LECCIÓN 8: EXIT Y WAIT

8.1 Finalización de un proceso. EXIT

La llamada al sistema exit es la primera en sys_call_table y es la función que termina con la ejecución de un proceso. Esta función se encarga de retirar los recursos que está utilizando el proceso, así como dejarlo preparado para su posterior eliminación.

Cuando un proceso termina debe de comunicar a su padre su finalización por medio de la señal SIGCHLD, de forma que una vez el padre haya sido informado y realice un wait, el hijo sea totalmente eliminado y quitado del planificador. Para reflejar este hecho se denomina al estado transitorio entre la comunicación de finalización y la eliminación total como estado "zombie".

Además de notificar al padre, el exit se encarga de liberar recursos como la memoria tomada por el proceso, así como el sistema de ficheros y las entradas en el mismo y los registros de estado del procesador. Si dicho proceso no tuviera padre, ya que este acabó antes que él, se eliminaría directamente del planificador en la llamada exit y no habría que esperar a comunicar su estado de terminación a nadie.

Muestra de uso

```
/*Programa en C*/

void main ()
{
   ...
   exit (código de error);
   ...
}
```

La llamada *exit* provoca que el proceso termine y le mande *(código de error)* al programa que invocó a "Programa en C".

8.1.1. ¿Cómo se ejecuta la llamada al sistema exit?

¿Qué secuencia de eventos tiene lugar desde que se encuentra una llamada al sistema exit en el código hasta que se empieza a ejecutar do_exit, la función principal del archivo exit.c?. Veámoslo paso a paso como una interrupción software que es:

1. La función exit 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_ exit.
4. La función sys_ exit llama a la función do_ exit, que es la función principal del archivo exit.c.

8.1.2. Descripción

Las principales tareas que lleva a cabo esta llamada el sistema son:

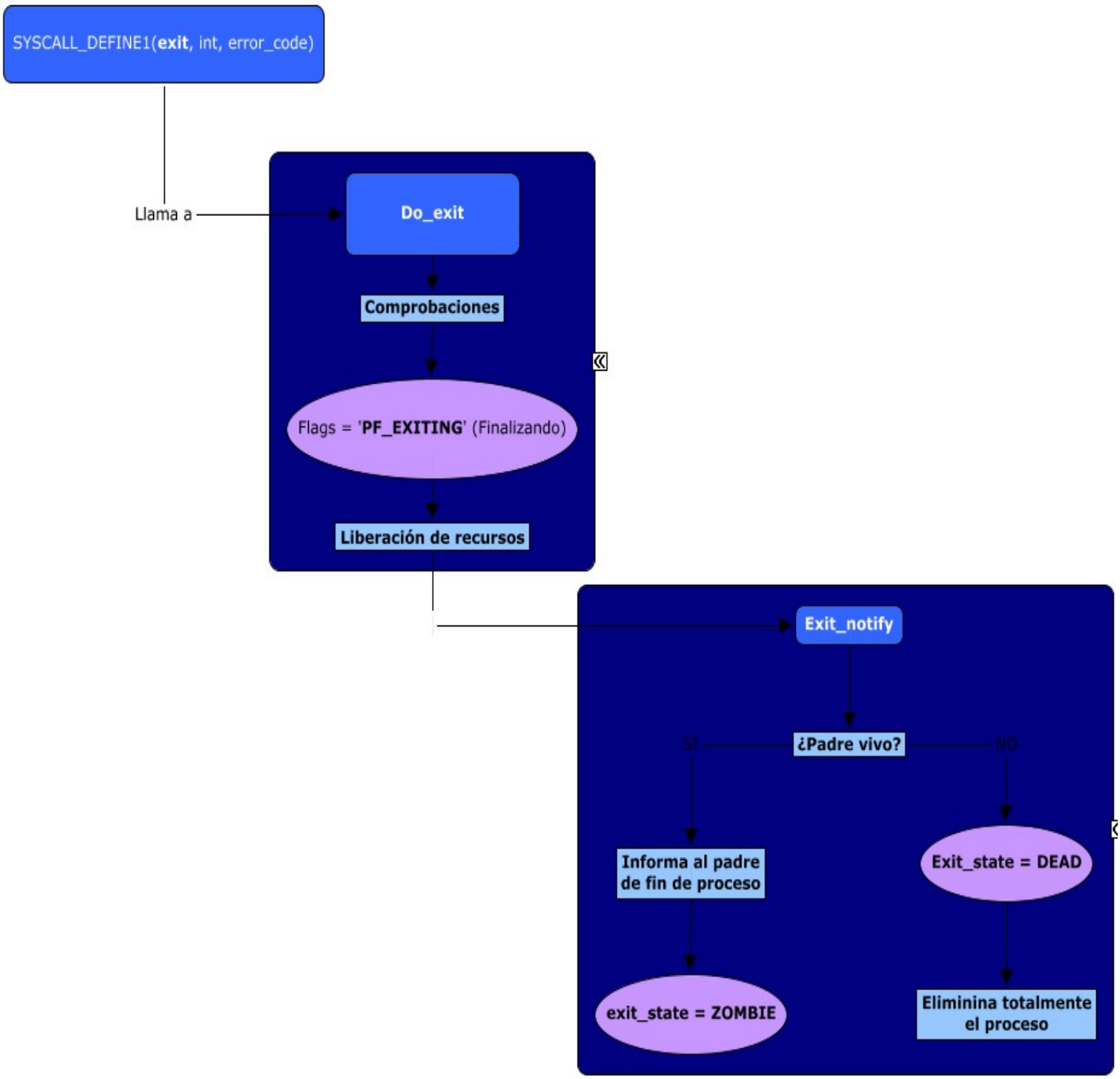
- Especifica que el proceso está finalizando.
  - Escribe el “exit_code”
- Liberación de recursos:
  - Elimina el temporizador del proceso

- ❖ Elimina la memoria asociada al proceso
- ❖ Elimina el proceso de las colas de los semáforos
- ❖ Cierra todos los ficheros que el proceso tenga abiertos
- ❖ Elimina la información del sistema de ficheros
- ❖ Elimina los registros de estado guardados
- ❖ Abre todos los cerrojos que haya cerrado el proceso
- ❖ Libera el enlace entre el proceso y el terminal
- ❖ Elimina las dependencias y entradas del sistema de ficheros
- ❖ Saca al proceso del grupo o subgrupo al que pertenece

- ➢ Notifica al padre de su terminación y establece el exit_state del proceso como "Zombie"; o si no está el padre, lo finaliza del todo (exit_state se establece a DEAD).
- ➢ Ejecuta el planificador

La función principal donde se realizan todas estas acciones es **do_exit**, ésta llama a **exit_notify** para notificar al padre de la finalización de uno de sus hijos. Y en caso de que el padre ya haya finalizado su ejecución antes que el hijo, do_exit llama a **release_task** para eliminar por completo el proceso hijo de la lista de procesos.

8.1.3. Esquema general del proceso



```
//------------------------------------------------------------------//
//                          kernel/exit.c                           //
//------------------------------------------------------------------//

//====================================================================
// Nombre: SYSCALL_DEFINE1(exit, int, error_code)
// Cometido: Convierte el "exit_code" al formato que espera la función
//              do_exit, y la invoca.
// Ubicación: linux/kernel/exit.c
//====================================================================


1146 SYSCALL_DEFINE1(exit, int, error_code)
1147 {
1148   do_exit((error_code&0xff)<<8);
1149 }

//====================================================================
// Nombre: do_exit(long code)
// Cometido: Libera recursos e invoca a "exit_notify" para notificar al
//              padre el exit de su hijo.
// Ubicación: linux/kernel/exit.c
//====================================================================


 998 NORET_TYPE void do_exit(long code)
 999 {
         //Recupera la task_struct del proceso a eliminar (Actual)
1000        struct task_struct *tsk = current;
1001        int group_dead;
1002
1003        profile_task_exit(tsk);
1004
1005        WARN_ON(atomic_read(&tsk->fs_excl));
1006
           // Comprobaciones:
           // Comprueba que el Proceso a Eliminar no sea ni
           // el manejador de interrupciones, ni el idle task.
           // Si no, se detiene el sistema.
1007        if (unlikely(in_interrupt()))
1008                panic("Aiee, killing interrupt handler!");
1009        if (unlikely(!tsk->pid))
1010                panic("Attempted to kill the idle task!");
1011
         // Le da un valor al campo exit_code del task_struc
1012        tracehook_report_exit(&code);
1013
1014        /*
1015         * We're taking recursive faults here in do_exit. Safest is to just
1016         * leave this task alone and wait for reboot.
1017         */
           // Verificamos que no estemos ante un fallo por recursividad
         // (el proceso actual ya estaba finalizándose previamente)
1018        if (unlikely(tsk->flags & PF_EXITING)) {
1019                printk(KERN_ALERT
```

```
1020                    "Fixing recursive fault but reboot is needed!\n");
```

```
1021                /*
1022                 * We can do this unlocked here. The futex code uses
1023                 * this flag just to verify whether the pi state
1024                 * cleanup has been done or not. In the worst case it
1025                 * loops once more. We pretend that the cleanup was
1026                 * done as there is no way to return. Either the
1027                 * OWNER_DIED bit is set by now or we push the blocked
1028                 * task into the wait for ever nirwana as well.
1029                 */
1030                tsk->flags |= PF_EXITPIDONE;
                   // Si está esperando operaciones de E/S, se elimina
                   // el descriptor de ésta.
1031                if (tsk->io_context)
1032                        exit_io_context();
                 // Cambiamos el estado del proceso y llamamos al
                 // planificador
1033                set_current_state(TASK_UNINTERRUPTIBLE);
1034                schedule();
1035        }
1036     // IMPORTANTE
          // Cambia el estado del proceso a PF_EXITING (Finalizando)
1037        exit_signals(tsk);  /* sets PF_EXITING */
1038        /*
1039         * tsk->flags are checked in the futex code to protect against
1040         * an exiting task cleaning up the robust pi futexes.
1041         */
1042        smp_mb(); // se utiliza en caso que haya varias CPU

1043        spin_unlock_wait(&tsk->pi_lock);
1044

          // Si la ejecución no es atómica, mostramos un mensaje de error
1045        if (unlikely(in_atomic()))
1046                printk(KERN_INFO "note: %s[%d] exited with preempt_count
%d\n",
1047                                current->comm, task_pid_nr(current),
1048                                preempt_count());
1049
1050        acct_update_integrals(tsk);
1051        if (tsk->mm) {
1052                update_hiwater_rss(tsk->mm);
1053                update_hiwater_vm(tsk->mm);
1054        }
          // IMPORTANTE: Comenzamos a liberar recursos
1055        group_dead = atomic_dec_and_test(&tsk->signal->live);
1056        if (group_dead) {
                   // Eliminamos el Temporizador del proceso
1057                hrtimer_cancel(&tsk->signal->real_timer);
1058                exit_itimers(tsk->signal);
1059        }
1060        acct_collect(code, group_dead);
1061        if (group_dead)
1062                tty_audit_exit();
```

```
1063        if (unlikely(tsk->audit_context))
1064                audit_free(tsk);
1065
          // Se escribe en la estructura del proceso el código de terminación
1066        tsk->exit_code = code;
1067        taskstats_exit(tsk, group_dead);
1068
          // Liberamos el espacio de memoria utilizado por el proceso
1069        exit_mm(tsk);
1070
1071        if (group_dead)
1072                acct_process();
1073        trace_sched_process_exit(tsk);
1074
  //          Eliminamos el proceso de todas las colas de semáforos por
  //          las que pudiera estar esperando
1075        exit_sem(tsk);
  //          Cierra todos los ficheros que el proceso tenga abiertos
1076        exit_files(tsk);
  //          Elimina la información sobre el sistema de ficheros
1077        exit_fs(tsk);
  //          Comprueba si comparte la pila de ejecución con otro proceso
1078        check_stack_usage();
  //          Elimina el hilo Actual (Proceso a eliminar). Liberando los
  //          registros usados en el procesador.
1079        exit_thread();
  //      Saca al proceso del grupo o subgrupo al que pertenece
1080        cgroup_exit(tsk, 1);
  //          Libera todos los cerrojos que el proceso haya cerrado
1081        exit_keys(tsk);
1082
1083        if (group_dead && tsk->signal->leader)
  //          Se Libera el Enlace Entre el Proceso y el Terminal
1084                disassociate_ctty(1);
1085
1086        module_put(task_thread_info(tsk)->exec_domain->module);
1087        if (tsk->binfmt)
1088                module_put(tsk->binfmt->module);
1089
1090        proc_exit_connector(tsk);
  //        IMPORTANTE: Notifica al padre de la eliminación de su hijo
1091        exit_notify(tsk, group_dead);
1092 #ifdef CONFIG_NUMA
1093        mpol_put(tsk->mempolicy);
1094        tsk->mempolicy = NULL;
1095 #endif
1096 #ifdef CONFIG_FUTEX
1097        /*
1098         * This must happen late, after the PID is not
1099         * hashed anymore:
1100         */
1101        if (unlikely(!list_empty(&tsk->pi_state_list)))
```

```
1102                exit_pi_state_list(tsk);
1103        if (unlikely(current->pi_state_cache))
1104                kfree(current->pi_state_cache);
1105 #endif
1106        /*
1107         * Make sure we are holding no locks:
1108         */
1109        debug_check_no_locks_held(tsk);
1110        /*
1111         * We can do this unlocked here. The futex code uses this flag
1112         * just to verify whether the pi state cleanup has been done
1113         * or not. In the worst case it loops once more.
1114         */
1115        tsk->flags |= PF_EXITPIDONE;
1116
```

```
  //          Si esta esperando operaciones de E/S, se elimina el
  //          descriptor de esta
1117        if (tsk->io_context)
1118                exit_io_context();
1119
1120        if (tsk->splice_pipe)
1121                __free_pipe_info(tsk->splice_pipe);
1122
1123        preempt_disable();
1124        /* causes final put_task_struct in finish_task_switch(). */
  //          Cambia el estado del proceso a TASK_DEAD
1125        tsk->state = TASK_DEAD;
1126
  //          Invoca al planificador, Esta será la ultima vez que este
  //          proceso tenga la CPU
1127        schedule();
1128        BUG();
1129        /* Avoid "noreturn function does return".  */
1130        for (;;)
1131                cpu_relax();    /* For when BUG is null */
1132 }
```

8.2 Funciones auxiliares de exit

```
//=========================================================================
// Nombre: exit_notify(struct task_struct *tsk)
// Cometido: Notifica al padre de la finalización de uno de sus hijos.
// Ubicación: linux/kernel/exit.c
```

```
//=====================================================================

 909 static void exit_notify(struct task_struct *tsk, int group_dead)
 910 {
 911         int signal;
 912         void *cookie;
 913
 914         /*
 915          * This does two things:
 916          *
 917          * A.  Make init inherit all the child processes
 918          * B.  Check to see if any process groups have become orphaned
 919          *      as a result of our exiting, and if they have any stopped
 920          *      jobs, send them a SIGHUP and then a SIGCONT. (POSIX
3.2.2.2)
 921          */
 922         forget_original_parent(tsk);
 923         exit_task_namespaces(tsk);
 924
 925         write_lock_irq(&tasklist_lock);
 926         if (group_dead)
 927                 kill_orphaned_pgrp(tsk->group_leader, NULL);
 928
```

```
 929         /* Let father know we died
 930          *
 931          * Thread signals are configurable, but you aren't going to use
 932          * that to send signals to arbitary processes.
 933          * That stops right now.
 934          *
 935          * If the parent exec id doesn't match the exec id we saved
 936          * when we started then we know the parent has changed security
 937          * domain.
 938          *
 939          * If our self_exec id doesn't match our parent_exec_id then
 940          * we have changed execution domain as these two values started
 941          * the same after a fork.
 942          */
           //IMPORTANTE: Si el proceso padre se encuentra aun en
           // ejecución se le notifica la finalización de su hijo
           // mediante "do_notify_parent" con la señal de salida adecuada.
 943         if (tsk->exit_signal != SIGCHLD && !task_detached(tsk) &&
 944             (tsk->parent_exec_id != tsk->real_parent->self_exec_id
||
 945              tsk->self_exec_id != tsk->parent_exec_id) &&
 946             !capable(CAP_KILL))
 947                 tsk->exit_signal = SIGCHLD;
 948
 949         signal = tracehook_notify_death(tsk, &cookie, group_dead);
 950         if (signal >= 0)
 951                 signal = do_notify_parent(tsk, signal);
 952
          // Primero pone el exit_state en estado DEAD y luego, una vez
          // liberados los recursos, en estado ZOMBIE
 953         tsk->exit_state = signal == DEATH_REAP ? EXIT_DEAD :
EXIT_ZOMBIE;
 954
 955         /* mt-exec, de_thread() is waiting for us */
 956         if (thread_group_leader(tsk) &&
 957             tsk->signal->group_exit_task &&
 958             tsk->signal->notify_count < 0)
 959                 wake_up_process(tsk->signal->group_exit_task);
 960
 961         write_unlock_irq(&tasklist_lock);
 962
 963         tracehook_report_death(tsk, signal, cookie, group_dead);
 964
           //          IMPORTANTE: Si el proceso está muerto lo elimina por
           //          completo con "release_task"
 965         /* If the process is dead, release it - nobody will wait for it */
 966         if (signal == DEATH_REAP)
 967                 release_task(tsk);
 968 }
```

```
//=========================================================================
// Nombre: release_task(struct task_struct * p)
// Cometido: Elimina por completo el proceso de la lista de procesos
// Ubicación: linux/kernel/exit.c
//=========================================================================


 161 void release_task(struct task_struct * p)
 162 {
 163         struct task_struct *leader;
 164         int zap_leader;
 165 repeat:
 166         tracehook_prepare_release_task(p);
 167         atomic_dec(&p->user->processes);
 168         proc_flush_task(p);
 169         write_lock_irq(&tasklist_lock);
 170         tracehook_finish_release_task(p);
   //          Libera la información del proceso. Entre otras cosas,
   //          libera la entrada pid de pidtask (llamando internamente a
   //          la función unhash_process) y los descriptores de acciones
   //          asociados a señales (llamando internamente a la función
   //          cleanup_sighand)
 171         __exit_signal(p);
 172
 173         /*
 174          * If we are the last non-leader member of the thread
 175          * group, and the leader is zombie, then notify the
 176          * group leader's parent process. (if it wants notification.)
 177          */
 178         zap_leader = 0;
 179         leader = p->group_leader;
 180         if (leader != p && thread_group_empty(leader) && leader-
>exit_state == EXIT_ZOMBIE){
 181                 BUG_ON(task_detached(leader));
 182                 do_notify_parent(leader, leader->exit_signal);
 183                 /*
 184                  * If we were the last child thread and the leader has
 185                  * exited already, and the leader's parent ignores SIGCHLD,
 186                  * then we are the one who should release the leader.
 187                  *
 188                  * do_notify_parent() will have marked it self-reaping in
 189                  * that case.
 190                  */
 191                 zap_leader = task_detached(leader);
 192
 193                 /*
 194                  * This maintains the invariant that release_task()
 195                  * only runs on a task in EXIT_DEAD, just for sanity.
 196                  */
 197                 if (zap_leader)
 198                         leader->exit_state = EXIT_DEAD;
 199         }
```

```
 200
 201        write_unlock_irq(&tasklist_lock);
 202        release_thread(p);
  //         Libera la memoria usada por la task_struct
 203        call_rcu(&p->rcu, delayed_put_task_struct);
 204
 205        p = leader;
 206        if (unlikely(zap_leader))
 207                goto repeat;
 208}
```

Al considerarse importante la forma en la que se libera la memoria de la task_struct se comenta en las siguientes líneas los enlaces a las funciones necesarias para llevarlo a cabo. La llamada:

```
203        call_rcu(&p->rcu, delayed_put_task_struct);
```

```
//=========================================================================
// Nombre: delayed_put_task_struct(struct rcu_head *rhp)
// Cometido: Invocar a " put_task_struct" para liberar la memoria del
//              task_struct
// Ubicación: linux/kernel/exit.c
//=========================================================================
```

```
 152 static void delayed_put_task_struct(struct rcu_head *rhp)
 153{
 154        struct task_struct *tsk = container_of(rhp, struct
task_struct, rcu);
 155
 156        trace_sched_process_free(tsk);
 157        put_task_struct(tsk);
 158}
```

```
//=========================================================================
// Nombre: put_task_struct(struct task_struct *t)
// Cometido: Invoca a "__put_task_struct" para realizar la liberación
// Ubicación: linux/include/linux/sched.h
//=========================================================================

1526 static inline void put_task_struct(struct task_struct *t)
1527 {
1528        if (atomic_dec_and_test(&t->usage))
1529                __put_task_struct(t);
1530 }
```

```
//=====================================================================
// Nombre: __put_task_struct(struct task_struct *tsk)
// Cometido: Libera la task_struct del proceso
// Ubicación: linux/kernel/fork.c
//=====================================================================


 144 void __put_task_struct(struct task_struct *tsk)
 145 {
 146         WARN_ON(!tsk->exit_state);
 147         WARN_ON(atomic_read(&tsk->usage));
 148         WARN_ON(tsk == current);
 149
   //          Comprueba si es seguro liberar el task_struct
 150         security_task_free(tsk);
   //          Libera el identificador del propietario del proceso
 151         free_uid(tsk->user);
 152         put_group_info(tsk->group_info);
 153         delayacct_tsk_free(tsk);
 154
   //          Libera la memoria del task_struct
 155         if (!profile_handoff_task(tsk))
 156                 free_task(tsk);
 157 }


//=====================================================================
// Nombre: free_task(struct task_struct *tsk)
// Cometido: Libera la memoria del task_struct
// Ubicación: linux/kernel/fork.c
//=====================================================================


135void free_task(struct task_struct *tsk)
 136{
 137         prop_local_destroy_single(&tsk->dirties);
   //          Libera la información de la pila del proceso
 138         free_thread_info(tsk->stack);
 139         rt_mutex_debug_task_free(tsk);
   //          Libera la memoria del núcleo ocupada por el task_struct
 140         free_task_struct(tsk);
 141}
```

8.3 Esperar por un proceso. Wait

La función **wait** suspende la ejecución del proceso actual hasta que un proceso

hijo haya terminado, o hasta que se produce una señal cuya acción es terminar el proceso actual o llamar a la función manejadora de dicha señal. Si un hijo ya había terminado (está en estado Zombie) la función wait vuelve inmediatamente y se encarga de eliminar de la tabla de procesos al hijo e informar al padre acerca de dicha eliminación.

En el núcleo de Linux, un hijo planificado por el núcleo no es una construcción distinta a un proceso. En su lugar, un hilo es simplemente un proceso que es creado usando la llamada al sistema única en Linux clone(2) ; otras rutinas como la llamada portable pthread_create(3) son implementadas usando clone(2). Desde la versión 2.4. de Linux un hilo puede, y por defecto lo hará, esperar a hijos de otros hilos en el mismo grupo de hilos.

Se puede esperar por un hijo mediante una familia de funciones que tienen como función principal esperar hasta que el estado de los hijos lanzados cambie y retornar información de dicho proceso. El cambio de estado e hijos por los que esperar viene determinado por las opciones de la llamada al sistema.

Las llamadas tiene esta sintaxis:

pid_t wait(int ****status*);**
pid_t waitpid(pid_t *pid***, int ****status***, int** *options***);**
int waitid(idtype_t *idtype***, id_t** *id* , **siginfo_t** * *infop* , **int** *options*);

La función **waitpid** suspende la ejecución del proceso en curso hasta que un hijo especificado por el argumento pid ha terminado, o hasta que se produce una señal cuya acción es finalizar el proceso actual o llamar a la función manejadora de la señal.

int ***status: puntero donde debe devolver información el hijo.*

***pid_t** pid: Un valor tal que:*

< -1: Esperar por cualquier proceso cuyo Process Group ID sea igual al valor absoluto de pid.

-1: Esperar por cualquier hijo.

0: Esperar por cualquier hijo cuyo Process Group ID sea igual al del proceso llamador.

> 0: Esperar por el hijo cuyo PID sea igual al valor de pid.

***int** options: Una combinación de las siguientes flags:*

***WEXITED**: Espera por hijos que hayan terminado.*

***WSTOPPED:** Espera por hijos que hayan sido parados por recibir una señal.*

***WNOHANG**: No esperar por un hijo que esta en ejecución.*

***WNOWAIT:** dejar al hijo sin modificar ni marcar en la tabla de procesos, tal que una posterior llamada se comportaría como si no hubiesemos hecho wait por dicho hijo.*

***WUNTRACED**: No esperar si el hijo está parado a no ser que este siendo trazado.*

***WCONTINUED**: Volver si un hijo ha continuado ejecutándose tras mandarle la señal SIGCONT (desde 2.6.10).*

*Las flags **WUNTRACED** y **WCONTINUED** solo son efectivas si SA_NOCLDSTOP no ha sido establecida para la señal **SIGCHLD**.*

*La llamada **waitid**() está disponible desde la versión 2.6.9 para dar más control sobre qué hijo y por qué cambio debemos esperar, para dicha llamada idtype y id_arguments seleccionan al hijo tal que:*

idtype == P_PID: Espera por el hijo con esa ID.

idtype == P_PGID: Espera por el hijo cuyo process group ID coincida.

idtype == P_ALL: Espera por cualquier hijo.

Si tiene éxito la función devuelve una estructura siginfo_t en infop con la siguiente información:

si_pid: PID del hijo.

si_uid: UID del hijo

si_signo: Siempre puesto a SIGCHLD.

si_status: El estado de salida del hijo o la señal que le hizo acabar, pararse o continuar.

si_code: Puesto a CLD_EXITED si el hijo salió, CLD_KILLED si el hijo fue terminado por una señal kill, CLD_STOPPED si el hijo fué parado o CLD_CONTINUED si el hijo continuó al llegarle una señal SIGCONT.

*Si **status** no es NULL, wait o waitpid almacena la información de estado en la memoria apuntada por status.*

Si el estado puede ser evaluado con las siguientes macros (dichas macros toman el buffer stat (un int) como argumento — ¡no un puntero al buffer!):

***WIFEXITED**(status) es distinto de cero si el hijo terminó normalmente.*

***WEXITSTATUS**(status) evalúa los ocho bits menos significativos del código de retorno del hijo que terminó, que podrían estar activados como el argumento de una llamada a exit() o como el argumento de un return en el programa principal. Esta macro solamente puede ser tenida en cuenta si WIFEXITED devuelve un valor distinto de cero.*

***WIFSIGNALED**(status) devuelve true si el proceso hijo terminó a causa de una señal nocapturada.*

***WTERMSIG**(status) devuelve el número de la señal que provocó la muerte del proceso hijo. Esta macro sólo puede ser evaluada si WIFSIGNALED devolvió un valor distinto de cero.*

***WIFSTOPPED**(status) devuelve true si el proceso hijo que provocó el retorno está actualmente pardo; esto solamente es posible si la llamada se hizo usando WUNTRACED o cuando el hijo está siendo rastreado (vea ptrace(2)).*

***WSTOPSIG**(status) devuelve el número de la señal que provocó la parada del hijo. Esta macro solamente puede ser evaluada si WIFSTOPPED devolvió un valor distinto de cero. Algunas versiones de Unix (p.e. Linux, Solaris, pero no AIX ni SunOS) definen también una macro WCOREDUMP(status) para comprobar si el proceso hijo provocó un volcado de memoria. Utilícela solamente encerrada entre #ifdef WCOREDUMP ... #endif.*

wait y **waitpid** devuelven el identificador del proceso hijo que ha finalizado. En caso de error devuelven -1. Si se hace uso de **waitpid** con la opción **WNOHANG** y ningún hijo estaba disponible, se devuelve 0. **waitid** devuelve 0 si tuvo éxito o si se utilizó **WNOHANG** y no había hijos por los que esperar. Si hubo error devuelve -1.

La llamada **waitid** es nueva (apareció en el kernel 2.6.9) y permite mayor control sobre por qué hijo esperar ya que incorpora nuevos flags en **idtype_t** *idtype,* y devuelve más información del hijo en la estructura **siginfo_t***).*

En resumen, **wait** solo permite esperar por hijos que hayan terminado, **waitpid** por hijos que hayan terminado/parado/continuado y **waitid** permite mayor control sobre por qué hijo esperar y más información de retorno.

Cuando un proceso muere, durante el tiempo que su padre este esperando por él, se convierte en un proceso "zombie", ya que se sigue guardando su entrada en la tabla de procesos en espera de que su padre lea su estado de salida. En caso que un proceso hijo sea huérfano, su padre ha muerto antes que él, todos los procesos "zombies" son adoptados por el proceso ***init*** que automáticamente hace **waits** para ir eliminando a los "zombies" de la tabla de procesos.

Internamente en el núcleo todas éstas llamadas al sistema están implementadas por:

```
     SYSCALL_DEFINE5(waitid, int, which, pid_t, upid, struct sig-
info __user *,
1759            infop, int, options, struct rusage __user *, ru)
```

y

```
1798  SYSCALL_DEFINE4(wait4, pid_t, upid, int __user *,
stat_addr,
1799            int, options, struct rusage __user *, ru)
```

```
1836 SYSCALL_DEFINE3(waitpid, pid_t, pid, int __user *,
stat_addr, int, options)
1837{
```

implementadas en *kernel/exit.c* . Siendo *sys_waitpid* un mero llamador para *sys_wait4* que se sigue mantiendo por razones históricas, ya que desde la propia glibc se utiliza siempre *sys_wait4*.

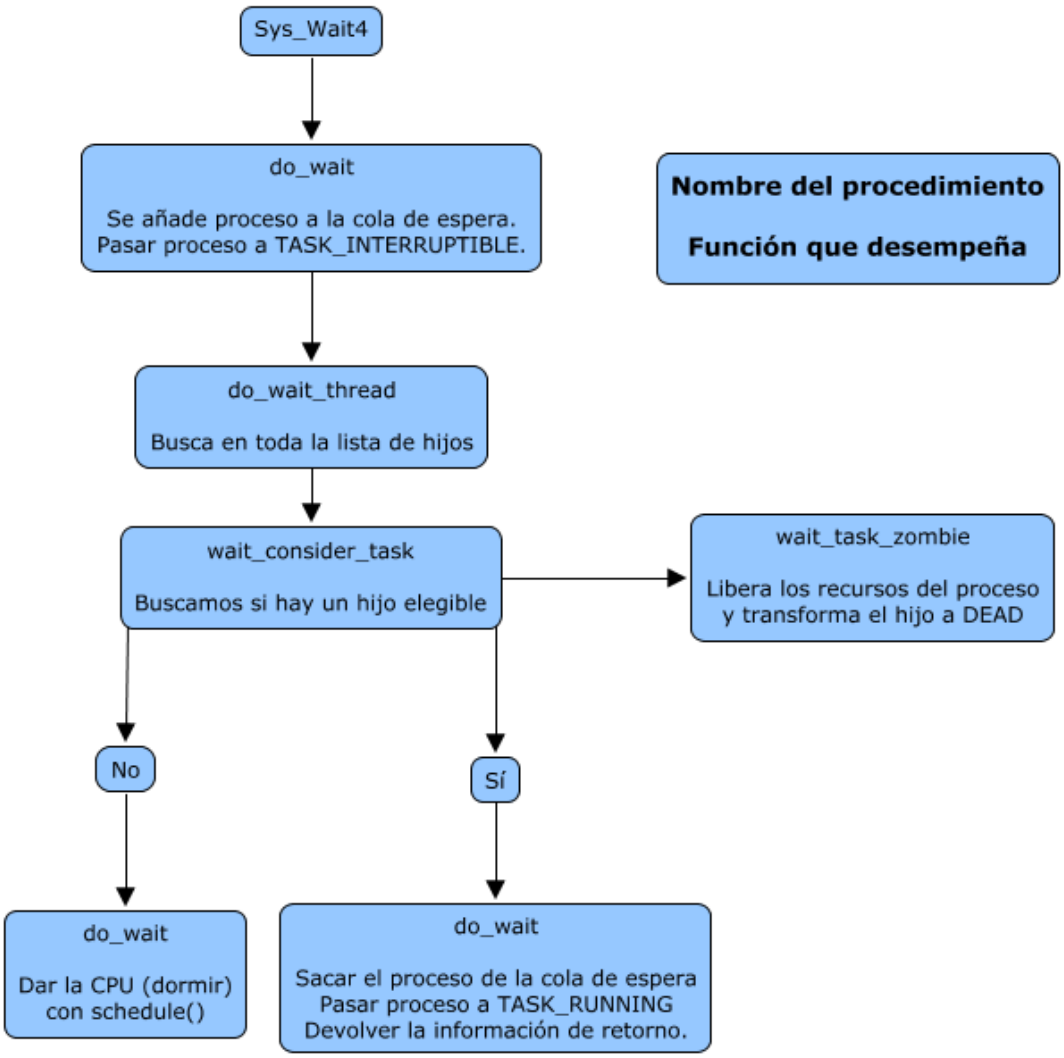
El código completo se muestra a continuación:

```
SYSCALL_DEFINE5(waitid, int, which, pid_t, upid, struct siginfo
__user *,
1759            infop, int, options, struct rusage __user *, ru)
1760{
1761        struct pid *pid = NULL;
1762        enum pid_type type;
1763        long ret;
1764
          //Comprueba que los parámetros de opciones son válidos
1765        if (options & ~(WNOHANG|WNOWAIT|WEXITED|WSTOPPED|WCON-
TINUED))
1766                return -EINVAL;
1767        if (!(options & (WEXITED|WSTOPPED|WCONTINUED)))
1768                return -EINVAL;
1769
1770        switch (which) {
1771        case P_ALL:
1772                type = PIDTYPE_MAX;
1773                break;
1774        case P_PID:
1775                type = PIDTYPE_PID;
1776                if (upid <= 0)
1777                        return -EINVAL;
1778                break;
1779        case P_PGID:
```

```
1780                type = PIDTYPE_PGID;
1781                if (upid <= 0)
1782                        return -EINVAL;
1783                break;
1784        default:
1785                return -EINVAL;
1786        }
1787
1788        if (type < PIDTYPE_MAX)
1789                pid = find_get_pid(upid);
1790        ret = do_wait(type, pid, options, infop, NULL, ru);
1791        put_pid(pid);
1792
1793        /* avoid REGPARM breakage on x86: */
1794        asmlinkage_protect(5, ret, which, upid, infop, op-
tions, ru);
1795        return ret;
1796}
1797
1798SYSCALL_DEFINE4(wait4, pid_t, upid, int __user *, stat_addr,
1799                int, options, struct rusage __user *, ru)
1800{
1801        struct pid *pid = NULL;
1802        enum pid_type type;
1803        long ret;
1804
          //Comprueba que los parámetros de opciones son válidos
1805        if (options & ~(WNOHANG|WUNTRACED|WCONTINUED|
1806                        __WNOTHREAD|__WCLONE|__WALL))
1807                return -EINVAL;
1808
1809        if (upid == -1)
1810                type = PIDTYPE_MAX;
1811        else if (upid < 0) {
1812                type = PIDTYPE_PGID;
1813                pid = find_get_pid(-upid);
1814        } else if (upid == 0) {
1815                type = PIDTYPE_PGID;
1816                pid = get_pid(task_pgrp(current));
1817        } else /* upid > 0 */ {
1818                type = PIDTYPE_PID;
1819                pid = find_get_pid(upid);
1820        }
1821
1822        ret = do_wait(type, pid, options | WEXITED, NULL,
stat_addr, ru);
1823        put_pid(pid);
1824
1825        /* avoid REGPARM breakage on x86: */
1826        asmlinkage_protect(4, ret, upid, stat_addr, options,
ru);
```

```
1827        return ret;
1828}
1829
1830#ifdef __ARCH_WANT_SYS_WAITPID
1831
1832/*
1833 * sys_waitpid() remains for compatibility. waitpid() should be
1834 * implemented by calling sys_wait4() from libc.a.
1835 */
1836SYSCALL_DEFINE3(waitpid, pid_t, pid, int __user *,
stat_addr, int, options)
1837{
1838        return sys_wait4(pid, stat_addr, options, NULL);
1839}
1840
```

Diagrama de tareas



En la función do_wait y sus auxiliares se realizan todo el proceso del wait

FUNCIÓN do_wait

```
1674static long do_wait(enum pid_type type, struct pid *pid, int
options,
1675                  struct siginfo __user *infop, int __user
*stat_addr,
1676                  struct rusage __user *ru)
1677{
1678       DECLARE_WAITQUEUE(wait, current);
1679       struct task_struct *tsk;
1680       int retval;
1681
1682       trace_sched_process_wait(pid);

          /* No se comprueban la validez de los parámetros puesto que ello se hace ya
en las llamadas que utilizan do_wait.

              IMPORTANTE

              Se añade el proceso actual (el que hace el wait) a una cola de espera y
se establece su estado a TASK_INTERRUPTIBLE, es decir, no dar CPU hasta que le llegue
una señal, en este caso sería la señal SIG_CHLD indicando que un hijo ha cambiado de
estado. */

1683
1684       add_wait_queue(&current->signal-
>wait_chldexit,&wait);
1685repeat:
1686       /*
1687        * If there is nothing that can match our critiera just get out.
1688        * We will clear @retval to zero if we see any child that might later
1689        * match our criteria, even if we are not able to reap it yet.
1690        */

          //Comprobaciones para agilizar el proceso de búsqueda de un hijo
1691       retval = -ECHILD;
1692       if ((type < PIDTYPE_MAX) && (!pid || hlist_empty(&pid-
>tasks[type])))
1693              goto end;
1694
1695       current->state = TASK_INTERRUPTIBLE;
1696       read_lock(&tasklist_lock);
1697       tsk = current;

          //Hace una búsqueda iterando en cada thread
1698       do {
1699              int tsk_result = do_wait_thread(tsk, &retval,
1700                                              type, pid, options,
1701                                              infop, stat_addr, ru);
1702              if (!tsk_result)
1703                     tsk_result = ptrace_do_wait(tsk, &retval,
1704                                                 type, pid, options,
1705                                                 infop, stat_addr, ru);
1706              if (tsk_result) {
```

```
1707                    /*
1708                     * tasklist_lock is unlocked and we have a final result.
1709                     */
1710                    retval = tsk_result;
1711                    goto end;
1712            }

                /* LA OPCIÓN WNOTHREAD es para no esperar por hijos de otros hilos
                           del mismo grupo de hilos */
1713
1714            if (options & __WNOTHREAD)
1715                    break;
                // pasa al siguiente hilo

1716            tsk = next_thread(tsk);
1717            BUG_ON(tsk->signal != current->signal);
1718    } while (tsk != current);

            /* Fin del bucle: comprobaciones. Hay que tener en cuenta que no siempre
            el algoritmo llega aquí, si se ha hecho un goto end se salta esta parte,
            aquí se llega solamente cuando ningún hijo ha cumplido las condiciones */
1719    read_unlock(&tasklist_lock);
//Si no está activada la opción WNOHANG (no esperar por hijos) ni hay señales pendientes
se suelta la CPU llamando a schedule() y cuando se despierta volverá a hacer el
procedimiento

1720
1721    if (!retval && !(options & WNOHANG)) {
1722            retval = -ERESTARTSYS;
1723            if (!signal_pending(current)) {
1724                    schedule();
1725                    goto repeat;
1726            }
1727    }
1728
1729 end:

        //IMPORTANTE
        /* Se cambia el estado del padre a TASK_RUNNING y el proceso padre se quita
        de la lista de procesos en espera */
1730      current->state = TASK_RUNNING;
1731      remove_wait_queue(&current->signal-
>wait_chldexit,&wait);

       //Entra en el if si do_wait ha sido llamado mediante waited (no es nuestro caso)
      /* infop es un puntero, comprueba si hay que rellenar la estructura (según el
      tipo de llamada) */

1732    if (infop) {
1733            if (retval > 0)
1734                    retval = 0;
1735            else {
1736                    /*
1737                     * For a WNOHANG return, clear out all the fields
1738                     * we would set so the user can easily tell the
1739                     * difference.
1740                     */
```

```
1741                    if (!retval)
1742                            retval = put_user(0, &infop-
>si_signo);
1743                    if (!retval)
1744                            retval = put_user(0, &infop-
>si_errno);
1745                    if (!retval)
1746                            retval = put_user(0, &infop-
>si_code);
1747                    if (!retval)
1748                            retval = put_user(0, &infop->si_pid);
1749                    if (!retval)
1750                            retval = put_user(0, &infop->si_uid);
1751                    if (!retval)
1752                            retval = put_user(0, &infop-
>si_status);
1753            }
1754    }
1755    return retval;
1756}
```

8.4 FUNCIONES AUXILIARES

FUNCIÓN do_wait_thread

```
1628static int do_wait_thread(struct task_struct *tsk, int
*notask_error,
1629                        enum pid_type type, struct pid *pid, int
options,
1630                        struct siginfo __user *infop, int __user
*stat_addr,
1631                        struct rusage __user *ru)
1632{
1633    struct task_struct *p;
1634
        //Busca en toda la lista de hijos donde p  tiene cada entrada de la lista
1635    list_for_each_entry(p, &tsk->children, sibling) {
1636            /*
1637             * Do not consider detached threads.
1638             */
1639            if (!task_detached(p)) {
1640                    int ret = wait_consider_task(tsk, 0, p,
notask_error,
1641                                                 type, pid, options,
1642                                                 infop, stat_addr,
ru);
1643                    if (ret)
1644                            return ret;
1645            }
1646    }
```

```
1647
1648        return 0;
1649}
1650
```

wait_consider_task

```
1557/*
1558 * Consider @p for a wait by @parent.
1559 *
1560 * -ECHILD should be in *@notask_error before the first call.
1561 * Returns nonzero for a final return, when we have unlocked tasklist_lock.
1562 * Returns zero if the search for a child should continue;
1563 * then *@notask_error is 0 if @p is an eligible child,
1564 * or another error from security_task_wait(), or still -ECHILD.
1565 */
1566static int wait_consider_task(struct task_struct *parent, int
ptrace,
1567                              struct task_struct *p, int
*notask_error,
1568                              enum pid_type type, struct pid *pid, int
options,
1569                              struct siginfo __user *infop,
1570                              int __user *stat_addr, struct rusage
__user *ru)
1571{
          //Elegible_child verifica si el hijo actual es elegible.
          //0 el proceso hijo no puede ser seleccionado
          //1 el proceso hijo puede ser seleccionado
1572        int ret = eligible_child(type, pid, options, p);
1573        if (!ret)
1574                return ret;
1575

          //Si no hemos podido aceder al proceso                                daremos
un error
1576        if (unlikely(ret < 0)) {
1577                /*
1578                 * If we have not yet seen any eligible child,
1579                 * then let this error code replace -ECHILD.
1580                 * A permission error will give the user a clue
1581                 * to look for security policy problems, rather
1582                 * than for mysterious wait bugs.
1583                 */
1584                if (*notask_error)
1585                        *notask_error = ret;
1586        }
1587
1588        if (likely(!ptrace) && unlikely(p->ptrace)) {
1589                /*
1590                 * This child is hidden by ptrace.
```

```
1591                 * We aren't allowed to see it now, but eventually we will.
1592                 */
1593                *notask_error = 0;
1594                return 0;
1595        }
1596
```

```
        //Si el hijo cumple los criterios se pasa a mirar su estado

        //IMPORTANTE
        // si el hijo ya está DEAD no nos interesa
1597        if (p->exit_state == EXIT_DEAD)
1598                return 0;
1599
1600        /*
1601         * We don't reap group leaders with subthreads.
1602         */
```

```
/* Éste es el caso más frecuente de llamadas a wait. El hijo ha acabado su ejecución y el
padre va a leer los resultados de la misma. wait_task_zombie hace el trabajo. Cambia el
estado del hijo a EXIT_DEAD, rellena las estadísticas de uso del proceso, elimina su
task_struct mediante release_task y retorna su pid */

1603        if (p->exit_state == EXIT_ZOMBIE && !
delay_group_leader(p))
1604                return wait_task_zombie(p, options, infop,
stat_addr, ru);
1605
1606        /*
1607         * It's stopped or running now, so it might
1608         * later continue, exit, or stop again.
1609         */
1610        *notask_error = 0;
1611

1612        if (task_is_stopped_or_traced(p))
1613                return wait_task_stopped(ptrace, p, options,
1614                                         infop, stat_addr, ru);
1615
1616        return wait_task_continued(p, options, infop,
stat_addr, ru);
1617}
```

wait_task_zombie

```
/* wait_task_zombie ignora el proceso y devuelve 0 si no cumple los criterios para
```

```
transformarlo a estado EXIT_DEAD, si no, libera los recursos del proceso. Y devuelve la
información de la llamada */
1255 /*
1256  * Handle sys_wait4 work for one task in state EXIT_ZOMBIE.  We hold
1257  * read_lock(&tasklist_lock) on entry.  If we return zero, we still hold
1258  * the lock and this task is uninteresting.  If we return nonzero, we have
1259  * released the lock and the system call should return.
1260  */
1261 static int wait_task_zombie(struct task_struct *p, int options,
1262                             struct siginfo __user *infop,
1263                             int __user *stat_addr, struct rusage
    __user *ru)
1264 {
1265         unsigned long state;
1266         int retval, status, traced;
1267         pid_t pid = task_pid_vnr(p);
1268
1269         if (!likely(options & WEXITED))
1270                 return 0;
1271

        /* noreap será verdadero si nos pasan como opción de la llamada el WNOWAIT
(dejarlo todo como estaba para que otra llamada que se encuentre con el proceso haga lo
mismo) */
1272         if (unlikely(options & WNOWAIT)) {
1273                 uid_t uid = p->uid;
1274                 int exit_code = p->exit_code;
1275                 int why, status;
1276
1277                 get_task_struct(p);
1278                 read_unlock(&tasklist_lock);
1279                 if ((exit_code & 0x7f) == 0) {
1280                         why = CLD_EXITED;
1281                         status = exit_code >> 8;
1282                 } else {
1283                         why = (exit_code & 0x80) ? CLD_DUMPED :
CLD_KILLED;
1284                         status = exit_code & 0x7f;
1285                 }

                /*Reporta la información del proceso muerto a la zona
                de memoria de usuario mediante la función sin tocar nada de la estructura
                wait_noreap_copyout*/
1286                 return wait_noreap_copyout(p, pid, uid, why,
1287                                            status, infop, ru);
1288         }
1289



1290         /*
1291          * Try to move the task's state to DEAD
1292          * only one thread is allowed to do this:
1293          */

        //Si no, hay que pasar el proceso a DEAD. Se intenta  cambiar el   estado a
        EXIT_DEAD
```

```
        //xchg es atómico
1294        state = xchg(&p->exit_state, EXIT_DEAD);

         // comprueba que pueda cambiarlo
1295        if (state != EXIT_ZOMBIE) {
1296                BUG_ON(state != EXIT_DEAD);
1297                return 0;
1298        }
1299
1300        traced = ptrace_reparented(p);
1301

//IMPORTANTE
// comprueba que sea el padre real para rellenar él las estadísticas
// (cosas de las llamadas al sistema de contabilidad de tiempo que se deben notificar al
padre)
1302        if (likely(!traced)) {
1303                struct signal_struct *psig;
1304                struct signal_struct *sig;
1305                struct task_cputime cputime;
1306
1307                /*
1308                 * The resource counters for the group leader are in its
1309                 * own task_struct.  Those for dead threads in the group
1310                 * are in its signal_struct, as are those for the child
1311                 * processes it has previously reaped.  All these
1312                 * accumulate in the parent's signal_struct c* fields.
1313                 *
1314                 * We don't bother to take a lock here to protect these
1315                 * p->signal fields, because they are only touched by
1316                 * __exit_signal, which runs with tasklist_lock
1317                 * write-locked anyway, and so is excluded here.  We do
1318                 * need to protect the access to p->parent->signal fields,
1319                 * as other threads in the parent group can be right
1320                 * here reaping other children at the same time.
1321                 *
1322                 * We use thread_group_cputime() to get times for the thread
1323                 * group, which consolidates times for all threads in the
1324                 * group including the group leader.
1325                 */
1326                spin_lock_irq(&p->parent->sighand->siglock);
1327                psig = p->parent->signal;
1328                sig = p->signal;
1329                thread_group_cputime(p, &cputime);
1330                psig->cutime =
1331                        cputime_add(psig->cutime,
1332                        cputime_add(cputime.utime,
1333                                    sig->cutime));
1334                psig->cstime =
1335                        cputime_add(psig->cstime,
1336                        cputime_add(cputime.stime,
1337                                    sig->cstime));
1338                psig->cgtime =
1339                        cputime_add(psig->cgtime,
```

```
1340                    cputime_add(p->gtime,
1341                    cputime_add(sig->gtime,
1342                                sig->cgtime)));
1343            psig->cmin_flt +=
1344                    p->min_flt + sig->min_flt + sig->cmin_flt;
1345            psig->cmaj_flt +=
1346                    p->maj_flt + sig->maj_flt + sig->cmaj_flt;
1347            psig->cnvcsw +=
1348                    p->nvcsw + sig->nvcsw + sig->cnvcsw;
1349            psig->cnivcsw +=
1350                    p->nivcsw + sig->nivcsw + sig->cnivcsw;
1351            psig->cinblock +=
1352                    task_io_get_inblock(p) +
1353                    sig->inblock + sig->cinblock;
1354            psig->coublock +=
1355                    task_io_get_oublock(p) +
1356                    sig->oublock + sig->coublock;
1357            task_io_accounting_add(&psig->ioac, &p->ioac);
1358            task_io_accounting_add(&psig->ioac, &sig->ioac);
1359            spin_unlock_irq(&p->parent->sighand->siglock);
1360    }
1361
1362    /*
1363     * Now we are sure this task is interesting, and no other
1364     * thread can reap it because we set its state to EXIT_DEAD.
1365     */
1366    read_unlock(&tasklist_lock);
1367

// tomamos la información del proceso que necesitamos para la retornar de la llamada al sistema.
1368    retval = ru ? getrusage(p, RUSAGE_BOTH, ru) : 0;
1369    status = (p->signal->flags & SIGNAL_GROUP_EXIT)
1370            ? p->signal->group_exit_code : p->exit_code;
1371    if (!retval && stat_addr)
1372            retval = put_user(status, stat_addr);
1373    if (!retval && infop)
1374            retval = put_user(SIGCHLD, &infop->si_signo);
1375    if (!retval && infop)
1376            retval = put_user(0, &infop->si_errno);
1377    if (!retval && infop) {
1378            int why;
1379
1380            if ((status & 0x7f) == 0) {
1381                    why = CLD_EXITED;
1382                    status >>= 8;
1383            } else {
1384                    why = (status & 0x80) ? CLD_DUMPED :
CLD_KILLED;
1385                    status &= 0x7f;
1386            }
1387            retval = put_user((short)why, &infop->si_code);
```

```
1388                if (!retval)
1389                        retval = put_user(status, &infop-
>si_status);
1390        }
1391        if (!retval && infop)
1392                retval = put_user(pid, &infop->si_pid);
1393        if (!retval && infop)
1394                retval = put_user(p->uid, &infop->si_uid);
1395        if (!retval)
1396                retval = pid;
1397
1398        if (traced) {
1399                write_lock_irq(&tasklist_lock);
1400                /* We dropped tasklist, ptracer could die and untrace */
1401                ptrace_unlink(p);
1402                /*
1403                 * If this is not a detached task, notify the parent.
1404                 * If it's still not detached after that, don't release
1405                 * it now.
1406                 */
1407                if (!task_detached(p)) {
1408                        do_notify_parent(p, p->exit_signal);
1409                        if (!task_detached(p)) {
1410                                p->exit_state = EXIT_ZOMBIE;
1411                                p = NULL;
1412                        }
1413                }
1414                write_unlock_irq(&tasklist_lock);
1415        }
        //IMPORTANTE
        /* Elimina por completo al proceso hijo mediante la función release_task */
1416        if (p != NULL)
1417                release_task(p);
1418
        //Retornamos el valor
1419        return retval;
1420}
```

wait_task_stopped

```
/* Comprueba si el proceso es susceptible de ser elegido con los parámetros de la
llamada al sistema, si no, devuelve cero. */
1422/*
1423 * Handle sys_wait4 work for one task in state TASK_STOPPED.  We hold
1424 * read_lock(&tasklist_lock) on entry.  If we return zero, we still hold
1425 * the lock and this task is uninteresting.  If we return nonzero, we have
1426 * released the lock and the system call should return.
1427 */
1428static int wait_task_stopped(int ptrace, struct task_struct *p,
```

```
1429                        int options, struct siginfo __user
*infop,
1430                        int __user *stat_addr, struct rusage
__user *ru)
1431{
1432        int retval, exit_code, why;
1433        uid_t uid = 0; /* unneeded, required by compiler */
1434        pid_t pid;
1435
    //Si tenemos la opción WUNTRACED no hacemos nada
1436        if (!(options & WUNTRACED))
1437                return 0;
1438
1439        exit_code = 0;
1440        spin_lock_irq(&p->sighand->siglock);
1441
1442        if (unlikely(!task_is_stopped_or_traced(p)))
1443                goto unlock_sig;
1444
1445        if (!ptrace && p->signal->group_stop_count > 0)
1446                /*
1447                 * A group stop is in progress and this is the group leader.
1448                 * We won't report until all threads have stopped.
1449                 */
1450                goto unlock_sig;
1451
/* No nos interesa si ya está siendo mirado por otro hilo del kernel o el grupo no ha
acabado */
1452        exit_code = p->exit_code;
1453        if (!exit_code)
1454                goto unlock_sig;
1455
1456        if (!unlikely(options & WNOWAIT))
1457                p->exit_code = 0;
1458
1459        uid = p->uid;
1460 unlock_sig:
1461        spin_unlock_irq(&p->sighand->siglock);
1462        if (!exit_code)
1463                return 0;
1464
1465        /*
1466         * Now we are pretty sure this task is interesting.
1467         * Make sure it doesn't get reaped out from under us while we
1468         * give up the lock and then examine it below.  We don't want to
1469         * keep holding onto the tasklist_lock while we call getrusage and
1470         * possibly take page faults for user memory.
1471         */
1472        get_task_struct(p);
1473        pid = task_pid_vnr(p);
1474        why = ptrace ? CLD_TRAPPED : CLD_STOPPED;
1475        read_unlock(&tasklist_lock);
```

```
1476
// Si nos pasan el WNOWAIT como parámetro, dejamos la tarea sin tocar
1477        if (unlikely(options & WNOWAIT))
1478                return wait_noreap_copyout(p, pid, uid,
1479                                           why, exit_code,
1480                                           infop, ru);
// rellena la información del proceso para devolver desde la llamada
1481
1482        retval = ru ? getrusage(p, RUSAGE_BOTH, ru) : 0;
1483        if (!retval && stat_addr)
1484                retval = put_user((exit_code << 8) | 0x7f,
stat_addr);
1485        if (!retval && infop)
1486                retval = put_user(SIGCHLD, &infop->si_signo);
1487        if (!retval && infop)
1488                retval = put_user(0, &infop->si_errno);
1489        if (!retval && infop)
1490                retval = put_user((short)why, &infop->si_code);
1491        if (!retval && infop)
1492                retval = put_user(exit_code, &infop->si_status);
1493        if (!retval && infop)
1494                retval = put_user(pid, &infop->si_pid);
1495        if (!retval && infop)
1496                retval = put_user(uid, &infop->si_uid);
1497        if (!retval)
1498                retval = pid;
1499        put_task_struct(p);
1500
1501        BUG_ON(!retval);
1502        return retval;
1503}
```

wait_task_continued

```
/* wait_task_continued es llamado desde wait_comsider_task cuando un proceso está
ejecutándose actualmente, por ejemplo cuando ha cambiado de stopped a running. En ese
caso toma la información de dicho proceso. Si devuelve 0 el proceso no nos interesa, en
otro caso sí */
1505/*
1506 * Handle do_wait work for one task in a live, non-stopped state.
1507 * read_lock(&tasklist_lock) on entry.  If we return zero, we still hold
1508 * the lock and this task is uninteresting.  If we return nonzero, we have
1509 * released the lock and the system call should return.
1510 */
1511static int wait_task_continued(struct task_struct *p, int
options,
1512                               struct siginfo __user *infop,
1513                               int __user *stat_addr, struct rusage
__user *ru)
1514{
1515        int retval;
1516        pid_t pid;
```

```
1517        uid_t uid;

// Si en el campo option esta WCONTINUED salimos del proceso
// Si no es que ha cambiado de estado (de stop a continue) ignorar proceso
1518
1519        if (!unlikely(options & WCONTINUED))
1520                return 0;
1521
1522        if (!(p->signal->flags & SIGNAL_STOP_CONTINUED))
1523                return 0;
1524
1525        spin_lock_irq(&p->sighand->siglock);
1526        /* Re-check with the lock held.  */
1527        if (!(p->signal->flags & SIGNAL_STOP_CONTINUED)) {
1528                spin_unlock_irq(&p->sighand->siglock);
1529                return 0;
1530        }
1531        if (!unlikely(options & WNOWAIT))
1532                p->signal->flags &= ~SIGNAL_STOP_CONTINUED;
1533        spin_unlock_irq(&p->sighand->siglock);
1534
1535        pid = task_pid_vnr(p);
1536        uid = p->uid;
1537        get_task_struct(p);
1538        read_unlock(&tasklist_lock);
1539
1540        if (!infop) {
1541                retval = ru ? getrusage(p, RUSAGE_BOTH, ru) : 0;
1542                put_task_struct(p);
1543                if (!retval && stat_addr)
1544                        retval = put_user(0xffff, stat_addr);
1545                if (!retval)
1546                        retval = pid;
1547        } else {
1548                retval = wait_noreap_copyout(p, pid, uid,
1549                                             CLD_CONTINUED, SIGCONT,
1550                                             infop, ru);
1551                BUG_ON(retval == 0);
1552        }
1553
1554        return retval;
1555}
```

8.5 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