Ione Falcón Sánchez

Carlos E. David Monzón

Exit – Descripción (I)

Exit ==!(Fork)

Exit – Descripción (II)

- Se encarga de liberar los recursos retenidos por el proceso:
  - **Memoria**
  - **Semáforos**
  - **Ficheros abiertos**
  - **Información sobre el sistema de ficheros**
  - **Pila de ejecución**
  - **Registros de la CPU**
  - **Cerrojos**
  - **NameSpaces**
  - **Enlace con el terminal**

Exit – Descripción (III)

- También se encarga de notificar al padre de la finalización de su hijo.

- Y eliminar completamente al proceso en casos especiales:
  - El padre no está en ejecución
  - Modo traza

Exit – Implementación

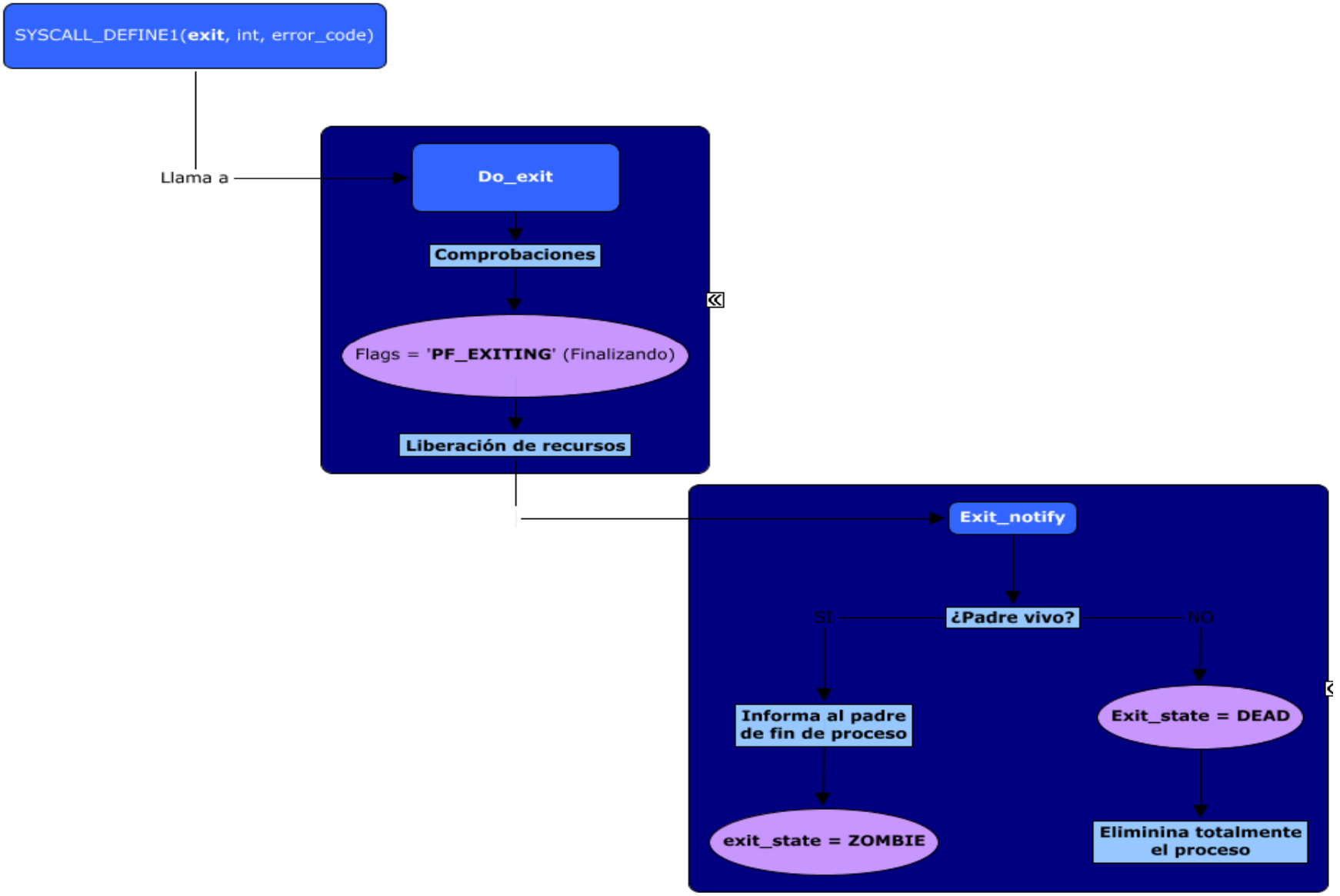
- Cuando un proceso finaliza su ejecución realiza la llamada al sistema **exit(n).**
  - n: código de salida entre 0 y 255 (0 = sin errores)

- Se implementa en la función **sys_exit**, la cual libera los recursos del proceso, termina su ejecución y lo notifica al proceso padre.

Exit – Posibles Estados de Terminación

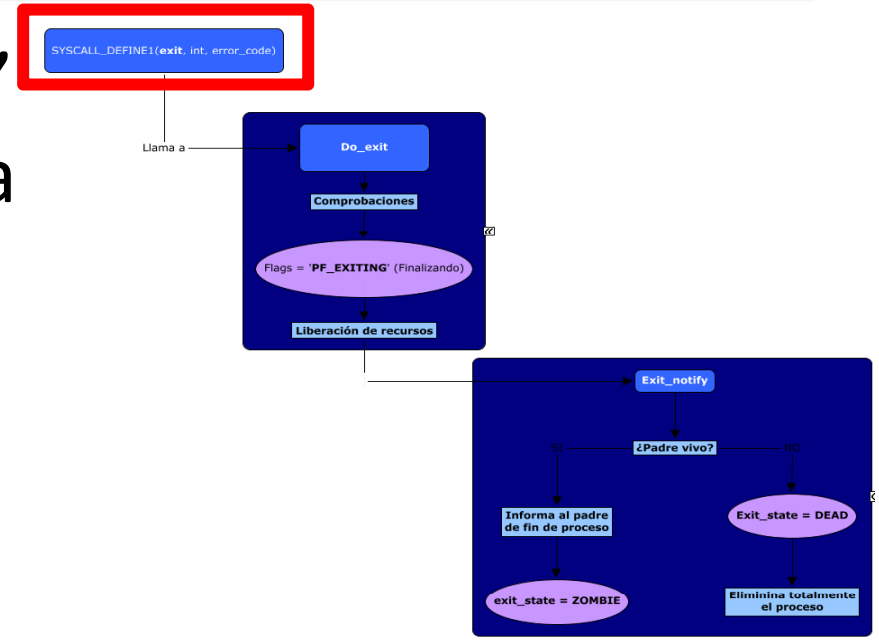
```
struct task_struct
{

[...]

flags      =   PF_EXITING     El Proceso está Terminando

    ...

exit_state = EXIT_ZOMBIE  Proceso sin recursos, espera al padre
             EXIT_DEAD    Proceso finalizado

}
```

Exit -Flujo de ejecución



SYSCALL_DEFINE1(**exit**, int, error_code)

- Convierte el “exit_code” al formato que espera la función **do_exit**
- Llama a dicha función




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

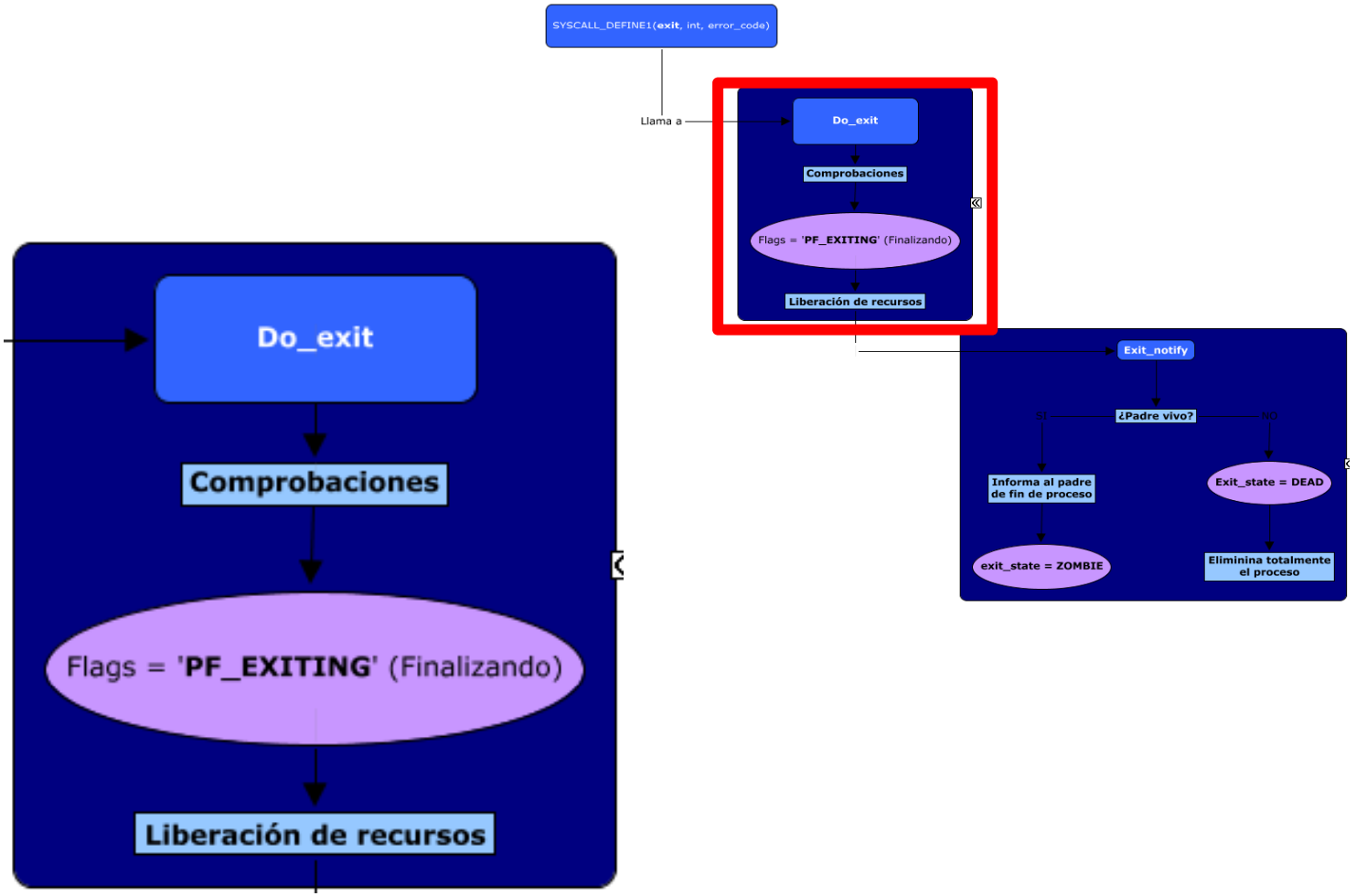
Do_exit - Descricpción

- Es la función que realmente ejecuta la liberación de recursos.

- También invoca a “exit_notify” para notificar al padre el exit de su hijo.

998 NORET_TYPE void do_exit(long code)

Do_exit - Diagrama de Flujo



Do_exit – Comprobaciones (I)

- Comprueba que no se vaya a eliminar el manejador de interrupciones o el “idle task”.

- Si así fuera, se detiene el SO

```
1007    if (unlikely(in_interrupt()))
1008        panic("Aiee, killing interrupt handler!");
1009    if (unlikely(!tsk->pid))
1010        panic("Attempted to kill the idle task!");
```

Do_exit – Comprobaciones (II)

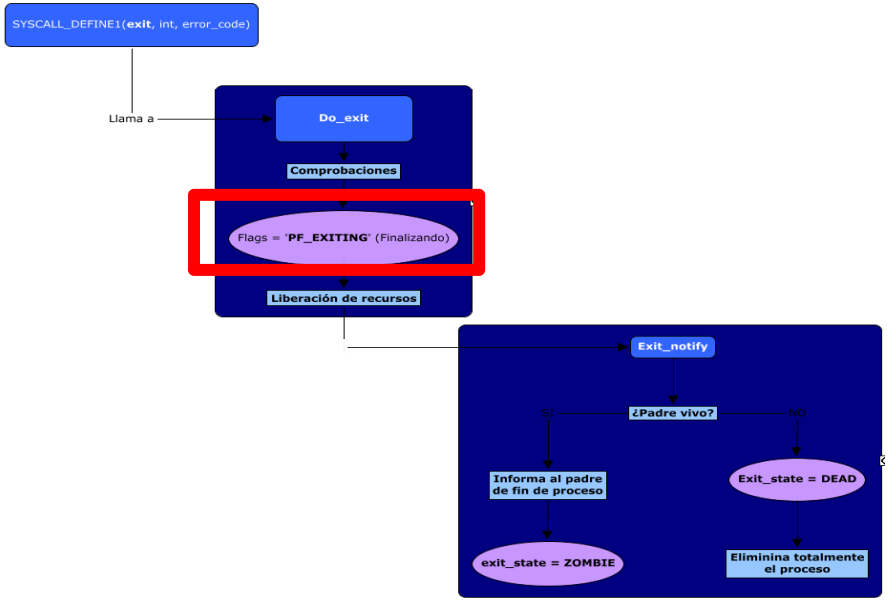
- Verifica que no esté ante un fallo por recursividad

```
1018        if (unlikely(tsk->flags & PF_EXITING)) {
1019            printk(KERN_ALERT
1020                "Fixing recursive fault but reboot is needed!\n");
```

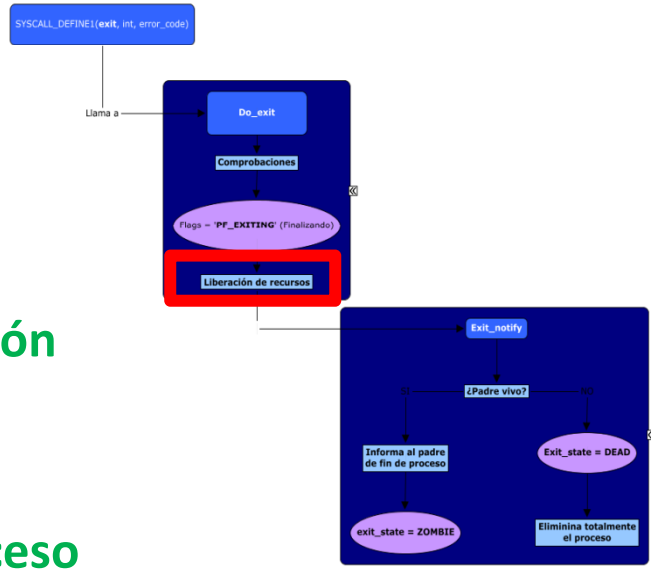
Do_exit - Cambio de estado

- Se actualiza el flag del proceso a **FINALIZANDO**

1037 exit_signals(tsk); ***/* sets PF_EXITING */***



Do_exit - Liberación de recursos (I)




```
// Elimina el Temporizador del proceso
1057    hrtimer_cancel(&tsk->signal->real_timer);
1058    exit_itimers(tsk->signal);
...
// Se escribe en el task-struck el código de terminación
1066    tsk->exit_code = code;
...
// Libera el espacio de memoria utilizado por el proceso
1069    exit_mm(tsk);
...
//Elimina 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);
```

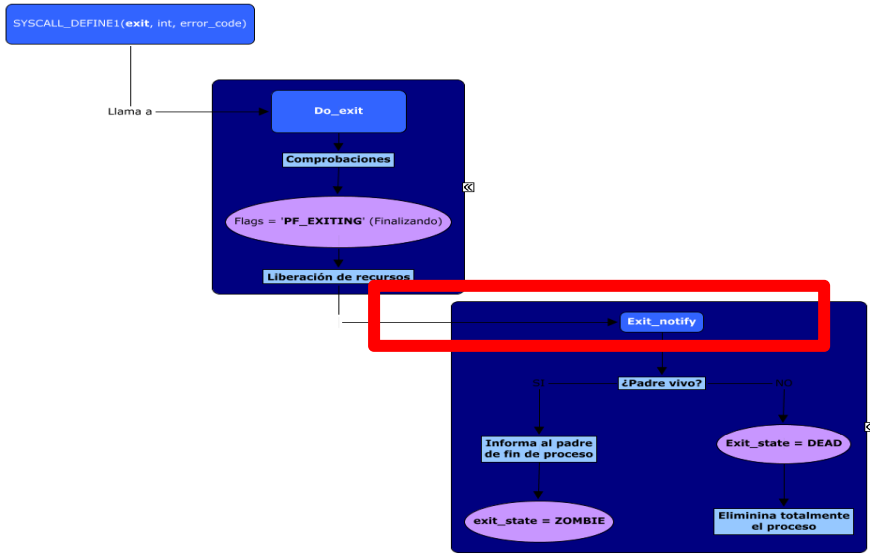
Do_exit - Liberación de recursos (II)

```
// 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);
```

Do_exit - Invocación a **exit_notify**

- A continuación, do_exit invoca a **exit_notify**

```
// Notifica al padre de la eliminación de su hijo
1091        exit_notify(tsk, group_dead);
```



Do_exit

- Por último, tras retornarse de dicha función, se cambia el estado del proceso a TASK_DEAD y se llama al planificador de procesos

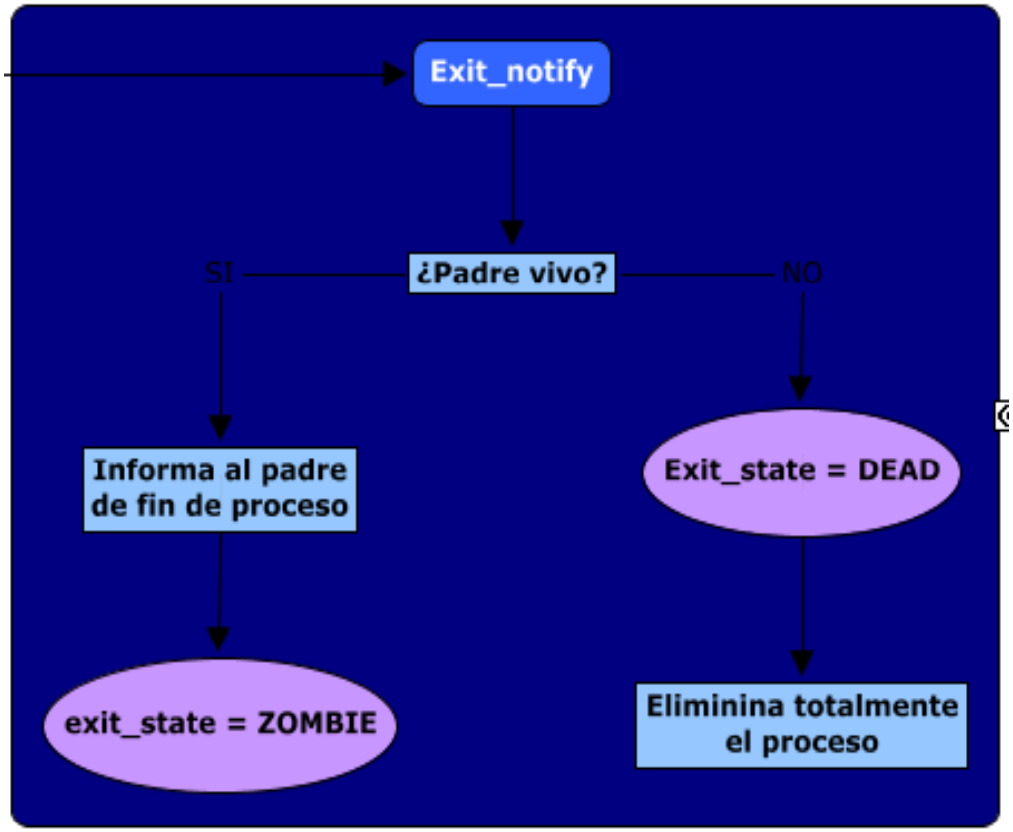
```
...
// Cambia el estado del proceso a TASK_DEAD
1125        tsk->state = TASK_DEAD;
1126
// Invoca al planificador, Esta será la ultima vez que este
// proceso tenga la CPU
1127        schedule();
```

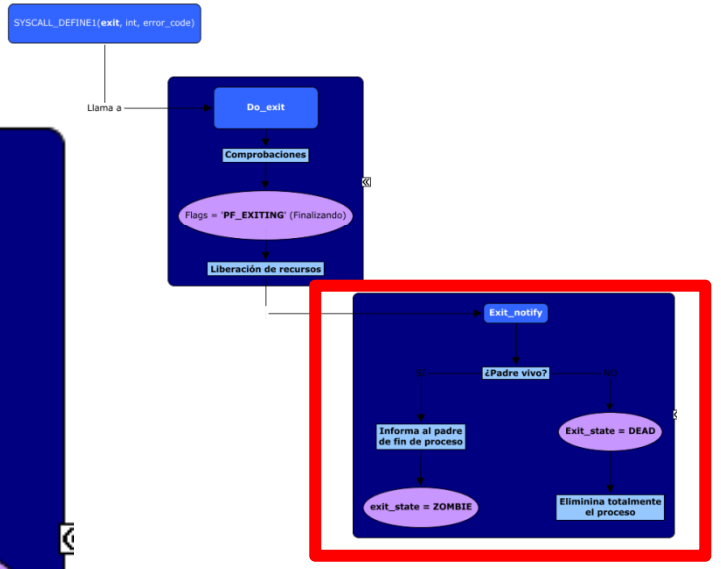
Exit_notify - Decripción

- Si el padre del proceso sigue ejecutándose, se encarga de notificarle acerca del fin del proceso hijo mediante una señal
- Sino, elimina el proceso hijo totalmente

```
909 static void exit_notify(struct task_struct *tsk, int group_dead)
```

Exit_notify - Diagrama de Flujo





Exit_notify - Padre en ejecución (I)

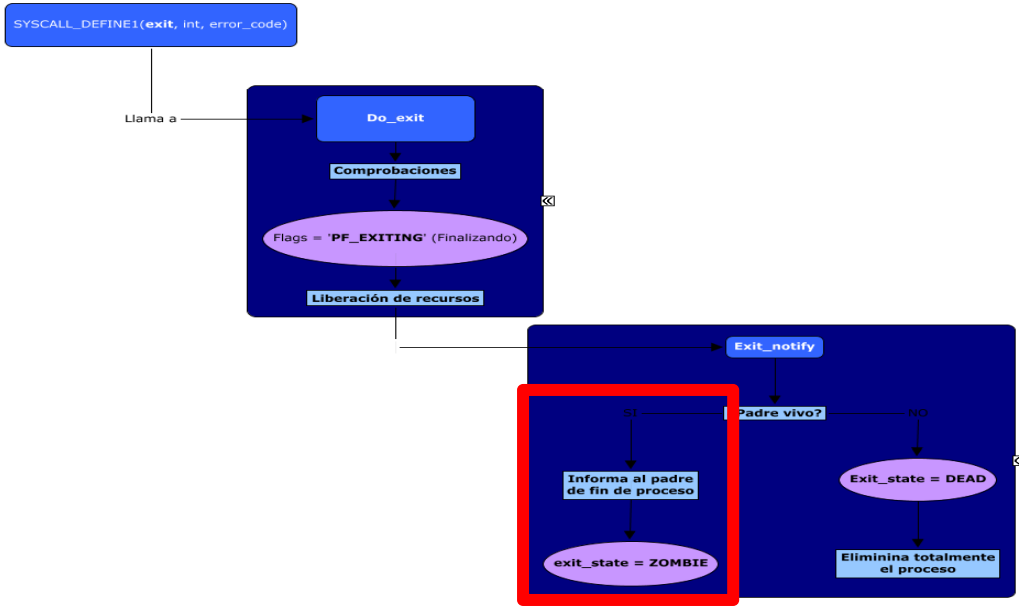
- Se invoca a **do_notify_parent**, que se encarga del envío de la señal

```
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);
```

Exit_notify - Padre en ejecución (II)

- El estado del proceso se establece a **EXIT_ZOMBIE**

```
 953      tsk->exit_state = signal == DEATH_REAP ? EXIT_DEAD : EXIT_ZOMBIE;
```



¿Qué significa estar en estado **ZOMBIE**?

- Un proceso "**Zombie**" es aquel al que se le han quitado todos los recursos, pero que se sigue manteniendo en la tabla de procesos.

- **¿Por qué?** – Al hacer un *exit()* se le manda una señal al padre que le indica que alguno de sus hijos ha terminado, el padre sabrá que (pid) mediante un *wait*.

- Al hacer el *wait* el proceso muere definitivamente (se elimina de la tabla).

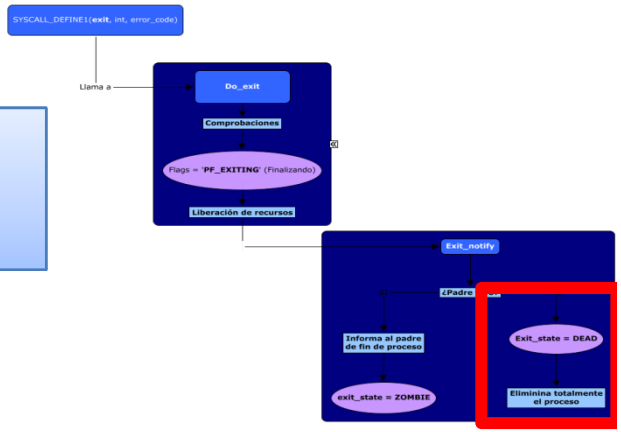
Exit_notify - Padre finalizado

- Si no existe proceso padre (ya ha finalizado), el hijo pasa al estado **EXIT_DEAD**

```
953     tsk->exit_state = signal == DEATH_REAP ? EXIT_DEAD : EXIT_ZOMBIE;
```

- Ya no es necesario mantenerlo en la tabla de procesos, por lo que se elimina completamente mediante la función **release_task**

```
966     if (signal == DEATH_REAP)
967         release_task(tsk);
```



Tareas más importantes
Release_task

1) Liberar información del proceso (_exit_signal)
    1.1) Liberar los descriptores de acciones asociadas a señales
    1.2) Liberar la entrada pid de pidtask
2) Liberar la memoria usada por la task_struct

```
 161 void release_task(struct task_struct * p)
171         __exit_signal(p); // 1)
 203       call_rcu(&p->rcu, delayed_put_task_struct); // 2)
```

WAIT

Introducción

- Espera por cambios en los hijos y devuelve información de su estado.

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

- **Wait** → suspende la ejecución del proceso actual hasta que acabe alguno de sus procesos hijo.

- **Waitpid** → suspende la ejecución del proceso actual hasta que cambie de estado el hijo identificado pid. (por defecto, que acabe)

Valor devuelto

```
_Pid_t   wait (int *statusp);
```

```
_Pid_t   waitpid (pid_t pid, int *statusp, int options);
```

- Devuelven el identificador del proceso hijo que ha finalizado
- En caso de error, se retorna –1
- Si se hizo uso de waitpid con la opción WNOHANG y ningún hijo estaba disponible, se devuelve 0

```
int     waitid (idtype_t idtype, id_t id , siginfo_t * infop , int options);
```

- Devuelve 0 si tiene éxito o si se usó con la opción WNOHANG y ningún hijo estaba disponible
- En caso de error, se retorna –1

Parámetros – información devuelta

```
Pid_t wait (int *statusp);
```

```
Pid_t waitpid (pid_t pid, int *statusp, int options);
```

- int *statusp : sirve para retornar el estado del proceso hijo si no es nulo:

-WIFEXITED(status): si terminó normalmente

-WEXITSTATUS(status): valor pasado al exit

-WIFSIGNALED(status): si terminó por una señal

-WTERMSIG(status): número de la señal

-WCOREDUMP(status): si produjo un core dump

-WIFSTOPPED(status): si el hijo está parado.

-WSTOPSIG(status): señal que hizo parase al hijo

-WIFCONTINUED(status) (2.6.10) si el hijo continuó tras un SIGCONT.

Parámetros – información devuelta

```
int     waitid (idtype_t idtype, id_t id , siginfo_t * infop , int options);
```

- 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 saliío, CLD_KILLED si el hijo fue terminado por una señal, CLD_STOPPED si el hijo fué parado o CLD_CONTINUED si el hijo continuó al llegarle una señal SIGCONT.

Waitpid – Parámetros propios

- pid_t pid : identifica al hijo por el que se espera

pid > 0 → especifica el número del proceso a esperar

pid=0 → espera por los hijos cuyo número de grupo de procesos sea igual al del padre

pid= -1 → espera por la finalización de alguno de sus hijos. En este caso es igual al wait

pid < -1 → espera por el proceso abs(pid)

```
Pid_t waitpid (pid_t pid, int *statusp, int options);
```

Waitpid – Parámetros propios

- int options : es un OR de cero o más de las siguientes constantes:

WNOHANG → vuelve inmediatamente si ningún hijo ha terminado.
WUNTRACED → también vuelve si hay hijos parados, y de cuyo estado no ha recibido notificación
WCONTINUED → vuelve si un hijo ha recontinuado la ejecución tras estar parado y recibir un SIGCONT (2.6.10)
__WCLONE→ solo esperar por hijos clones.
__WALL→ esperar por cualquier hijo.
__WNOTHREAD→ no esperar por cualquier hijo del grupo de threads.

```
Pid_t waitpid (pid_t pid, int *statusp, int options);
```

Waitid – Parámetros propios

idtype, id: seleccionan por quién esperar:

- 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.*

```
int     waitid (idtype_t idtype, id_t id , siginfo_t * infop , int options);
```

Waitid – Parámetros propios

- int options : es un OR de cero o más de las siguientes constantes:

WEXITED → Espera por hijos que hayan terminado.

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

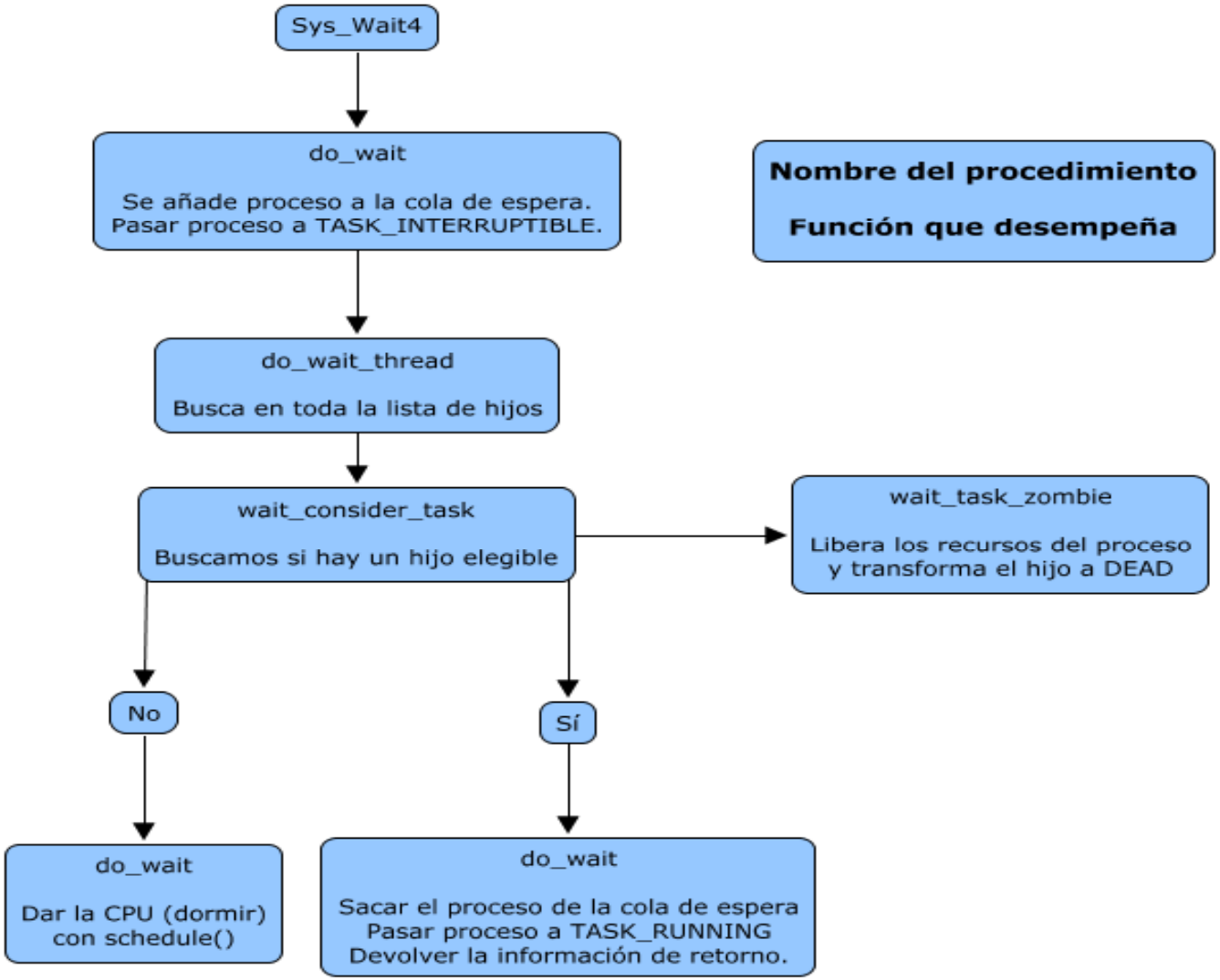
WCONTINUED → Espera por hijos que continúen su ejecución tras recibir un SIGCONT tras estar parados.

WNOHANG → volver inmediatamente si ho hay hijo por el que esperar.

WNOWAIT → dejar al hijo sin modificar ni marcar en la tabla de procesos, una posterior llamada esperaría otra vez por él.

```
Pid_t waitpid (pid_t pid, int *statusp, int options);
```

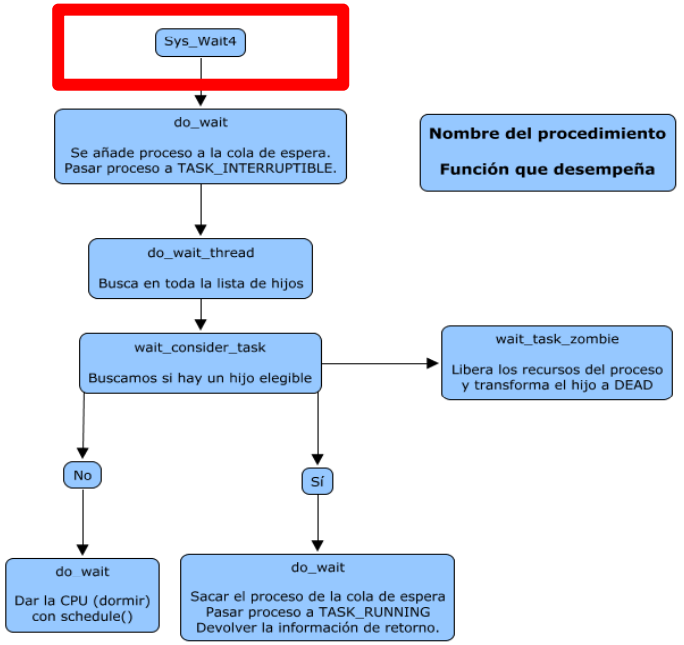
Flujo de Ejecución



Syscall_define4(wait4, pid_t, upid, int user *, stat addr, int, option struct rusage user*, ru)

- Comprueba que el parámetro de opciones es correcto
- Llama a la función **do_wait**

```
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}
```



do_wait

- Es la función que realmente realiza el trabajo
- Parámetros:

  - pid Identificador del proceso por el que se va a esperar.

  - options Define el comportamiento de la función.

  - infop Estructura con información del proceso.

  - stat_addr Dirección donde se copiará el estado de finalización.

  - ru Dirección donde se indica la información de recursos usados por el proceso a esperar.

```
1674 static 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)
```

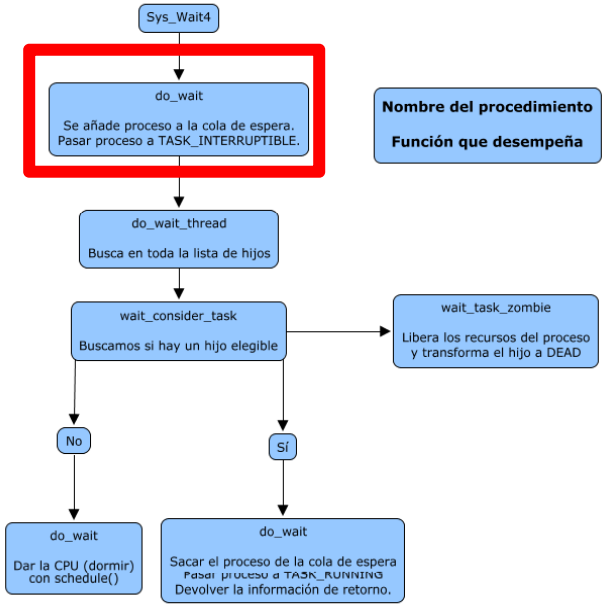
do_wait - Inicio

- El proceso pasa de la lista de procesos en ejecución a la lista de procesos en espera.
- Estado del padre = INTERRUPTIBLE, esto es, está detenido pero puede ser interrumpido por señales (SIG_CHLD).
- Llamamos a la función do_wait_thread

```
1684 add_wait_queue(&current->signal->wait_chldexit,&wait);

//El proceso pasa de la lista RUNNING a la lista WAIT
```

```
1695 current->state = TASK_INTERRUPTIBLE;
//Detenido, pero responde a señales
```

```
1699 int tsk_result = do_wait_thread(tsk, &retval,
1700                                  type, pid, options,
1701                                  infop, stat_addr, ru);
```



Do_wait_thread

Crea y busca en toda la lista de hijos y

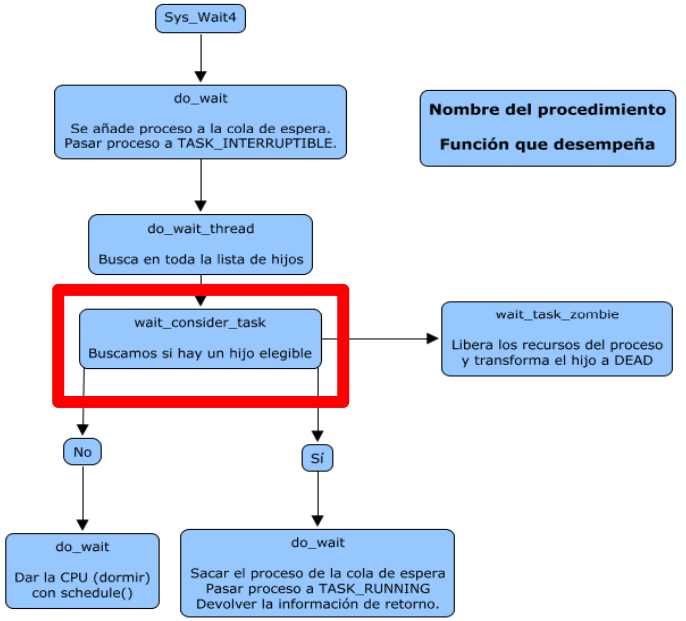
llamamos a la función wait_consider_task

```
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
          //Crea y busca en toda la lista de hijos donde p  tiene cada entrada de la lista
1635        list_for_each_entry(p, &tsk->children, sibling) {
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}
```

wait_consider_task

```
 //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        }
```

- Miramos si el hijo es elegible
- Después comprobamos si hemos podido acceder al proceso
- Si el hijo cumple estos criterios iremos a mirar su estado.



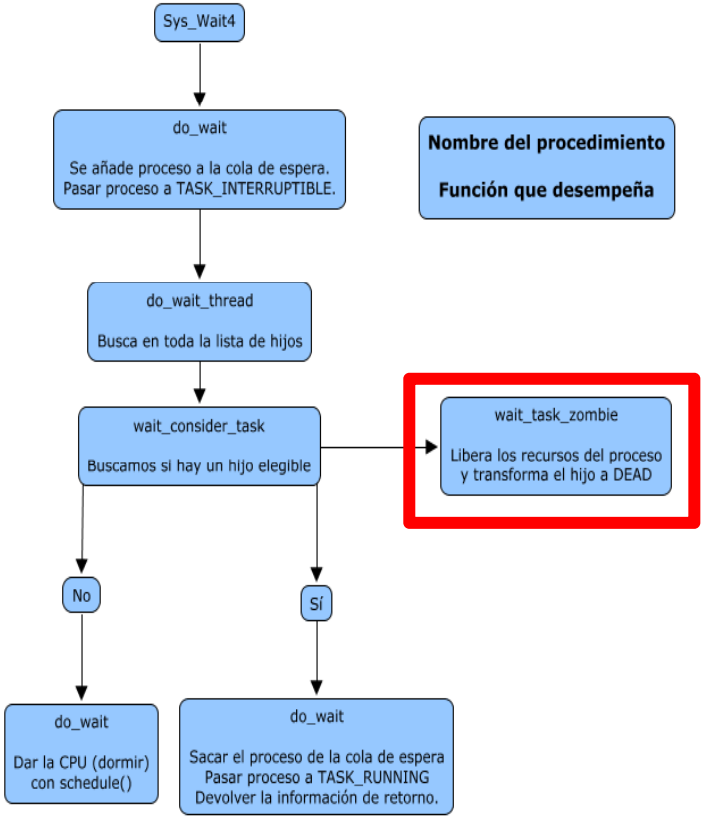
wait_consider_task

```
1597        if (p->exit_state == EXIT_DEAD)
1598                return 0;
1599
1600        /*
1601         * We don't reap group leaders with subthreads.
1602         */

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}
```

- Si el hijo está muerto no nos interesa.
- Si el hijo está en estado zombie entraremos en wait_task_zombie.
- Si está parado o en modo traza vamos a wait_task_stopped.
- Si no se cumple ninguna de esas condiciones, es decir, todavía esta continuando su ejecución vamos a wait_task_continued.

wait_task_zombie

```
//Si no, hay que pasar el proceso a DEAD. Se intenta
//cambiar el   estado a EXIT_DEAD
1294state = xchg(&p->exit_state, EXIT_DEAD);

// tomamos la información del proceso que necesitamos
//para retornar la llamada al sistema.
1368retval = ru ? getrusage(p, RUSAGE_BOTH, ru) : 0;

/* Elimina por completo al proceso hijo mediante la
//función release_task */
1416        if (p != NULL)
1417                release_task(p);

//Retornamos el valor
1419        return retval;
```



Otros casos

wait_task_stopped

Comprueba si el proceso es susceptible de ser elegido con los parámetros de la llamada al sistema, si no, devuelve cero.

```
    if (task_is_stopped_or_traced(p))
1613  return wait_task_stopped(ptrace, p, options,
1614  infop, stat_addr, ru);
```

wait_task_continued

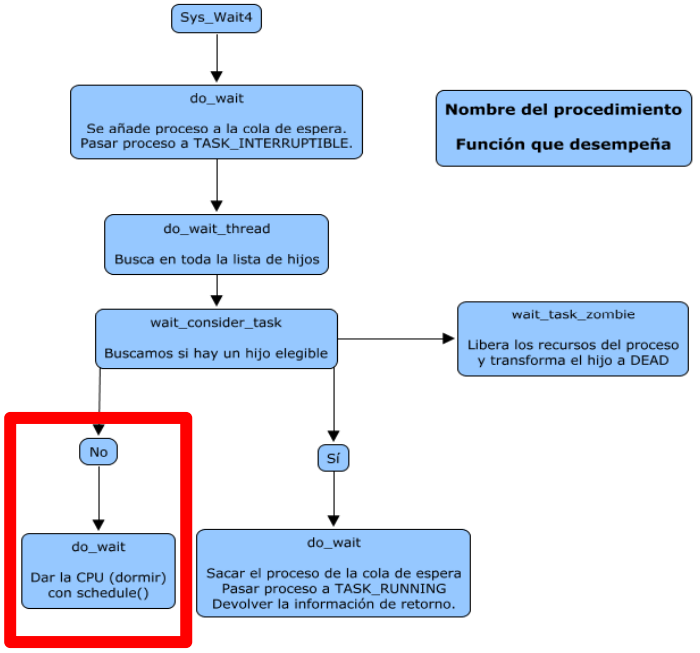
Cuando un proceso está ejecutándose actualmente, por ejemplo cuando ha cambiado de stopped a running. En ese caso toma la información de dicho proceso. Si devuelve 0 el proceso no nos interesa, en otro caso sí.

```
1616        return wait_task_continued(p, options, infop, stat_addr, ru);
```

do_wait - No

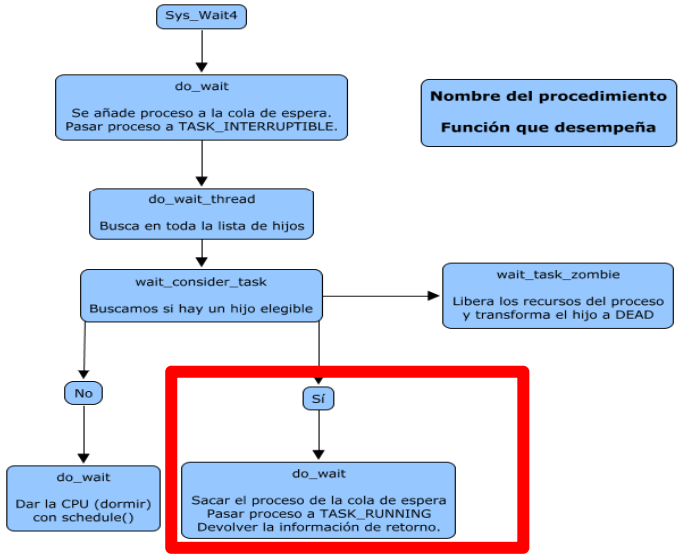
Si no se ha elegido la opción WNOHANG y no hemos encontrado ningún resultado, entonces llamamos al planificador para que pase a otro proceso y cuando nos devuelvan la CPU volvemos a rehacer la búsqueda.

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



do_wait - Sí

- Se cambia el estado del padre a TASK_RUNNING y se quita de la lista de procesos en espera
- Devuelve la información del proceso




```
1730    current->state = TASK_RUNNING;
        remove_wait_queue(&current->signal >wait_chldexit,&wait);


1755    return retval;
```

FIN