LECCIÓN 7: CREACION DE PROCESOS. FORK.

7.1 Introducción: procesos, hilos, tareas.

Antes de hablar de la llamada al sistema fork propiamente dicha, conviene hablar sobre procesos e hilos. Dos conceptos muy parecidos y relacionados, pero con un conjunto de sutiles diferencias.

Uno de los principales motivos de la existencia de la informática es imitar el comportamiento de la mente humana. En un comienzo surgieron los algoritmos, que no son más que una secuencia de pasos para conseguir un objetivo, a partir de los cuales surgió el pensamiento de “por qué no hacer varias cosas a la vez” y es precisamente de esta inquietud de donde surgen los hilos o threads.

Si queremos que nuestro programa empiece a ejecutar varias cosas "a la vez", tenemos dos opciones. Por una parte podemos crear un nuevo proceso y por otra, podemos crear un nuevo hilo de ejecución (un thread). En realidad nuestro ordenador, salvo que tenga varias CPU’s, no ejecutará varias tareas a la vez esto se refiere a que el sistema operativo, es este caso Linux, irá ejecutando los threads según la política del mismo, siendo lo mas usual mediante rodajas de tiempo muy rápidas que dan la sensación de simultaneidad.

Procesos

Un proceso es un concepto manejado por el sistema operativo que consiste en el conjunto formado por:

- Las instrucciones de un programa destinadas a ser ejecutadas por el microprocesador.
- Su estado de ejecución en un momento dado, esto es, los valores de los registros de la CPU para dicho programa.
- Su memoria de trabajo, es decir, la memoria que ha reservado y sus contenidos.
- Otra información que permite al sistema operativo su planificación.

En un sistema Linux, que como ya sabemos es multitarea (sistema operativo multihilo), se pueden estar ejecutando distintas acciones a la par, y cada acción es un proceso que consta de uno o más hilos, memoria de trabajo compartida por todos los hilos e información de planificación. Cada hilo consta de instrucciones y estado de ejecución.

Cuando ejecutamos un comando en el shell, sus instrucciones se copian en algún sitio de la memoria RAM del sistema para ser ejecutadas. Cuando las

instrucciones ya cumplieron su cometido, el programa es borrado de la memoria del sistema, dejándola libre para que más programas se puedan ejecutar a la vez. Por tanto cada uno de estos programas son los procesos.

Los procesos son creados y destruidos por el sistema operativo, pero lo hace a petición de otros procesos. El mecanismo por el cual un proceso crea otro proceso se denomina bifurcación (fork). Los nuevos procesos son independientes y no comparten memoria (es decir, información) con el proceso que los ha creado.

En definitiva, es posible crear tanto hilos como procesos. La diferencia estriba en que un proceso solamente puede crear hilos para sí mismo y en que dichos hilos comparten toda la memoria reservada para el proceso.

Hilos

Los hilos son similares a los procesos ya que ambos representan una secuencia simple de instrucciones ejecutada en paralelo con otras secuencias. Los hilos son una forma de dividir un programa en dos o más tareas que corren simultáneamente, compitiendo, en algunos casos, por la CPU.

La diferencia más significativa entre los procesos y los hilos, es que los primeros son típicamente independientes, llevan bastante información de estados, e interactúan sólo a través de mecanismos de comunicación dados por el sistema. Por otra parte, los hilos generalmente comparten la memoria, es decir, acceden a las mismas variables globales o dinámicas, por lo que no necesitan costosos mecanismos de comunicación para sincronizarse. Por ejemplo un hilo podría encarguese de la interfaz gráfica (iconos, botones, ventanas), mientras que otro hace una larga operación internamente. De esta manera el programa responde más ágilmente a la interacción con el usuario.

En sistemas operativos que proveen facilidades para los hilos, es más rápido cambiar de un hilo a otro dentro del mismo proceso, que cambiar de un proceso a otro.

Es posible que los hilos requieran de operaciones atómicas para impedir que los datos comunes sean cambiados o leídos mientras estén siendo modificados. El descuido de esto puede generar estancamiento.

La tabla 1 resume algunas diferencias entre procesos e hilos, y la figura 1 muestra una representación de los conceptos proceso e hilo. Nótese cómo en la figura 1 se puede apreciar que los procesos son entidades independientes, mientras que los hilos son entidades relacionadas por la sección de datos en el interior del proceso que los contiene.

PROCESOS	HILOS

Manejados por el S.O.	Manejados por los procesos
Independientes de otros procesos	Relacionados con otros hilos del mismo proceso
Memoria privada, se necesitan mecanismos de comunicación para compartir información	Memoria compartida con el resto de hilos que forman el proceso

Tabla 1: Procesos e hilos

Figura1: Relación entre procesos e hilos

Creación de procesos: fork y clone

A la hora de crear procesos linux provee de dos funciones para dicho cometido, la función clone() y la función fork(). Ambas crean un nuevo proceso a partir del proceso padre pero de una manera distinta.

Cuando utilizamos la llamada al sistema fork, el proceso hijo creado es una copia exacta del padre (salvo por el PID y la memoria que ocupa). Al proceso hijo se le facilita una copia de las variables del proceso padre y de los descriptores de fichero. Es importante destacar que las variables del proceso hijo son una copia de las del padre (no se refieren físicamente a la misma variable), por lo que modificar una variable en uno de los procesos no se refleja en el otro.

La llamada al sistema clone es mucho más genérica y flexible que el fork, ya que nos permite definir qué van a compartir los procesos padre e hijo. La tabla 2 resume las diferencias entre las llamadas al sistema fork y clone.

Las llamadas al sistema fork y clone tienen la misma funcionalidad, pero distintas características:

fork: En el momento de la llamada a fork el proceso hijo:

- es una copia exacta del padre excepto el PID.
- tiene las mismas variables y ficheros abiertos.
- las variables son independientes (padre e hijo tienen distintas memorias).
- los ficheros son compartidos (heredan el descriptor).

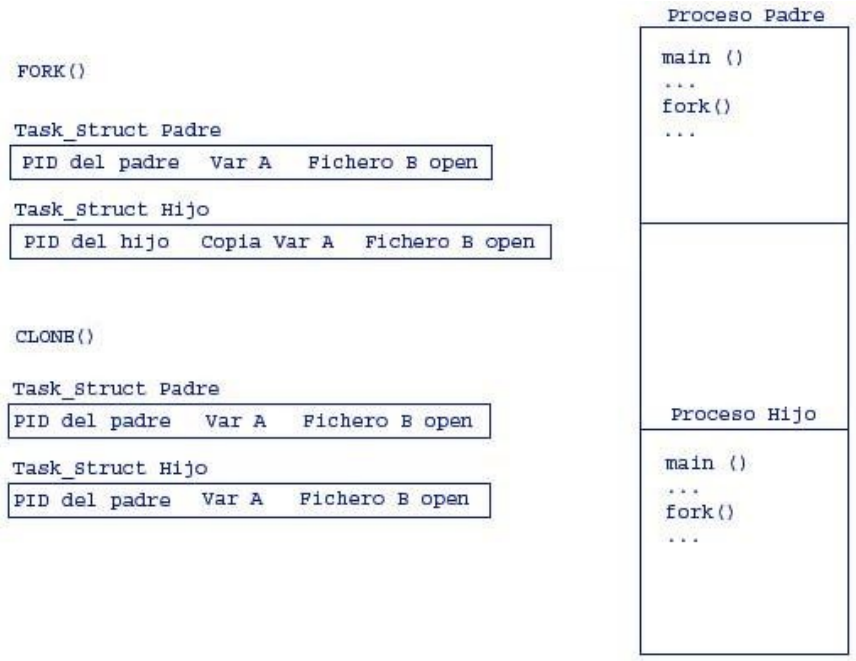
clone: permite especificar qué queremos que compartan padre e hijo.

- espacio de direccionamiento
- información de control del sistema de archivos (file system)
- descriptores de archivos abiertos.
- gestores de señales o PID.

FORK	CLONE
El hijo es una copia exacta del padre (salvo por el PID y memoria)	Permite especificar qué comparten padre e hijo
Ambos procesos disponen de las mismas variables, aunque éstas son independientes	
El hijo hereda los descriptores de fichero del padre	

Tabla 2: Fork y clone

En la figura 2 se puede observar cómo trabajan las llamadas al sistema fork y clone. A la derecha, en vertical, se muestra una representación de la memoria. Tanto al hacer un fork como un clone en su modalidad por defecto (se puede cambiar el comportamiento de clone con una serie de flags), el proceso padre se copia en la zona de memoria del proceso hijo.



En el fork() el hijo creado obtiene una copia de todos los campos del task_struct del padre y su propio identificador. En el clone el hijo en principio dispondrá de exactamente los mismos campos del task_struct del padre y sólo realizará una copia de estos en caso de modificar alguno de ellos. Si esto ocurre debe asignarsele al hijo su propio PID.

7.2 Llamada al sistema fork

Los procesos en Linux tienen una estructura jerárquica, es decir, un proceso padre puede crear un nuevo proceso hijo y así sucesivamente. La forma en que un proceso arranca a otro es mediante una llamada al sistema fork o clone.

Cuando se hace un fork, se crea un nuevo task_struct a partir del task_struct del proceso padre. Al hijo se le asigna un PID propio y se le copian las variables del proceso padre. Sin embargo, vemos como en la llamada al sistema clone (figura 2) el task_struct del proceso padre se copia y se deja tal cual, por lo que el hijo tendrá el mismo PID que el proceso padre y obtendrá (físicamente) las mismas variables que el proceso padre.
El proceso hijo creado es una copia del padre (mismas instrucciones, misma memoria). Lo normal es que a continuación el hijo ejecute una llamada al sistema exec. En cuanto al valor devuelto por el fork, se trata de un valor numérico que depende tanto de si el fork se ha ejecutado correctamente como de si nos encontramos en el proceso padre o en el hijo.

- Si se produce algún error en la ejecución del fork, el valor devuelto es -1.

- Si no se produce ningún error y nos encontramos en el proceso hijo, el fork devuelve un 0.
- Si no se produce ningún error y nos encontramos en el proceso padre, el fork devuelve el PID asignado al proceso hijo.

A la variable errno se le asigna un código de error determinado cada vez que se produce algún problema. Una llamada al sistema fork (o clone) puede provocar dos tipos de problemas: bien se ha alcanzado el máximo número de procesos, bien no queda suficiente memoria para crear el nuevo proceso. La tabla 3 muestra los valores que obtiene la variable errno en función del tipo de error producido.

Error	Significado
EAGAIN	Se ha llegado al número máximo de procesos del usuario actual o del sistema
EANOMEM	El núcleo no ha podido asignar suficiente memoria para crear un nuevo proceso

Tabla 3: Problemas en el fork

La figura 3 muestra un ejemplo de utilización de la llamada al sistema fork, en este caso combinada con una llamada al sistema exec. El objetivo es que el padre imprima por pantalla el PID de su hijo y que el hijo ejecute la herramienta date, que imprime por pantalla la fecha y hora actuales.

```
#include <stdio.h>
#include <unistd.h>

void main() {
  pid_t pid;
  pid = fork();

  if (pid == -1) {
    printf("Error al crear proceso hijo\n");
    exit(0);
  }

  if (pid) {
    //Proceso padre
    printf("Soy el padre, y el PID de mi hijo es %d\n", pid);
  } else {
    //Proceso hijo
    printf("Soy el hijo, y voy a ejecutar la herramienta date\n");
    execve("/bin/date",NULL,NULL);
  }
}
```

Figura 3: Ejemplo de uso de fork

¿Cómo se ejecuta la llamada al sistema fork? ¿Qué secuencia de eventos tiene lugar desde que se encuentra una llamada al sistema fork en el código hasta que se empieza a ejecutar do_fork, la función principal del archivo fork.c? Veámoslo paso a paso:

1. La función fork de la librería libc coloca los parámetros de la llamada en los registros del procesador y se ejecuta la instrucción INT 0x80.
2. Se conmuta a modo núcleo y, mediante las tablas IDT y GDT, se llama a la función sys_call.
3. La función sys_call busca en la sys_call_table la dirección de la llamada al sistema sys_fork.
4. La función sys_fork llama a la función do_fork, que es la función principal del archivo fork.c.

Nótese que la función do_fork implementa tanto la llamada al sistema fork como la llamada al sistema clone. Dada la gran flexibilidad y versatilidad de la llamada al sistema clone, la función do_fork necesita una cierta cantidad de información para implementar la llamada al sistema clone. Esta información se conoce como los flags de control del clone. La tabla 4 muestra los flags de control de la llamada al sistema clone.

FLAG	DESCRIPCIÓN
CLONE_VM	Si se pone CLONE_VM, los procesos padre e hijo se ejecutan en el mismo espacio de memoria (por lo que una modificación en un proceso se refleja también en el otro).
CLONE_FS	Si se pone CLONE_FS, los procesos padre e hijo comparten la misma información del sistema de ficheros. Ésta incluye la raíz del sistema de ficheros, el directorio de trabajo actual y el valor de umask.
CLONE_FILES	Si se pone CLONE_FILES, los procesos padre e hijo comparten la misma estructura de descriptores de fichero. Los descriptores de fichero siempre se refieren a los mismos ficheros en el padre y en el proceso hijo.
CLONE_SIGHAND	Si se pone CLONE_SIGHAND, los procesos padre e hijo comparten la misma estructura de manejadores de señales.
CLONE_PTRACE	Si el proceso padre está siendo depurado, el proceso hijo también estará siendo depurado.
CLONE_VFORK	Utilizada por vfork para indicar que despierte al padre al morir.
CLONE_PARENT	Si CLONE_PARENT está presente, el padre del nuevo hijo será el mismo que el del proceso invocador. Si CLONE_PARENT no está presente, el padre del hijo es el proceso invocador.

CLONE_THREAD	Si CLONE_THREAD está presente, el proceso hijo se pone en el mismo grupo de hilos que el proceso invocador y fuerza al hijo a compartir el descriptor de señal con el padre.
CLONE_NEWNS	Esta activa si la copia necesita su propio namespace, es decir, su propia perspectiva de los ficheros montados.
CLONE_SYSVSEM	Padre e hijo comparten una única lista de valores de deshacer para un semáforo de tipo System V.
CLONE_SETTLS	Crea un nuevo segmento de almacenamiento local para un hilo.
CLONE_PARENT_SETTID	Almacena el PID del hijo en la variable parent_tidptr tanto en la zona de memoria del padre como en la del hijo.
CLONE_CHILD_CLEARTID	Elimina el PID del hijo de child_tidptr del espacio de memoria en caso de que éste exista. Además despierta el “futex” en esa dirección.
CLONE_UNTRACED	Usada por el núcleo para deshabilitar CLONE_PTRACE.
CLONE_CHILD_SETTID	Escribe el PID del hijo en child_tidptr en el espacio de memoria del hijo.
CLONE_STOPPED	Fuerza al hijo a comenzar en estado TASK_STOPPED.
CLONE_NEWID	Nuevo ID del namespace del hijo

Tabla 4: Flags de control de la llamada al sistema clone

7.3 do_fork

La función do_fork es la función principal del archivo fork.c y la que implementa las llamadas al sistema fork y clone.
A grandes rasgos, podemos explicar el funcionamiento de la función do_fork como un procedimiento en que se realizan 3 pasos:

1. Invocar a copy_process para crear un nuevo proceso, creando una nueva task_struct, y asignandole los recursos.
2. Obtener el identificador del proceso hijo que se creo en copy_process
3. Devolver el identificador del proceso hijo.

A continuación se explica paso a paso la función do_fork, y se concluye este apartado con el código al completo de la función do_fork.

Paso 1

Este es el paso más importante de la función do_fork, ya que es cuando se invoca a la función copy_process para crear un nuevo proceso con su estructura task_struct y sus recursos. Si todo sale bien,

copy_process devuelve la dirección del task_struct creado (que almacenamos en la variable p). Si no conseguimos crear el nuevo task_struct, se devuelve un error.

```
    p = copy_process(clone_flags, stack_start, regs,
stack_size,child_tidptr, NULL, trace);

    if (!IS_ERR(p)) {
    ...
    } else {
      nr = PTR_ERR(p);
    }
    return nr;
```

Paso 2

Creamos la estructura vfork y generamos el identificador del proceso hijo a partir del puntero *p*.

```
    struct completion vfork;

    trace_sched_process_fork(current, p);

    nr = task_pid_vnr(p);
```

Paso 3

En caso de que el flag CLONE_PARENT_SETTID este activado padre e hijo compartirán la variable parent_tidptr, de tal forma que el pid del hijo la encontraremos en la zona de memoria del padre como del hijo.

En caso de que el flag CLONE_VFORK este activado significa que se crea un proceso hijo y se bloquea al padre. Precisamente la función que realiza esto es la función vfork() que se encuentra en el fichero *unistd.h*, por lo que se le pasa la dirección de la función a p->vfork->done.

```
if (clone_flags & CLONE_PARENT_SETTID)
   put_user(nr, parent_tidptr);

if (clone_flags & CLONE_VFORK) {
   p ->vfork->done = &vfork;
   init_completion(&vfork);
}
```

Paso 4

Se lleva a cabo unas ultimas configuariones del hijo a través de *audit_finish_fork(p)*.

Comprobamos si el hijo empezará a ejecutarse en modo depuración a través de *tracehook_report_clone()*

Se establece un flag PF_STARTING para diferenciar un hijo que desea entrar en modo de ejecución, en lugar de otros que se va a ejecutar en modo depuración.

```
audit_finish_fork(p);
tracehook_report_clone(trace, regs, clone_flags, nr, p);
p ->flags &= ˜PF_STARTING;
```

Paso 5

En caso de que el hijo deba empezar en modo TASK_STOPPED, se especifica con el flag CLONE_STOPPED, asignamos el estado de TASK_STOPPED al proceso hijo mediante la función *set_task_state().*

Si el flag CLONE_STOPPED no está activo, se invoca a la función wake_up_new_task que realiza las siguientes tareas:

- ✓ Ajusta los parámetros de planificación tanto en el padre como en el hijo.
- ✓ Si el hijo va a ejecutarse en la misma CPU que el proceso padre, se fuerza a que éste se ejecute después de su hijo, insertando al hijo en la cola de ejecución antes que su padre.
- ✓ En otro caso, ya sea porque el proceso hijo vaya a ser ejecutado en otra cpu o bien porque vaya a compartir memoria con su padre, el hijo es colocado en la ultima posición de la cola del padre.

```
if (unlikely(clone_flags & CLONE_STOPPED)) {
        sigaddset(&p >pending.signal, SIGSTOP);
        set_tsk_thread_flag(p, TIF_SIGPENDING);
        #define __set_task_state(tsk, state_value)
        do { (tsk)->state = (state_value); } while (0)
        __set_task_state(p, TASK_STOPPED);

} else {
        wake_up_new_task(p, clone_flags);
}
```

Paso 6

Si el flag CLONE_VFORK está activo, se inserta el proceso padre en una cola de procesos en espera y se suspende hasta que el hijo termina o ejecuta un nuevo programa. El procedimiento wait_for_completation() es en donde espera el padre a que el hijo termine.

```
tracehook_report_clone_complete(trace, regs,clone_flags, nr, p);
```

```
if (clone_flags & CLONE_VFORK) {
                freezer_do_not_count();
                wait_for_completion(&vfork);    //espera por el hijo a que termine
                freezer_count();
                tracehook_report_vfork_done(p, nr);
        }
```

Paso 7

Para acabar la función do_fork, se devuelve el PID del hijo.

```
1428    return nr;
```

A continuación se muestra el código de la función do_fork al completo, para la versión 2.6.28.7 del núcleo, que encontramos en el fichero kernel/fork.c

```
long do_fork(unsigned long clone_flags,
1350              unsigned long stack_start,
1351              struct pt_regs *regs,
1352              unsigned long stack_size,
1353              int __user *parent_tidptr,
1354              int __user *child_tidptr)
1355{
1356        struct task_struct *p;
1357        int trace = 0;
1358        long nr;
1359
1360        /*
1361         * We hope to recycle these flags after 2.6.26
1362         */
1363        if (unlikely(clone_flags & CLONE_STOPPED)) {
1364                static int __read_mostly count = 100;
1365
1366                if (count > 0 && printk_ratelimit()) {
1367                        char comm[TASK_COMM_LEN];
1368
1369                        count--;
1370                        printk(KERN_INFO "fork(): process `%s'
used deprecated "
1371                                                "clone flags 0x%lx\n",
1372                                get_task_comm(comm, current),
1373                                clone_flags & CLONE_STOPPED);
1374                }
1375        }
1376
1377        /*
1378         * When called from kernel_thread, don't do user tracing
stuff.
1379         */
1380        if (likely(user_mode(regs)))
1381                trace = tracehook_prepare_clone(clone_flags);
1382
1383        p = copy_process(clone_flags, stack_start, regs,
```

```
stack_size,
1384                          child_tidptr, NULL, trace);
1385        /*
1386         * Do this prior waking up the new thread - the thread
pointer
1387         * might get invalid after that point, if the thread exits
quickly.
1388         */
1389        if (!IS_ERR(p)) {
1390                struct completion vfork;
1391
1392                trace_sched_process_fork(current, p);
1393
1394                nr = task_pid_vnr(p);
1395
1396                if (clone_flags & CLONE_PARENT_SETTID)
1397                        put_user(nr, parent_tidptr);
1398
1399                if (clone_flags & CLONE_VFORK) {
1400                        p->vfork_done = &vfork;
1401                        init_completion(&vfork);
1402                }
1403
1404                audit_finish_fork(p);
1405                tracehook_report_clone(trace, regs, clone_flags,
nr, p);
1406
1407                /*
1408                 * We set PF_STARTING at creation in case tracing
wants to
1409                 * use this to distinguish a fully live task from
one that
1410                 * hasn't gotten to tracehook_report_clone() yet.
Now we
1411                 * clear it and set the child going.
1412                 */
1413                p->flags &= ~PF_STARTING;
1414
1415                if (unlikely(clone_flags & CLONE_STOPPED)) {
1416                        /*
1417                         * We'll start up with an immediate
SIGSTOP.
1418                         */
1419                        sigaddset(&p->pending.signal, SIGSTOP);
1420                        set_tsk_thread_flag(p, TIF_SIGPENDING);
1421                        __set_task_state(p, TASK_STOPPED);
1422                } else {
1423                        wake_up_new_task(p, clone_flags);
1424                }
1425
1426                tracehook_report_clone_complete(trace, regs,
1427                                                clone_flags, nr,
p);
1428
1429                if (clone_flags & CLONE_VFORK) {
1430                        freezer_do_not_count();
1431                        wait_for_completion(&vfork);
1432                        freezer_count();
1433                        tracehook_report_vfork_done(p, nr);
1434                }
1435        } else {
```

```
1436                    nr = PTR_ERR(p);
1437            }
1438            return nr;
1439}
```

7.4 Función copy_process

Copy_Process

La función do_fork se sustenta en la función copy_process en la que delega toda la responsabilidad de crear las estructuras para el proceso nuevo.

La función copy_process() tiene como misión dos cosas.

- ✓ Duplicar el task_struct del padre para asignárselo al hijo (función dup_task_struct).
- ✓ Inicializar los campos del task_struct, haciendo incapié en las funciones copy_fs, copy_sighand, copy_signal, copy_mm, copy_namespace ...

Se comprueba que los flags CLONE_NEWNS y CLONE_FS estén activos al mismo tiempo se lanza un error.

```
if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))
        return ERR_PTR( EINVAL);
```

Se comprueba que esten activos los flags clone_thread y clone_sighand al mismo tiempo , ya que en ambos casos, padre e hijo comparten los descriptores de señales. En caso de que uno esto activo y otro no, se lanza un error.

Si el flag CLONE_SIGHAND está activo y el flag CLONE_VN no; se devuelve un error ya que el primero indica que ambos procesos deben compartir la tabla de manejadores de señales, mientras que el segundoº indica que comparten el mismo espacio de memoria. En caso de que el primero esté activo, el segundo flag debería estarlo también.

```
if ((clone_flags & CLONE_SIGHAND) && !(clone_flags & CLONE_VM))
                return ERR_PTR( EINVAL);
```

En primer lugar realizamos algunos chequeos de seguridad mediante la función security_task_create. En caso de que hubiese algún error esta function devuelve cero, por lo que saldríamos de la función a través de fork_out.

Se asigna provisionalmente a retval ENOMEM, que significa que el núcleo no ha podido asignar suficiente memoria para crear el nuevo proceso.

Se llama a la función dup_task_struct(), en el que se duplica el task_struct del padre, para el hijo mediante memoria dinámica. La variable p es un puntero a esta estructura que se acaba de crear. Despues de esta función se llevará a cabo la inicialización de la estructura.

```
retval = security_task_create(clone_flags);

if (retval)
        goto fork_out;

retval = ENOMEM;

p = dup_task_struct(current);
```

En caso de que no se haya podido duplicar el task_struct se sale de la función por *fork_out*.

Se le asigna a retval el valor eagain provisionalmente, que significaque se ha llegado al máximo número de procesos del usuario actual o del sistema. El valor anterior que tenía era ENOMEM, que significaba que no se podía generar más hijos ya que no había mas memoria. Este valor ya no tiene sentido ya que con la función **dup_task_struct** el sistema ya habría creado el proceso hijo.

Comprueba si el valor almacenado en: current->signal-> rlim[RLIMIT_NPROC].rlim_cur es mayor o igual al número actual de los procesos creados por el usuario.
Si es así se sale del fork, a menos que el proceso tenga privilegios de root.
La función atomic_read obtiene el número actual de procesos creados por el usuario.

```
if (!p)
        goto fork_out;
rt_mutex_init_task(p);

#ifdef CONFIG_PROVE_LOCKING
        DEBUG_LOCKS_WARN_ON(!p >hardirqs_enabled);
        DEBUG_LOCKS_WARN_ON(!p >softirqs_enabled);
#endif
retval = EAGAIN;

if (atomic_read(&p ->user ->processes) >=p ->signal
>rlim[RLIMIT_NPROC].rlim_cur) {
        if (!capable(CAP_SYS_ADMIN) && !capable(CAP_SYS_RESOURCE) &&
        p ->user != current ->nsproxy ->user_ns ->root_user)
                goto bad_fork_free;
}
```

Se aumenta el número de contadores de manera atómica. Posible sección crítica.

```
atomic_inc(&p ->user ->count);
atomic_inc(&p ->user ->processes);
get_group_info(p ->group_info);
```

Se comprueba que el número de procesos en el sistema (almacenado en la variable nr_threads) no exceda el valor indicado por la variable max_threads.

Si el proceso padre usa algún modulo, se incrementan los contadores correspondientes de referencia (módulo son las partes del kernel que se pueden cargar para usarlas y descargar cuando ya nadie las usa).

Se fijan algunos campos cruciales relacionados con el estado de proceso.

Indicamos que el nuevo hijo no ha hecho un exec.

Copiamos los flags del proceso padre al hijo.

```
if (nr_threads >= max_threads)
    goto bad_fork_cleanup_count

if (!try_module_get(task_thread_info(p) ->exec_domain ->module))
    goto bad_fork_cleanup_count;

if (p- >binfmt && !try_module_get(p ->binfmt ->module))
    goto bad_fork_cleanup_put_domain;

p ->did_exec = 0;
delayacct_tsk_init(p); /_ Must remain after dup_task_struct() _ /

copy_flags(clone_flags, p);
```

A continuación se continúa inicializando la task_struct .

Se inicializa la lista de los hijos del proceso que se está creando, así como se inicializa un puntero (p->sibling) a los otros hijos de mi padre (los hermanos).

Se inicializa el campo vfork_done el cual obtendrá mas adelante si es necesario la dirección de vfork, que se utiliza para que el padre espere a que el hijo termine.

Se inicializa la cerradura que interviene en el acceso de mm, files, fs, tty, keyrings mediante la función *spin_lock_init()*

Se inicializa funciones pendientes mediante la función *init_sigpending.*

Se inicializa los contadores de tiempo del proceso hijo.

```
INIT_LIST_HEAD(&p ->children);
INIT_LIST_HEAD(&p ->sibling);

#ifdef CONFIG_PREEMPT_RCU
        p ->rcu_read_lock_nesting = 0;
        p ->rcu_flipctr_idx = 0;
1020 #endif /_ #ifdef CONFIG_PREEMPT_RCU _ /

p ->vfork_done = NULL;

spin_lock_init(&p ->alloc_lock);
clear_tsk_thread_flag(p, TIF_SIGPENDING);

init_sigpending(&p ->pending);

p ->utime = cputime_zero;
p ->stime = cputime_zero;
p ->gtime = cputime_zero;
p ->utimescaled = cputime_zero;
p ->stimescaled = cputime_zero;
p ->prev_utime = cputime_zero;
p ->prev_stime = cputime_zero;
p ->default_timer_slack_ns = current ->timer_slack_ns;

/*A partir de aquí no tengo ni idea que pasa-maxpower*/
#ifdef CONFIG_DETECT_SOFTLOCKUP
p ->last_switch_count = 0;
        p >last_switch_timestamp = 0;
#endif
task_io_accounting_init(&p >ioac);
acct_clear_integrals(p);
posix_cpu_timers_init(p);
p >lock_depth = 1; /_ 1 = no lock _ /
do_posix_clock_monotonic_gettime(&p >start_time);
p >real_start_time = p >start_time;
monotonic_to_bootbased(&p >real_start_time);

#ifdef CONFIG_SECURITY
        p >security = NULL;
#endif

p >cap_bset = current >cap_bset;
p >io_context = NULL;
p >audit_context = NULL;
cgroup_fork(p);

#ifdef CONFIG_NUMA
        p >mempolicy = mpol_dup(p >mempolicy);
        if (IS_ERR(p >mempolicy)) {
                retval = PTR_ERR(p >mempolicy);
                p >mempolicy = NULL;
                goto bad_fork_cleanup_cgroup;
        }
        mpol_fix_fork_child_flag(p);
#endif

#ifdef CONFIG_TRACE_IRQFLAGS
```

```
        p >irq_events = 0;
        #ifdef __ARCH_WANT_INTERRUPTS_ON_CTXSW
                p >hardirqs_enabled = 1;
        #else
                p >hardirqs_enabled = 0;
        #endif
        p >hardirq_enable_ip = 0;
        p >hardirq_enable_event = 0;
        p >hardirq_disable_ip = _THIS_IP_;
        p >hardirq_disable_event = 0;
        p >softirqs_enabled = 1;
        p >softirq_enable_ip = _THIS_IP_;
        p >softirq_enable_event = 0;
        p >softirq_disable_ip = 0;
        p >softirq_disable_event = 0;
        p >hardirq_context = 0;
        p >softirq_context = 0;
#endif
#ifdef CONFIG_LOCKDEP
        p >lockdep_depth = 0; /_ no locks held yet _ /
        p >curr_chain_key = 0;
        p >lockdep_recursion = 0;
#endif

#ifdef CONFIG_DEBUG_MUTEXES
        p >blocked_on = NULL; /_ not blocked yet _ /
#endif
```

Se invoca a la función sched_fork() para completar la inicialización de la estructura de datos del planificador del nuevo proceso. La función también fija el estado del nuevo proceso a TASK_RUNNING y fija el campo preempt_count de la estructura tHRead_info a 1, para evitar el cambio de contexto de modo núcleo a modo usuario.

```
sched_fork(p, clone_flags);
```

Se realiza unos chequeos de seguridad mediante *security_task_alloc() y audit_alloc()*

Se lleva a cabo la copia o la *compartición* (se especifica con los flags clone) de las componentes del padre al hijo. Ficheros abiertos, Sistema de ficheros abiertos, descriptores de señales, espacio de direccionamiento, espacio de nombres …

Probablemente esta sea la parte más importante de copy_process

```
//Chequeos de seguridad
if ((retval = security_task_alloc(p)))
  goto bad_fork_cleanup_policy;

//Chequeos de seguridad
if ((retval = audit_alloc(p)))
        goto bad_fork_cleanup_security;

if ((retval = copy_semundo(clone_flags, p)))
```

```
        goto bad_fork_cleanup_audit;

/*Copia la lista de ficheros abiertos del padre.*/
if ((retval = copy_files(clone_flags, p)))
        goto bad_fork_cleanup_semundo;

/*Copia información del sistema de ficheros sobre incluye la raíz del sistema de
ficheros, el directorio de trabajo actual y el valor de umask.*/
if ((retval = copy_fs(clone_flags, p)))
        goto bad_fork_cleanup_files;

/*Copia las funciones manejadoras asociadas a cada señal que tiene el proceso
padre, en el proceso hijo.*/
if ((retval = copy_sighand(clone_flags, p)))
        goto bad_fork_cleanup_fs;

/*Copia información asociada a señales como límites de recursos, cola de hijos
lanzados, lista de señales pendientes de atender, etc.*/
if ((retval = copy_signal(clone_flags, p)))
        goto bad_fork_cleanup_sighand;
```

La función copy_mm() hace una copia de la región de memoria del padre en el hijo, es decir, copia el contendido del espacio de direccionamiento del proceso.

```
if ((retval = copy_mm(clone_flags, p)))
        goto bad_fork_cleanup_signal;


if ((retval = copy_keys(clone_flags, p)))
        goto bad_fork_cleanup_mm;

/*Copia espacio de nombres
muestra un mismo sistema de archivos en forma diferente para cada usuario, ya que
define la capa superior de visibilidad, dicha capa recibe el nombre de Espacio de
Nombres.*/
if ((retval = copy_namespaces(clone_flags, p)))
        goto bad_fork_cleanup_keys;

/*estrada salida*/
if ((retval = copy_io(clone_flags, p)))
        goto bad_fork_cleanup_namespaces;
```

Se llama a copy_thread que inicializar la pila en modo kernel del hijo con los valores de los registros contenidos en la CPU. En caso de que no se pudiese se devolvería un cero y saldríamos de copy_process.

```
retval = copy_thread(0, clone_flags, stack_start, stack_size, p, regs);

if (retval)
        goto bad_fork_cleanup_io;
```

Si el pid no está inicializado, le asignamos un pid a través de la función alloc_pid(). En la versión anterior del kernel esta función era una de las primeras instrucciones de la función do_fork. En esta versión nos la encontramos aquí.

```
/*SI el pid no esta inicializado*/
if (pid != &init_struct_pid) {

        retval = ENOMEM;
        /*Se asigna un Nuevo pid*/
        pid = alloc_pid(task_active_pid_ns(p));

        if (!pid)
                goto bad_fork_cleanup_io;
                if (clone_flags & CLONE_NEWPID) {
                        retval =
pid_ns_prepare_proc(task_active_pid_ns(p));
                        if (retval < 0)
                                goto bad_fork_free_pid;
                }
        }
        global id, i.e. the id seen from the init namespace

        p ->pid = pid_nr(pid);
        p ->tgid = p ->pid;


        if (clone_flags & CLONE_THREAD)
                p ->tgid = current ->tgid;
        if (current ->nsproxy != p ->nsproxy) {
                retval = ns_cgroup_clone(p, pid);
        if (retval)
                goto bad_fork_free_pid;
}
        p ->tgid = current ->tgid;
        if (current ->nsproxy != p ->nsproxy) {
                retval = ns_cgroup_clone(p, pid);
        if (retval)
                goto bad_fork_free_pid;
}
p ->set_child_tid = (clone_flags &
CLONE_CHILD_SETTID) ? child_tidptr : NULL;
/_
_ Clear TID on mm_release()?
_ /

p ->clear_child_tid = (clone_flags &
CLONE_CHILD_CLEARTID) ? child_tidptr: NULL;

#ifdef CONFIG_FUTEX
        p >robust_list = NULL;
#ifdef CONFIG_COMPAT
        p >compat_robust_list = NULL;
#endif
```

Se lleva a cabo la inicialización de otra lista.

```
INIT_LIST_HEAD(&p ->pi_state_list);
p >pi_state_cache = NULL;
```

Se llevan a cabo distintas asignaciones e inicializaciones.

Se inicializa otra lista e asignamos las CPUs que tiene permitidas el padre al hijo.

```
/*Distintas asignaciones*/
p ->parent_exec_id = p ->self_exec_id;

p ->exit_signal = (clone_flags & CLONE_THREAD) ? 1 : (clone_flags & CSIGNAL);
p ->pdeath_signal = 0;
p ->exit_state = 0;

/*Se inicializa otra lista*/
p ->group_leader = p;
INIT_LIST_HEAD(&p->thread_group);

/*CPU permitidas del hijo son las mismas del padre*/
p >cpus_allowed = current >cpus_allowed;
p >rt.nr_cpus_allowed = current >rt.nr_cpus_allowed;
```

Nos aseguramos de que la cpu asignada al proceso hijo es la misma que la del padre, mediante el campo cpus_allowed y la función smp_processor_id.

Cuando usamos el flag CLONE_PARENT significa que al padre del nuevo hijo será el mismo que el del proceso invocador

Se toma la cerradura SpinLocks (son un tipo de semáforo)

```
if (unlikely(!cpu_isset(task_cpu(p), p ->cpus_allowed) ||!
cpu_online(task_cpu(p))))
    set_task_cpu(p, smp_processor_id());

if (clone_flags & (CLONE_PARENT|CLONE_THREAD))
        p ->real_parent = current ->real_parent;
else
        p >real_parent = current;  //si no el padre es current

 spin_lock(&current ->sighand ->siglock);
```

En el caso de que llegue una señal en medio del, liberamos los semáforos (spin_unlock, write_unlock_irq)y el fork se aborta

```
if (signal_pending(current)) {
        spin_unlock(&current >sighand >siglock);
        write_unlock_irq(&tasklist_lock);
        retval = ERESTARTNOINTR;
        goto bad_fork_free_pid;
                                        }
```

Si CLONE_THREAD está presente, el proceso hijo se pone en el mismo grupo de hilos que el proceso invocador y fuerza al hijo a compartir el descriptor de señal con el padre. Por este motivo se le asigna el group_leader del padre al hijo.

¿Qué son los grupos? Cuando se crea un nuevo hilo, se coloca en un grupo, bien indicándolo explícitamente, o bien dejando que el sistema lo coloque en el grupo por defecto. Los grupos de hilos permiten que sea posible recoger varios hilos de ejecución en un solo objeto y manipularlo como un grupo, en vez de individualmente. Por ejemplo, se pueden regenerar los hilos de un grupo mediante una sola sentencia.

Se asigna el puntero a los hermanos del nuevo hijo creado mediante la función list_add_tail.

Termina el clonado. Si el proceso padre está siendo depurado, el proceso hijo también estará siendo depurado.

```
if (likely(p ->pid)) {
        list_add_tail(&p ->sibling, &p- >real_parent ->children);

tracehook_finish_clone(p, clone_flags, trace);
```

En caso de que el hijo sea el hilo principal de un grupo de hilos se realizan unas ultimas tareas.

Se invoca a attack_pid para incluir en el hash el nuevo PID del proceso, y de esta forma quede registrado en el sistema.

Se incrementa el número de hilos, y el número de forks realizados.

Se libera las cerraduras y se devuelve el puntero al descriptor del fichero.

```
/*Comprueba si el hijo es el hilo principal de un grupo de hilos o si el hijo
pertenece al grupo de hilos del padre.*/

if (thread_group_leader(p)) {

        if (clone_flags & CLONE_NEWPID)
                p ->nsproxy ->pid_ns ->child_reaper = p;

        p ->signal ->leader_pid = pid;
        tty_kref_put(p >signal >tty);
        p ->signal >tty = tty_kref_get(current >signal >tty);
        set_task_pgrp(p, task_pgrp_nr(current));
        set_task_session(p, task_session_nr(current));
        attach_pid(p, PIDTYPE_PGID, task_pgrp(current));
        attach_pid(p, PIDTYPE_SID, task_session(current));
        list_add_tail_rcu(&p >tasks, &init_task.tasks);
        __get_cpu_var(process_counts)++;
}
// Se invoca a la función attach_pid()  para incluir en el hash en nuevo PID del
proceso.
attach_pid(p, PIDTYPE_PID, pid);
//incrementamos el número de hilos
nr_threads++;
 }

//incrementamos el numero total de forks
total_forks++;

//se libera la cerradura
spin_unlock(&current >sighand >siglock);
//se libera otro lipo de cerradura
write_unlock_irq(&tasklist_lock);
proc_fork_connector(p);
cgroup_post_fork(p);

 return p
```

A continuación mostramos el código completo de la función copy_process()

```
941     static struct task_struct *copy_process(unsigned long clone_flags,
 942                                     unsigned long stack_start,
 943                                     struct pt_regs *regs,
 944                                     unsigned long stack_size,
 945                                     int __user *child_tidptr,
 946                                     struct pid *pid,
 947                                     int trace)
 948{
 949     int retval;
 950     struct task_struct *p;
 951     int cgroup_callbacks_done = 0;
 952
 953     if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|
CLONE_FS))
 954             return ERR_PTR(-EINVAL);
 955
 956     /*
 957      * Thread groups must share signals as well, and detached threads
 958      * can only be started up within the thread group.
 959      */
```

```
 960        if ((clone_flags & CLONE_THREAD) && !(clone_flags & CLONE_SIGHAND))
 961                return ERR_PTR(-EINVAL);
 962
 963        /*
 964         * Shared signal handlers imply shared VM. By way of the above,
 965         * thread groups also imply shared VM. Blocking this case allows
 966         * for various simplifications in other code.
 967         */
 968        if ((clone_flags & CLONE_SIGHAND) && !(clone_flags & CLONE_VM))
 969                return ERR_PTR(-EINVAL);
 970
 971        retval = security_task_create(clone_flags);
 972        if (retval)
 973                goto fork_out;
 974
 975        retval = -ENOMEM;
 976        p = dup_task_struct(current);
 977        if (!p)
 978                goto fork_out;
 979
 980        rt_mutex_init_task(p);
 981
 982#ifdef CONFIG_PROVE_LOCKING
 983        DEBUG_LOCKS_WARN_ON(!p->hardirqs_enabled);
 984        DEBUG_LOCKS_WARN_ON(!p->softirqs_enabled);
 985#endif
 986        retval = -EAGAIN;
 987        if (atomic_read(&p->user->processes) >=
 988                        p->signal->rlim[RLIMIT_NPROC].rlim_cur) {
 989                if (!capable(CAP_SYS_ADMIN) && !capable(CAP_SYS_RESOURCE) &&
 990                    p->user != current->nsproxy->user_ns->root_user)
 991                        goto bad_fork_free;
 992        }
 993
 994        atomic_inc(&p->user->__count);
 995        atomic_inc(&p->user->processes);
 996        get_group_info(p->group_info);
 997
 998        /*
 999         * If multiple threads are within copy_process(), then this check
1000         * triggers too late. This doesn't hurt, the check is only there
1001         * to stop root fork bombs.
1002         */
1003        if (nr_threads >= max_threads)
1004                goto bad_fork_cleanup_count;
1005
1006        if (!try_module_get(task_thread_info(p)->exec_domain->module))
1007                goto bad_fork_cleanup_count;
1008
1009        if (p->binfmt && !try_module_get(p->binfmt->module))
1010                goto bad_fork_cleanup_put_domain;
1011
1012        p->did_exec = 0;
1013        delayacct_tsk_init(p);  /* Must remain after dup_task_struct() */
1014        copy_flags(clone_flags, p);
1015        INIT_LIST_HEAD(&p->children);
1016        INIT_LIST_HEAD(&p->sibling);
1017#ifdef CONFIG_PREEMPT_RCU
1018        p->rcu_read_lock_nesting = 0;
1019        p->rcu_flipctr_idx = 0;
```

```
1020#endif /* #ifdef CONFIG_PREEMPT_RCU */
1021        p->vfork_done = NULL;
1022        spin_lock_init(&p->alloc_lock);
1023
1024        clear_tsk_thread_flag(p, TIF_SIGPENDING);
1025        init_sigpending(&p->pending);
1026
1027        p->utime = cputime_zero;
1028        p->stime = cputime_zero;
1029        p->gtime = cputime_zero;
1030        p->utimescaled = cputime_zero;
1031        p->stimescaled = cputime_zero;
1032        p->prev_utime = cputime_zero;
1033        p->prev_stime = cputime_zero;
1034
1035        p->default_timer_slack_ns = current->timer_slack_ns;
1036
1037#ifdef CONFIG_DETECT_SOFTLOCKUP
1038        p->last_switch_count = 0;
1039        p->last_switch_timestamp = 0;
1040#endif
1041
1042        task_io_accounting_init(&p->ioac);
1043        acct_clear_integrals(p);
1044
1045        posix_cpu_timers_init(p);
1046
1047        p->lock_depth = -1;             /* -1 = no lock */
1048        do_posix_clock_monotonic_gettime(&p->start_time);
1049        p->real_start_time = p->start_time;
1050        monotonic_to_bootbased(&p->real_start_time);
1051#ifdef CONFIG_SECURITY
1052        p->security = NULL;
1053#endif
1054        p->cap_bset = current->cap_bset;
1055        p->io_context = NULL;
1056        p->audit_context = NULL;
1057        cgroup_fork(p);
1058#ifdef CONFIG_NUMA
1059        p->mempolicy = mpol_dup(p->mempolicy);
1060        if (IS_ERR(p->mempolicy)) {
1061                retval = PTR_ERR(p->mempolicy);
1062                p->mempolicy = NULL;
1063                goto bad_fork_cleanup_cgroup;
1064        }
1065        mpol_fix_fork_child_flag(p);
1066#endif
1067#ifdef CONFIG_TRACE_IRQFLAGS
1068        p->irq_events = 0;
1069#ifdef __ARCH_WANT_INTERRUPTS_ON_CTXSW
1070        p->hardirqs_enabled = 1;
1071#else
1072        p->hardirqs_enabled = 0;
1073#endif
1074        p->hardirq_enable_ip = 0;
1075        p->hardirq_enable_event = 0;
1076        p->hardirq_disable_ip = _THIS_IP_;
1077        p->hardirq_disable_event = 0;
1078        p->softirqs_enabled = 1;
1079        p->softirq_enable_ip = _THIS_IP_;
```

```
1080        p->softirq_enable_event = 0;
1081        p->softirq_disable_ip = 0;
1082        p->softirq_disable_event = 0;
1083        p->hardirq_context = 0;
1084        p->softirq_context = 0;
1085#endif
1086#ifdef CONFIG_LOCKDEP
1087        p->lockdep_depth = 0; /* no locks held yet */
1088        p->curr_chain_key = 0;
1089        p->lockdep_recursion = 0;
1090#endif
1091
1092#ifdef CONFIG_DEBUG_MUTEXES
1093        p->blocked_on = NULL; /* not blocked yet */
1094#endif
1095
1096        /* Perform scheduler related setup. Assign this task to a CPU. */
1097        sched_fork(p, clone_flags);
1098
1099        if ((retval = security_task_alloc(p)))
1100                goto bad_fork_cleanup_policy;
1101        if ((retval = audit_alloc(p)))
1102                goto bad_fork_cleanup_security;
1103        /* copy all the process information */
1104        if ((retval = copy_semundo(clone_flags, p)))
1105                goto bad_fork_cleanup_audit;
1106        if ((retval = copy_files(clone_flags, p)))
1107                goto bad_fork_cleanup_semundo;
1108        if ((retval = copy_fs(clone_flags, p)))
1109                goto bad_fork_cleanup_files;
1110        if ((retval = copy_sighand(clone_flags, p)))
1111                goto bad_fork_cleanup_fs;
1112        if ((retval = copy_signal(clone_flags, p)))
1113                goto bad_fork_cleanup_sighand;
1114        if ((retval = copy_mm(clone_flags, p)))
1115                goto bad_fork_cleanup_signal;
1116        if ((retval = copy_keys(clone_flags, p)))
1117                goto bad_fork_cleanup_mm;
1118        if ((retval = copy_namespaces(clone_flags, p)))
1119                goto bad_fork_cleanup_keys;
1120        if ((retval = copy_io(clone_flags, p)))
1121                goto bad_fork_cleanup_namespaces;
1122        retval = copy_thread(0, clone_flags, stack_start, stack_size, p, regs);
1123        if (retval)
1124                goto bad_fork_cleanup_io;
1125
1126        if (pid != &init_struct_pid) {
1127                retval = -ENOMEM;
1128                pid = alloc_pid(task_active_pid_ns(p));
1129                if (!pid)
1130                        goto bad_fork_cleanup_io;
1131
1132                if (clone_flags & CLONE_NEWPID) {
1133                        retval = pid_ns_prepare_proc(task_active_pid_ns(p));
1134                        if (retval < 0)
1135                                goto bad_fork_free_pid;
1136                }
1137        }
1138
1139        p->pid = pid_nr(pid);
```

```
1140        p->tgid = p->pid;
1141        if (clone_flags & CLONE_THREAD)
1142                p->tgid = current->tgid;
1143
1144        if (current->nsproxy != p->nsproxy) {
1145                retval = ns_cgroup_clone(p, pid);
1146                if (retval)
1147                        goto bad_fork_free_pid;
1148        }
1149
1150        p->set_child_tid = (clone_flags & CLONE_CHILD_SETTID) ? child_tidptr :
NULL;
1151        /*
1152         * Clear TID on mm_release()?
1153         */
1154        p->clear_child_tid = (clone_flags & CLONE_CHILD_CLEARTID) ? child_tidptr:
NULL;
1155#ifdef CONFIG_FUTEX
1156        p->robust_list = NULL;
1157#ifdef CONFIG_COMPAT
1158        p->compat_robust_list = NULL;
1159#endif
1160        INIT_LIST_HEAD(&p->pi_state_list);
1161        p->pi_state_cache = NULL;
1162#endif
1163        /*
1164         * sigaltstack should be cleared when sharing the same VM
1165         */
1166        if ((clone_flags & (CLONE_VM|CLONE_VFORK)) == CLONE_VM)
1167                p->sas_ss_sp = p->sas_ss_size = 0;
1168
1169        /*
1170         * Syscall tracing should be turned off in the child regardless
1171         * of CLONE_PTRACE.
1172         */
1173        clear_tsk_thread_flag(p, TIF_SYSCALL_TRACE);
1174#ifdef TIF_SYSCALL_EMU
1175        clear_tsk_thread_flag(p, TIF_SYSCALL_EMU);
1176#endif
1177        clear_all_latency_tracing(p);
1178
1179        /* Our parent execution domain becomes current domain
1180           These must match for thread signalling to apply */
1181        p->parent_exec_id = p->self_exec_id;
1182
1183        /* ok, now we should be set up.. */
1184        p->exit_signal = (clone_flags & CLONE_THREAD) ? -1 : (clone_flags &
CSIGNAL);
1185        p->pdeath_signal = 0;
1186        p->exit_state = 0;
1187
1188        /*
1189         * Ok, make it visible to the rest of the system.
1190         * We dont wake it up yet.
1191         */
1192        p->group_leader = p;
1193        INIT_LIST_HEAD(&p->thread_group);
1194
1195        /* Now that the task is set up, run cgroup callbacks if
1196         * necessary. We need to run them before the task is visible
```

```
1197        * on the tasklist. */
1198       cgroup_fork_callbacks(p);
1199       cgroup_callbacks_done = 1;
1200
1201       /* Need tasklist lock for parent etc handling! */
1202       write_lock_irq(&tasklist_lock);
1203
1204       /*
1205        * The task hasn't been attached yet, so its cpus_allowed mask will
1206        * not be changed, nor will its assigned CPU.
1207        *
1208        * The cpus_allowed mask of the parent may have changed after it was
1209        * copied first time - so re-copy it here, then check the child's CPU
1210        * to ensure it is on a valid CPU (and if not, just force it back to
1211        * parent's CPU). This avoids alot of nasty races.
1212        */
1213       p->cpus_allowed = current->cpus_allowed;
1214       p->rt.nr_cpus_allowed = current->rt.nr_cpus_allowed;
1215       if (unlikely(!cpu_isset(task_cpu(p), p->cpus_allowed) ||
1216                       !cpu_online(task_cpu(p))))
1217             set_task_cpu(p, smp_processor_id());
1218
1219       /* CLONE_PARENT re-uses the old parent */
1220       if (clone_flags & (CLONE_PARENT|CLONE_THREAD))
1221             p->real_parent = current->real_parent;
1222       else
1223             p->real_parent = current;
1224
1225       spin_lock(&current->sighand->siglock);
1226
1227       /*
1228        * Process group and session signals need to be delivered to just the
1229        * parent before the fork or both the parent and the child after the
1230        * fork. Restart if a signal comes in before we add the new process to
1231        * it's process group.
1232        * A fatal signal pending means that current will exit, so the new
1233        * thread can't slip out of an OOM kill (or normal SIGKILL).
1234        */
1235       recalc_sigpending();
1236       if (signal_pending(current)) {
1237             spin_unlock(&current->sighand->siglock);
1238             write_unlock_irq(&tasklist_lock);
1239             retval = -ERESTARTNOINTR;
1240             goto bad_fork_free_pid;
1241       }
1242
1243       if (clone_flags & CLONE_THREAD) {
1244             p->group_leader = current->group_leader;
1245             list_add_tail_rcu(&p->thread_group, &p->group_leader-
>thread_group);
1246       }
1247
1248       if (likely(p->pid)) {
1249             list_add_tail(&p->sibling, &p->real_parent->children);
1250             tracehook_finish_clone(p, clone_flags, trace);
1251
1252             if (thread_group_leader(p)) {
1253                   if (clone_flags & CLONE_NEWPID)
1254                         p->nsproxy->pid_ns->child_reaper = p;
1255
```

```
1256                    p->signal->leader_pid = pid;
1257                    tty_kref_put(p->signal->tty);
1258                    p->signal->tty = tty_kref_get(current->signal->tty);
1259                    set_task_pgrp(p, task_pgrp_nr(current));
1260                    set_task_session(p, task_session_nr(current));
1261                    attach_pid(p, PIDTYPE_PGID, task_pgrp(current));
1262                    attach_pid(p, PIDTYPE_SID, task_session(current));
1263                    list_add_tail_rcu(&p->tasks, &init_task.tasks);
1264                    __get_cpu_var(process_counts)++;
1265            }
1266            attach_pid(p, PIDTYPE_PID, pid);
1267            nr_threads++;
1268    }
1269
1270    total_forks++;
1271    spin_unlock(&current->sighand->siglock);
1272    write_unlock_irq(&tasklist_lock);
1273    proc_fork_connector(p);
1274    cgroup_post_fork(p);
1275    return p;
1276
1277bad_fork_free_pid:
1278    if (pid != &init_struct_pid)
1279            free_pid(pid);
1280bad_fork_cleanup_io:
1281    put_io_context(p->io_context);
1282bad_fork_cleanup_namespaces:
1283    exit_task_namespaces(p);
1284bad_fork_cleanup_keys:
1285    exit_keys(p);
1286bad_fork_cleanup_mm:
1287    if (p->mm)
1288            mmput(p->mm);
1289bad_fork_cleanup_signal:
1290    cleanup_signal(p);
1291bad_fork_cleanup_sighand:
1292    __cleanup_sighand(p->sighand);
1293bad_fork_cleanup_fs:
1294    exit_fs(p); /* blocking */
1295bad_fork_cleanup_files:
1296    exit_files(p); /* blocking */
1297bad_fork_cleanup_semundo:
1298    exit_sem(p);
1299bad_fork_cleanup_audit:
1300    audit_free(p);
1301bad_fork_cleanup_security:
1302    security_task_free(p);
1303bad_fork_cleanup_policy:
1304#ifdef CONFIG_NUMA
1305    mpol_put(p->mempolicy);
1306bad_fork_cleanup_cgroup:
1307#endif
1308    cgroup_exit(p, cgroup_callbacks_done);
1309    delayacct_tsk_free(p);
1310    if (p->binfmt)
1311            module_put(p->binfmt->module);
1312bad_fork_cleanup_put_domain:
1313    module_put(task_thread_info(p)->exec_domain->module);
1314bad_fork_cleanup_count:
1315    put_group_info(p->group_info);
```

```
1316        atomic_dec(&p->user->processes);
1317        free_uid(p->user);
1318bad_fork_free:
1319        free_task(p);
1320fork_out:
1321        return ERR_PTR(retval);
1322}
```

7.5 Funciones auxiliares

dup_task_struct()

Esta función realiza las siguientes acciones :

- En versiones anteriores del kernel, se invoca a unlazy_fpu() a través de la función prepare_to_copy. En la nueva versión del kernel esta función no hace nada, ya que está comentada dentro de la función prepare_to_copy.
- Ejecuta la macro del alloc_task_struct() para crear un descriptor de proceso (estructura del task_struct) para el nuevo proceso, y almacena su dirección en la variable local del tsk.
- Ejecuta la macro alloc_thread_info para conseguir un área de memoria libre para almacenar la estructura thread_info y la pila en modo núcleo del nuevo proceso, y almacena su dirección en la variable local ti.
- Copia el contenido del descriptor de proceso del hilo current en la estructura task_struct señalada por tsk, después asigna tsk->thread_info al ti.
- Inicializa el contador para el nuevo descriptor del proceso (tsk->usage) a 2 para especificar que el descriptor del proceso esta en uso y indicando que el proceso está activo (su estado no es EXIT_ZOMBIE o EXIT_DEAD).
- Devuelve un puntero al descriptor del nuevo proceso (tsk).

```
static struct task_struct *dup_task_struct(struct task_struct *orig)
 209{
 210        struct task_struct *tsk;
 211        struct thread_info *ti;
 212        int err;
 213
 214        prepare_to_copy(orig);
 215
 216        tsk = alloc_task_struct();
 217        if (!tsk)
 218                return NULL;
```

```
 219
 220        ti = alloc_thread_info(tsk);
 221        if (!ti) {
 222                free_task_struct(tsk);
 223                return NULL;
 224        }
 225
 226        err = arch_dup_task_struct(tsk, orig);
 227        if (err)
 228                goto out;
 229
 230        tsk->stack = ti;
 231
 232        err = prop_local_init_single(&tsk->dirties);
 233        if (err)
 234                goto out;
 235
 236        setup_thread_stack(tsk, orig);
 237
 238#ifdef CONFIG_CC_STACKPROTECTOR
 239        tsk->stack_canary = get_random_int();
 240#endif
 241
 242        /* One for us, one for whoever does the "release_task()" (usually parent) */
 243         atomic_set(&tsk->usage,2);
 244        atomic_set(&tsk->fs_excl, 0);
 245#ifdef CONFIG_BLK_DEV_IO_TRACE
 246        tsk->btrace_seq = 0;
 247#endif
 248        tsk->splice_pipe = NULL;
 249        return tsk;
 250
 251out:
 252        free_thread_info(ti);
 253        free_task_struct(tsk);
 254        return NULL;
 255}
```

copy_files()

La función copy_files() crea una nueva estructura files_struct para el proceso hijo, inicializando los campos correspondientes y copiando la lista de ficheros abiertos del padre. Por otro lado, se actualiza el puntero files de la task_struct del proceso hijo para que apunte a la estructura files_struct recién creada, devolviendo cero si la operación se ha ejecutado correctamente.

Hay que notar que copy_files() puede no realizar una copia, sino dejar que el proceso padre e hijo compartan los ficheros abiertos. Esto se consigue con el flag CLONE_FILES.

```
/*La función copy_files() crea una nueva estructura files_struct para el proceso hijo,
inicializando los campos correspondientes y copiando (si procede) la lista de ficheros
abiertos del padre*/
static int copy_files(unsigned long clone_flags, struct task_struct * tsk)
{
	struct files_struct *oldf, *newf;

	//indicamos que no hay error
	int error = 0;

	/*En primer lugar se almacena en oldf la dirección de la files_struct del proceso actual.
Si ocurre un fallo al	almacenar dicha dirección, la función copy_files termina.*/

	oldf = current->files;
	if (!oldf)
		goto out;

/*Si el flag CLONE_FILES está activado, padre e hijo compartirán la misma estructura
files_struct y copy_files no creará ninguna nueva. Esto se ejecutará si se llama a clone()*/

	if (clone_flags & CLONE_FILES) {
		atomic_inc(&oldf->count);
		goto out;
	}

//copiar la structura de arhcivos del padre al hijo. Si se llama al fork se ejecutará esto
	newf = dup_fd(oldf, &error);
	if (!newf)
		goto out;

//Se le asigna a tsk->files (p->files) al hijo, por lo que completará la copia (fork)
	tsk->files = newf;
	error = 0;  //no hay error
out:
	return error;
}
```

Como se ha nombrado antes, La función copy_files(), crea la estructura files_struct, a continuación se explica la funcionalidad de esta:

```
659 static struct files_struct *dup_fd(struct files_struct *oldf, int *errorp)
660 {
661         struct files_struct *newf;
662         struct file **old_fds, **new_fds;
663         int open_files, size, i;
664         struct fdtable *old_fdt, *new_fdt;
665
666         *errorp = -ENOMEM;
```

Se reserva memoria para la nueva estructura.

```
667         newf = alloc_files();
668         if (!newf)
669                 goto out;
670
671         spin_lock(&oldf->file_lock);
```

```
672         old_fdt = files_fdtable(oldf);
673         new_fdt = files_fdtable(newf);
```

Se cuenta en número de ficheros abiertos del proceso padre.

```
674         open_files = count_open_files(old_fdt);
675
680         if (open_files > new_fdt->max_fds) {
681                 new_fdt->max_fds = 0;
682                 spin_unlock(&oldf->file_lock);
683                 spin_lock(&newf->file_lock);
684                 *errorp = expand_files(newf, open_files-1);
685                 spin_unlock(&newf->file_lock);
686                 if (*errorp < 0)
687                         goto out_release;
688                 new_fdt = files_fdtable(newf);
694                 spin_lock(&oldf->file_lock);
695                 old_fdt = files_fdtable(oldf);
696         }
697
698         old_fds = old_fdt->fd;
699         new_fds = new_fdt->fd;
700
```

Copia el vector de descriptores de fichero abiertos.

```
701         memcpy(new_fdt->open_fds->fds_bits,
702                 old_fdt->open_fds->fds_bits, open_files/8);
```

Copia el vector de descriptores de fichero a cerrar cuando el proceso haga un exec.

```
703         memcpy(new_fdt->close_on_exec->fds_bits,
704                 old_fdt->close_on_exec->fds_bits, open_files/8);
705
```

Copia la información de cada fichero abierto contenida en la estructura file (linux/file.h).

```
706         for (i = open_files; i != 0; i--) {
707                 struct file *f = *old_fds++;
708                 if (f) {
709                         get_file(f);
710                 } else {
717                         FD_CLR(open_files - i, new_fdt->open_fds);
718                 }
719                 rcu_assign_pointer(*new_fds++, f);
720         }
721         spin_unlock(&oldf->file_lock);
```

```
724         size = (new_fdt->max_fds - open_files) * sizeof(struct file *);
727         memset(new_fds, 0, size);
728
729         if (new_fdt->max_fds > open_files) {
730                 int left = (new_fdt->max_fds-open_files)/8;
731                 int start = open_files / (8 * sizeof(unsigned long));
732
733                 memset(&new_fdt->open_fds->fds_bits[start], 0, left);
734                 memset(&new_fdt->close_on_exec->fds_bits[start], 0, left);
735         }
736
737         return newf;
738
739 out_release:
740         kmem_cache_free(files_cachep, newf);
741 out:
742         return NULL;
743 }
```

Copy_fs()

En el núcleo existe una lista enlazada con los sistemas de ficheros montados. Cada sistema de ficheros está ligado a un directorio y tiene un descriptor correspondiente la estructura vsfmount. La única tarea que realiza la función copy_fs es invocar a la función __copy_fs_struct, que es la que hace todo el trabajo. Si el flag CLONE_FS está activado, padre e hijo compartirán la misma estructura fs_struct y copy_fs no creará ninguna nueva.

```
23 struct vfsmount
24 {
25         struct list_head mnt_hash;
26         struct vfsmount *mnt_parent;
27         struct dentry *mnt_mountpoint;
28         struct dentry *mnt_root;
29         struct super_block *mnt_sb;
30         struct list_head mnt_mounts;
31         struct list_head mnt_child;
32         atomic_t mnt_count;
33         int mnt_flags;
34         int mnt_expiry_mark;
35         char *mnt_devname;
36         struct list_head mnt_list;
37         struct list_head mnt_fslink;
38         struct namespace *mnt_namespace;
39 };
```

Esta función copia las estructuras vfsmount necesarias, que están a su vez contenidas en la estructura fs_struct. Como vemos contiene punteros al root del sistema de ficheros en el cual se ha ejecutado el proceso, así como el path en el cual se ha ejecutado.

```
7 struct fs_struct {
8       atomic_t count;
9       rwlock_t lock;
10        int umask;
11        struct dentry * root, * pwd, * altroot;
12        struct vfsmount * rootmnt, * pwdmnt, * altrootmnt;
13 };
```

La función copy_fs() llama a la función copy_fs_struct() que es la que realmente actualiza la información referente al sistema de ficheros en el cual se ha ejecutado el proceso, como vemos a continuación:

```
 static inline int copy_fs(unsigned long clone_flags, struct task_struct * tsk)
 609{
 610       if (clone_flags & CLONE_FS) {
 611             atomic_inc(&current->fs->count);
 612             return 0;
 613       }
 614       tsk->fs = __copy_fs_struct(current->fs);
 615       if (!tsk->fs)
 616             return -ENOMEM;
 617       return 0;
```

```
571 static inline struct fs_struct *__copy_fs_struct(struct fs_struct *old)
572 {
573       struct fs_struct *fs = kmem_cache_alloc(fs_cachep, GFP_KERNEL);
574
575       if (fs) {
576             atomic_set(&fs->count, 1);
577             rwlock_init(&fs->lock);
578             fs->umask = old->umask;
579             read_lock(&old->lock);
580             fs->rootmnt = mntget(old->rootmnt);
581             fs->root = dget(old->root);
582             fs->pwdmnt = mntget(old->pwdmnt);
583             fs->pwd = dget(old->pwd);
584             if (old->altroot) {
585                   fs->altrootmnt = mntget(old->altrootmnt);
586                   fs->altroot = dget(old->altroot);
587             } else {
588                   fs->altrootmnt = NULL;
```

```
589                     fs->altroot = NULL;
590             }
591             read_unlock(&old->lock);
592         }
593         return fs;
594 }
```

Copy_sighand()

Copia los descriptores de acciones asociados a señales que tiene el proceso padre, en el proceso hijo. La estructura task_struct tiene un campo puntero a sighand_struct (struct sighand_struct *sighand) que es la que se copia en el nuevo proceso durante el fork:

```
261   struct sighand_struct {
262         atomic_t            count;
263         struct k_sigaction     action[_NSIG];
264         spinlock_t          siglock;
265   };
```

La función copy_sighand reserva memoria para la nueva estructura, la copia y devuelve cero si todo ha sido correcto.

```
 819      sig = kmem_cache_alloc(sighand_cachep, GFP_KERNEL);
 824      memcpy(sig->action, current->sighand->action, sizeof(sig-> action));
 825      return 0;
```

A continuación mostramos el código completo de la función **copy_sighand()**

```
/*Copia los descriptores de acciones asociados a cada señal que tiene el proceso
padre, en el proceso hijo.*/
static int copy_sighand(unsigned long clone_flags, struct task_struct *tsk)
{

/*La estructura task_struct tiene un campo puntero a sighand_struct (struct
sighand_struct sighand) que es la que se copia en el nuevo proceso durante el fork.*/

      struct sighand_struct *sig;
      if (clone_flags & (CLONE_SIGHAND | CLONE_THREAD)) {
            atomic_inc(&current->sighand->count);
            return 0;
      }
/*La func
ión copy_sighand reserva memoria para la nueva estructura, la copia y devuelve cero
si todo ha sido correcto.*/

      sig = kmem_cache_alloc(sighand_cachep, GFP_KERNEL);
      rcu_assign_pointer(tsk->sighand, sig);
      if (!sig)
            return -ENOMEM;
```

```
        atomic_set(&sig->count, 1);
        memcpy(sig->action, current->sighand->action, sizeof(sig->action));
        return 0;
}
```

Copy_signal()

Cada proceso tiene una estructura que define campos con información asociada a señales, como por ejemplo, límites de recursos, lista de señales pendientes de atender, cola de hijos lanzados, etc. Esta información se guarda en la estructura signal_struct, habiendo en la task_struct correspondiente a cada proceso un puntero a esta estructura:

```
399 struct signal_struct {
400         atomic_t            count;
401         atomic_t            live;
402
403         wait_queue_head_t       wait_chldexit;
406         struct task_struct      *curr_target;
409         struct sigpending       shared_pending;
412         int                 group_exit_code;
418         struct task_struct      *group_exit_task;
419         int                 notify_count;
422         int                 group_stop_count;
423         unsigned int            flags;
426         struct list_head posix_timers;
429         struct hrtimer real_timer;
430         struct task_struct *tsk;
431         ktime_t it_real_incr;
434         cputime_t it_prof_expires, it_virt_expires;
435         cputime_t it_prof_incr, it_virt_incr;
438         pid_t pgrp;
439         pid_t tty_old_pgrp;
440
441         union {
442             pid_t session __deprecated;
443             pid_t __session;
444         };
445
447         int leader;
449         struct tty_struct *tty;
450
457         cputime_t utime, stime, cutime, cstime;
458         unsigned long nvcsw, nivcsw, cnvcsw, cnivcsw;
```

```
459         unsigned long min_flt, maj_flt, cmin_flt, cmaj_flt;
460
467         unsigned long long sched_time;
478         struct rlimit rlim[RLIM_NLIMITS];
480         struct list_head cpu_timers[3];
481
484 #ifdef CONFIG_KEYS
485         struct key *session_keyring;
486         struct key *process_keyring;
487 #endif
488 #ifdef CONFIG_BSD_PROCESS_ACCT
489         struct pacct_struct pacct;
490 #endif
491 #ifdef CONFIG_TASKSTATS
492         struct taskstats *stats;
493 #endif
494 };
```

La función copy_signal inicializa todos estos campos, exceptuando los límites sobre los recursos que el nuevo proceso puede consumir, ya que estos los hereda del padre.

```
858     sig->flags = 0;
859     sig->group_exit_code = 0;
860     sig->group_exit_task = NULL;
861     sig->group_stop_count = 0;
862     sig->curr_target = NULL;
```

```
871     sig->it_virt_expires = cputime_zero;
872     sig->it_virt_incr = cputime_zero;
873     sig->it_prof_expires = cputime_zero;
874     sig->it_prof_incr = cputime_zero;
875
876     sig->leader = 0;        /* session leadership doesn't inherit */
877     sig->tty_old_pgrp = NULL;
```

A continuación mostramos el código completo de la función copy_signal()

```
/*La función copy_signal inicializa todos estos campos, exceptuando los límites sobre los recursos que el nuevo proceso puede consumir, ya que estos los hereda del padre.*/
static int copy_signal(unsigned long clone_flags, struct task_struct *tsk)
{
        struct signal_struct *sig;
        int ret;
        if (clone_flags & CLONE_THREAD) {
                ret = thread_group_cputime_clone_thread(current);
                if (likely(!ret)) {
                        atomic_inc(&current->signal->count);
                        atomic_inc(&current->signal->live);
                }
                return ret;
```

```
        }
        sig = kmem_cache_alloc(signal_cachep, GFP_KERNEL);
        tsk->signal = sig;
        if (!sig)
                return -ENOMEM;
        ret = copy_thread_group_keys(tsk);
        if (ret < 0) {
                kmem_cache_free(signal_cachep, sig);
                return ret;
        }
        atomic_set(&sig->count, 1);
        atomic_set(&sig->live, 1);
        init_waitqueue_head(&sig->wait_chldexit);
        sig->flags = 0;
        sig->group_exit_code = 0;
        sig->group_exit_task = NULL;
        sig->group_stop_count = 0;
        sig->curr_target = tsk;
        init_sigpending(&sig->shared_pending);
        INIT_LIST_HEAD(&sig->posix_timers);
        hrtimer_init(&sig->real_timer, CLOCK_MONOTONIC,
HRTIMER_MODE_REL);
        sig->it_real_incr.tv64 = 0;
        sig->real_timer.function = it_real_fn;
        sig->leader = 0;        /* session leadership doesn't inherit */
        sig->tty_old_pgrp = NULL;
        sig->tty = NULL;
        sig->cutime = sig->cstime = cputime_zero;
        sig->gtime = cputime_zero;
        sig->cgtime = cputime_zero;
        sig->nvcsw = sig->nivcsw = sig->cnvcsw = sig->cnivcsw = 0;
        sig->min_flt = sig->maj_flt = sig->cmin_flt = sig->cmaj_flt = 0;
        sig->inblock = sig->oublock = sig->cinblock = sig->coublock = 0;
        task_io_accounting_init(&sig->ioac);
        taskstats_tgid_init(sig);
        task_lock(current->group_leader);
        memcpy(sig->rlim, current->signal->rlim, sizeof sig->rlim);
        task_unlock(current->group_leader);
        posix_cpu_timers_init_group(sig);

        acct_init_pacct(&sig->pacct);
        tty_audit_fork(sig);
        return 0;
}
```

Copy_mm()

Hace una copia de la región de memoria del padre en el hijo, es decir, copia el contendido del espacio de direccionamiento del proceso. El núcleo mantiene en memoria una descripción de las regiones utilizadas por un proceso, ya que el espacio de direccionamiento de los procesos puede estar formado por varias regiones.

A su vez, Linux mantiene un descriptor del espacio de direccionamiento accesible por el campo mm contenido en el descriptor del proceso (task_struct). Cada proceso posee normalmente un descriptor propio, pero dos procesos clones pueden compartir el mismo descriptor según el flag CLONE_VM. Este descriptor del espacio de direccionamiento viene dado por la estructura mm_struct.

La función copy_mm() duplica estos descriptores haciendo una copia de cada una de las regiones del proceso padre para el proceso hijo, a menos que el flag CLONE_VM defina que las regiones serán compartidas.

```
/*La función copy_mm() hace una copia de la región de memoria del padre en el hijo, es decir, copia el contendido del espacio de direccionamiento del proceso. */

static int copy_mm(unsigned long clone_flags, struct task_struct * tsk)
{
	struct mm_struct * mm, *oldmm;
	int retval;

	tsk->min_flt = tsk->maj_flt = 0;
	tsk->nvcsw = tsk->nivcsw = 0;

/*Inicializamos el espacio de direccionamiento a null*/
	tsk->mm = NULL;
	tsk->active_mm = NULL;

/*Si estamos clonando simplemente se hace una asignación de punteros*/
	oldmm = current->mm;
	if (!oldmm)
		return 0;

	if (clone_flags & CLONE_VM) {
		atomic_inc(&oldmm->mm_users);
		mm = oldmm;
		goto good_mm;
	}

//En el caso de que no se clone hacemos una copia del espacio de direccionamiento
	retval = -ENOMEM;
	mm = dup_mm(tsk);
	if (!mm)
		goto fail_nomem;
good_mm:

	    /* Initializing for Swap token stuff */
	mm->token_priority = 0;
	mm->last_interval = 0;
```

```
        //Hacemos la asignación del tsk (p) ya sea si fue por el camino del clone o el del fork
      tsk->mm = mm;
      tsk->active_mm = mm;
      return 0;
fail_nomem:
      return retval;
}
```

dup_mm

La función dup_mm asigna una nueva estructura mm y rellena su contenido desde la estructura mm pasada dentro de task_ struct.

```
 570struct mm_struct *dup_mm(struct task_struct *tsk)
 571{
 572        struct mm_struct *mm, *oldmm = current->mm;
 573        int err;
 574
 575        if (!oldmm)
 576                return NULL;
 577
 578        mm = allocate_mm();
 579        if (!mm)
 580                goto fail_nomem;
 581
 582        memcpy(mm, oldmm, sizeof(*mm));
 583
 584        /* Initializing for Swap token stuff */
 585        mm->token_priority = 0;
 586        mm->last_interval = 0;
 587
 588        if (!mm_init(mm, tsk))
 589                goto fail_nomem;
 590
 591        if (init_new_context(tsk, mm))
 592                goto fail_nocontext;
 593
 594        dup_mm_exe_file(oldmm, mm);
 595
 596        err = dup_mmap(mm, oldmm);
 597        if (err)
 598                goto free_pt;
 599
 600        mm->hiwater_rss = get_mm_rss(mm);
 601        mm->hiwater_vm = mm->total_vm;
 602
 603        return mm;
 604
 605free_pt:
 606        mmput(mm);
 607
 608fail_nomem:
 609        return NULL;
 610
 611fail_nocontext:
 616        mm_free_pgd(mm);
```

```
 617        free_mm(mm);
 618        return NULL;
 619}
```

7.6 BIBLIOGRAFÍA

Linux cross reference

http://lxr.linux.no/

Versión del núcleo 2.6.28.7

Understandind the Linux Kernel (3ª Edición)

Daniel P. Bovet, Marco Cesati

Ed. O'Reilly

2005

Maxvell

Remy Card