo y principalmente por ser la implementada en la distribución que se ha instalado en el laboratorio, es por lo que pasaremos a describirlo brevemente.

Como ya hemos dicho, el proceso INIT, es un proceso que el núcleo ejecuta cuando arranca el sistema. Se encarga de inicializar todos los procesos normales que se necesiten ejecutar en el momento de arrancar; incluyendo los terminales que le permiten acceder al sistema, los demonios y cualquier proceso que quiera ejecutar el usuario cuando su máquina arranque.

El proceso INIT del Sistem V se está convirtiendo rápidamente en el estándar del mundo de Linux para manejar la puesta en marcha del Software en el momento del inicio. Esto se debe a que es más fácil su uso, más potente y más flexible.

Los ficheros de configuración se localizan dentro de un subdirectorio de "/etc", concretamente se llama rc.d. En el podemos encontrar el fichero rc.sysinit y los siguientes directorios:

· init.d

· rc0.d

· rc1.d

· rc2.d

· rc3.d

· rc4.d

· rc5.d

· rc6.d

El directorio init.d contiene una colección de scripts. Básicamente se requiere un scripts por cada servicio que pueda necesitar inicializar en el momento de arranque.

Veamos cómo es la sucesión de eventos:

· El núcleo busca el proceso INIT en diferentes partes (/sbin/init; /etc/init; /bin/init) y ejecuta el primero que encuentre.

· El proceso init ejecuta /etc/rc.d/rc.sysinit.

· El proceso init ejecuta todos los scripts del nivel de ejecución por defecto. Dicho nivel se decide en /etc/inittab. El proceso init arrancará todos los servicios en el orden en que se encuentren.

· El proceso init ejecuta rc.local.

Una vez terminadas todas estas tareas, la máquina esta lista para interactuar con el usuario.

Cada vez que un hijo termina, init guarda el hecho y razón por la que murió en /var/run/utmp y /var/log/wtmp(probando que existen).

5.4 INIT/MAIN.C

```
1/*
   2 *  linux/init/main.c
   3 *
   4 *  Copyright (C) 1991, 1992  Linus Torvalds
   5 *
   6 *  GK 2/5/95  -  Changed to support mounting root fs via NFS
   7 *  Added initrd & change_root: Werner Almesberger & Hans Lermen, Feb
'96
   8 *  Moan early if gcc is old, avoiding bogus kernels - Paul Gortmaker,
May '96
   9 *  Simplified starting of init:  Michael A. Griffith <grif@acm.org>
  10 */
  11
  12#include <linux/types.h>
  13#include <linux/module.h>
  14#include <linux/proc_fs.h>
  15#include <linux/kernel.h>
  16#include <linux/syscalls.h>
  17#include <linux/string.h>
  18#include <linux/ctype.h>
  19#include <linux/delay.h>
  20#include <linux/utsname.h>
  21#include <linux/ioport.h>
  22#include <linux/init.h>
  23#include <linux/smp_lock.h>
  24#include <linux/initrd.h>
  25#include <linux/bootmem.h>
  26#include <linux/tty.h>
  27#include <linux/gfp.h>
  28#include <linux/percpu.h>
  29#include <linux/kmod.h>
  30#include <linux/vmalloc.h>
  31#include <linux/kernel_stat.h>
  32#include <linux/start_kernel.h>
  33#include <linux/security.h>
  34#include <linux/smp.h>
  35#include <linux/workqueue.h>
  36#include <linux/profile.h>
  37#include <linux/rcupdate.h>
  38#include <linux/moduleparam.h>
  39#include <linux/kallsyms.h>
  40#include <linux/writeback.h>
  41#include <linux/cpu.h>
  42#include <linux/cpuset.h>
  43#include <linux/cgroup.h>
  44#include <linux/efi.h>
  45#include <linux/tick.h>
  46#include <linux/interrupt.h>
```

```
  47#include <linux/taskstats_kern.h>
  48#include <linux/delayacct.h>
  49#include <linux/unistd.h>
  50#include <linux/rmap.h>
  51#include <linux/mempolicy.h>
  52#include <linux/key.h>
  53#include <linux/unwind.h>
  54#include <linux/buffer_head.h>
  55#include <linux/page_cgroup.h>
  56#include <linux/debug_locks.h>
  57#include <linux/debugobjects.h>
  58#include <linux/lockdep.h>
  59#include <linux/pid_namespace.h>
  60#include <linux/device.h>
  61#include <linux/kthread.h>
  62#include <linux/sched.h>
  63#include <linux/signal.h>
  64#include <linux/idr.h>
  65#include <linux/ftrace.h>
  66
  67#include <asm/io.h>
  68#include <asm/bugs.h>
  69#include <asm/setup.h>
  70#include <asm/sections.h>
  71#include <asm/cacheflush.h>
  72
  73#ifdef CONFIG_X86_LOCAL_APIC
  74#include <asm/smp.h>
  75#endif
  76
  77/*
  78 * This is one of the first .c files built. Error out early if we have
compiler
  79 * trouble.
  80 */
  81
  82#if __GNUC__ == 4 && __GNUC_MINOR__ == 1 && __GNUC_PATCHLEVEL__ == 0
   83#warning gcc-4.1.0 is known to miscompile the kernel.   A different
compiler version is recommended.
  84#endif
  85
  86static int kernel_init(void *);
  87
  88extern void init_IRQ(void);
  89extern void fork_init(unsigned long);
  90extern void mca_init(void);
  91extern void sbus_init(void);
  92extern void prio_tree_init(void);
  93extern void radix_tree_init(void);
  94extern void free_initmem(void);
  95#ifdef  CONFIG_ACPI
  96extern void acpi_early_init(void);
  97#else
  98static inline void acpi_early_init(void) { }
  99#endif
 100#ifndef CONFIG_DEBUG_RODATA
 101static inline void mark_rodata_ro(void) { }
 102#endif
 103
 104#ifdef CONFIG_TC
 105extern void tc_init(void);
```

```
106#endif
107
108enum system_states system_state;
109EXPORT_SYMBOL(system_state);
110
111/*
112 * Boot command-line arguments
113 */
114#define MAX_INIT_ARGS CONFIG_INIT_ENV_ARG_LIMIT
115#define MAX_INIT_ENVS CONFIG_INIT_ENV_ARG_LIMIT
116
117extern void time_init(void);
118/* Default late time init is NULL. archs can override this later. */
119void (*late_time_init)(void);
120extern void softirq_init(void);
121
122/* Untouched command line saved by arch-specific code. */
123char __initdata boot_command_line[COMMAND_LINE_SIZE];
124/* Untouched saved command line (eg. for /proc) */
125char *saved_command_line;
126/* Command line for parameter parsing */
127static char *static_command_line;
128
129static char *execute_command;
130static char *ramdisk_execute_command;
131
132#ifdef CONFIG_SMP
133/* Setup configured maximum number of CPUs to activate */
134unsigned int __initdata setup_max_cpus = NR_CPUS;
135
136/*
137 * Setup routine for controlling SMP activation
138 *
139 * Command-line option of "nosmp" or "maxcpus=0" will disable SMP
140 * activation entirely (the MPS table probe still happens, though).
141 *
142 * Command-line option of "maxcpus=<NUM>", where <NUM> is an integer
143 * greater than 0, limits the maximum number of CPUs activated in
144 * SMP mode to <NUM>.
145 */
146#ifndef CONFIG_X86_IO_APIC
147static inline void disable_ioapic_setup(void) {};
148#endif
149
150static int __init nosmp(char *str)
151{
152        setup_max_cpus = 0;
153        disable_ioapic_setup();
154        return 0;
155}
156
157early_param("nosmp", nosmp);
158
159static int __init maxcpus(char *str)
160{
161        get_option(&str, &setup_max_cpus);
162        if (setup_max_cpus == 0)
163                disable_ioapic_setup();
164
165        return 0;
166}
```

```
 167
 168early_param("maxcpus", maxcpus);
 169#else
 170#define setup_max_cpus NR_CPUS
 171#endif
 172
 173/*
  174  *  If  set,  this  is  an  indication  to  the  drivers  that  reset  the
underlying
  175  * device before going ahead with the initialization otherwise driver
might
 176  * rely on the BIOS and skip the reset operation.
 177  *
 178  * This is useful if kernel is booting in an unreliable environment.
  179  * For ex. kdump situaiton where previous kernel has crashed, BIOS has
been
 180  * skipped and devices will be in unknown state.
 181  */
 182unsigned int reset_devices;
 183EXPORT_SYMBOL(reset_devices);
 184
 185static int __init set_reset_devices(char *str)
 186{
 187        reset_devices = 1;
 188        return 1;
 189}
 190
 191__setup("reset_devices", set_reset_devices);
 192
 193static char * argv_init[MAX_INIT_ARGS+2] = { "init", NULL, };
 194char * envp_init[MAX_INIT_ENVS+2] = { "HOME=/", "TERM=linux", NULL, };
 195static const char *panic_later, *panic_param;
 196
 197extern struct obs_kernel_param __setup_start[], __setup_end[];
 198
 199static int __init obsolete_checksetup(char *line)
 200{
 201        struct obs_kernel_param *p;
 202        int had_early_param = 0;
 203
 204        p = __setup_start;
 205        do {
 206                int n = strlen(p->str);
 207                if (!strncmp(line, p->str, n)) {
 208                        if (p->early) {
 209                                /* Already done in parse_early_param?
 210                                 * (Needs exact match on param part).
 211                                 * Keep iterating, as we can have early
 212                                 * params and __setups of same names 8(
*/
 213                                if (line[n] == '\0' || line[n] == '=')
 214                                        had_early_param = 1;
 215                        } else if (!p->setup_func) {
 216                                   printk(KERN_WARNING "Parameter %s is
obsolete,"
 217                                          " ignored\n", p->str);
 218                                return 1;
 219                        } else if (p->setup_func(line + n))
 220                                return 1;
 221                }
 222                p++;
```

```
 223        } while (p < __setup_end);
 224
 225        return had_early_param;
 226}
 227
 228/*
 229 * This should be approx 2 Bo*oMips to start (note initial shift), and
will
 230 * still work even if initially too large, it will just take slightly
longer
 231 */
 232unsigned long loops_per_jiffy = (1<<12);
 233
 234EXPORT_SYMBOL(loops_per_jiffy);
 235
 236static int __init debug_kernel(char *str)
 237{
 238        console_loglevel = 10;
 239        return 0;
 240}
 241
 242static int __init quiet_kernel(char *str)
 243{
 244        console_loglevel = 4;
 245        return 0;
 246}
 247
 248early_param("debug", debug_kernel);
 249early_param("quiet", quiet_kernel);
 250
 251static int __init loglevel(char *str)
 252{
 253        get_option(&str, &console_loglevel);
 254        return 0;
 255}
 256
 257early_param("loglevel", loglevel);
 258
 259/*
 260 * Unknown boot options get handed to init, unless they look like
 261 * failed parameters
 262 */
 263static int __init unknown_bootoption(char *param, char *val)
 264{
 265          /* Change NUL term back to "=", to make "param" the whole
string. */
 266        if (val) {
 267                /* param=val or param="val"? */
 268                if (val == param+strlen(param)+1)
 269                        val[-1] = '=';
 270                else if (val == param+strlen(param)+2) {
 271                        val[-2] = '=';
 272                        memmove(val-1, val, strlen(val)+1);
 273                        val--;
 274                } else
 275                        BUG();
 276        }
 277
 278        /* Handle obsolete-style parameters */
 279        if (obsolete_checksetup(param))
 280                return 0;
```

```
 281
 282        /*
 283         * Preemptive maintenance for "why didn't my misspelled command
 284         * line work?"
 285         */
 286        if (strchr(param, '.') && (!val || strchr(param, '.') < val)) {
 287                printk(KERN_ERR "Unknown boot option `%s': ignoring\n",
param);
 288                return 0;
 289        }
 290
 291        if (panic_later)
 292                return 0;
 293
 294        if (val) {
 295                /* Environment option */
 296                unsigned int i;
 297                for (i = 0; envp_init[i]; i++) {
 298                        if (i == MAX_INIT_ENVS) {
 299                                panic_later = "Too many boot env vars
at `%s'";
 300                                panic_param = param;
 301                        }
 302                        if (!strncmp(param, envp_init[i], val - param))
 303                                break;
 304                }
 305                envp_init[i] = param;
 306        } else {
 307                /* Command line option */
 308                unsigned int i;
 309                for (i = 0; argv_init[i]; i++) {
 310                        if (i == MAX_INIT_ARGS) {
 311                                panic_later = "Too many boot init vars
at `%s'";
 312                                panic_param = param;
 313                        }
 314                }
 315                argv_init[i] = param;
 316        }
 317        return 0;
 318}
 319
 320#ifdef CONFIG_DEBUG_PAGEALLOC
 321int __read_mostly debug_pagealloc_enabled = 0;
 322#endif
 323
 324static int __init init_setup(char *str)
 325{
 326        unsigned int i;
 327
 328        execute_command = str;
 329        /*
 330         * In case LILO is going to boot us with default command line,
 331         * it prepends "auto" before the whole cmdline which makes
 332         * the shell think it should execute a script with such name.
 333         * So we ignore all arguments entered _before_ init=... [MJ]
 334         */
 335        for (i = 1; i < MAX_INIT_ARGS; i++)
 336                argv_init[i] = NULL;
 337        return 1;
 338}
```

```
339__setup("init=", init_setup);
340
341static int __init rdinit_setup(char *str)
342{
343        unsigned int i;
344
345        ramdisk_execute_command = str;
346        /* See "auto" comment in init_setup */
347        for (i = 1; i < MAX_INIT_ARGS; i++)
348                argv_init[i] = NULL;
349        return 1;
350}
351__setup("rdinit=", rdinit_setup);
352
353#ifndef CONFIG_SMP
354
355#ifdef CONFIG_X86_LOCAL_APIC
356static void __init smp_init(void)
357{
358        APIC_init_uniprocessor();
359}
360#else
361#define smp_init()      do { } while (0)
362#endif
363
364static inline void setup_per_cpu_areas(void) { }
365static inline void setup_nr_cpu_ids(void) { }
366static inline void smp_prepare_cpus(unsigned int maxcpus) { }
367
368#else
369
370#if NR_CPUS > BITS_PER_LONG
371cpumask_t cpu_mask_all __read_mostly = CPU_MASK_ALL;
372EXPORT_SYMBOL(cpu_mask_all);
373#endif
374
375/* Setup number of possible processor ids */
376int nr_cpu_ids __read_mostly = NR_CPUS;
377EXPORT_SYMBOL(nr_cpu_ids);
378
 379/* An arch may set nr_cpu_ids earlier if needed, so this would be
redundant */
380static void __init setup_nr_cpu_ids(void)
381{
382        int cpu, highest_cpu = 0;
383
384        for_each_possible_cpu(cpu)
385                highest_cpu = cpu;
386
387        nr_cpu_ids = highest_cpu + 1;
388}
389
390#ifndef CONFIG_HAVE_SETUP_PER_CPU_AREA
391unsigned long __per_cpu_offset[NR_CPUS] __read_mostly;
392
393EXPORT_SYMBOL(__per_cpu_offset);
394
395static void __init setup_per_cpu_areas(void)
396{
397        unsigned long size, i;
398        char *ptr;
```

```
 399        unsigned long nr_possible_cpus = num_possible_cpus();
 400
 401        /* Copy section for each CPU (we discard the original) */
 402        size = ALIGN(PERCPU_ENOUGH_ROOM, PAGE_SIZE);
 403        ptr = alloc_bootmem_pages(size * nr_possible_cpus);
 404
 405        for_each_possible_cpu(i) {
 406                __per_cpu_offset[i] = ptr - __per_cpu_start;
 407                        memcpy(ptr, __per_cpu_start, __per_cpu_end -
__per_cpu_start);
 408                ptr += size;
 409        }
 410}
 411#endif /* CONFIG_HAVE_SETUP_PER_CPU_AREA */
 412
 413/* Called by boot processor to activate the rest. */
 414static void __init smp_init(void)
 415{
 416        unsigned int cpu;
 417
 418        /*
 419         * Set up the current CPU as possible to migrate to.
 420         * The other ones will be done by cpu_up/cpu_down()
 421         */
 422        cpu = smp_processor_id();
 423        cpu_set(cpu, cpu_active_map);
 424
 425        /* FIXME: This should be done in userspace --RR */
 426        for_each_present_cpu(cpu) {
 427                if (num_online_cpus() >= setup_max_cpus)
 428                        break;
 429                if (!cpu_online(cpu))
 430                        cpu_up(cpu);
 431        }
 432
 433        /* Any cleanup work */
  434                      printk(KERN_INFO   "Brought   up   %ld   CPUs\n",
(long)num_online_cpus());
 435        smp_cpus_done(setup_max_cpus);
 436}
 437
 438#endif
 439
 440/*
 441 * We need to store the untouched command line for future reference.
 442 * We also need to store the touched command line since the parameter
 443 * parsing is performed in place, and we should allow a component to
 444 * store reference of name/value for future reference.
 445 */
 446static void __init setup_command_line(char *command_line)
 447{
 448          saved_command_line = alloc_bootmem(strlen (boot_command_line)
+1);
 449        static_command_line = alloc_bootmem(strlen (command_line)+1);
 450        strcpy (saved_command_line, boot_command_line);
 451        strcpy (static_command_line, command_line);
 452}
 453
 454/*
 455 * We need to finalize in a non-__init function or else race conditions
 456 * between the root thread and the init thread may cause start_kernel
```

```
to
 457 * be reaped by free_initmem before the root thread has proceeded to
 458 * cpu_idle.
 459 *
 460 * gcc-3.4 accidentally inlines this function, so use noinline.
 461 */
 462
 463static void noinline __init_refok rest_init(void)
 464        __releases(kernel_lock)
 465{
 466        int pid;
 467
 468        kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
 469        numa_default_policy();
 470        pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
 471        kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
 472        unlock_kernel();
 473
 474        /*
 475         * The boot idle thread must execute schedule()
 476         * at least once to get things moving:
 477         */
 478        init_idle_bootup_task(current);
 479        preempt_enable_no_resched();
 480        schedule();
 481        preempt_disable();
 482
 483        /* Call into cpu_idle with preempt disabled */
 484        cpu_idle();
 485}
 486
 487/* Check for early params. */
 488static int __init do_early_param(char *param, char *val)
 489{
 490        struct obs_kernel_param *p;
 491
 492        for (p = __setup_start; p < __setup_end; p++) {
 493                if ((p->early && strcmp(param, p->str) == 0) ||
 494                    (strcmp(param, "console") == 0 &&
 495                     strcmp(p->str, "earlycon") == 0)
 496                ) {
 497                        if (p->setup_func(val) != 0)
 498                                printk(KERN_WARNING
 499                                       "Malformed early option '%s'\n",
param);
 500                }
 501        }
 502        /* We accept everything at this stage. */
 503        return 0;
 504}
 505
 506/* Arch code calls this early on, or if not, just before other parsing.
*/
 507void __init parse_early_param(void)
 508{
 509        static __initdata int done = 0;
 510        static __initdata char tmp_cmdline[COMMAND_LINE_SIZE];
 511
 512        if (done)
 513                return;
 514
```

```
 515        /* All fall through to do_early_param. */
 516        strlcpy(tmp_cmdline, boot_command_line, COMMAND_LINE_SIZE);
  517                parse_args("early  options",  tmp_cmdline,  NULL,  0,
do_early_param);
 518        done = 1;
 519}
 520
 521/*
 522 *      Activate the first processor.
 523 */
 524
 525static void __init boot_cpu_init(void)
 526{
 527        int cpu = smp_processor_id();
 528          /* Mark the boot cpu "present", "online" etc for SMP and UP
case */
 529        cpu_set(cpu, cpu_online_map);
 530        cpu_set(cpu, cpu_present_map);
 531        cpu_set(cpu, cpu_possible_map);
 532}
 533
 534void __init __weak smp_setup_processor_id(void)
 535{
 536}
 537
 538void __init __weak thread_info_cache_init(void)
 539{
 540}
 541
```

Comienzo de la function start_kernel

```
 542asmlinkage void __init start_kernel(void)
 543{
 544        char * command_line;
 545        extern struct kernel_param __start___param[], __stop___param[];
 546
 547        smp_setup_processor_id();
 548
 549        /*
 550         * Need to run as early as possible, to initialize the
 551         * lockdep hash:
 552         */
 553        unwind_init();
 554        lockdep_init();
 555        debug_objects_early_init();
 556        cgroup_init_early();
 557
 558        local_irq_disable();
 559        early_boot_irqs_off();
 560        early_init_irq_lock_class();
 561
 562/*
 563 * Interrupts are still disabled. Do necessary setups, then
 564 * enable them
 565 */
```

Las interrupciones están deshabilitadas. Se hacen las configuraciones necesarias, y luego se activan nuevamente.

Se pone un cerrojo sobre todo el núcleo que no se libera hasta que se haya creado el

proceso init. Impide que otras partes del núcleo interfieran en el codigo siguiente, para evitar interferencias de otros procesadores en máquinas multiprocesador.

```
566         lock_kernel();
 567         tick_init();
 568         boot_cpu_init();
 569         page_address_init();
 570         printk(KERN_NOTICE);
```

Imprime la versión del SO, así como la información del compilador, contenido del fichero linux/init/version.c.

```
571         printk(linux_banner);
```

Inicializaciones hardware y contabiliza la memoria RAM disponible, utilizando la información que setup.S había obtenido de la BIOS. Invoca también a paging_init(), que inicializa el esquema de paginación, mapeando toda la memoria física a partir de PAGE_OFFSET, perteneciente al espacio de direcciones lógicas del núcleo.

```
 572         setup_arch(&command_line);
 573         mm_init_owner(&init_mm, &init_task);
 574         setup_command_line(command_line);
 575         unwind_setup();
 576         setup_per_cpu_areas();
```

Marca la CPU inicial como "online", así puede llamar al manejador de consola con printk() y puede acceder a la memoria asignada a la cpu. SMP: Symmetric Multiprocessing

```
577         setup_nr_cpu_ids();
578         smp_prepare_boot_cpu(); /* arch-specific boot-cpu hooks */
```

Iniciliza las entradas de la tabla de procesos. Todas disponibles salvo la tarea idle o init_task, considerada el proceso 0 del sistema con PID=0. La tarea idle no es lo mismo que el primer proceso de usuario (init).

```
 579
 580         /*
 581          * Set up the scheduler prior starting any interrupts (such as
the
 582          * timer interrupt). Full topology setup happens at smp_init()
 583          * time - but meanwhile we still have a functioning scheduler.
 584          */
 585         sched_init();
 586         /*
 587          * Disable preemption - early bootup scheduling is extremely
 588          * fragile until we cpu_idle() for the first time.
 589          */
 590         preempt_disable();
 591         build_all_zonelists();
```

Se inicializa la Unidad de Manejo de Memoria MMU de la CPU.

```
 592      page_alloc_init();
 593     printk(KERN_NOTICE "Kernel command line: %s\n", boot_command_line);
 594     parse_early_param();
```

Se llama a parse_args, que lleva a cabo una verificación de la correctitud de los parámetros pasados en el arranque. También inicializa las variables de entorno y los parámetros que se pasan al main.

```
595                     parse_args("Booting   kernel",   static_command_line,
__start___param,
 596                    __stop___param - __start___param,
 597                    &unknown_bootoption);
 598         if (!irqs_disabled()) {
 599                    printk(KERN_WARNING "start_kernel(): bug: interrupts
were "
 600                                       "enabled *very* early, fixing it\n");
 601                 local_irq_disable();
 602         }
```

Como este núcleo soporta procesadores de 64 bits, nos podemos encontrar la tabla de excepciones desordenada, por lo tanto se llama a sort_main_extable();

```
603         sort_main_extable();
```

Inicializa la IDT instalando los vectores de intrrupcion de las excepciones producidas por el procesador, asi como la interrupción utilizada para las llamadas al sistema SYSCALL_VECTOR. También invoca a cpu_init para inicializar el estado de la CPU (GDT, IDT, LDT, TSS, etc).

```
604         trap_init();
```

RCU Read-copy update, es un sistema para establecer bloqueos interprocesadores que consigue un mejor rendimiento del sistema.

```
605         rcu_init();
```

Inicializa todos los vectores de interrupción de las interrupciones externas (periféricos), programa el chip de reloj para que interrumpa 100 veces por segundo.

```
606         init_IRQ();
```

Crea la tabla hash para buscar procesos en la tabla de procesos task_struct

```
 607         pidhash_init();
 608         init_timers();
 609         hrtimers_init();
```

Inicializa las interrupciones software

```
610         softirq_init();
 611         timekeeping_init();
```

Lee la hora (número de segundos) actual y la guarda en xtime. Esta variable se mantendrá actualizando los ticks de reloj. Instala el manejador de interrupciones de la

interrupción de reloj.

```
612         time_init();
 613         sched_clock_init();
 614         profile_init();
 615         if (!irqs_disabled())
 616         printk("start_kernel(): bug: interrupts were enabled early\n");
 617         early_boot_irqs_on();
```

Activa las interrupciones del procesador principal

```
618         local_irq_enable();
```

Hace una primera inicialización de la consola para poder mostrar mensajes del núcleo en el inicio, antes que el sistema completo de consola virtual esté inicializado

```
619
 620         /*
 621          * HACK ALERT! This is early. We're enabling the console before
 622          * we've done PCI setups etc, and console_init() must be aware
of
 623           * this. But we do want output early, in case something goes
wrong.
 624          */
 625         console_init();
 626         if (panic_later)
 627                 panic(panic_later, panic_param);
 628
 629         lockdep_info();
 630
 631         /*
 632          * Need to run this when irqs are enabled, because it wants
 633          * to self-test [hard/soft]-irqs on/off lock inversion bugs
 634          * too:
 635          */
 636         locking_selftest();
 637
 638#ifdef CONFIG_BLK_DEV_INITRD
 639         if (initrd_start && !initrd_below_start_ok &&
  640                     page_to_pfn(virt_to_page((void  *)initrd_start))  <
min_low_pfn) {
 641                     printk(KERN_CRIT "initrd overwritten (0x%08lx < 0x
%08lx) - "
 642                     "disabling it.\n",
 643                     page_to_pfn(virt_to_page((void *)initrd_start)),
 644                     min_low_pfn);
 645                 initrd_start = 0;
 646         }
 647#endif
 648         vmalloc_init();
 649         vfs_caches_init_early();
 650         cpuset_init_early();
651         page_cgroup_init();
```

Se inicia el manejador de memoria

```
652         mem_init();
```

Inicializa el subsistema de memoria dinámica del propio núcleo

```
 653         enable_debug_pagealloc();
 654         cpu_hotplug_init();
 655         kmem_cache_init();
 656         debug_objects_mem_init();
 657         idr_init_cache();
 658         setup_per_cpu_pageset();
 659         numa_policy_init();
 660         if (late_time_init)
 661                 late_time_init();
```

Calculamos el BogoMIPS contando el número de ticks de reloj desde el inicio, que es el número de veces por segundo que la CPU puede ejecutar un determinado bucle de retraso (lo utilizan algunos drivers de dispositivo como tiempo de espera).

```
662         calibrate_delay();
663         pidmap_init();
```

Inicializa la tabla de páginas de la cache

```
664         pgtable_cache_init();
 665         prio_tree_init();
```

VMA: Virtual Memory Area

```
666         anon_vma_init();
```

EFI, tecnología de firmware que se aplica sobre arquitecturas IA-64 o x86-64.

```
667#ifdef CONFIG_X86
668         if (efi_enabled)
669                 efi_enter_virtual_mode();
670#endif
```

Inicializa las variables que controlan el número máximo de hilos permitidos

```
671         thread_info_cache_init();
672         fork_init(num_physpages);
```

Inicializa buffer de memoria para el sistema de ficheros proc

```
673         proc_caches_init();
674         buffer_init();
```

Crea e inicializa las estructuras con los nombres de los dispositivos

```
675         key_init();
```

Se inicializan medidas de seguridad

```
676         security_init();
```

Inicializa memoria cache para el sistema virtual de ficheros VFS

```
677         vfs_caches_init(num_physpages);
```

La cache está organizada en un tipo de árbol especial llamada radix-tree

```
678         radix_tree_init();
```

Se inicializa el sistema de señales

```
679         signals_init();
```

El sistema de fichero root puede necesitar reescritura de página

```
680         /* rootfs populating might need page-writeback */
 681         page_writeback_init();
```

El sistema de fichero root puede necesitar reescritura de página

```
 682#ifdef CONFIG_PROC_FS
 683         proc_root_init();
 684#endif
 685         cgroup_init();
 686         cpuset_init();
 687         taskstats_init_early();
 688         delayacct_init();
```

La function check_bugs se encarga de buscar los fallos del procesador, y establecer políticas de software para evitar que estos fallos se produzcan

```
690         check_bugs();
691
692         acpi_early_init(); /* before LAPIC and SMP init */
693
694         ftrace_init();
```

rest_init se encarga de desbloquear el núcleo, limpiar los restos que han ido dejando las inicializaciones anteriores y de lanzar un nuevo hilo de ejecución utilizando la función kernel_thread el cual ejecutará la función init() para que termine el proceso de arranque del sistema.

```
 695
 696         /* Do the rest non-__init'ed, we're now alive */
 697         rest_init();
 698}
 699
 700static int initcall_debug;
 701core_param(initcall_debug, initcall_debug, bool, 0644);
 702
```

```
 703int do_one_initcall(initcall_t fn)
 704{
 705        int count = preempt_count();
 706        ktime_t delta;
 707        char msgbuf[64];
 708        struct boot_trace it;
 709
 710        if (initcall_debug) {
 711                it.caller = task_pid_nr(current);
 712                printk("calling  %pF @ %i\n", fn, it.caller);
 713                it.calltime = ktime_get();
 714        }
 715
 716        it.result = fn();
 717
 718        if (initcall_debug) {
 719                it.rettime = ktime_get();
 720                delta = ktime_sub(it.rettime, it.calltime);
 721                it.duration = (unsigned long long) delta.tv64 >> 10;
 722                   printk("initcall %pF returned %d after %Ld usecs\n",
fn,
 723                         it.result, it.duration);
 724                trace_boot(&it, fn);
 725        }
 726
 727        msgbuf[0] = 0;
 728
 729        if (it.result && it.result != -ENODEV && initcall_debug)
 730                sprintf(msgbuf, "error code %d ", it.result);
 731
 732        if (preempt_count() != count) {
  733                              strlcat(msgbuf,  "preemption  imbalance  ",
sizeof(msgbuf));
 734                preempt_count() = count;
 735        }
 736        if (irqs_disabled()) {
  737                               strlcat(msgbuf,  "disabled  interrupts  ",
sizeof(msgbuf));
 738                local_irq_enable();
 739        }
 740        if (msgbuf[0]) {
 741                printk("initcall %pF returned with %s\n", fn, msgbuf);
 742        }
 743
 744        return it.result;
 745}
 746
 747
     748extern     initcall_t     __initcall_start[],     __initcall_end[],
__early_initcall_end[];
 749
 750static void __init do_initcalls(void)
 751{
 752        initcall_t *call;
 753
 754         for (call = __early_initcall_end; call < __initcall_end; call+
+)
 755                do_one_initcall(*call);
 756
  757           /* Make sure there is no pending stuff from the initcall
sequence */
```

```
 758        flush_scheduled_work();
 759}
 760
 761/*
 762 * Ok, the machine is now initialized. None of the devices
 763 * have been touched yet, but the CPU subsystem is up and
 764 * running, and memory and process management works.
 765 *
 766 * Now we can finally start doing some real work..
 767 */
 768static void __init do_basic_setup(void)
 769{
 770        rcu_init_sched(); /* needed by module_init stage. */
 771        init_workqueues();
 772        usermodehelper_init();
 773        driver_init();
 774        init_irq_proc();
 775        do_initcalls();
 776}
 777
 778static void __init do_pre_smp_initcalls(void)
 779{
 780        initcall_t *call;
 781
 782        for (call = __initcall_start; call < __early_initcall_end; call
++)
 783                do_one_initcall(*call);
 784}
 785
 786static void run_init_process(char *init_filename)
 787{
 788        argv_init[0] = init_filename;
 789        kernel_execve(init_filename, argv_init, envp_init);
 790}
 791
 792/* This is a non __init function. Force it to be noinline otherwise gcc
 793 * makes it inline to init() and it becomes part of init.text section
 794 */
 795static int noinline init_post(void)
 796{
```

Se liberan todas las páginas donde estaban las variables y funcione de inicialización que ya no se necesitan.

```
797        free_initmem();
```

Se libera la exclusión mutua.

```
798        unlock_kernel();
 799        mark_rodata_ro();
```

El estado del sistema se cambia a SYSTEM_RUNNING (sistema funcionando)

```
800        system_state = SYSTEM_RUNNING;
 801        numa_default_policy();
```

Se abren los ficheros estándar de entrada, salida y error. Se abre la consola 0 entrada estándar.

```
803     if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
804             printk(KERN_WARNING "Warning: unable to open an initial
console.\n");
```

Asigna el 1 para la salida estándar y el 2 para el estándar error duplicando el canal de consola

```
806         (void) sys_dup(0);
 807         (void) sys_dup(0);
 808
 809         current->signal->flags |= SIGNAL_UNKILLABLE;
 810
 811         if (ramdisk_execute_command) {
 812                 run_init_process(ramdisk_execute_command);
 813                 printk(KERN_WARNING "Failed to execute %s\n",
 814                                 ramdisk_execute_command);
 815         }
 816
```

Si tiene valor 'execute_command' ejecutamos el comando que tenga almacenado.

```
817         /*
 818          * We try each of these until one succeeds.
 819          *
 820          * The Bourne shell can be used instead of init if we are
 821          * trying to recover a really broken machine.
 822          */
 823         if (execute_command) {
 824                 run_init_process(execute_command);
 825                  printk(KERN_WARNING "Failed to execute %s.  Attempting
"
 826                                                         "defaults...\n",
execute_command);
 827         }
```

Se busca el fichero init en varias localizaciones y se lanza desde la primera que se encuentre.

```
828         run_init_process("/sbin/init");
 829         run_init_process("/etc/init");
 830         run_init_process("/bin/init");
```

Si no se encuentra se lanza un shell para que el root arregle el fallo y reinicie la máquina.

```
831         run_init_process("/bin/sh");
```

Le mostramos un mensaje indicando que no se ha encuentrado el Init.

```
833         panic("No init found.  Try passing init= option to kernel.");
 834}
```

Init

La función init() es la última función de inicialización del núcleo y prepara al sistema para el lanzamiento del proceso init. Algunas de sus funciones son:

· Inicializar los dispositivos.

· Liberar la memoria que se utilizó para las inicializaciones previas.

· Activar el planificador de tareas.

· Establecer los canales de comunicación con la consola.

· Lanzar el proceso init.

```
836static int __init kernel_init(void * unused)
 837{
```

Se llama a la función lock_kernel para obtener la exclusión mutua y evitar que otros procesadores interfieran

```
838         lock_kernel();
```

Inicializa procesadores disponibles para que comiencen a trabajar

```
839         /*
 840          * init can run on any cpu.
 841          */
 842         set_cpus_allowed_ptr(current, CPU_MASK_ALL_PTR);
 843         /*
 844          * Tell the world that we're going to be the grim
 845          * reaper of innocent orphaned children.
 846          *
 847          * We don't want people to have to make incorrect
 848          * assumptions about where in the task array this
 849          * can be found.
 850          */
 851         init_pid_ns.child_reaper = current;
 852
 853         cad_pid = task_pid(current);
 854
 855         smp_prepare_cpus(setup_max_cpus);
```

Establece los mecanismos para la migración de procesos entre distintos procesadores

```
857         do_pre_smp_initcalls();
858         start_boot_trace();
 859
```

Inicializa el hardware relacionado con el APIC (Advanced Programmable Interrupt Controller)

```
860         smp_init();
```

Inicializa el planificador relacionado con el APIC (Advanced Programmable Interrupt Controller)

```
861         sched_init_smp();
 862
 863          cpuset_init_smp();
 864
```

Realiza el resto de las inicilalizaciones del sistema (el bus PCI, el bus MCA, el sistema ISA, etc). Posteriormente la función do_init_calls llama a todas las funciones de inicialización que se hayan declarado. Estas funciones son las que lanzan los diferentes hilos de ejecución del núcleo.

```
865         do_basic_setup();
 866
 867          /*
 868            * check if there is an early userspace init.  If yes, let it
do all
 869           * the work
 870           */
 871
 872          if (!ramdisk_execute_command)
 873                  ramdisk_execute_command = "/init";
```

Si el usuario le especifica un proceso init lo ejecuta. El prepare_namespace() monta el sistema de ficheros raiz y el de dispositivos

```
874
 875      if (sys_access((const char __user *) ramdisk_execute_command, 0) != 0) {
 876              ramdisk_execute_command = NULL;
 877              prepare_namespace();
 878      }
 879
 880      /*
 881       * Ok, we have completed the initial bootup, and
 882       * we're essentially up and running. Get rid of the
```

```
883         * initmem segments and start the user-mode stuff..
884         */
885        stop_boot_trace();
886        init_post();
887        return 0;
888}
 889
```

5.5 DOCUMENTACIÓN

- Linux cross reference http://lxr.linux.no/

- [MAX99] - Maxwell, Scott; Linux Core Kernel commentary. Ed. Scottsdale, Arizona: The Coriolis group, 1999.