TEMA 2. GESTIÓN DE PROCESOS

2.1. Estados y transiciones de un proceso.
- 2.1.1. Ciclo de vida de un proceso en UNIX. Conjunto de estados.
- 2.1.2. Control sobre algunas transiciones a nivel de usuario.
- 2.1.3. Ciclo de vida de un proceso en Linux. Conjunto de estados.

2.2. Estructuras de datos del sistema para la gestión de procesos.
- 2.2.1. Tabla de procesos.
  - 2.2.1.1. Tabla de procesos en Linux.
- 2.2.2. U-Área.

2.3. Organización de la memoria del sistema.
- 2.3.1. Regiones.
- 2.3.2. Páginas y tablas de páginas.
- 2.3.3. Organización de la memoria del *kernel.*
- 2.3.4. Mapa de memoria para el u-Area (Área de usuario).

2.4. El contexto de un proceso. Concepto y tipos.

2.5. Manipulación del contexto de un proceso.
- 2.5.1. Interrupciones y excepciones. Tratamiento.
- 2.5.2. Interfaz de llamadas al sistema (algoritmo sys_call).
- 2.5.3. Cambio de contexto de un proceso.

2.6. Manipulación del espacio de direcciones de un proceso. Algoritmos.

2.7. Llamadas al sistema para la gestión de procesos.
- 2.7.1. Creación de un proceso (fork). Duplicación del proceso actual.
  - 2.7.1.1. Acciones más importantes del kernel al llamar a fork.
  - 2.7.1.2. Algoritmo detallado para la llamada al sistema fork.
  - 2.7.1.3. Observaciones sobre la creación de un proceso (padre e hijo).
  - 2.7.1.4. Clonado en Linux (clone).
- 2.7.2. Terminación y espera por la terminación de procesos. EXIT y WAIT.
  - 2.7.2.1. Terminación de procesos, exit.
  - 2.7.2.2. Espera para la terminación de procesos, wait.
  - 2.7.2.3. Ejemplo de la sincronización de procesos padre e hijo utilizando exit y wait.
- 2.7.3. Dormir (sleep) y despertar (wakeup) procesos.
  - 2.7.3.1. Contexto de ejecución de procesos que están dormidos.
  - 2.7.3.2. Eventos y direcciones en los que duermen los procesos.
  - 2.7.3.3. Problemas.
  - 2.7.3.4. Acciones para dormir (sleep).
  - 2.7.3.5. Acciones para despertar (wakeup).
  - 2.7.3.6. Conclusiones sobre sleep y wakeup.
- 2.7.4. Llamadas a otros programas. Familia de funciones exec.
  - 2.7.4.1. Estructura de un archivo ejecutable.
  - 2.7.4.2. Acciones: exec(nombre_archivo_ejecutable, parámetros).
  - 2.7.4.3. Comentarios sobre exec.

2.7.5. Información sobre procesos. Identificadores de proceso, Identificadores de usuario y grupo, Variables de entorno y Parámetros relativos a archivos.
- 2.7.5.1. Identificadores de proceso.
- 2.7.5.2. Identificadores de usuario y de grupo.
- 2.7.5.3. Variables de entorno.
- 2.7.5.4 Parámetros relativos a archivos.

2.8. Sincronización de procesos en Linux.
- 2.8.1. Bottom-halves.
- 2.8.2. Temporizadores del kernel (timers).
- 2.8.3. Colas de tareas.
- 2.8.4. Colas de espera.
- 2.8.5. Semáforos.
- 2.8.6. Algunos aspectos de implementación en la sincronización de procesos.

2.9. Señales y funciones de tiempo.
- 2.9.1. Concepto de señal.
- 2.9.2. Tipos de señales.
- 2.9.3. Tratamiento de las señales.
- 2.9.4. Descriptores de señales.
- 2.9.5. Funciones de tiempo.
- 2.9.6. Señales en Linux.

2.10. Nociones básicas de planificación de procesos en UNIX.
- 2.10.1. Algoritmo de planificación.
- 2.10.2. Parámetros de planificación.
- 2.10.3. Cálculo y control de prioridad de un proceso.
- 2.10.4. Scheduler de Linux.

2.1. ESTADOS Y TRANSICIONES DE UN PROCESO.

2.1.1. Ciclo Vida de un Proceso en UNIX. Conjunto Estados (Figura 2.1.).

Ya sabemos que un programa es una colección de instrucciones y de datos que se encuentran almacenados en un archivo que tiene en su inodo un atributo que lo identifica como ejecutable. Puede ser ejecutado por el propietario, por el grupo y por el resto de los usuarios, dependiendo de los permisos que tenga el archivo. Cuando un programa es leído del disco (a través del sistema de archivos) por el *kernel* y es cargado en memoria para ejecutarse, se convierte en un *proceso*. También debemos recordar que en un proceso no sólo hay una copia del programa, sino que además el *kernel* le añade información adicional para poder manejarlo.

El tiempo de vida de un proceso se puede dividir en un conjunto de estados, cada uno con unas características determinadas. Es decir, que un proceso no permanece siempre en un mismo estado, sino que está continuamente cambiando de acuerdo con unas reglas bien definidas. Estos cambios de estados vienen impuestos por la competencia que existe entre los procesos para compartir un recurso tan escaso como es la CPU (*scheduler*). La transición entre los diferentes estados (Figura 2.1) da lugar a un Diagrama de Transición de Estados. Un diagrama de transición de estados es un grafo dirigido, cuyos nodos representan los estados que pueden alcanzar los procesos y cuyos arcos representan los eventos que hacen que un proceso cambie de un estado a otro.

En la Figura 2.1. podemos ver el diagrama completo y los estados que en él se reflejan son:

- Entrada del proceso en el modelo, creación de un proceso: estado Creado (8).
  - Este estado es el estado inicial para todos los procesos excepto el proceso 0 (swapper).
  - Llamada al sistema fork del padre ⇒ Jerarquía de procesos.
  - El proceso acaba de ser creado y está en un estado de transición; el proceso existe, pero ni está preparado para ejecutarse (estado 3), ni durmiendo (estado 4)
- Proceso ejecutándose en modo usuario (1). Los eventos que desencadenan una transición pueden ser los siguientes:
  - [Evento = fin quantum] ⇒ transición al estado de "ejecutándose en modo *kernel*" (2).
    - Gestión de la interrupción. Esta función la lleva a cargo el módulo de control del hardware, que es la parte del *kernel* encargada del manejo de interrupciones y de la comunicación con la máquina.
    - Planificación a cargo del scheduler.
      - Elegir otro proceso si es el siguiente para ser ejecutado según el scheduler.
      - Proceso actual pasa a → Requisado (7) [Evento = requisar (o apropiar), en el que el *kernel* se apodera del proceso y hace un cambio de contexto, pasando otro proceso a ejecutarse en modo usuario] → posiblemente a listo para ejecutar en modo usuario.
  - [Evento = llamada al sistema] ⇒ transición al estado de "ejecutándose en modo *kernel*" (2).
    - Operación de Entrada/Salida.
      - El proceso pasa al estado "durmiendo en memoria" (4) [Evento = dormir].
    - Fin de la operación de Entrada/Salida.
      - El hardware interrumpe la CPU. Como sabemos los dispositivos pueden interrumpir la CPU mientras está ejecutando un proceso. Si esto ocurre, el *kernel* debe reanudar la ejecución del proceso después de atender a la interrupción. Las interrupciones no son atendidas por procesos, sino por funciones especiales, codificadas en el *kernel*, que son llamadas durante la ejecución de cualquier proceso.
      - Estado listo para ejecutarse en memoria (3) [Evento = despertar].

- Ejecución en modo *kernel* (2).
  - Gestión interrupciones y llamadas al sistema, planificación, etc. Esta transición se debe a eventos que sólo el *kernel* puede controlar y no pueden pasar a otro estado de tránsito [Evento = interrupción / volver de interrupción]
  - Si viene de recién creado (8) → completa su parte del fork ante un evento de "hay suficiente memoria".
  - Cuando el proceso termine de ejecutarse en modo *kernel* → pasa a ejecutarse en modo usuario (1) [Evento = volver].
- Listo para ejecutarse en memoria (3) o en el área de swap en memoria secundaria (5).
  - Listo para ejecutarse en memoria (3).
    + El proceso no se está ejecutando, pero está listo para ejecutarse tan pronto como el scheduler lo ordene [Evento = orden de ejecución por parte del scheduler]. Puede haber varios procesos simultáneamente en este estado.
    + Si viene de recién "Creado" (8), completa su parte del fork [Evento = hay memoria suficiente].
  - Listo para ejecutarse en el área de swap, en memoria secundaria (5).
    + No hay memoria principal suficiente para todos los procesos [Evento = memoria principal insuficiente].
    + El proceso está listo para ejecutarse pero el swapper (proceso 0) debe cargar el proceso en memoria secundaria (swap out) antes de que el *kernel* pueda ordenar que pase a ejecutarse.
- Durmiendo cargando en memoria (4) o en el área de swap en memoria secundaria (6).
  - El proceso está durmiendo cargado en memoria principal (4). Un proceso entra en este estado cuando no puede proseguir su ejecución porque está esperando a que se complete una operación de entrada/salida [Evento = dormir].
  - El proceso está durmiendo y el swapper ha descargado el proceso hacia memoria secundaria (6) (en el área de swap) para poder crear espacio en memoria principal para poder cargar otros procesos [Evento = sacar de memoria principal para enviar a disco (swap out)]. Una vez, que el proceso ha estado durmiendo en memoria secundaria, según un evento despertar [Evento = despertar] puede pasar al estado "listo para ejecutarse en memoria secundaria (área de swap)".
- Requisado o expulsado o apropiado (7).
  - El proceso está volviendo del modo *kernel* al modo usuario, pero el *kernel* se apropia del proceso y hace un cambio de contexto, pasando otro proceso a ejecutarse en modo usuario [Evento = requisar (cambio de contexto)].
  - Desde este estado puede pasar el proceso al estado de "ejecutándose en modo usuario" gracias a un evento de volver al modo usuario [Evento = volver al modo usuario].
- Estado Zombie (9).
  - Fin de ejecución de un proceso pero sigue siendo referenciado en el sistema ⇒ Llamada al sistema exit [Evento = exit] y pasa a estado *Zombie*. El proceso ya no existe, pero deja para su proceso padre un registro que contiene el código de salida y algunos datos estadísticos tales como los tiempos de ejecución. El estado *Zombie* es el estado final de un proceso
    + Estado de proceso ejecutándose en modo *kernel* [Evento = exit] ⇒ Estado *Zombie*.
  - Recurso: tabla de procesos.

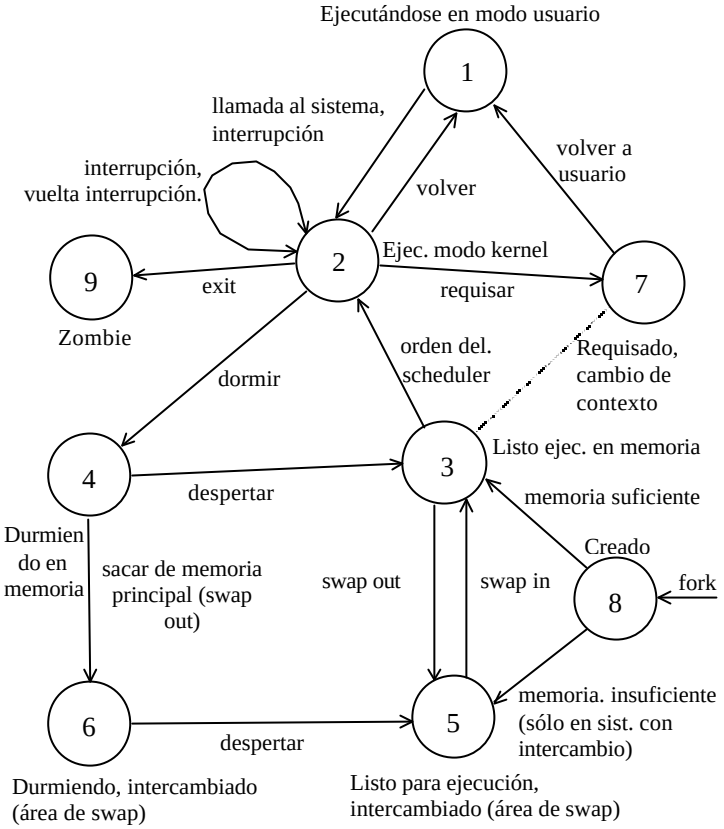

Figura 2.1. Diagrama de transición de estados de un proceso en un sistema UNIX.

Recordemos que existen dos niveles de ejecución para un proceso: modo *kernel* (2) y modo usuario (1).

- Modo *kernel* (modo privilegiado). En este modo no se impone ninguna restricción al *kernel* del sistema. El *kernel* podrá:
  - Utilizar todas las instrucciones del procesador.
  - Manipular toda la memoria.
  - Dialogar directamente con todos los controladores (drivers) de dispositivos, ...
- Modo usuario. Modo de ejecución normal de un proceso. En este modo el proceso no posee ningún privilegio.
  - Ciertas instrucciones están prohibidas.
  - Sólo tiene acceso a las zonas que se le han asignado al proceso.
  - No puede interactuar con la máquina físicas (hardware).
  - Sólo efectúa operaciones en su entorno, sin interferir con los demás procesos.
  - Puede ser interrumpido en cualquier momento, pero esto no obstaculiza su funcionamiento.

Recordemos también que un proceso que se ejecute en modo usuario no puede acceder directamente a los recursos de la máquina, para ello debe efectuar "llamadas al sistema". Una llamada al sistema es una petición transmitida por un proceso al *kernel*. El *kernel* trata la petición en modo *kernel*, con todos los privilegios, y envía el resultado al proceso, que prosigue su ejecución en modo usuario. Una llamada al sistema provoca un cambio: (1) el proceso ejecuta una instrucción del procesador que le hace pasar a modo *kernel* (interfaz de llamadas al sistema), (2) el *kernel* ejecuta una función de tratamiento vinculada a la llamada al sistema que ha efectuado, y (3) el proceso vuelve a modo usuario para proseguir con su ejecución. De este modo, el propio proceso trata su llamada al sistema, ejecutando una llamada al *kernel*. Esta función que ejecuta el *kernel*, se supone que es fiable y puede ejecutarse en modo *kernel*.

2.1.2. Control sobre Algunas Transiciones que se dan a Nivel Usuario.

- Crear otro proceso.
- Llamadas al sistema.
  - "ejecutándose en modo usuario" ⇒ "ejecutándose en modo *kernel*".
- Puede finalizar (exit) por voluntad propia.
- El resto de las transiciones → Modelo rígido codificado en el *kernel* (Diagrama de Transición de Estados).

2.1.3. Ciclo de Vida de un Proceso en Linux. Conjunto de Estados (Figura 2.2.).

En Linux, un proceso o tarea, está representado por una estructura de datos *task_struct*. Linux mantiene una task, que es un vector lineal de punteros a cada estructura *task_struct* definida en un instante dado. La estructura *task_struct* contiene información de varios tipos:

- **Estado**. Representa el estado de ejecución de un proceso (en ejecución, interrumpible, no interrumpible, parado y zombie).
- **Información de planificación**. Representa a la información necesaria para la planificación de procesos en Linux. Un proceso puede ser normal o en tiempo real y tiene una prioridad. Los procesos en tiempo real se planifican antes que los procesos normales y se utilizan prioridades relativas dentro de cada categoría. El tiempo en el que un proceso puede ejecutarse se controla mediante un contador.
- **Identificadores**. Cada proceso tiene un único identificador de proceso, y tiene también identificadores de usuario y de grupo. Un identificador de grupo se utiliza para asignar privilegios de acceso a recursos de un grupo de usuarios.
- **Comunicación entre procesos**. Linux soporta los mecanismo de IPC de UNIX
- **Vínculos**. Cada proceso incluye un vínculo con su proceso padre, vínculos con sus hermanos (procesos con el mismo padre) y vínculos con todos sus hijos.
- **Tiempos y temporizadores**. Incluye el instante de creación del proceso y la cantidad de tiempo del procesador consumido hasta el instante. Un proceso puede tener también asociados uno o más temporizadores de intervalo. Un proceso define un temporizador de intervalo mediante una llamada al sistema; cuando el tiempo termina se envía una señal al proceso. Un temporizador puede ser de un solo uso o periódico.
- **Sistema de archivos**. Incluye punteros a cualquier archivo abierto por el proceso.
- **Memoria virtual**. Define la memoria virtual asignada al proceso.
- **Contexto específico del procesador**. La información de registros y pila que forma el contexto del proceso (espacio de direccionamiento del proceso).

Los estados de ejecución de un proceso Linux son los siguientes:

- **En Ejecución (TASK_RUNNING)**. El proceso es ejecutado por el procesador.
- **Listo (TASK_RUNNING)**. El proceso podría ser ejecutado, pero otro proceso se está ejecutando en ese momento.
- **Interumpible (TASK_INTERRUPTIBLE)**. Es un estado de bloqueo, en el cual el proceso espera un suceso, como la terminación de una operación de entrada/salida, liberación de un recurso o una señal de otro proceso.
- **No interumpible (TASK_UNINTERRUPTIBLE)**. Es otro estado de bloqueo. La diferencia entre éste y el estado Interumpible es que en un estado No Interumpible, un proceso espera directamente en una condición de hardware y, por tanto, no acepta señales.
- **Parado (TASK_STOPPED)**. El proceso ha sido detenido por una intervención externa y sólo puede reanudarse por una acción positiva de otro proceso. Por ejemplo, un proceso puede estar en estado Parado durante la ejecución de un programa de depuración.
- **Zombie (TASK_ZOMBIE)**. El proceso ha terminado pero, por alguna razón, su estructura *task* debe permanecer aún en la tabla de procesos.

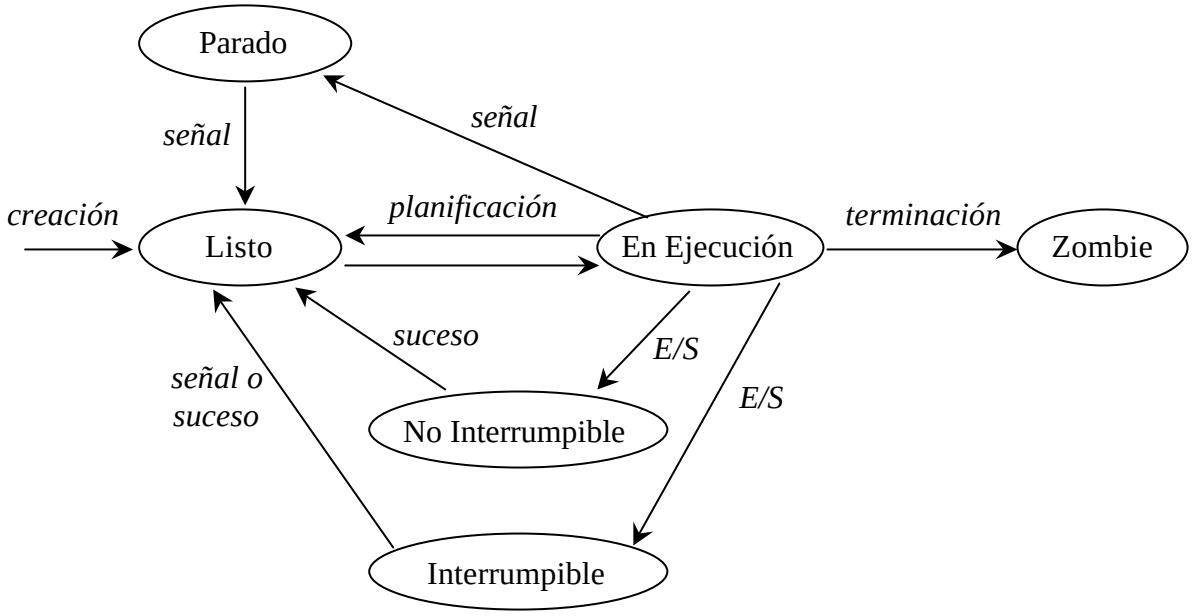

Figura 2.2. Diagrama de transición de estados de un proceso en un sistema Linux.

2.2. ESTRUCTURAS DE DATOS DEL SISTEMA PARA LA GESTIÓN DE PROCESOS.

Todo proceso tiene asociada una entrada en la Tabla de Procesos y un Área de Usuario (*u-Area*). Estas dos estructuras van a describir el estado del proceso y le van a permitir al *kernel* su control. La Tabla de Procesos tiene campos que van a ser accesibles desde el *kernel*, pero los campos del Área de Usuario sólo necesitan ser visibles por el proceso. Una posición (entrada) por proceso.

Las Áreas de Usuario se reservan cuando se crea un proceso y no es necesario que una entrada de la Tabla de Procesos que no aloja a ningún proceso tenga reservada un área de usuario.

2.2.1. Tabla de Procesos.

Cada proceso se referencia por un descriptor (descriptor del proceso). Este descriptor contiene los atributos del proceso, así como la información que permite gestionarlo.

Campos que siempre deben ser accesibles por el *kernel* y que tiene cada una de las entradas (descriptor de un proceso) de la Tabla de Procesos.

- Campo de estado, que identifica el estado del proceso (9 posibles valores estudiados previamente).
- Campos que permiten al *kernel* localizar al proceso y su área de usuario (u-Area) en memoria principal o en memoria secundaria.
  - El *kernel* usa esta información para realizar los cambios de contexto cuando el proceso pasa de un estado a otro.
    - Listo para ejecutarse en memoria (cambio de contexto) $\Rightarrow$ ejecutándose en modo *kernel.*
    - Requisado (cambio de contexto) $\Rightarrow$ ejecutándose en modo usuario.
  - El *kernel* también utiliza esta información cuando traslada el proceso de la memoria principal al área de swap, y viceversa $\Rightarrow$ Intercambio de procesos desde o hacia la memoria principal.
  - En estos campos también hay información sobre el tamaño del proceso, para que el *kernel* sepa cuánto espacio de memoria debe reservar para él.

- Algunos identificadores de usuario (UID) para determinar los privilegios del proceso. Por ejemplo, el campo UID delimita a qué proceso puede enviar señales el proceso actual y qué procesos pueden enviárselas a él. Cada señal se corresponde con un evento particular, y permite a los procesos reaccionar a los eventos provocados por ellos mismos o por otros procesos, es decir, las señales son interrupciones software que pueden ser enviadas a un proceso para informarle de algún evento asíncrono o situación especial.
- Identificadores del proceso (PID) que especifican las relaciones de unos procesos con otros. Estos identificadores son establecidos cuando se crea el proceso mediante una llamada fork.
- Descriptores de eventos, por ejemplo cuando el proceso está en el estado "durmiendo" y que serán utilizados al "despertar".
- Parámetros de planificación, que permiten al *kernel* determinar el orden en el que los procesos pasan del estado "ejecutándose en modo *kernel*" a "ejecutándose en modo usuario".
- Un campo de señales que enumera las señales que han sido recibidas, pero que no han sido tratadas todavía.
- Tabla de regiones de ducho proceso, que representan las regiones de memoria contenidas en el espacio de direcciones de un proceso.
- Algunos temporizadores,
  - Que indican:
    + Tiempo de ejecución del proceso.
    + Utilización de recursos del *kernel.*
  - Estos campos se utilizan también para:
    + Llevar la contabilidad de los procesos.
    + Calcular la prioridad dinámica que se le asigna a cada proceso y que luego utilizará el scheduler para decidir qué proceso debe ser ejecutado por el procesador (CPU).
    + Aquí también hay un temporizador que puede programar el usuario y que se utiliza para enviarle la señal SIGALRM.

2.2.1.1. Tabla de Procesos en Linux.

En Linux, inicialmente, la Tabla de Procesos era un vector de tamaño fijo de *task_strcut <linux/sched.h>*, con lo que el número máximo de procesos estaba limitado. Actualmente, la Tabla de Procesos es realmente una lista doblemente enlazada mediante los punteros *next_task* y *prev_task*.

Estructura del descriptor de procesos en Linux (PCB = Process Control Block).

En Linux el PCB (Process Control Block) es una estructura denominada *task_strcut* en el archivo *include/linux/sched.h*. En ella aparece tipo de información sobre cada uno de los procesos. Muchas partes del sistema operativo hacen uso de esta estructura de datos, por lo que es necesario conocer los campos más importantes de *task_struct.*

- *volatile long state*: Estado del proceso (activo o bloqueado). En la línea 76 de *include/linux/sched.h* están los posibles valores que puede tomar el estado.
- *unsigned long flags*: Estado detallado del proceso, a nivel del *kernel.*
- *long counter*: Número de "ticks" (ciclos de reloj) que restan al proceso actual para que se le acabe el quantum.
- *long priority*: Prioridad estática del proceso. A partir de este valor, el scheduler asigna el valor de *counter* cada vez que se agota.
- *struct task_struct *next_run, *prev_run*: Punteros para implementar la lista de procesos activos. El scheduler busca en esta lista cada vez que tiene que elegir un proceso nuevo para ejecutar. Cuando un proceso se suspenda, se extraerá inmediatamente de esta lista.
- *int exit_code, exit_signal*: Contiene el valor de terminación de un proceso, en caso de que haya finalizado mediante la llamada al sistema *exit(2)*. Si termina por una señal, contendrá el número de señal que lo mató.
- *int pid*: Número identificador del proceso. Cada proceso tiene un PID distinto.

- *struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr*: Punteros para acceder a toda la familia de procesos: op(Original Parent), p(Parent), c(Youngest Child), ys(Youngest Sibling, hermano más joven), os(Oldest Sibling, hermano más viejo). Los inicia lizan las macros SET_LINKS y REMOVE_LINKS llamadas al crear un proceso (fork) y al destruirlo (exit, kill).
- *unsigned long policy, rt_priority*: Indican la política de planificación del proceso y la prioridad dentro de esa política.
- *long start_time*: Instante de creación de este proceso.
- *uid_t uid, euid, suid, fsuid*: Usuario propietario de este proceso, tanto real (uid), como efectivo (euid), y atributos más específicos.
- *gid_t gid, egid, sgid, fsgid*: Grupo propietario de este proceso, tanto real (gid), como efectivo (egid), y atributos más específicos.
- *unsigned long signal, blocked*: *signal* es un mapa de bits con las señales pendientes de ser enviadas a este proceso. Y *blocked* es otro mapa de bits con las señales que están temporalmente bloqueadas.
- *struct fs_struct *fs*: Información relativa al sistema de archivos. Los campos de fs no están directamente dentro de *task_struct* para facilitar que varios procesos compartan la información que contienen. Esta funcionalidad la usan los threads a nivel del *kernel* (llamada al sistema clone(2)).
- *struct files_struct *files, struct mm_struct *mm, struct signal_struct *sig*. Esto es reflejo de que los threads de ejecución pueden compartir también los archivos abiertos, su espacio de memoria y las señales. Contiene los siguientes campos:
  - *atomic_t count*: Número de procesos que comparten esta estructura.
  - *int umask*: Mascara de creación de archivos. La llamada al sistema umask permite consultar y modificar esta variable. La orden interna del shell umask permite modificarla desde la línea de órdenes. Cuando un proceso crea un archivo nuevo (open, creat), el *kernel* quitará los bits de permisos que estén activos en umask: permisos = permisos_creat ~umask.
  - *struct dentry * root*: Inodo del directorio raíz de este proceso. Con la llamada al sistema chroot(2) se puede cambiar este atributo.
  - *struct dentry * pwd*: (Process Working Directory) inodo del directorio de trabajo. chdir (2) es la llamada al sistema que permite modificarlo.
- *struct files_struct *files*; Esta estructura contiene toda la información relativa a los archivos abiertos por el proceso. Algunos de sus campos son:
  - *int count*: Número de procesos que comparten estos archivos abiertos.
  - *int next_fd*: Primer identificador libre.
  - *fd_set close_on_exec*: Conjunto bits que indica qué archivos se tienen que cerrar cuando este proceso realice una operación exec(2). Ver la llamada al sistema fcntl(2).
  - *fd_set open_fds*: Mapa de bits que indica los archivos que actualmente tiene abiertos este proceso.
  - *struct file *fd[NR_OPEN]*: Vector de descriptores de archivos en uso.

Organización de la tabla de procesos en Linux (lista doblemente enlazada).

- Los descriptores de proceso los asigna dinámicamente el *kernel* llamando a una función especial (kmalloc). El array *task* contiene punteros a estos descriptores. El array *current_set* contiene punteros a los descriptores de procesos en curso de ejecución en cada procesador.
- La variable *init_task* contiene el descriptor del primer proceso creado en el arranque del sistema. Tras el arranque, este proceso sólo se ejecuta cuando ninguno más este listo para ejecución, y su descriptor sirve para recuperar el inicio de la Tabla de Procesos.
- Los descriptores de procesos se organizan en forma de una lista doblemente enlazada, por los punteros *next_task* y *prev_task*. Los descriptores de procesos que estén listos para su ejecución o en curso de su ejecución se colocan en otra lista doblemente enlazada, mediante los campos *next_run* y *prev_run*.
- Los campos *p_optr*, *p_pptr*, *p_cptr*, *p_ysptr* y *p_osptr*, se utilizan para gestionar las filiaciones entre procesos. Entendiéndose por filiación (relación familiar) la relación que existe entre los procesos dentro de la jerarquía de procesos.

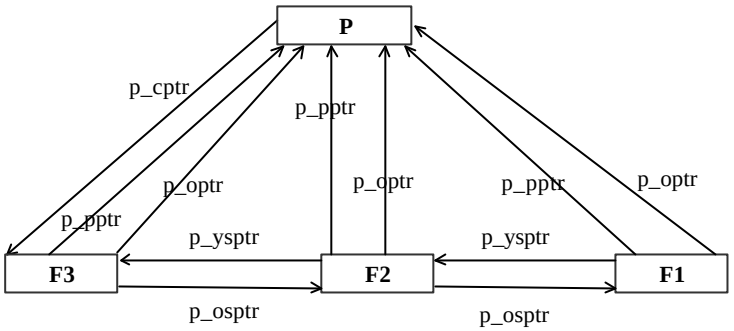

Figura 2.3. Relaciones (filiaciones) entre descriptores de procesos en el caso de que un proceso P ha creado sucesivamente tres procesos hijos F1, F2 y F3.

- Cuando un proceso se duplica, llamando a la primitiva fork:
  - Los punteros "p_optr" y "p_pptr" del descriptor del proceso hijo, contienen la dirección del proceso padre.
  - El puntero "p_osptr" del descriptor del proceso hijo toma el valor del puntero "p_cptr" del descriptor del proceso padre.
  - El puntero "p_ysptr" del proceso hermano más reciente (referenciado por el puntero "p_cptr" del descriptor del proceso padre) contiene la dirección del descriptor del nuevo proceso hijo.
  - El puntero "p_cptr" del descriptor del proceso padre contienen la dirección del descriptor del proceso hijo.
- La gestión de la tabla de procesos (listas de descriptores de procesos) es llevada a cabo por el scheduler.

2.2.2. u-Area

El área de usuario contiene información que es necesaria sólo cuando el proceso se está ejecutando.

Campos que sólo necesitan ser accesibles para el proceso en ejecución.

- Puntero a la entrada de la Tabla de Procesos correspondiente al proceso.
- Identificadores del usuario real y efectivo (UID), que determina algunos privilegios permitidos al proceso, tales como permiso de acceso a un archivo. El identificador del usuario real del proceso es el identificador del usuario que ha lanzado el proceso. El identificador del usuario efectivo es el identificador que utiliza el sistema para los controles de acceso a archivos para determinar el propietario de los archivos recién creados y los permisos para enviar señales a otros procesos. Puede ser distinto del identificador de usuario real, por ejemplo, en el caso de programas que tienen el bit *setuid*.
- Identificadores de grupo real y efectivo. El identificador de grupo real del proceso es el identificador del grupo primario que ha lanzado el proceso. El identificador de grupo efectivo es el identificador que utiliza el sistema para los controles de acceso para determinar el propietario de los archivos recién creados y los permisos para enviar señales a otros procesos. Puede ser distinto del identificador de grupo real, por ejemplo, en el caso de programas que tienen el bit *setuid*. Además existe una lista de identificadores de grupo ya que todo usuario puede pertenecer a varios grupos de forma simultánea y el *kernel* mantiene una lista de grupos asociados a cada proceso a fin de proceder a los controles de acceso.
- Temporizadores (contadores) que registran el tiempo empleado por el proceso (y sus descendientes) ejecutándose en modo *kernel* y modo usuario.
- Un array que indica cómo ha de reaccionar el proceso ante las señales que recibe.
- Campo de terminal que identifica el "terminal de login" (inicio de sesión) asociado al proceso, que indica cuál es, si existe, el terminal de control.
- Campo de errores que registra los errores producidos durante una llamada al sistema.

- Campo de valor de retorno que almacena el resultado de las llamadas al sistema efectuadas por el proceso.
- Parámetros de E/S que describen la cantidad de datos a transferir, la dirección del array origen o destino de la transferencia y los punteros de Lectura/Escritura del archivo al que se refiere la operación de Entrada/Salida.
- El directorio de trabajo actual y el directorio raíz asociados al proceso.
- Tabla de descriptores de archivos que identifica los archivos que tiene abiertos el proceso.
- Campos de límite que restringen el tamaño del proceso y el tamaño de algún archivo sobre el que puede escribir.
- Una máscara de permisos que va a ser utilizada cada vez que se cree un nuevo archivo.

En resumen, para gestionar los procesos, el *kernel*, en general, divide (conceptualmente y en la implementación) la información del proceso en dos apartados: la necesaria para localizarlo y planificarlo (Tabla de Procesos) y el resto de informaciones referentes la proceso (u-Area). *Tabla de Procesos* ⇒ localizar el proceso y su u-Area; *u-Area* ⇒ caracterización detallada del proceso.

Tabla de Procesos	**u-Area**
Global del Kernel (todos los procesos)	Local, asociada al proceso (sólo visible al proceso en ejecución)
Estática (n entradas)	Dinámica (solo para procesos en el sistema)
Estado	Puntero a la tabla de procesos
Localización de la u-Area	Identificadores (permisos), máscaras de permisos (create)
Tamaño del proceso	Detalle de tiempos
Identificadores (grupos, señales)	Acción ante señales
Evento en el que duerme	Terminal asociada, cuotas de proceso y archivos
Parametros de planificación	Campo de error en llamadas al sistema
Señales recibidas no atendidas	Valor de retorno en llamadas al sistema
Tiempos y recursos consumidos	Parámetros para E/S, Curr_dir y Root_dir, Tabla de archivos

2.3. ORGANIZACIÓN DE LA MEMORIA DEL SISTEMA.

- La memoria física de una máquina es direccionable desde el byte 0 hasta el byte que se corresponde con la memoria total de la máquina.
- En UNIX, el *kernel* genera direcciones para un proceos en un espacio de direcciones virtual dentro de un rango dado.
  - No hay otros procesos en dicho rango de direcciones virtual ⇒ no hay conflictos para el uso de la memoria entre procesos.
  - La memoria es infinita.
- Unidad de gestión de memoria de la máquina.
  - Direcciones virtuales ⇒ Direcciones físicas.
- Los subsistemas del *kernel* y del hardware (control del hardware) cooperan para realizar la traducción de direcciones virtuales a físicas ⇒ **subsistema de gestión de memoria**.

Espacio de direccionamiento de un proceso.

A todo proceso se le asocia un espacio de direccionamiento que representa las zonas de memoria asignadas a dicho proceso. Este espacio de direcciones disponibles se descompone en regiones de memoria utilizables por el proceso. Este espacio de direcciones incluye:

- El código (texto) del proceso.
- Los datos del proceso, que se descompone en dos segmentos, por una parte *data*, que contiene las variables inicializadas, y por otra parte *bss* (*block started symbol*) que contiene las variables no inicializadas.
- La pila utilizada por el proceso

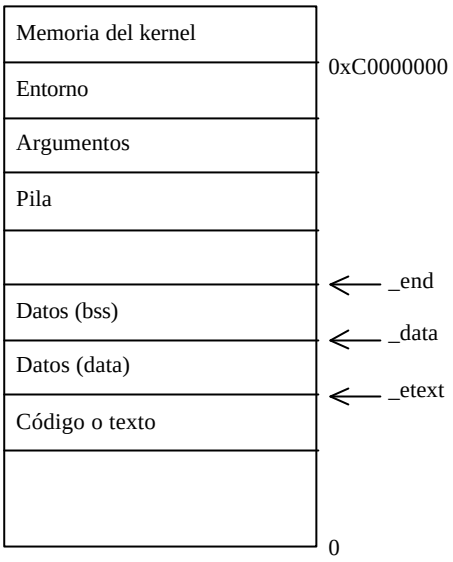

Figura 2.4. Espacio de direccionamiento de un proceso.

Asignación de memoria.

Cuando un proceso empieza su ejecución, sus segmentos (código, datos y pila) poseen un tamaño fijo. Sin embargo, existen funciones de asignación y liberación de memoria, que permiten a un proceso manipular variables cuyo número o tamaño no es conocido en el momento de su compilación ⇒ asignación y liberación de memoria de forma dinámica (funciones malloc, calloc, realloc, free).

Las asignaciones y liberaciones se efectúan modificando el tamaño del segmento de datos del proceso. Cuando debe asignarse un dato, el segmento de datos aumenta en el número de bytes necesario y el dato puede almacenarse en el espacio de memoria así asignado. Cuando un dato situado al final del segmento de datos deja de utilizarse, su liberación consiste simplemente en reducir el tamaño del segmento.

2.3.1. Regiones.

- UNIX divide el espacio de direcciones virtual del proceso en zonas lógicas. Como se ha expuesto anteriormente el espacio de direccionamiento de un proceso se compone de varias regiones de memoria. Cada región de memoria se caracteriza por varios atributos:
  - Sus direcciones de inicio y fin.
  - Los derechos de acceso que tiene asociados.
  - El objeto asociado (por ejemplo, un archivo ejecutable que contiene el código ejecutable para el proceso).
- Región: Área contigua del espacio de direcciones virtuales de un proceso que puede ser tratada como un objeto a ser protegido o compartido.
  - Como entidad, se puede tratar como un objeto cuyas operaciones principales son:
    + Creación y eliminación de regiones de memoria.
    + Proteger regiones de memoria.
    + Modificaciones para las protecciones de las regiones de memoria.
    + Reasignar regiones de memoria.
    + Compartir regiones de memoria.
  - Concepto de Región: Independiente de la política de gestión de memoria implementada en el sistema operativo.
- Las regiones de memoria contenidas en el espacio de direccionamiento de un proceso pueden determinarse mostrando el contenido del archivo *maps*, situado en el directorio de cada proceso en el sistema de archivos */proc*.

Descripción de Regiones.

- Tabla de regiones del Sistema.
  - Esta tabla es gestionada por el *kernel* (global a todo el sistema).
  - Una entrada se corresponde con cada región activa del sistema.
    + Información necesaria para localizar a la región en memoria física. Direcciones de inicio y fin.
    + Tamaño de la región.
- Tabla de regiones por proceso o *pregion*.
  - Esta tabla es privada para cada proceso.
  - Esta tabla está relacionada con la tabla de procesos.
  - Cada entrada.
    + Apunta a una entrada en la tabla de regiones.
    + Direcciones virtuales de comienzo y de final de la región en el proceso.
    + Los derechos de acceso que tiene asociados la región. (r = lectura, w = escritura, x = ejecución y p = indica si la región puede ser compartida por otros procesos)
    + Otros campos que indican la información relativa al objeto asociado a la región de memoria: el desplazamiento del inicio de la región en el objeto, número de dispositivo que tiene asociado el objeto y el número de inodo del objeto.
- Regiones compartidas ⇒ diferentes direcciones virtuales para cada proceso (Figura 2.5). En dicha figura podemos ver dos procesos compartiendo la mismo región de código, pero la misma región está situada en dos direcciones virtuales diferentes (l 8 para el proceso *a* y la 4 para el proceso *b*)

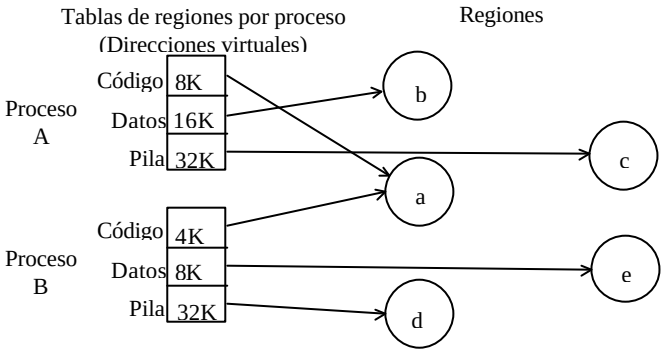

Figura 2.5. Procesos y regiones.

Este esquema de regiones facilita la compartición de memoria, ya que para ello bastará que desde dos tablas de regiones de proceso se *apunte* a la misma zona de memoria. Por esta razón, es necesario un campo con la dirección de inicio de la región, ya que la misma región podría estar en dos direcciones virtuales diferentes para dos procesos diferentes. Además, observe que para poder compartir las regiones necesitamos una Tabla global de regiones donde llevemos control del número de procesos que comparten la región, indicación de que la región puede ser compartida, etc. y también surgirán algunas limitaciones para algoritmos que se puedan ejecutar sobre este tipo de regiones (por ejemplo el crecimiento dinámico de la región).

2.3.2. Páginas y Tablas de Páginas.

UNIX utiliza los mecanismos de memoria virtual proporcionados por el procesador sobre el que se ejecuta (paginación y segmentación). Las direcciones manipuladas por el *kernel* y los procesos son direcciones lógicas y el procesador junto con el sistema operativo efectúan una conversión para transformar una dirección lógica en dirección física en memoria principal. Ahora, vamos a presentar como modelo de trabajo un esquema de gestión de memoria por *Páginas* (bloques de memoria de igual tamaño [512B...4KB], cuyo formato de dirección es del tipo *(página, desplazamiento)*. El *kernel* asigna páginas a las regiones (sin continuidad ni orden) lo que le proporciona gran flexibilidad, con un nivel controlado de fragmentación (en la última página de cada región) y un coste razonable de gestión (memoria para las tablas).
La idea en que se basa la memoria virtual es que el tamaño combinado del código, datos y pila puede exceder la cantidad de memoria física disponible. El sistema operativo mantiene en memoria principal las partes del

programa que actualmente se están utilizando y el resto en disco. La memoria virtual también puede funcionar en un sistema de tiempo compartido, manteniendo segmentos de muchos programas en memoria a la vez. Mientras que un programa está esperando que se traiga a la memoria principal una de sus partes, está esperando una operación de E/S y no puede ejecutarse, así que puede otorgarse la CPU a otro proceso, lo mismo que cualquier otro sistema de tiempo compartido.

- Las direcciones generadas por los programas ⇒ direcciones virtuales ⇒ Espacio de direcciones virtuales. El espacio de direcciones virtuales se divide en unidades de nominadas páginas.
- Arquitectura de gestión de memoria basada en páginas ⇒ hardware de gestión de memoria divide la memoria física en una serie de bloques de igual tamaño denominados páginas.
- Tamaño página ⇒ definido por el hardware.
- Cada posición direccionable de la memoria está contenida en una página, direccionada de la siguiente forma (Figura 2.6): *(número de página, desplazamiento dentro de la página)*.

  El mecanismo de conversión es el siguiente: una dirección de memoria virtual (lineal) se descompone en dos partes, un número de página y un desplazamiento dentro de la página. El número de página se utiliza como índice en una tabla, llamada tabla de páginas, lo que proporciona una dirección física de página en memoria principal. A esta dirección se le añade el desplazamiento para obtener la dirección física de la palabra de memoria en concreto.
- La Tabla de Páginas ⇒ su objetivo es transformar páginas virtuales en páginas físicas. Realmente es una función con el número de página virtual como parámetro y el número de página física como resultado. Es decir, *f(Número_Página_Virtual) = Número_Página_Física*. Si la tabla de página llega a ser muy grande en tamaño que consume mucha memoria, la mejor solución es crear tablas de páginas multinivel.
- Algoritmos de Reemplazamiento de páginas:
  - Óptimo.
  - NRU (No utilizada recientemente).
  - FIFO (Primero en llegar, primero en salir).
  - Clock (Sustitución de página por reloj).
  - LRU (Menos utilizada recientemente).

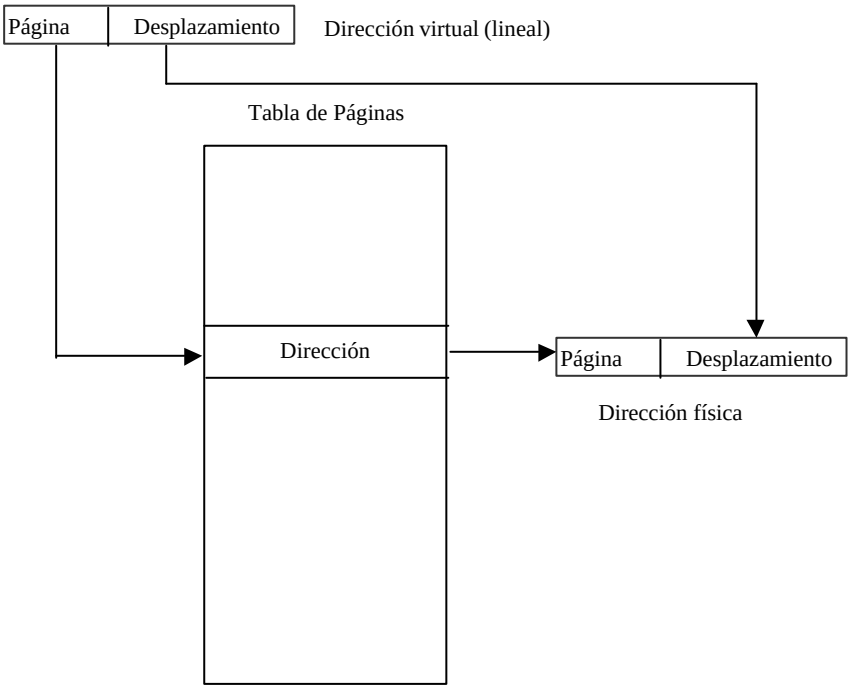

Figura 2.6. Conversión de dirección virtual (lineal) en dirección física.

- Entrada en la tabla de regiones (Figura 2.7).
  - Puntero a una tabla de números de página físicas, denominada tabla de páginas.

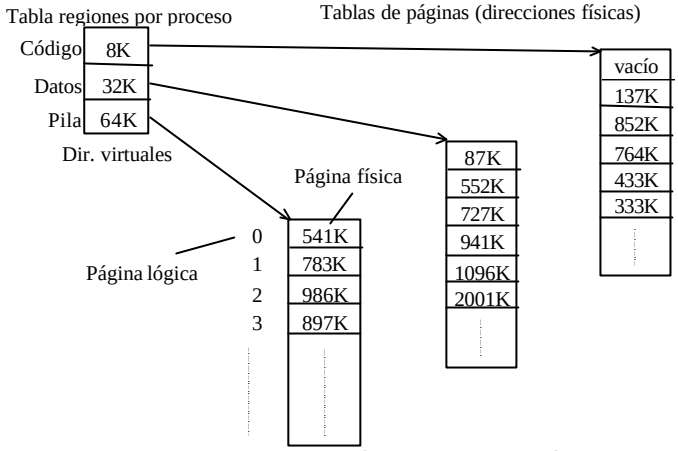

Figura 2.7. Tabla de páginas de la región.

Puesto que una región es un conjunto continuo de direcciones virtuales, el número de página dentro de una región es utilizado como un índice a una *tabla de páginas de la región* en la cual se almacena la dirección de la página física. Para ello, desde la tabla de regiones se apunta a la tabla de páginas de dicha región (recuerde que la tabla de páginas contiene además información del tipo dependiente-del-hardware para implementar algoritmos de reemplazo, bits de Lectura/Escritura, etc).

- Entradas en la tabla de páginas.
  - Información dependiente de la máquina.
  - Bits de permiso para lectura o escritura en la página.
  - Problema de este mecanismo de traducción de direcciones $\Rightarrow$ gran tamaño de la tabla de paginas en memoria.
- Registros hardware y caches se utilizan para incrementar la velocidad de traducción de direcciones virtuales a físicas (Figura 2.6).
- Debido al tamaño del espacio de memoria direccionable por los procesadores, las tablas de páginas raramente se implementa en forma de una sola tabla contigua en memoria. Como la tabla de páginas debe ser residente en memoria, necesitaría un exceso de memoria únicamente para esta tabla. Por ejemplo, los procesadores de la arquitectura x86 pueden direccionar 4 Gigabytes, el tamaño de las páginas de memoria es de 4 Kbytes, y cada entrada de la tabla ocupa 4 bytes; con tales procesadores una tabla de página completa utilizaría 1048576 entradas, ocupando 4 Megabytes de memoria.
  - Solución al problema. Descomponer la tabla de páginas en varios niveles, con un mínimo de dos niveles.
    + Un directorio de tablas de páginas contiene las direcciones de las páginas que contienen partes de dicha tabla.
    + Las partes utilizadas de la tabla de páginas se cargan en memoria.
  - El interés de esta tabla de páginas a dos niveles se basa en que la tabla de páginas no necesita cargarse completamente en memoria. Si un proceso utiliza 6 Megabytes de memoria en un procesador x86, sólo se utilizan tres páginas para la tabla de páginas:
    + La página que contiene el directorio de tablas de páginas.
    + La página que contiene la parte de la tabla de páginas correspondiente a los primeros 4 Megabytes de memoria.
    + La página que contiene la parte de la tabla de páginas correspondiente a los 4 Megabytes de memoria siguientes (de la que sólo se utilizan la mitad de las entradas).

  - **Tabla de páginas gestionadas por Linux** ⇒ 3 niveles. Linux gestiona la memoria central y las tablas de páginas utilizadas para convertir las direcciones virtuales en direcciones físicas. Implementa una gestión de la memoria que es ampliamente independiente del procesador sobre el que se ejecuta. En realidad, la gestión de la memoria implementada por Linux considera que dispone de una tabla de páginas a tres niveles:
    + El directorio global de tablas páginas (page global directory) cuyas entradas contienen las direcciones de páginas que contienen tablas intermedias.
    + El directorio intermedio de tablas páginas (page middle directory) cuyas entradas contienen las direcciones de las páginas que contienen las tablas de páginas.
    + Las tablas de páginas (page table) cuyas entradas contienen las direcciones de páginas de memoria que contienen el código o los datos utilizados por el *kernel* o los procesos de usuario.

    Este modelo no siempre se corresponde al procesador sobre el cual Linux se ejecuta (los procesadores x86, utilizan una tabla de páginas que sólo posee dos niveles). El *kernel* efectúa una correspondencia entre el modelo implementado por el procesador y el modelo de Linux (en los procesadores x86, el *kernel* considera que la tabla intermedia sólo tiene una entrada).
- Segmentación.
  - La memoria virtual es unidimensional ya que las direcciones van desde 0 hasta la dirección máxima, una dirección tras otra.
  - Segmentos ⇒ Espacio de direcciones completamente independientes, cada segmento es una secuencia lineal de direcciones, desde 0 hasta algún máximo. La longitud de cada segmento puede ser cualquiera desde 0 hasta un máximo permitido y puede cambiar de longitud en tiempo de ejecución.
  - La protección tiene sentido en una memoria segmentada, ya que cada segmento contiene sólo una clase de objetos y estos a su vez pueden estar en diferentes páginas.
  - La implementación de la segmentación difiere de la paginación en el sentido de que el tamaño de la página es fijo pero el de los segmentos no.
  - Segmentación con paginación (Pentium de Intel). 16K segmentos independientes cada uno con hasta 1000 millones de palabras de 32 bits. El funcionamiento de la memoria virtual Pentium esta basada en 2 tablas:
    + LDT (tabla de descriptores de segmentos local) ⇒ una para cada proceso y describe los segmentos locales a cada proceso (código, datos y pila).
    + GDT (tabla de descriptores de segmentos global) ⇒ tabla compartida por todos los procesos e incluye todos los segmentos del sistema.
  - Linux utiliza el mecanismo de segmentación sobre Pentium para separar las zonas de memoria asignadas al *kernel* y a los procesos, de este modo el código y los datos se protegen de los accesos erróneos y mal intencionados por parte de procesos en modo usuario.
- El sistema contiene un conjunto de tripletas de registros de gestión de memoria.
  - Primer registro ⇒ dirección de la tabla de páginas en memoria física.
  - Segundo registro ⇒ primera dirección virtual accesible vía la tripleta.
  - Tercer registro ⇒ información de control.
    + Número de páginas de la tabla de páginas.
    + Permisos de acceso a las páginas (sólo-lectura, lectura-escritura).
- Cuando el *kernel* prepara un proceso para su ejecución ⇒ carga el conjunto de tripletas de registros de gestión de memoria con los datos correspondientes almacenados en las entradas en la *pregion*.
- Cuando un proceso intenta acceder a una dirección que está:
  - Fuera de su espacio de direccionamiento.
  - No tiene permiso de acceso.

  Entonces el sistema operativo genera una *excepción*.

2.3.3. Organización de la Memoria del *Kernel*.

Aunque el *kernel* ejecuta en el contexto de un proceso, su espacio virtual es independiente de todos los procesos. El *kernel* permanece en memoria en todo momento y su código y datos se comparte. En el arranque *(boot)* se carga su código y se preparan las tablas y registros para *mapear* su espacio virtual. Estas páginas y el mecanismo para *mapearlas* es el mismo usado para los procesos, pero estas tablas son visibles solo cuando se ejecuta en modo *kernel* (en general el *kernel* y el hardware colaboran para permitir/inhibir este acceso, ya sea cargando determinados registros cuando se ejecuta en modo *kernel* o dividiendo el espacio físico.

- Las tablas de páginas del *kernel* y mecanismos de traducción de direcciones virtuales a físicas son análogas a las de los procesos.
- Espacio de direcciones virtuales de un proceso.
  - Clases sistema.
  - Clases usuario.
  - Cada clase tiene su propia tabla de páginas.
- Colaboración del hardware + sistema operativo.
  - Cuando un proceso se ejecuta en modo *kernel* permite accesos a direcciones *kernel.*
  - Prohibidos cuando se ejecuta en modo usuario.
- Tripletas de registros del *kernel*, para mapear las tablas de páginas de los proceso y del *kernel.*
  - Modo *kernel.*
  - Cambio de modo *kernel* a modo usuario ⇒ el sistema permita o no referencias a direcciones vía las tripletas de registros del *kernel.*
    + Dirección de la Tabla de Páginas (en memoria).
    + Primera dirección virtual *mapeada* (base de la región).
    + Longitud de la Tabla de Páginas (permisos, tipo de página, etc.).

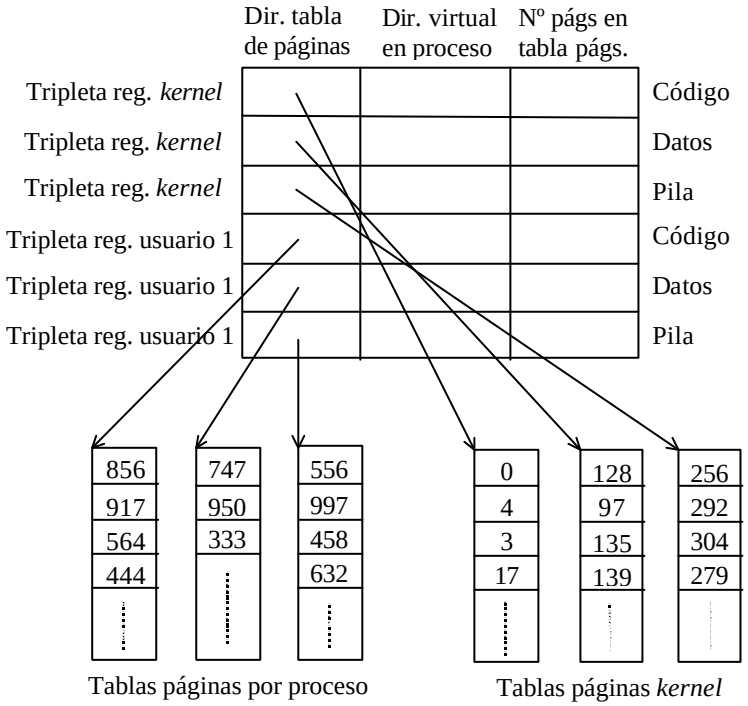

Figura 2.8. Organización de la memoria del *kernel* y de los procesos. Se usan tablas de páginas y un conjunto de registros para *mapear* las direcciones de inicio de estas páginas. Es necesario decir que se dispone de una *pila* por cada proceso (el código y los datos son el mismo, es decir, es compartido por todos los procesos) cuando éste se ejecuta en modo *kernel.*

2.3.4. Mapa de Memoria para el u-Area (Área de Usuario).

- Zona del espacio de direcciones virtuales asociado a cada proceso es accesible sólo en modo *kernel*.
- Cada proceso del sistema ⇒ Área de usuario (u-Area) privada.
- El *kernel* accede al Área de Usuario (u-Area) del proceso en ejecución.
  - Como si sólo estuviese ésta en el sistema.
  - El *kernel* cambia su mapa de traducción de direcciones virtuales de acuerdo al proceso en ejecución.
- En tiempo de compilación ⇒ Se le asigna una dirección virtual fija conocida por el resto del sistema operativo.
- El *kernel* sabe en dónde, dentro de sus tabla de gestión de memoria, se realiza la traducción de la dirección virtual del Área de Usuario (u-Area) ⇒ Cambia dinámicamente la dirección del Área de Usuario (u-Area) a una nueva dirección física.
  - Dos direcciones físicas ⇒ dos Áreas de usuario (u-Areas) de dos procesos.
  - El *kernel* accede a ellas con la misma dirección virtual.

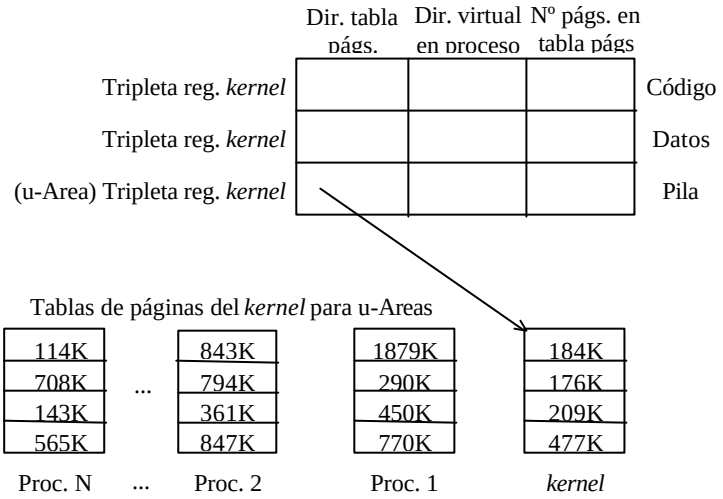

Figura 2.9. Mapa de memoria para el u-Area en el *kernel*.

2.4. CONTEXTO DE UN PROCESO. CONCEPTO Y TIPOS.

- El contexto de un proceso está formado por el contenido: de su espacio de direcciones (user), de los registros hardware del procesador y de las estructuras de datos del *kernel* relativas al proceso. Es decir, el contexto de un proceso es su estado, definido por:
  - Su código.
  - Los valores de sus variables de usuario globales y de sus estructuras de datos.
  - El valor de los registros de la CPU.
  - Los valores almacenados en su entrada de la tabla de procesos y en su área de usuario.
  - Y el contenido de sus pilas (*stacks*) de usuario y *kernel*.
- El código del sistema operativo y sus estructuras de datos globales, son compartidas por todos los procesos, pero no son consideradas como parte del contexto del proceso.

Uso del contexto de un proceso.

- Cuando se ejecuta un proceso ⇒ se dice que el sistema se está ejecutando en el contexto de un proceso.
- Cuando el *kernel* decide que debe ejecutar otro proceso ⇒ *cambio de contexto* ⇒ el *kernel* guarda la información necesaria para poder continuar con la ejecución del proceso interrumpido en el mismo punto que la dejó.

No se considera cambio de contexto.

- No se considera cambio de contexto el cambio en el modo de ejecución de un proceso ⇒ cuando un proceso cambia su modo de ejecución, del modo usuario al modo *kernel*, el *kernel* guarda información para cuando el proceso tenga que volver a modo usuario. Este cambio de modo usuario a modo *kernel* y viceversa no se considera un cambio de contexto.

Desde el punto de vista formal el contexto de un proceso es la unión de:

- Su contexto del nivel de usuario.
- Su contexto del nivel de registros.
- Su contexto del nivel de sistema.

Contexto del nivel de usuario.

- El contexto del nivel de usuario se compone de:
  - Segmento de Texto (código) del proceso.
  - Segmento de Datos del proceso.
  - Segmento de Pila del proceso.
  - Espacio de direcciones virtuales compartidas del proceso (de usuario) que se encuentran en su zona de direcciones virtuales.
  - Las partes del espacio de direcciones virtuales que periódicamente no residen en memoria principal debido al swapping.

Contexto del nivel de registros.

- Contenido de los registros hardware.
  - El contador de programa (PC) ⇒ que contiene la dirección de la siguiente instrucción que debe ejecutar la CPU ⇒ Esta dirección es una dirección virtual tanto en modo *kernel* como en usuario.
  - Registro de estado del procesador (PS), que especifica el estado del hardware de la máquina. El PS normalmente contiene campos para indicar que el resultado de la última operación de cálculo ha sido cero, positiva, negativa, si ha habido overflow, acarreo, etc. Las operaciones que causan la modificación del PS son realizadas en un determinado proceso, por lo tanto el PS contiene el estado del hardware en relación a ese proceso. Otros campos importantes son los que indican el nivel de ejecución actual del procesador (en relación con las interrupciones) y el modo de ejecución (*kernel* o usuario). El campo que indica el modo de ejecución determina si un proceso puede ejecutar instrucciones privilegiadas y si un proceso puede acceder al espacio de direcciones del *kernel*.
  - Puntero de pila (SP), que contiene la dirección de la siguiente entrada en la pila de usuario o de *kernel*. La arquitectura de la máquina dicta si el puntero de pila debe apuntar a la siguiente entrada libre o a la última entrada utilizada. De la misma forma, es la arquitectura de la máquina la que impone si la pila crece hacia direcciones altas o bajas de memoria.
  - Registros de propósito general, que contienen datos generados por el proceso durante la ejecución.

Contexto del nivel de sistema.

- El contexto del nivel de sistema de un proceso tiene una parte estática y otra dinámica. Todo proceso tiene una única parte estática del contexto del nivel de usuario, pero puede tener un número variable de partes dinámicas. La parte dinámica es vista como una pila de capas de contexto que el *kernel* puede apilar y desapilar según los eventos que se produzcan.
- Parte estática.
  - La entrada en la tabla de procesos. Define el estado del proceso y contiene información de control que es siempre accesible al *kernel.*
  - El Área de Usuario (u-Area). Contiene información de control del proceso que necesita ser accedida sólo en el contexto del proceso.
  - Entradas en la *pregion* (tabla de regiones por proceso), tabla de regiones y tablas de páginas, que definen el mapa de transformación entre las direcciones del espacio virtual y las direcciones físicas. Si varios procesos comparten regiones comunes, estas regiones también son consideradas parte del contexto de cada proceso, ya que cada proceso las manipula independientemente. Es trabajo del módulo de gestión de memoria indica qué partes del espacio de direcciones virtuales de un proceso no están cargadas en memoria.
- Parte dinámica.
  - La pila del *kernel*, que contiene marcos de pila (stack frames) de las funciones ejecutadas en modo *kernel.* Si bien todos los procesos ejecutan el mismo código *kernel*, hay una copia privada de la pila del *kernel* para cada uno de ellos que da cuenta de las llamadas que cada proceso hace a las funciones del *kernel.* Por ejemplo, un proceso puede ejecutar la llamada a *creat* y ponerse a dormir hasta que se le asigne un inodo, y otro proceso puede llamar a *read* y dormir hasta que se efectúe la transferencia entre el disco y la memoria. Ambos procesos están ejecutando funciones del *kernel*, pero tienen pilas separadas que contienen la secuencia de llamadas a funciones que ha realizado cada uno. El *kernel* debe ser capaz de recuperar el contenido de la pila *kernel* y la posición del puntero de pila para reanudar la ejecución de un proceso en modo *kernel.* La pila *kernel* está vacía cuando el proceso se está ejecutando en modo usuario.
  - Pila de capas de contexto del nivel de sistema que el *kernel* apila o desapila de acuerdo a una serie de eventos. Cada capa contiene la información necesaria para recuperar la capa anterior, incluyendo el contexto del nivel de registros de la capa anterior.

Apilar (salvar) y Desapilar (restaurar) capas de contexto.

- Apilar (salvar) ⇒ el *kernel* apila una capa de contexto cuando se produce:
  - Una interrupción,
  - Una llamada al sistema, o
  - Un cambio de contexto, restaurándolo.
- Desapilar (restaurar) ⇒ las capas de contexto son desapiladas cuando:
  - El *kernel* vuelve del tratamiento de una interrupción.
  - El proceso vuelve a modo usuario después de ejecutar una llamada al sistema.
  - Se produce un cambio de contexto.
- La capa de contexto apilada es la del último proceso que se estaba ejecutando y la desapilada es la del proceso que se va a pasar a ejecutar.
- Cada una de las estradas de la tabla de procesos contiene información suficiente para poder recuperar la capa contexto actual.
- Figura 2.10 ⇒ Componente que forman parte del contexto de proceso. En el lado izquierdo tenemos la parte estática del contexto y en el derecho la dinámica.

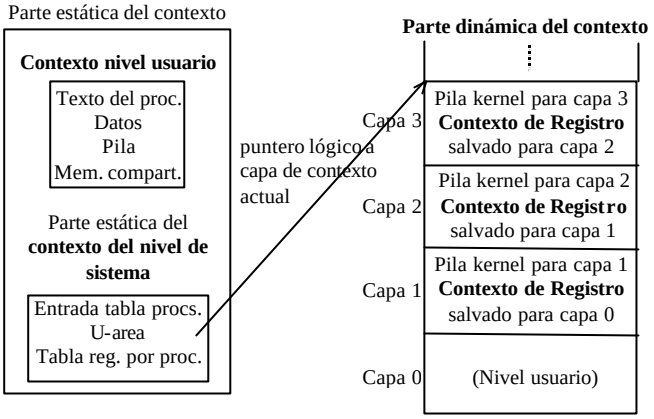

Figura 2.10. Componentes del contexto de un proceso.

Ejecución de un proceso.

- Un proceso se ejecuta dentro de su capa de contexto actual.
- El número de capas de contexto está limitado por el número de niveles de interrupción que soporta la máquina.
  - Por ejemplo, si una máquina soporta 5 niveles de interrupción (interrupciones software, de terminales, de disco, de varios periféricos y de reloj), un proceso podrá tener al menos 7 capas de contexto: Una por cada nivel de interrupción, una para las llamadas al sistema y otra para el nivel de usuario. Estas 7 capas son suficientes para contener todas las capas posibles, incluso si las interrupciones se dan en la secuencia más desfavorable, ya que las interrupciones de un nivel determinado son bloqueadas (la CPU no las atiende) mientras que le *kernel* está atendiendo una interrupción en un nivel igual o superior.
- Ejecución del *kernel* siempre en contexto de proceso, pero la función en ejecución puede no pertenecer a ese proceso.

2.5. MANIPULACIÓN DEL CONTEXTO DE UN PROCESO.

- Salvar el contexto de un proceso ⇒ El *kernel* apila una nueva capa de contexto del nivel de sistema cuando:
  - Recibe una interrupción.
  - El proceso realiza una llamada al sistema.
  - El *kernel* realiza un cambio de contexto.
- Restaurar el contexto de un proceso ⇒ desapila una nueva capa de contexto del nivel de sistema cuando:
  - Vuelve de una interrupción.
  - Vuelve a modo usuario después de una llamada al sistema.
  - Cambio de contexto.

2.5.1. Interrupciones y Excepciones. Tratamiento.

El kernel debe manejar las interrupciones ya sea que provengan del hardware (clock, E/S), software (instrucciones programadas para producir una *software-interrupt*) o como excepciones (como fallos de página). Si el procesador esta en un nivel de ejecución menor que el nivel de la interrupción, ésta se acepta antes de decodificar la siguiente instrucción, se incrementa *(raises)* el nivel de ejecución para no procesar otras interrupciones de ese nivel o inferior mientras atiende la actual de tal forma que preserva la coherencia de sus estructuras de datos.

- Las interrupciones son señales eléctricas manejadas directamente por un controlador o manejador de interrupciones (*hardware*) o codificadas a nivel de software.
- Además de las interrupciones *hardware* y *software* (señales), diversas condiciones de error de la CPU pueden causar la iniciación de una excepción. Las excepciones pueden servir: para estimular al sistema operativo para que suministre un servicio, para suministrar más memoria a un proceso, ...

algoritmo *int_hand*
entrada: ninguna
salida: ninguna
{
salvar (apilar) la capa de contexto actual y apilar una nueva capa de contexto;
// cuando se recibe una interrupción en general se consigue un valor que se usa como un offset en una
// tabla de interrupciones que apunta a la función que la manejará.
determinar la fuente de interrupción;
buscar en vector de interrupción la dirección del procedimiento del servicio de esa interrupción;
invocar el manejador de la interrupción por parte del *kernel*;
// en este momento la pila del kernel es diferente de la pila en el contexto anterior. Algunas
// implementaciones usan la pila del kernel del proceso que estaba ejecutando y otras usan un stack
// global que garantiza un retorno del manejador sin cambio de contexto.
el manejador de interrupción finaliza su trabajo;
restaurar (desapilar) la capa de contexto anterior (en el punto en el que se produjo la interrupción);
}

- Por ejemplo, supongamos el caso de un proceso que ejecuta una llamada al sistema y recibe una interrupción de disco mientras se encuentra en dicha llamada al sistema. Cuando está ejecutando el código del manejador de la interrupción de disco, el sistema recibe una interrupción de reloj y ejecuta el manejador de la interrupción de reloj. Cada vez que el sistema recibe una interrupción o realiza una llamada al sistema se crea una nueva capa y salva los registros que definen contexto del nivel salvado.

- Si el nivel de ejecución de la CPU < nivel de una interrupción recibida.
  - Acepta la interrupción antes de decodificar la siguiente instrucción.
  - Elevar el nivel de ejecución del procesador ⇒ evitar que interrupciones del mismo o más bajo nivel sean atendidas. Preservando la integridad de las estructuras de datos del *kernel.*
- El **vector de interrupciones** contiene la dirección del procedimiento de servicios de interrupciones y está situado en una posición cerca de la memoria. Supongamos que se está ejecutando un proceso y ocurre una interrupción de disco. El hardware de interrupciones mete el contador de programa, la palabra de estado del programa y los valores de los registros en la pila (actual), a continuación la computadora salta a la dirección especificada en el vector de interrupciones. Esto es todo lo que hardware hace y a partir de ahora es el software el que decide.
- **Gestión de interrupción** ⇒ Realización de una serie de acciones.
  - Obtiene el número de la interrupción ⇒ Desplazamiento en el vector de interrupciones ⇒ Dirección del manejador de interrupciones para cada interrupción.
- **Manejo de interrupción** (algoritmo de manejo de interrupciones *(int_hand)*).
  - Salva contexto de registro actual del proceso en ejecución y apila una nueva capa de contexto ⇒ Puntero a zona donde haya salvado los registros.
  - Determina la causa de la interrupción, identificando tipo y dispositivo concreto ⇒ manejador de interrupción.
  - *Kernel* invoca el manejador de interrupciones ⇒ Nueva capa en la pila de *kernel* para la nueva capa de contexto.
  - Manejador de interrupciones finaliza su trabajo y retorna.
    - El *kernel* restaura contexto de registro y la pila de *kernel* de la capa de contexto anterior ⇒ Como eran antes de la interrupción.
    - El *kernel* reanuda la ejecución del contexto restaurado.

2.5.2. Interfaz de Llamadas al Sistema (Algoritmo *sys_call*).

- Un proceso que se ejecuta en modo usuario no puede acceder directamente a los recursos de la máquina, para ello debe ejecutar llamadas al sistema. Una llamada al sistema es una petición transmitida por el proceso al *kernel.* El *kernel* trata la petición en modo *kernel* con todos los privilegios y envía el resultado al proceso que la llamó, prosiguiendo su ejecución.
- Una llamada al sistema provoca un cambio:
  - El proceso ejecuta una instrucción del procesador que le hace pasar a modo *kernel.*
  - A continuación, el proceso en modo *kernel* ejecuta la función de tratamiento vinculada a la llamada al sistema que ha efectuado.
  - El proceso vuelve a modo usuario para proseguir su ejecución.
- Diferencia entre llamadas al sistema / llamadas a funciones normales ⇒ llamadas a funciones normales no se cambia el modo de ejecución (de modo usuario a modo *kernel*) ⇒ Uso pila de *kernel.*
- Compilador C usa una librería de funciones ⇒ nombres de llamadas al sistema. Estas funciones afectan al nombre de las llamadas al sistema. Para ejecutar una llamada al sistema, basta con llamar a la función correspondiente.
- Las funciones de librería deben aislar al usuario de las llamadas al sistema y hacer que la vea como una llamada a una función o procedimiento normal ⇒ Operaciones adicionales.
  - Preparación de la llamada al sistema (modo usuario).
  - Cambio a modo *kernel* e invocación de la llamada al sistema.
  - Recuperación y vuelva a modo usuario.
- Una llamada al sistema se caracteriza por un nombre y un número que la identifica. Preparar la llamada al sistema ⇒ Informar del número de la llamada al sistema al *kernel.*
- Invocar la instrucción que da lugar a un cambio de modo de ejecución, de modo usuario a modo *kernel* (*trap* del S.O.) y causa que el *kernel* ejecute código para resolver la llamada al sistema.

- **Inicio de la ejecución del código de llamadas al sistema.**
  - Funciones de librería pasan al *kernel* un número único para cada llamada al sistema ⇒ el *kernel* determina la llamada al sistema según el número, y los parámetros de las llamadas al sistema se colocan en ciertos registros del procesador.
  - Interfaz de llamadas al sistema ⇒ caso similar al manejador de interrupciones. Se provoca un bloqueo desencadenando una interrupción.
  - Manejo del *trap* (sentencia software) ⇒ el *kernel* realizará las operaciones de manejo de interrupción.
    + Salvar el contexto del proceso ⇒ paso del proceso a modo *kernel.*
    + Busca el número de llamada al sistema en la tabla de llamadas al sistema ⇒ rutina en el *kernel* (algoritmo para las llamadas al sistema *(sys-call)*).
    + Calcula la dirección de usuario del primer parámetro para la llamada al sistema.
    + Copia los parámetros de usuario al Área de Usuario (u-Area).
    + Invoca la función correspondiente a la llamada al sistema.
    + Después de ejecutarse el código de la llamada al sistema, el *kernel* determina si hubo algún error y retorna. Este retorno vuelve a pasar al proceso a modo usuario.
- **Recuperación de la llamada.**
  - Cuando el *kernel* retorna del *trap* del sistema operativo a modo usuario, retorna a la instrucción de librería posterior al *trap*.
  - La librería interpreta los valores devueltos por el *kernel.*
  - Devuelve el valor al programa de usuario que no sabe si lo que sufrió fue una llamada al sistema o a una función de usuario.

```
algoritmo sys_call(num)        // entrada: número de la llamada al sistema. salida: resultado
{
    colocar los parámetros de la llamada al sistema en ciertos registros del procesador;
    buscar la entrada en la tabla de llamadas al sistema del número de la llamada al sistema;
    determinar el número de parámetros de la llamada al sistema;
    copiar los parámetros del espacio del proceso de usuario al Área de Usuario (u-Area);
    salvar el contexto actual para retorno forzoso (abortive return);
    invocar el código en el kernel de la llamada al sistema;
    if (se produjo un error durante la llamada al sistema)
    {
        guardar el número de error en el contexto de registro del usuario;
        activar el bit de acarreo en el registro PS (registro de estado del procesador) del contexto de
            registro del proceso salvado;
    }
    else
        actualizar los registros en el contexto de registro del proceso salvado para retornar valores de
            la llamada al sistema;
}
```

2.5.3. Cambio de Contexto de un Proceso.

- El *kernel* permite un cambio de contexto bajo determinadas circunstancias, como pueden ser que empiece una operación de entrada/salida o cuando expira el quantum asociado a un proceso.
- El *kernel* evita hacer un cambio de contexto en cualquier situación debido a que debe mantener la coherencia de sus estructuras (las listas están bien enlazadas, los bloqueos han sido activados, etc). Claramente se hace un cambio de contexto como conclusión de la llamada *exit* (no queda más por hacer) y cuando el proceso se bloquea (hace que duerma) voluntariamente a la espera de un recurso *(sleep)*. Las otras dos situaciones se refieren a la aparición de un proceso de mayor prioridad (activado por una interrupción).
- Procedimiento de cambio de contexto.
  - Similar al manejo de interrupciones y de las llamadas al sistema, excepto que el *kernel* restaura la capa de contexto de nivel de sistema de un proceso distinto.

Pasos en el cambio de contexto de un proceso:
1. Decidir si realizar un cambio de contexto y si éste es permisible ahora.
2. Salvar el contexto del proceso antiguo.
3. Buscar el mejor proceso para planificar *(scheduler)* su ejecución.
4. Restaurar su contexto.

- *Abortive Return*. Finalización anormal de la secuencia de ejecución del *kernel* ⇒ Reejecutar desde el contexto previo *save_context* {crea una nueva capa de contexto}, *setjmp* {almacena la capa de contexto en la u-Area}, *longjmp* {restaura la capa de contexto desde la u-Area}.
- Copia de datos en el *kernel* ⇔ Usuario. Los modos *kernel* y usuario son excluyentes, pero necesitan intercambiar datos (Usuario ⇒ *kernel*: parámetros, *kernel* ⇒ Usuario: resultados).

2.6. MANIPULACIÓN DEL ESPACIO DE DIRECCIONES DE UN PROCESO. ALGORITMOS.

- Hemos visto como el *kernel* realiza cambios de contexto metiendo y sacando contextos de la pila, pero viendo el contexto de usuario como un objeto estático que no cambia durante los cambios de contexto. Sin embargo, hay varias llamadas al sistema que manipulan el espacio virtual de un proceso. Para revisar estas llamadas al sistema necesitamos definir las estructuras que describen las regiones y algunos algoritmos básicos que serán utilizados por las llamadas al sistema.
- El espacio de direcciones de un proceso ⇒ Representación lógica como regiones. Este espacio de direccionamiento incluye:
  - El código del proceso.
  - Los datos del proceso, que se descompone en dos segmentos, por una parte *data*, que contiene las variables inicializadas, y por otra parte *bss* que contiene las variables no inicializadas.
  - La pila utilizada por el proceso.
- Cada región de memoria se caracteriza por varios atributos:
  - Sus direcciones de inicio y fin.
  - Los derechos de acceso que tiene asociada.
  - El objeto asociado (por ejemplo, un archivo ejecutable que contiene el código ejecutable por el proceso).
- Tabla de regiones.
  - Información necesaria para describir una región (descriptor de regiones). Los campos más importantes son los siguientes:
    + Puntero al inodo del archivo cuyo contenido está cargado inicialmente en la región.
    + Tipo de la región (texto (código), memoria compartida, datos privados o pila).
    + Tamaño de la región.
    + Localización de la región en memoria física.
    + Estado de la región ⇒ combinación de:
      * Bloqueada.
      * En demanda.
      * En proceso a ser cargada en memoria (cargándose).
      * Válida (cargada en memoria).
    + Cuenta de referencia ⇒ número de procesos que la referencian (región compartida).
  - Organización de los descriptores de regiones organizados en Linux. Se mantiene una lista de descriptores organizados como un árbol AVL (árbol equilibrado en memoria), lo que permite reducir la complejidad de la búsqueda de O(n) a O(log n). Destacar que a partir de la versión 2.4.9 del kernel de Linux se utilizan *árboles roji-negros* para acelerar las búsquedas de un descriptor de región particular.
- Operaciones que manipulan regiones:
  - Crear / suprimir regiones de memoria.
  - Bloquear / desbloquear una región para prevenit accesos mientras se manipula una región.
  - Asignar / liberar una región. Asignar ⇒ mover de la lista de regiones libres a la lista de regiones activas, iniciándola. Liberar ⇒ liberar la región y los recursos asociados a ella.

- Reasignar regiones de memoria.
- Vincular / desvincular (asociar / desligar) una región al espacio de memoria de un proceso. Asociar ⇒ fijar el espacio de direcciones del proceso, completando la *pregion*, formando una nueva tripleta y la tabla de páginas.
- Cambiar el tamaño de una región.
- Duplicar el contenido de una región.
- Modificar las protecciones asociadas a regiones de memoria.
- Cargar una región con un ejecutable.

Algunos de los algoritmos que realizar las operaciones sobre regiones son los siguientes:

- **Asignar una Región**. El *kernel* asigna una región *(allocreg)* en las llamadas al sistema *fork*, *exec* y *shmget (shared memory)*. Con pocas excepciones cada proceso está asociado con un archivo ejecutable como resultado de una *exec*, entonces la región se asocia al *i-nodo* (con ello se identificará una región que se desee compartir y ya se encuentre cargada). El contador del *i-nodo* se incrementa para evitar que por ejemplo el ejecutable sea borrado mientras va a ser utilizado.

algoritmo *allocreg(i-nodo, tipo)*
{
    Retirar primera región de libres;
    Asignar tipo de región e i-nodo;
    if (i-nodo)
        Incrementar contador de referencias en inodo;
    Insertar la región en lista de activas;
    Return región bloqueada;
}

- **Asociar una Región a un Proceso**. En las llamadas al sistema *fork*, *exec* y *shmget* el *kernel* asocia el espacio de direcciones de un proceso con una región vía este algoritmo *(attachreg)*. La región puede ser nueva o ya existir *(shared)*. La región a ser asociada al proceso lo será en una dirección virtual para el proceso, por lo que será necesario controlar la legalidad (tamaño final del espacio de direcciones del proceso).

algoritmo *attachreg(region, proceso, dir_Virtual, tipo)*
{
    Asignar una tabla de *pregion* para el proceso;
    Inicializar tabla de *pregion*;
        Puntero a la región;
        Tipo de región;
        Dirección virtual de la región para el proceso;
    Controlar legalidad de dir_Virtual, tamaño de región;
    Incrementar contador de referencias a la región;
    Incrementar tamaño del proceso de acuerdo a la región asociada;
    Inicializar registros hardware para el proceso (tripleta);
    Return (tabla *pregion*);
}

- **Cambio de tamaño de una región**. Un proceso puede expandir o contraer su espacio virtual con la llamada al sistema *sbrk*, aunque la pila del proceso lo hace automáticamente de acuerdo a la profundidad en que se pueden anidar llamadas al sistema. Internamente la función *sbrk* llama al algoritmo *growreg* para cambiar el tamaño. Cuando una región se expande el *kernel* debe asegurar que no se solapen regiones o que no se superen los límites impuestos a los procesos. Las regiones compartidas no pueden ser expandidas sin causar efectos laterales en los otros procesos, por esta razón *kernel* usa *growreg* en las regiones privadas (datos y automáticamente en la pila). Se asignan tablas de páginas (dependiente del modelo usado, programa entero, demanda de páginas, etc). Debe asegurarse que existe memoria antes de hacer la llamada.

```
algoritmo growreg(pregión, incremento)
{
    if (incremento es positivo)
    {
        Controlar legalidad del nuevo tamaño;
        Asignar tablas de paginas;
        if (modelo de programa entero)
        {
            Asignar memoria física;
            Inicialización de tablas necesarias;
        }
    }
    else
    {
        Liberar memoria física;
        Liberar tablas;
    }
    Otras inicializaciones;
    Cambiar tamaño de proceso en la Tabla de Procesos;
}
```

- **Cargar una región**. Si el sistema soporta paginación por demanda, el sistema puede *mapear* el espacio de direcciones del ejecutable durante la *exec* de tal forma que sólo cuando sea necesario leerá las páginas a memoria. Si el modelo es de programa entero se debe copiar el ejecutable cargando las regiones en las direcciones virtuales que se especifique en el archivo (posibles páginas vacías). Para cargar el archivo en la región debe expandirla de acuerdo a la memoria que necesite. Marca la región en el estado *Cargándose* y usa una variación de la llamada al sistema *read* para leer el ejecutable (porque puede ganarse velocidad, y porque en un modelo de paginación por demanda se podría producir un fallo de página durante una *read*). El *offset* que se indica como parámetro indica el desplazamiento dentro del archivo del inicio de la región y *count* el número de datos a leer.

```
algoritmo loadreg(pregión, dir_Virtual, i-node, Offset , count)
{
    Incrementar tamaño (growreg);
    Marcar estado Cargándose;
    Desbloquear región (unlock);
    Poner parámetros de lectura en u-Area;
    Leer ejecutable en la región (variación de read);
    Bloquear región;
    Marcar estado Válida;
    Despertar procesos que esperan por región cargándose;
}
```

- **Liberar una Región**.

algoritmo *freereg(region-bloqueada)*
{
    if (algún proceso está usando la región)
    {
        Retirar el bloqueo de la región;
        if (región tiene un i-nodo asociado)
            Retirar el bloqueo del i-nodo;
        Return;
    }
    if (región tiene un i-nodo asociado)
        Liberar i-nodo *(iput)*;
    Liberar memoria física;
    Liberar tablas de páginas;
    Limpiar campos de la región;
    Pasar la región a lista de libres;
    Retirar el bloqueo de la región;
}

- **Desligar (desasociar) región y proceso**. Esta acción se realiza en las llamadas al sistema *exec*, *exit* y *shmdt*. Un contador de referencias a la región que sea diferente de cero indica que hay más procesos utilizando la región y por tanto no se debe liberar la región. El uso del *sticky-bit* (bit pegajoso), que se verá mas adelante, aunque podemos adelantar algo de su uso. Para algunos procesos de uso muy frecuente resulta de interés mantener su texto en memoria aún cuando no hay ningún proceso que haga uso de él (es decir, un compilador, editor, etc que son frecuentemente utilizados). La llamada al sistema *chmod* puede permitir activar un *sticky-bit* a un archivo de tal forma que cuando se encuentra en memoria sus páginas no son liberadas aún cuando no queden procesos referenciándolas (esto permitirá que cuando se lance un nuevo proceso que utilice este texto lo encontrará ya en memoria asignado a una región).

algoritmo *deattachreg(pregion)*
{
    Invalidar las tablas auxiliares asociadas al proceso (pregion, tripletas, etc.);
    Disminuir el tamaño del proceso;
    Disminuir el contador de referencias de la región;
    if (contador llega a 0 y la región no tiene *sticky bit*)
        Liberar la región; *// freereg*
    else *// el contador no es 0 o tiene sticky-bit*
    {
        Retirar el bloqueo del i-nodo, si hay i-nodo asociado a la región;
        Retirar el bloqueo a la región;
    }
}

- **Duplicar una región**. Este algoritmo es usado por *fork*. Si la región es compartida no es necesario hacer una nueva copia de la región, bastará incrementar el contador de referencias y padre/hijo compartirán la región. Si no es compartida se debe realizar físicamente la copia para lo cual se necesita una nueva entrada en la tabla de regiones, tablas de páginas y memoria física para las páginas.

```
algoritmo dupreg(Old_region, New_region)
{
    if (Old_region es compartida)
    {
        El proceso que llama incrementará el contador de referencias, que referencian esa región;
        Return New_region = Old_region;
    }
    Asignar nueva región (allocreg);
    Preparar tablas iguales a las de Old_region;
    Asignar memoria física para el contenido de New_region;
    Copiar el contenido de Old_region en New_region;
    Return puntero a New_region;
}
```

2.7. LLAMADAS AL SISTEMA PARA GESTIÓN DE PROCESOS.

2.7.1. Creación de un Proceso (fork). Duplicación del Proceso Actual.

- La única forma de crear un nuevo proceso en UNIX ⇒ llamada al sistema *fork*.
  - Proceso padre ⇒ Proceso que llama a *fork*.
  - Proceso hijo ⇒ Proceso creado.
- Fin de la llamada a *fork*, procesos padre e hijo ⇒ La llamada a *fork* hace que el proceso actual se duplique.
  - Igual contexto a nivel usuario ⇒ A la salida de *fork*, los dos procesos tienen una copia idéntica del contexto de nivel de usuario excepto para el valor del PID.
  - Diferente valor de retorno del *fork*.
    - Proceso padre, el PID del proceso creado (proceso hijo).
    - Proceso hijo, el valor 0 para el PID.
- El proceso 0, creado por el *kernel* cuando se arranca el sistema (*swapper*, encargado de la gestión de memoria virtual), es el único que no se crea vía *fork*.
- Si la llamada a *fork* falla, devolverá el valor –1 y en *errno* estará el código del error producido.
- Al volver de *fork*, dos procesos, el proceso padre y el proceso hijo, comparten el mismo código.

2.7.1.1. Acciones más Importantes del *Kernel* al Llamar a *fork*.

- Busca una entrada libre en la tabla de procesos y la reserva para el proceso hijo.
- Asigna un identificador de proceso (PID) para el proceso hijo. Este número es único e invariable durante toda la vida del proceso y es la clave para poder controlarlo desde otros procesos.
- Realiza un copia del contexto del nivel de usuario del proceso padre para el proceso hijo. Las secciones que deban ser compartidas, como el código o las zona de memoria compartida, no se copian, sino que se incrementan los contadores que indican cuántos procesos comparten esas zonas.
- Las tablas de control de archivos locales al proceso, como pueden ser la tabla de descriptores de archivo, también se copian del proceso padre al proceso hijo, ya que forman parte del contexto de nivel de usuario. En las tablas globales del *kernel*, tabla de archivos y tabla de inodos, se incrementan los contadores que indican cuántos procesos tienen abiertos esos archivos.
- Retorna al proceso padre el PID del proceso hijo, y al proceso hijo le devuelve el valor 0 para su PID.

```
int pid;
if ((pid = fork()) == -1)
        perror("Error en la llamada a fork.");
else if (pid == 0)
        // Código que va a ejecutar el proceso hijo.
else
        // Código que va a ejecutar el proceso padre.
```

2.7.1.2. Algoritmo Detallado para la Llamada al Sistema *fork*.

- ¿Hay disponibles recursos suficientes para poder completar la llamada a *fork* con éxito? ⇒ Estado "Creado" (8) ⇒ en el diagrama completo de transición de estados de un proceso UNIX.
  - Recurso 1: Sistema swapping ⇒ hay espacio en memoria o disco para el proceso hijo.
    + Si hay memoria principal suficiente ⇒ Proceso en estado "Listo para ejecutarse en Memoria Principal" (3).
    + Si no hay memoria principal suficiente ⇒ Swapper ⇒ Proceso en estado "Listo para ejecutarse en el área de swap en memoria secundaria" (5).
  - Recurso 2: El sistema demanda páginas (Paginación) ⇒ también espacio para tablas auxiliares.
- **NO**
  - ERROR.
- **SI**
  - Construcción del contexto del nivel de usuario (segmentos de código, datos y pila, zonas de memoria compartida que se encuentran en la zona de direcciones virtuales del proceso y las partes del espacio de direcciones virtuales que periódicamente no residen en memoria principal debido al swapping o a la paginación) y la parte estática del contexto del nivel de sistema (entrada en la tabla de procesos, el área de usuario y las entradas de la tabla de regiones por proceso (*pregion*), tabla de regiones y tabla de páginas) del proceso hijo.
    + Comprobaciones de la existencia de los recursos necesarios en el proceso padre para que pueden ser copiados en el proceso hijo y actualizados a nivel global.
    + Buscar una entrada libre en la tabla de procesos para el proceso hijo ⇒ Buscar una entrada libre en la tabla de procesos y la reserva para el proceso hijo.
    + Número de ID (PID) único para el proceso nuevo. Determinar un identificador de proceso para el proceso hijo. Este número es único e invariable durante toda la vida del proceso y es la clave para poder controlarlo desde otros procesos.
    + Proceso hijo en estado "Creado", aunque necesita completar algunos parámetros para pasar al estado de "Listo para su ejecución en memoria principal o en el área de swap".
    + Inicializar la entrada del proceso hijo en la tabla de procesos ⇒ copiar la entrada en la tabla de procesos del proceso padre.
    + Asignar el ID (PID) del proceso hijo a la entrada de la tabla de procesos del proceso padre y al proceso hijo el PID 0.
    + Inicializar parámetros para la planificación (scheduler).
    + Incrementar la cuenta de referencias (contadores) al inodo del directorio actual y la raíz del proceso padre ⇒ En las tablas globales del *kernel*, tabla de archivos y la tabla de inodos, se incrementan los contadores que indican cuántos procesos tienen abiertos esos archivos.
    + El proceso hijo hereda la raíz del proceso padre.
    + Asociar al proceso hijo los archivos asociados al proceso padre ⇒ Copiar del proceso padre al proceso hijo la tabla de control de archivos locales al proceso, como puede ser la tabla de descriptores de archivos.
    + Reservar memoria para u-Area, regiones y tablas de páginas para el proceso hijo.
    + Duplicar cada región del proceso padre asignándosela al proceso hijo.
    + Sistema swapping ⇒ Duplicar regiones de memoria no compartidas en el área de swap.
    + Las secciones que deben ser compartidas, como el código o las zonas de memoria compartida, no se copian, sino que se incrementan los contadores asociados a ellas que indican cuántos procesos comparten esas zonas.
    + Las áreas de usuario (u-Areas) son iguales excepto en el puntero de entrada a la tabla de procesos.

- Construcción de la parte dinámica del contexto del proceso hijo a nivel de sistema (la pila *kernel* que contiene marcos de pila de las funciones ejecutadas en modo *kernel* por el proceso y una serie de capas que se almacenan en forma de pila, que contiene la información necesaria para poder recuperar la capa anterior incluyendo el contexto de nivel de registros de la capa anterior).
  + El *kernel* copia o apila la capa (1) del proceso *padre*.
    * Contexto de nivel de registros de usuario salvados.
    * Capa de la pila *kernel* para la llamada al sistema, *fork*.
  + El *kernel* crea y apila una capa de contexto falsa (2) para el proceso *hijo*.
    * Contexto de nivel de registros salvado para la capa (1).
    * Datos necesarios para permitir al proceso hijo reconocerse y empezar la ejecución desde este momento cuando sea planificado por el *kernel* a través del scheduler (contador del programa, PC (registro de estado del procesador que especifica el estado del hardware), puntero de la pila, otros registros, etc.).
- **if** (proceso padre)
  + El *kernel* cambia el estado del proceso hijo de "Creado" a "Listo para su ejecución en memoria principal" o "Listo para su ejecución en el área de swap".
  + El *kernel* retorna el PID del proceso creado (hijo) al proceso padre, y al proceso hijo le devuelve el valor 0.

  else
  + Inicializa los campos de tiempo en el u-Area (área de usuario).
  + retorna 0.

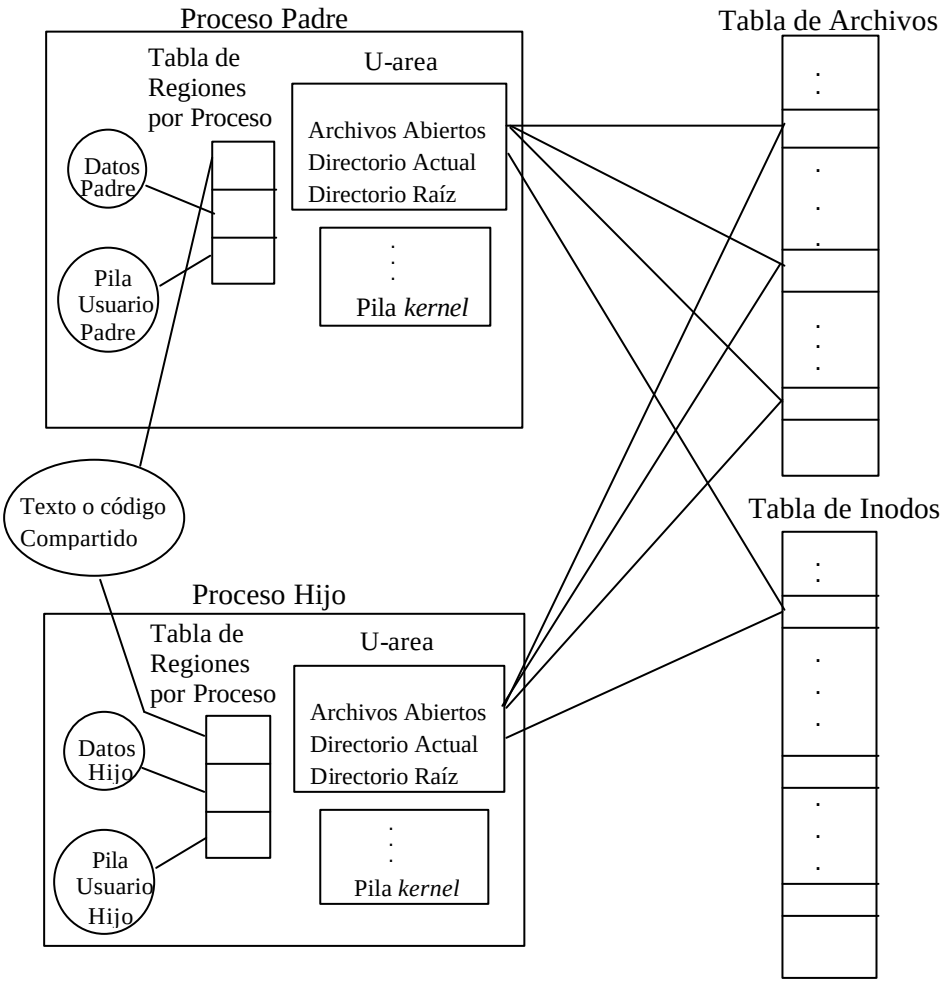

Figura 2.11. Procesos padre e hijo después de un *fork*. Creación de un nuevo contexto mediante *fork*.

2.7.1.3. Observaciones sobre la Creación de un Proceso (Padre e Hijo).

- Las u-Areas son iguales a excepción del puntero a la Tabla de Procesos. Una vez creado el contexto estático va por el dinámico. El *kernel* copia la primera capa (capa 1) que contiene el nivel de usuario y también la pila del *kernel* (con la capa de llamada a fork). Si la implementación incluye pila del *kernel* en la uArea ya en el punto anterior se hace esta copia (en cualquier caso la pila del padre e hijo son iguales). Ahora hace una copia falsa en la pila del hijo para simular que se ha ejecutado antes. En esta capa se debe incluir el contexto de registros de la capa 1.
- Observar que padre e hijo *comparten* los archivos abiertos antes del fork (los que se abran después ya van a cuenta de cada proceso), por lo que en el siguiente código ejemplo los dos procesos jamás leen o escriben en el mismo *offset* de los archivos descritos por *fr* y *fw*. En este caso ambos procesos compartirán las tablas de archivos cada uno con los valores para *fr* y *fw* apuntando a la misma entrada en la Tabla de Archivos. Aunque de la apariencia que se copia el archivo por dos veces, lo cierto es que el contenido del archivo destino dependerá de como se planifiquen los procesos.

2.7.1.4. Clonado en Linux (clone).

- Diferentes libros sobre sistemas operativos definen un *proceso* de diferentes formas, empezando por "instancia de un programa en ejecución" y finalizando con "lo que es producido por las llamadas del sistema fork o clone". Bajo Linux, hay tres clases de procesos:
  - el/los thread(s) vacío(s),
  - threads del *kernel*,
  - tareas de usuario.
- El thread vacío es creado en tiempo de compilación para la primera CPU; es entonces creado "manualmente" para cada CPU por medio de la función específica de la arquitectura fork_by_hand() en *arch/i386/kernel/smpboot.c*, el cual desenrolla la llamada al sistema fork a mano (en algunas arquitecturas). Las tareas vacías comparten una estructura init_task pero tienen una estructura privada TSS, en la matriz de cada CPU init_tss. Todas las tareas vacías tienen pid = 0 y ninguna otra tarea puede compartir el pid, esto es, usar el flag CLONE_PID en *clone*.
- Los threads del *kernel* son creados usando la función kernel_thread() la cual invoca a la llamada al sistema *clone* en modo *kernel*. Los threads del *kernel* usualmente no tienen espacio de direcciones de usuario, esto es p->mm = NULL, porque ellos explícitamente hacen exit_mm(), ej. a través de la función daemonize(). Los threads del *kernel* siempre pueden acceder al espacio de direcciones del *kernel* directamente. Ellos son asignados a números pid en el rango bajo. Funcionando en el anillo del procesador 0 (en x86) implica que los threads del núcleo disfrutan de todos los privilegios de E/S y no pueden ser pre-desocupados por el scheduler o planificador.
- Las tareas de usuario son creadas por medio de las llamadas al sistema clone o fork, las cuales internamente invocan a *kernel/fork.c:do_fork()*.
- Por tanto Linux, permite crear clones de procesos. Un proceso clon se crea por la llamada la sistema clone por duplicación de su proceso padre. Pero contrariamente a un proceso hijo clásico, puede compartir una parte de su contexto con su padre. Según las opciones especificadas en la llamada al sistema clone, se pueden compartir una o más partes de su contexto:
  - El espacio de direccionamiento. Los dos procesos comparten los mismos segmentos de código y de datos. Toda modificación efectuada por uno es visible por parte del otro.
  - Las informaciones de control del sistema de archivos. Los dos procesos comparten los mismos directorios raíz y actual. Si uno de estos directorios es modificado por uno de los procesos (por la primitivas chdir y chroot), la modificación es efectiva para el otro.
  - Los descriptores de archivos abiertos. Los dos procesos comparten los mismos descriptores de archivos abiertos. Si uno de ellos cierra un archivo, el otro ya no puede acceder a él.
  - Los gestores de señales: los dos procesos comparten la tabla de funciones llamada en la recepción de una señal. Toda modificación por parte de uno de los procesos (sigaction), provoca el cambio de la rutina de tratamiento de la señal por el otro proceso.
  - El identificador de proceso. Los dos procesos pueden compartir el mismo número de proceso.

- En el caso extremo en que los procesos compartan el máximo de cosas, se diferencian únicamente por el valor de los registros del procesador y por su segmento de pila. Esta posibilidad de clonado permite, entre otras cosas, implementar servidores en los que se ejecuten varias actividades (threads). Estas actividades pueden compartir simplemente datos, sin emplear mecanismos de comunicación interprocesos (IPC).
- La llamada al sistema clone crea un proceso actual. Antes de llamar a clone, debemos asignar un segmento de pila (llamando a la función malloc) y los parámetros de la función a ejecutar deben apilarse en esta zona de memoria. A continuación, los registros del procesador deben inicializarse de la siguiente forma (en un procesador x86): (1) eax debe contener el código de la llamada clone, representado por la constante _NR_clone; (2) ebx debe contener una combinación de constantes (CLONE_VM | CLONE_FS | CLONE_FILE | CLONE_SIGHAND | SIGCHLD); y (3) ecx debe contener el puntero de pila para el proceso hijo.

2.7.2. Terminación y Espera por la Terminación de Procesos. EXIT y WAIT.

Una situación muy típica en programación UNIX es que cuando un proceso crea a otro, el proceso padre se queda esperando a que termine el proceso hijo antes de continuar su ejecución. Un ejemplo de esta situación es la forma de operar de los intérpretes de órdenes. Cuando escribimos una orden, la *shell* arranca un proceso para ejecutar la orden y no devuelve el control hasta que no se ha ejecutado completamente. Naturalmente, esto no es aplicable cuando la orden se ejecuta en *background*. Para poder sincronizar los procesos padre e hijo se utilizan las llamadas *exit* y *wait*.

2.7.2.1. Terminación de Procesos, *exit*.

- Llamada al sistema *exit*(estado) implícita o explícitamente termina la ejecución de un proceso y devuelve el valor de estado (0..255) que hay que comunicar al proceso padre $\Rightarrow$ Hay que tener cuidado si el proceso es un proceso líder asociado con el terminal.
- Un retorno *(return)* efectuado desde la función principal de un programa C *(main)* tiene el mismo efecto que una llamada a *exit*. Si efectuamos el *return* sin devolver ningún valor en concreto, el resultado devuelto al sistema estará indefinido.
- La función *exit* es uno de los pocos ejemplos de llamada al sistema que no devuelve ningún valor, esto es en cierta medida lógico, ya que el proceso que la llama deja de existir después de haberla ejecutado.
- Por convenio, un proceso debe devolver el valor 0 en caso de finalización normal, y un valor no nulo en caso de finalización debida a un error.
- Antes de terminar la ejecución del proceso, *exit* ejecuta las funciones eventualmente registradas por las llamadas a las funciones de librería *atexit* y *on_exit*. Las funciones registradas por *atexit* son llamadas en orden inverso a como fueron registradas. La función *atexit* permite indicarle al sistema qué acciones queremos que se ejecuten al producirse la terminación de un proceso.
- Si el proceso actual posee procesos hijos, estos se vinculan al proceso número 1 (PID), que ejecuta el programa *init*. La señal SIGCHLD se envía al proceso padre para prevenirle de la finalización de uno de sus procesos hijos.
- Consecuencias en las llamadas al sistema, *exit*.
  - Las funciones registradas por *atexit* son llamadas en orden inverso a como fueron registradas. La función *atexit* permite indicarle al sistema qué acciones queremos que se ejecuten al producirse la terminación de un proceso.
  - El contexto del proceso es descargado de memoria, lo que implica que la tabla de descriptores de archivos es cerrada y sus archivos asociados cerrados, si no quedan más procesos que los tengan abiertos.
  - Si el proceso padre del que ejecuta la llamada a *exit* está ejecutando una llamada a *wait*, se le notifica la terminación de su proceso hijo y se le envían los 8 bits menos significativos de *estado*. Con esta información, el proceso padre puede saber en qué condiciones ha terminado el proceso hijo.

- Pasar al estado *zombie*. Si el proceso padre no está ejecutando una llamada *wait*, el proceso hijo se transforma en un proceso *zombie* después de realizar la llamada a *exit*. Un proceso *zombi*e sólo ocupa una entrada en la tabla de procesos del sistema y su contexto es descargado de memoria principal.
- Por último, el proceso, ya Zombie, ejecuta un cambio de contexto para que *kernel* pueda planificar a otro. Recuerde que si un programa no incluye *exit* en su código, la rutina de arranque que el compilador C incluye a todos los programas la ejecutará cuando el proceso regrese desde el *main*.

algoritmo *exit(status)*
{
Ignorar todas las señales;
if (proceso líder que tiene asociada la terminal)
{
Enviar señal SIGHUP *(hangup)* a todo el grupo *(kill)*;
Poner el identificador de grupo de todos los miembros a 0
}
Cerrar todos los archivos abiertos *(close)*;
Liberar directorio actual o CurDir *(iput)* y directorio raíz o RootDir *(iput)*;
Liberar regiones y memoria asociada al proceso *(freereg)*;
Almacenar estadística y contabilidad (y el status del *exit*) en la Tabla de Procesos:
Marcar proceso en estado de Zombie.
Cambiar el ID del padre de sus procesos hijos a 1 (*init* los adopta);
Si algún hijo está Zombie enviar SIGCHLD al proceso *init*;
Enviar SIGCHLD al proceso padre;
Hacer cambio de contexto;
}

2.7.2.2. Espera para la Terminación de Procesos, *wait*.

- La función *wait* suspende la ejecución del proceso padre que la llama hasta que alguno de sus procesos hijos termina (SIGCHLD).
- Su función es la de sincronizar la ejecución del proceso padre con la finalización de un proceso hijo.
- En la llamada a esta función (pid_t wait(int *estado)), ésta devuelve el PID de alguno de los procesos hijo *zombies*. El parámetro *estado* es la variable donde vamos a almacenar el valor que el proceso hijo le envía al proceso padre mediante la llamada *exit* y que da idea de la condición de finalización del proceso hijo. Si queremos ignorar este valor, podemos pasarse a wait un puntero NULL.
- Si durante la llamada a *wait* se produce algún error, la función devuelve –1 y en *errno* estará el código del tipo de error producido.
- El proceso que ejecuta *wait* queda bloqueado a nivel interrumpible a la espera de que algún hijo acabe (pase a Zombie). EL proceso que espera se despertará al recibir una señal y responderá a ellas como ya se ha revisado, a excepción de la señal SIGCHLD, a la que responderá de la siguiente manera:
  - En el caso por defecto, el proceso se despertará en del *sleep* que ha hecho en *wait*. Recordemos que *sleep* controla (al despertar) si hay señales pendientes y las procesa con *issig*. Este *issig* reconocerá el caso especial de SIGCHLD y devolverá FALSO. En consecuencia, el *kernel* no hace un *longjmp* desde el *sleep* sino que finaliza allí y vuelve a *wait*. Allí el proceso encontrará un hijo Zombie (el que le ha despertado) y acumulará datos, liberará la entrada en la tabla de procesos del proceso hijo y regresará del *wait*.
  - Si el proceso ha indicado un manejador de señal, él la procesará la SIGCHLD.
  - Si el proceso ha indicado ignorar la SIGCHLD, el *kernel* permanecerá en el *loop* del *wait* procesando al hijo Zombie y esperando por otros hijos.

- Llamada al sistema, *wait*.
  - Si el proceso que hace *wait*, no tiene hijos:
    + Error.
  - Para cada uno de sus procesos hijos *zombies*:
    + Añadir el uso de CPU del proceso hijo al proceso padre.
    + Liberar la entrada en la tabla de procesos del proceso hijo.
  - Si no tiene procesos hijos *zombies*:
    + Evento Dormir: hasta que el proceso hijo hace *exit*.

algoritmo *wait*
```
{
    if (proceso no tiene hijos)
        Return Error;
    for (;;)
    {
        if (proceso que espera tiene hijos Zombies)
        {
            Tomar uno de ellos;
            Acumular estadísticas del hijo en el padre;
            Liberar la entrada en la tabla de procesos del proceso hijo;
            Devolver (ID-hijo y código del exit);
        }
        if (proceso no tiene hijos)
            Return Error;
        Dormir a estado interrumpible (hasta que un hijo finalice);
    }
}
```

2.7.2.3. Ejemplo de la Sincronización de Procesos Padre e Hijo utilizando *exit* y *wait*.

```
int pid, estado
...
if ((pid = fork()) == -1)
    // Error en la creación del proceso hijo.
else if (pid == 0)
{
    // Código del proceso hijo.
    exit(10);
}
else
{
    // Código del proceso padre
    pid = wait(&estado);
    // Cuando el proceso hijo llame a exit(), le pasará al padre el valor 10, que éste pude recibir
    // a través de la variable estado (estado = 10).
}
```

2.7.3. Dormir *(sleep)* y Despertar *(wakeup)* procesos.

Dormir.

- Cambia el estado del proceso de "ejecutándose en modo *kernel*" a "durmiendo en memoria".

Despertar.

- Cambia el estado del proceso de "dormido" (durmiendo en memoria principal o en memoria secundaria) a "listo para ejecutarse" en memoria principal o en memoria secundaria (área de swap).

2.7.3.1. Contexto de Ejecución de Procesos que están Dormidos.

- Los procesos duermen durante la ejecución de una llamada al sistema.
  - El proceso entra en modo *kernel* (capa de contexto 1) cuando ejecuta un *trap* (interrupción software para pasar de modo usuario a modo *kernel*) del sistema operativo.
  - El proceso duerme *(sleep)* esperando un recurso (operación de E/S) ⇒ cambio de contexto ⇒ apila la capa de contexto actual y se ejecuta en la capa de contexto 2 (Figura 2.12).
- El proceso duerme también cuando ocurre una falta de página (páginación).
  - Duerme mientras el *kernel* lee el contenido de la página requerida, bien desde el búffer caché, de memoria principal o de memoria secundaria.

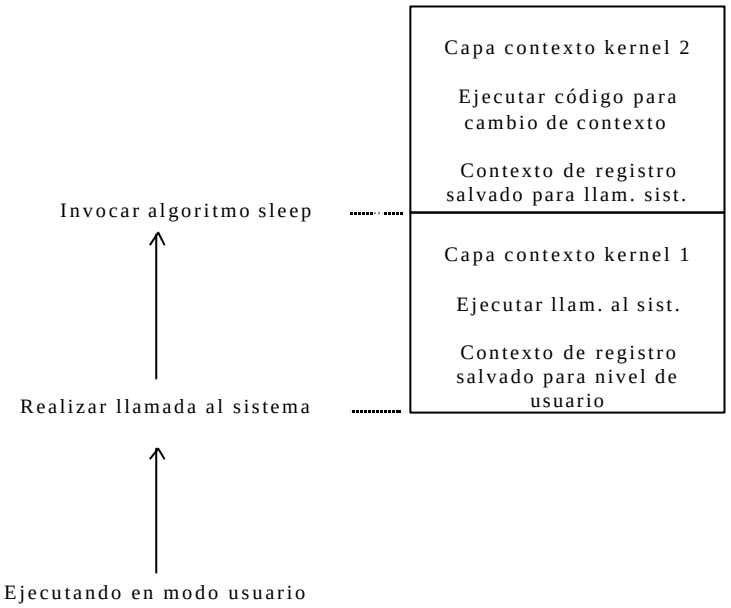

Figura 2.12. Capas de contexto típicas de un proceso dormido.

2.7.3.2. Eventos y Direcciones en los que Duermen los Procesos.

- Abstracción = Modelo para manejar procesos (supone).
  - Los procesos se duermen en un evento (dormir) ⇒ estado "dormido" en el diagrama de transición de estados de un proceso UNIX.
  - Ocurre el evento (despertar) ⇒ despiertan y pasan al estado "listo para ejecutarse".
- Implementación ⇒ El *kernel* establece una correspondencia *(mapea)* entre el conjunto de eventos y el conjunto de direcciones virtuales del *kernel* asociados a esos eventos.
  - Las clases de eventos que podríamos establecer son: esperando finalización de E/S, esperando búffer, esperando inodo y esperando entrada por terminal.
- Ni abstracción ni implementación del evento distinguen el número de procesos que esperan en el evento (duermen), es decir esta abstracción de eventos es genérica ⇒ anomalías debidas al número de procesos que esperan en el evento.
  - Anomalía 1.
    - + Se produce un evento (dormir) en el cual tenemos varios procesos.
    - + Llamada a *wakeup* para procesos que dormían en dicho evento.
    - + Todos ellos se despiertan ⇒ pasan a estado "listo para ejecutarse" ⇒ compiten por el mismo recurso.
    - + Muchos volverán a dormirse después de pasar por "ejecución en modo *kernel*".
  - Anomalía 2.
    - + Varios eventos pueden corresponder en una única dirección virtual. Figura 2.13 (esperando finalización de E/S y esperando búffer).
      - * Se completa la operación de E/S para el búffer.
      - * El *kernel* despierta procesos que dormían en ambos eventos.
      - * Proceso en espera de finalización E/S ⇒ bloquea búffer.
      - * Procesos en espera de búffer libre ⇒ duermen otra vez.

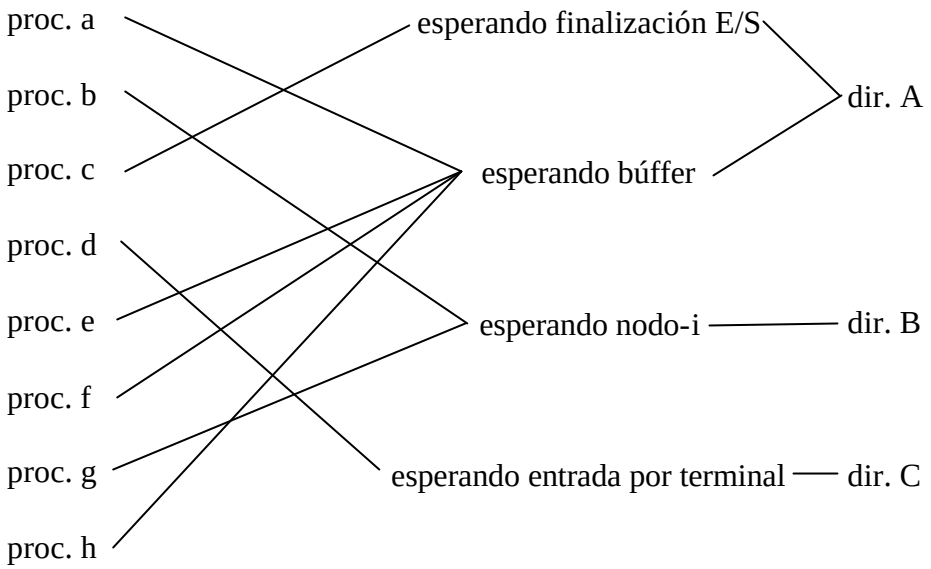

Figura 2.13. Procesos durmiendo en eventos y correspondencia entre eventos y direcciones virtuales.

2.7.3.3. Problemas.

- El proceso duerme hasta que ocurre el evento apropiado ⇒ Procesos duermen habitualmente en eventos que ocurren con seguridad.
- Procesos dormidos en eventos que pueden no producirse ⇒ Mecanismo para retomar el control y continuar la ejecución ⇒ El *kernel* interrumpe al proceso que duerme enviándole una señal (interrupción software).
- Podemos distinguir varios tipos "dormido" (procesos que han alcanzado este estado), dependiendo de prioridades.
  - El *kernel* fija la prioridad de planificación del proceso "dormido" cuando pasa a dicho estado, basada en la prioridad con la que duerme (niveles de prioridad para procesos en estado "dormido").
  - Si la prioridad del proceso "dormido" está por encima de un valor umbral ⇒ Proceso no despertará ante la recepción de una señal.
  - Si la prioridad del proceso "dormido" está por debajo de dicho umbral ⇒ Despertará inmediatamente ante la recepción de la señal.
- Si un proceso despierta ante una señal.
  - El *kernel* realiza acciones especiales (*longjmp*, que restaura el entorno salvado por *setjmp*) para restaurar el contexto salvado si no puede completar la llamada al sistema en ejecución. Y que, no es suficiente con restaurar la capa de contexto del nivel de sistema anterior.

2.7.3.4. Acciones para Dormir (sleep).

- El *kernel* eleva el nivel de ejecución del procesador (prioridades) ⇒ Bloquear las interrupciones.
- El *kernel* marca el proceso como "dormido", en el campo de estado asociado a su descriptor de proceso.
- El *kernel* guarda la dirección virtual sobre la que duerme (dirección de bloqueo) y la prioridad en la tabla de procesos.
- El *kernel* coloca al proceso en una cola *hash* de procesos "dormidos".
- Si el proceso que duerme no puede ser interrumpido por señales, entonces *sleep* no puede ser interrumpido por ninguna señal (caso más sencillo).
  - Cambio de contexto y pasa a estado "dormido".
  - Cuando despierta pasa a "listo para ejecutarse" y el *kernel* lo planificará en algún momento (scheduler), retomando su ejecución vía un cambio de contexto.
  - Su ejecución continuará justo a partir de este momento.
    + El proceso retorna de su cambio de contexto en el *sleep*
    + Restaura el nivel de ejecución del procesador al valor inicial.
    + Retorna.

- Si el proceso lo despertó una señal.
  - Elimina el proceso de la cola de procesos dormidos.
  - Restaura el nivel de ejecución del procesador al valor inicial.
  - Restaura contexto (*longjmp*, que restaura el entorno salvado por *setjmp*).
- Si no hay señales pendientes.
  - Cambio de contexto y pasa a dormir.
  - Cuando despierta, el *kernel* lo planificará en algún momento (scheduler).
  - Su ejecución continuará justo a partir de este momento.
    + Si no hay señales pendientes ⇒ retorno normal.
      * Restaura el nivel de ejecución del procesador al valor inicial.
      * Retorna.

algoritmo *sleep(dirección_bloqueo, prioridad)*

```
{
        Subir nivel de ejecución (todas las interrupciones;)
        Estado_Proceso = sleep;
        Colocar proceso en tablas de bloqueo (usar dirección_bloqueo);
        Salvar dirección en la Tabla de Procesos;
        Almacenar el nivel de prioridad en la Tabla de Procesos;
        if (proceso es No interrumpible)
        {
                Hacer cambio de contexto;
                // cuando el proceso despierte lo hará en este punto
                Restaurar nivel de ejecución del procesador al valor antes de dormir;
                Return 0;
        }
        // Aquí el proceso duerme a un a prioridad interrumpible
        if (no hay señales pendientes para este proceso)

                Hacer cambio de contexto;
                // cuando el proceso despierte lo hará en este punto
                if (no hay señales pendientes para este proceso)
                {
                        Restaurar nivel de ejecución del procesador al valor antes de dormir;
                        Return 0;
                }
        Retirar proceso de la colas de bloqueo;
        Restaurar nivel de ejecución del procesador al valor antes de dormir;
        if (prioridad del proceso permite responder a señales)
                Return 1;
        else
                Hacer longjmp;
}
```

2.7.3.5. Acciones para Despertar (wakeup).

- El *kernel* eleva el nivel de ejecución del procesador en *wakeup*.
- Para cada proceso que duerme en la dirección de entrada:
  - Estado del proceso = "listo para ejecutarse".
  - Elimina de la lista enlazada de procesos dormidos, el proceso en cuestión.
  - Coloca en la lista enlazada de procesos elegibles para planificar por el scheduler ("listos para ejecutarse").
  - Limpia el campo de la tabla de procesos asociado al proceso que indicaba la dirección de dormido (estado del proceso).
  - Si el proceso despertado no residía en memoria principal ⇒ Despierta al swapper, para que intercambie el proceso a memoria principal (si no hay paginación por demanda).
  - Si reside en memoria principal y el proceso despertado es "más elegible para ejecutar" que el proceso que se está ejecutando (según la prioridad que tiene asociada) ⇒ Activar el flag de planificación para que el scheduler le asigne la CPU.

algoritmo *wakeup(dirección_bloqueo)*

```
{
    Subir nivel de ejecución (todas las interrupciones);
    Encontrar la tabla hash para la dirección_bloqueo;
    foreach (proceso en la cola)
    {
        Retirar proceso de la cola;
        Marcarlo como listo para ejecutarse;
        Ponerlo en cola de Listos;
        Borrar dirección_bloqueo de la Tabla de Procesos;
        if (proceso no está en memoria)
            Despertar swapper (proceso 0);
        else
            if (proceso despertado es de mayor prioridad que el proceso en ejecución)
                Activar flag de planificación;
    }
    Restaurar nivel de ejecución
}
```

2.7.3.6. Conclusiones sobre *sleep* y *wakeup*.

- Debemos distinguir al menos dos situaciones, procesos que duermen en eventos que son seguros de ocurrir y eventos que no lo son. El *kernel* asigna una prioridad de sueño basado en el conocimiento de la seguridad de ocurrencia del evento y dispone de un umbral para determinar si el proceso será despertado o no al recibir una señal. Si el proceso ya tiene una señal cuando entra al algoritmo *sleep* y su prioridad es interrumpible no entrará a dormir (puede que la señal no se repita y el proceso se quede dormido sin despertar). Si su prioridad es mayor que el umbral, se irá a dormir esperando por un *wakeup* explícito.
- Cuando un proceso es despertado por efecto de una señal (o no fue a dormir porque tenía una pendiente) el *kernel* puede hacer un *longjmp* dependiendo de la razón por la que se invocó *sleep*. El *kernel* hace un *longjmp* para restaurar el contexto previamente salvado (si no hay forma de completar la llamada al sistema que estaba ejecutando). Por ejemplo, si un *read* de terminal es interrumpido porque se apaga el terminal, el *read* no debería continuar esperando, sino retornar con una indicación de error. Esto es genérico para las llamadas al sistema que pueden ser interrumpidas mientras el proceso está dormido. El proceso no debería continuar normalmente después de despertar ya que la llamada al sistema no ha sido satisfecha. Por ello el *kernel* salva el contexto del proceso al inicio de la llamada al sistema usando *setjmp* anticipándose a la posible necesidad de usar un posterior *longjmp*.

- Hay otras ocasiones en que el *kernel* necesita despertar el proceso al recibir una señal pero no hacer el *longjmp*. El *kernel* invoca el algoritmo con una prioridad especial que suprime la ejecución del *longjmp* y produce que el algoritmo devuelva 1. Esto es más efectivo que hacer un *setjmp* justo antes de la llamada a *sleep* para luego volver a hacer *longjmp* para restaurar contexto. El propósito de esto es permitir al *kernel* limpiar estructuras de datos locales. Por ejemplo, un manejador de dispositivo puede asignar estructuras locales privadas e ir a dormir a una prioridad interrumpible. Si es despertado por una señal debería liberar las estructuras y luego hacer el *longjmp* si fuera necesario. El usuario no tiene control sobre cuando un proceso hace o no un *longjmp*, ello depende de la razón por la cual el proceso se va a dormir y que estructuras del *kernel* necesitan modificarse antes de que el proceso regrese de la llamada al sistema.

2.7.4. Llamadas a Otros Programas. Familia de Funciones *exec*.

- Existe toda una familia de funciones *exec* que podemos utilizar para ejecutar programas. Dentro de esta familia, cada función tiene su interfaz propia, pero todas tienen aspectos comunes y obedecen al mismo tipo de funcionamiento.
- Llamada al sistema *exec* ⇒ permite a un proceso invocar o llamar a otro programa.
- El resultado que se consigue con estas funciones: Copia del archivo ejecutable ⇒ Cargar un programa en memoria en el espacio de direcciones del proceso (zona de memoria) que ejecuta la llamada, sobrescribiendo los segmentos del programa antiguo con los del nuevo.
- El contenido del contexto del nivel de usuario del proceso que llama a *exec* deja de ser accesible y es reemplazado de acuerdo con el nuevo programa. Es decir, el programa antiguo es sustituido por el nuevo y nunca retornaremos a él para proseguir su ejecución, ya que es el programa nuevo el que pasa a ejecutarse.
- La familia *exec* esta formada por las siguientes funciones: *execl*, *execv*, *execle*, *execve*, *execlp* y *execvp*. En todas ellas destacamos que hay un parámetro que se corresponde con el nombre absoluto o relativo de un archivo ordinario ejecutable.
- Si *exec* devuelve el control al programa que la llama, es porque no se ha ejecutado correctamente. En este caso devuelve el valor –1, y en *errno* estará el código del tipo de error producido.

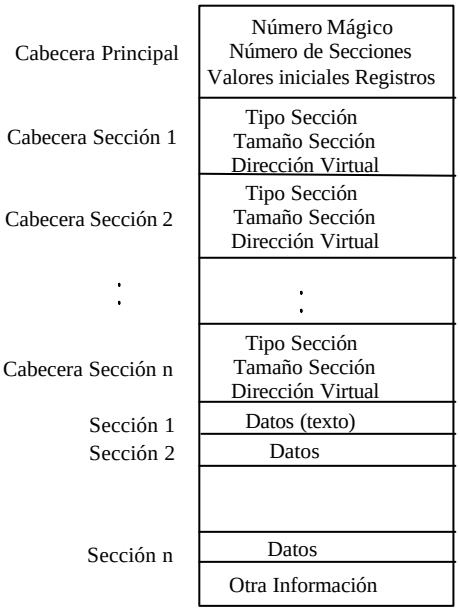

Figura 2.14. Estructura de un archivo ejecutable en UNIX (a.out).

2.7.4.1. Estructura de un Archivo Ejecutable (Figura 2.14).

Básicamente todo el programa consta de una cabecera principal y una serie de secciones, cada una de las cuales (secciones) se compone de una cabecera y una zona de datos. Los programas son generados por el linkador (enlazador, programa *ld*) y sus partes son las siguientes:

- La *cabecera principal* describe:
  - Cuántas secciones hay en el archivo (programa).
  - La dirección virtual de inicio o comienzo para la ejecución del proceso.
  - El *número mágico* del programa que identifica qué tipo de archivo es (ejecutable).
- Las *cabeceras de sección* describen cada una de las secciones del archivo, como:
  - El tamaño de la sección.
  - El rango de direcciones virtuales que la sección debería ocupar cuando se ejecute el archivo.
  - Otra información, como por ejemplo, el tipo de sección.
- Las *secciones* que contienen el código del programa y las variables globales.
- *Otras secciones* que pueden contener tablas de símbolos y otros datos útiles para el depurado, etc.

2.7.4.2. Acciones: *exec(nombre_archivo_ejecutable, parámetros)*.

- Accede al archivo.
- Verifica que es ejecutable, regular y tiene el permiso del usuario para ejecutarlo.
- Lee la cabecera del archivo.
- Almacena parámetros y libera recursos de memoria que forman el contexto del nivel de usuario del proceso que lo ejecuta.
- Para cada región especificada en el módulo de carga:
  - Asignar región.
  - Vincular región.
  - Cargar región.
- Almacena los parámetros del *exec* en la pila del usuario.
- Borra las direcciones de los manejadores de señales del usuario del área de usuario (u-Area).
- Inicializa los registros de usuario salvados (contexto de nivel de registros), para poder volver a modo usuario.
- Libera el inodo del archivo ejecutable.

algoritmo *exec(archivo, argums, env)*
{
Obtener i-nodo asociado al archivo *(namei)*;
Controlar que sea un ejecutable, permisos, etc.;
Leer el layout del archivo (descripción del ejecutable: número de secciones, dirección de inicio de ejecución, tipo de ejecutable. Para cada sección: tamaño, dirección virtual que ocupa...). Esta parte del algoritmo también se debe ocupar de cargar el código, inicializar variables, etc);
Copiar *argums* al espacio del *kernel*. Esto se debe a que los *argums* pertenecen al espacio de direcciones del usuario *(user)*, y este espacio va a ser liberado en los siguientes pasos. La implementación se puede realizar en la pila del *kernel*, pero si hay limitaciones en el tamaño de ésta se debe implementar sobre páginas dedicadas a ellas (recordemos que el tamaño de los *argums* puede ser relativamente grande);
Para cada región asociada al proceso.
Desligar la región *(deattachreg)*. Debe haber un tratamiento especial para las regiones de texto compartidas. También observar que en este punto el nuevo proceso no tiene contexto de usuario, así que cualquier error que se detecte más adelante debe abortar via una señal;

Para cada región que se especifique en el módulo de carga.
Asignar nuevas regiones *(allocreg)*;
Asociar la región *(attachereg)*;
Si procede, cargar la región en memoria *(loadreg)*. La región de datos es inicialmente dividida en dos partes: datos inicializados en compilación y datos no inicializados *(bss)*. La asignación se realiza para datos inicializados y después debe usar *growreg* para los no inicializados y ponerlos a cero. Por último, asigna una región para la pila del proceso y la asocia;
Copiar *argums* a la nueva pila (stack) del proceso (en la pila creada en el punto anterior);
Proceso especial para el *setuid*. El *kernel* inicializa el array de direcciones de tratamiento de señales en la u-Area (las pendientes se quedarán sin ser atendidas);
Preparar contenido de registros para regresar a modo usuario. Ahora el *kernel* modifica los registros en el contexto de usuario (primera capa) colocando el nuevo valor del puntero a pila y el contador de programa;
Liberar el i-nodo *(iput)*, para balancear el *namei* (equivale a haber hecho un *open* y un *close* del archivo ejecutable, a excepción que no hay una entrada en la Tabla de Archivos.
}

2.7.4.3. Comentarios sobre *exec*.

- Cuando el proceso llame *exec* ⇒ ejecutará el código del nuevo programa.
- Es el mismo proceso que antes del *exec*.
  - Mismo PID de proceso y posición en la jerarquía de procesos.
  - Diferente el contexto del nivel de usuario.
- Texto (código) y datos en secciones separadas del programa ejecutable ⇒ regiones separadas del proceso.
- Ventajas de separar el texto (código) y los datos: protección y compartición.
- Optimización: Compartir regiones (segmento de código) ⇒ *sticky-bit*.
- *sticky-bit* ⇒ el *sticky-bit* es uno de los bits de la palabra de modo de un archivo, sólo es aplicable a archivos ejecutables e indica que el segmento de código del programa (archivo ejecutable) puede ser compartido por varios procesos. Es decir, este bits indica al *kernel* que dicho archivo es un programa con capacidad para que varios procesos compartan el segmento de código (texto) y que este segmento se debe mantener en memoria, aun cuando alguno de los procesos que lo utiliza deje de ejecutarse o pase al área de intercambio (*swap*). La técnica de compartir el mismo código entre varios procesos permite gran ahorro de memoria en el caso de programas muy utilizados, como editores de texto, compiladores, etc. Cuando el *kernel* tiene que liberar memoria asignada al proceso (bien porque ha terminado su ejecución o bien porque debe pasar al área de *swap*), si hay otros procesos compartiendo el segmento de código, este segmento no es descargado de memoria. Solo el *superusuario* puede modificar el *sticky-bit* de un programa. El empleo de este bit (*sticky-bit*) supone una optimización en el trasiego de información entre el disco y la memoria, en el caso de programas muy utilizados como son editores, compiladores, etc. Cuando finaliza la ejecución de un proceso que tiene el *sticky-bit* activo, el *kernel* no libera su región de texto; pero en otras situaciones es necesario hacerlo:
  - Si un proceso abre el archivo para escritura; si se modifica el contenido del archivo, se invalida el contenido de la región.
  - Si un proceso cambia los permisos del archivo *(chmod)* y desactiva el *sticky-bit*.
  - Si un proceso ejecuta *unlink* sobre el archivo.
  - Si se desmonta el sistema de archivos que contiene el archivo
  - Si *kernel* se quedara sin espacio en *swap*, podría intentar aliviarse liberando regiones.
- Activar modo *sticky-bit* para archivos ejecutables usados frecuentemente.
- Ventaja ⇒ Texto o código está en memoria ⇒ Tiempo de *startup* del proceso es menor.

2.7.5. Información sobre Procesos. Identificadores de Proceso, Identificadores de Usuario y Grupo, Variables de Entorno y Parámetros Relativos a Archivos.

En este apartado vamos a estudiar las llamadas necesarias para consultar y fijar algunos de los parámetros más importantes de un proceso, los que describen cómo se relaciona el proceso con el resto del sistema. Nos vamos a centrar en los siguientes aspectos: identificadores asociados a un proceso, identificadores de usuario y grupo asociados al proceso, variables de entorno y parámetros relativos a archivos.

2.7.5.1. Identificadores de Proceso.

- Todo proceso tiene asociado dos números desde el momento de su creación:
  - El identificador de proceso ⇒ Este identificador (PID) es un número entero positivo que actúa a modo de nombre de proceso.
  - El identificador del proceso padre ⇒ Este identificador (PPID) es el PID del proceso que ha creado al actual (proceso hijo).
- El PID de un proceso no cambia durante el tiempo de vida de éste; si embargo, su PPID sí puede variar. Esta situación se da cuando el proceso padre muere, pasando el PPID del proceso hijo a tomar el valor 1 (PID del proceso *init*).
- Para leer los valores de PID y PPID se utilizarán las llamadas *getpid* y *getppid* respectivamente.
- En UNIX, los procesos van a estar agrupados en conjuntos de procesos que tienen alguna característica común (por ejemplo, tener un mismo proceso padre). A estos conjuntos se les conoce como grupos de procesos y desde el sistema son controlados a través de un identificador de grupo de procesos (PGRP). Para determinar a qué grupo pertenece un proceso, utilizaremos la llamada *getpgrp*.
- El identificador de grupo de procesos (PGRP) es heredado por los procesos hijo después de realizar una llamada a *fork*, pero también puede cambiarse creando un nuevo grupo de procesos, con la llamada a *setpgrp*.
- Con la llamada a *setpgrp* hace que el proceso actual se convierta en líder (leader) de un grupo de procesos. El identificador del grupo de procesos (PGRP) va a coincidir con el PID del proceso líder y es el valor retornado por *setpgrp*. Si la llamada a *serpgrp* falla, devolverá el valor –1 y en *errno* estará el código del error producido.

2.7.5.2 Identificadores de Usuario y de Grupo.

- El *kernel* asocia a cada proceso dos identificadores de usuario y dos identificadores de grupo:
  - Los identificadores de usuario son:
    - Identificador de usuario real (UID).
    - Identificador de usuario efectivo (EUID).
  - Los identificadores de grupo son:
    - Identificador de grupo real (GID).
    - Identificador de grupo efectivo (EGID).
- El identificador de usuario real (UID) ⇒ Identifica al usuario que es responsable de la ejecución del proceso. Y el identificador de grupo real (GID) identifica al grupo al cual pertenece el usuario.
- El identificador de usuario efectivo (EUID) se utiliza para:
  - Determinar el propietario de los archivos que se creen.
  - Comprobar la máscara de permisos de acceso a archivos y permisos para enviar señales a otros procesos. Es decir, comprobar los permisos de usuario para permitir acceder a archivos de otros usuarios.
- Normalmente, el UID y el EUID coinciden, pero si un proceso ejecuta un programa que pertenece a otro usuario y que tiene activo el bit S_ISUID (cambiar el UID del usuario al ejecutar), el proceso va a cambiar su EUID y va a tomar el valor del UID del nuevo usuario. Es decir, a efectos de comprobación de permisos de usuario, va a tener los mismos permisos que tiene el usuario cuyo UID coincide con el EUID del proceso. S_ISUID ⇒ este es otro de los bits de la palabra de modo de un archivo y le indica al *kernel* que cuando un proceso acceda a este archivo, cambie el identificador de usuario (UID) y le ponga el del propietario del archivo (UID). Esto tiene aplicación cuando intentamos acceder a archivos que son de otro usuario y no tenemos permisos para escribir en ellos.

- Con respecto al identificador de grupo efectivo (EGID), se aplica la misma norma, y el EGID es el GID del grupo al cual pertenece el usuario indicado en el EUID. S_ISGID ⇒ otro de los bits de la palabra de modo de un archivo (cambia el GID del grupo al ejecutar) y tiene un significado parecido al de S_ISUID, pero referido al grupo de usuarios al que pertenece el propietario del archivo. Así, cuando ejecutamos un programa que tiene activo este bit, nuestro EGID (identificador de grupo efectivo) toma el valor del GID del propietario del programa.
- Para determinar qué valores toman estos identificadores utilizamos:
  - La llamada *getuid*, que devuelve el identificador del usuario real o UID.
  - La llamada *geteuid*, que devuelve el identificador del usuario efectivo o EUID.
  - La llamada *getgid*, que devuelve el identificador del grupo real o GID.
  - La llamada *getegid*, que devuelve el identificador del grupo efectivo o EGID.
- Tanto *setuid* como *setgid* van a tener un comportamiento u otro, dependiendo si el valor que tiene el EUID del proceso (identificador de usuario efectivo) es el del superusuario o no.
  - En el caso de *setuid(uid)*:
    + Si el EUID del proceso (identificador de usuario efectivo) es el del superusuario, el *kernel* va a cambiar el UID y el EUID (en la tabla de procesos y en el u-Area) del proceso para que tomen el valor del parámetro *uid*.
    + Si el EUID del proceso (identificador de usuario efectivo) no es el de superusuario, pero el valor del parámetro *uid* coincide con UID (identificador de usuario real), EUID va a tomar el valor de *uid*. Esto se hace para restaurar el valor de EUID después de que el proceso ejecute algún programa que tiene activo el bit S_ISUID. En estos casos, *setuid* devuelve 0 para indicar que no se ha producido ningún error y en cualquier otro caso devolverá –1 y en *errno* estará el código del error producido.
  - En el caso de *setgid(gid)*, se aplica lo dicho en *setuid*, pero referido al GID y al EGID. En concreto, si el EUID del proceso es el del superusuario, entonces el GID y el EGID cambian para tomar el valor del parámetro *gid*, y si no lo es, pero *gid* tiene el valor GID, entonces el EGID toma el valor de *gid*. En cualquier otro caso se producirá un error.
- Programa *login* llama a *setuid* y *setgid* (ejemplo de aplicación de estas dos llamadas).
  - Este programa (*login*) se ejecuta con el EUID (identificador de usuario efectivo) del superusuario (usuario *root*).
  - Nos pregunta nuestro nombre de usuario *(username)* y nuestra palabra de paso *(password)*.
  - Consulta en el archivo */etc/passwd* cuáles son nuestros UID y GID, para hacer sendas llamadas a *setuid* y *setgid*, y que los identificadores UID, EUID, GID y EGID pasen a ser los del usuario que quiere iniciar la sesión de trabajo.
  - Luego llama a *exec* para ejecutar un intérprete de órdenes que nos dé servicio. Esta *shell* se va a ejecutar con los identificadores de usuario y de grupo, tanto reales (UID y GID) como efectivos (EUID y EGID), de acuerdo con el usuario que haya entrado en sesión.
- Los procesos hijo van a heredar el UID, EUID, GID y EGID del padre.

2.7.5.3. Variables de Entorno.

- Cuando un proceso empieza a través de una llamada a *exec*, el sistema pone a su disposición un array de cadenas de caracteres conocido como *entorno (environ)*.
- Para los programas que se ejecuten mediante una llamada a *execl*, *execv*, *execlp* o *execvp*, el entorno es accesible únicamente a través de una variable global que se declara como: *extern char **environ;*.
- Para las llamadas *execle* o *execve*, el entorno es accesible también a través del tercer parámetro de la función *main*, el parámetro *char **envp*.
- Tanto *environ* como *envp* tienen una estructura de array de cadenas de caracteres terminado con un puntero NULL. Por convenio, cada una de estas cadenas tiene la forma: *Variable_entorno = Valor_variable*.

- Algunas de la variables utilizadas por programas estándar:
  - HOME: Directorio de inicio de sesión de un usuario. */users/antonio*.
  - LANG: Identifica el idioma del Soporte en Lengua Nativa (NLS, Native Language Support).
  - PATH: Identifica la secuencia de directorios en los que programas como *time*, *nice*, *nohup*, etc., van a buscar cuando se especifica un archivo mediante su nombre y no mediante su *path name*. *.:/bin:/usr/bin:usr/local/bin*.
  - TERM: Identifica el tipo de terminal hacia el que se va a dirigir la salida. Es utilizado por programas vi, ed, etc. *vt100*.
- Se pueden ver el valor de todas las variables de entorno asociadas a nuestra sesión de trabajo mediante la orden *env*.
- El entorno de un proceso es heredado por todos los hijos, después de la llamada *fork*.
- Para preguntar por el valor de una variable de entorno determinada o para declarar nuevas variables, podemos utilizar las siguientes funciones: *getenv* y *putenv*.
  - La función *getenv(name)*, busca en la zona de variables de entorno una cadena de caracteres que tenga la forma *name=value* y devuelve un puntero a *value* en esta zona. Si la cadena no existe devuelve un puntero a NULL.
  - La función *putenv(string)*, permite declarar una nueva variable de entorno o modificar el valor de una ya existente. El parámetro *string* debe tener la forma *"name=value"*. Hay que tener en cuenta que la zona de memoria a la que apunta *string* es añadida al entorno y se producirá un error si *string* es una variable local a una función, ya que al abandonar una función deja de existir. La función *putenv* manipula el entorno referenciado por *environ*, pero no el referenciado por *envp* (tercer parámetro de la función *main*).
- Debemos indicar también que las variables de entorno son locales a cada proceso, de forma que las modificaciones que se realicen sobre ellas no se conservan cuando muere un proceso. Esto no es aplicable a las variables que han sido declaradas como globales mediante la orden *export*.

2.7.5.4 Parámetros Relativos a Archivos.

- Todo proceso tiene asociado un directorio de trabajo (CWD) y un directorio raíz.
  - El directorio de trabajo (CWD) indica dónde van a estar referidos los accesos a archivos que se realicen mediante un nombre de archivo y no mediante un *path name* (nombre de archivo absoluto). La llamada para cambiar el CWD asociado a un proceso es *chdir*.
  - El directorio raíz indica cuál es, dentro del sistema de archivos, el directorio que va a considerar el proceso como su directorio raíz. Por lo general, el directorio raíz va a coincidir con "/", pero lo podemos cambiar con la llamada *chroot.*
- Relacionado con la creación de archivos, cada proceso va a tener una máscara que indica qué bits, dentro de los bits de permiso en el modo del archivo, deben estar inactivos. La llamada *umask* es la permite fijar esta máscara. Por ejemplo, crear un archivo con la máscara de permisos (lectura, escritura y ejecución para el propietario, grupo y otros), rwxr--r--, ya que la máscara de creación de archivo se ha fijado a 000 011 011 ⇔ 0033, *umask(0033)*.
- Los procesos van a tener limitado el tamaño máximo de los archivos que pueden crear y el tamaño máximo de memoria que pueden tener asignado. Para consultar estos parámetros se utiliza la función *ulimit*.

2.8. SINCRONIZACIÓN DE PROCESOS EN LINUX.

En un instante dado, sólo un proceso puede ejecutarse en modo *kernel*. Aunque es posible que se apliquen interrupciones hardware y software a este proceso, Linux no provoca la planificación (scheduler) si el proceso actual está activo en modo *kernel*. Un proceso que se ejecuta en modo *kernel* puede provocar, sin embargo, un cambio del proceso actual suspendiendo su ejecución (dormir). Esta suspensión voluntaria se debe generalmente a la espera de un evento, tal como el fin de una entrada/salida o la terminación de un proceso hijo. Linux proporciona varios mecanismos que permiten a los procesos sincronizarse en modo *kernel* (implementadas en el código del *kernel* = servicios internos): las *"bottom-halves"*, los temporizadores (timers), las *colas de tareas*, las *colas de espera* y los *semáforos* (control de acceso a un recurso).

2.8.1. Bottom-halves.

- Las "Bottom-halves" (BHs) que constituyen la versión primitiva de las funciones utilizadas para dejar para más adelante los aspectos menos urgentes de las rutinas de servicio de las interrupciones. En muchos manejadores de interrupción no en necesario ni conveniente, realizar todo el trabajo de atención y procesado de la interrupción durante la propia atención de la interrupción: (1) parte del trabajo se puede retrasar utilizando en mecanismo denominado "bottom-half" (segunda parte). (2) La implementación es poco eficiente, a costa de ser poco flexible y oscura. (3) Se utiliza un mapa de bits (representado por un entero) para marcar como activa una de las bottom-halves. Por tanto, como máximo se pueden tener 32 rutinas de bottom-half (tantos bits como tiene un entero). Éste fue el mecanismo que inicialmente se diseñó en el *kernel*, y para eliminar la limitación de 32 BHs, se añadió el mecanismo de colas de espera.
- En include/linux/interrupt.h están las estructuras de datos relativas a las BHs y en include/asm-i386/softirq.h las funciones (las funciones definidas como inline se expanden en el código que las invoca, eliminando la sobrecarga de un salto a función). (1) init_bh(nr, routine), inicializa la BH nr para que se ejecuta la rutina routine. (2) mark_bh(nr), activa el bit correspondiente mapa de bits bh_active con lo que esta BH se ejecutará una vez. (3) disable_bh(nr), elimina el bit de esa BH del mapa de bits bh_mask, con lo que la BH queda temporalmente deshabilitada. (4) enable_bh(nr), restaura el bit en la máscara bh_mask, reactivando la BH. Estas dos últimas funciones se suelen utilizar desde de las propias rutinas de la BHs para evitar que mientras se está sirviendo la BH se puede volver a llamar a esa misma BH.
- Para poder utilizar una BH, primero hay que llamar a init_bh que asociará una función con nuestra BH. Por ejemplo desde la función sched_init se inicializan varias BHs, entre ellas la del reloj. Luego, dentro del manejadores de interrupciones podremos "marcar" nuestra BH para que se ejecute antes de volver a código en modo usuario. Por ejemplo, la rutina do_timer llama a mark_bh(TIMER_BH) para activar su BH, tras lo que el sistema continua atendiendo esta interrupción.
- Desde tres lugares (do_irq, ret_from_sys_call, schedule), se llama a do_bottom-half, que invoca por fin a todas las funciones postpuestas. En el archivo, include/linux/interrupt.h están definidas las constantes que identifican a cada una de las BH de todo el sistema. Es importante tener declarados en un único lugar del código todas las BHs para evitar que dos "drivers" del *kernel* utilicen el mismo número de BH (sólo hay disponibles 32 BHs).

2.8.2. Temporizadores del Kernel (timers).

- Linux gestiona una lista de timers con el objetivo de poder poner procesos en espera durante un periodo de tiempo determinado. Estos timers se organizan en forma de una lista circular doblemente enlazada. Los timers a desencadenar en un futuro próximo se colocan al principio de la lista, mientras que los elementos a desencadenar en un futuro lejano se al final de dicha lista. La estructura timer_list, se define en el archivo linux/timer.h según la siguiente estructura

```
struct timer_list {
    struct list_head list;
    unsigned long expires;
    unsigned long data;
    void (*function) (unsigned long);
    volatile int running;
};
```

- La variable timer_head, definida en el archivo kernel/sched.c, contiene la dirección del primer elemento (timer) de la lista. La fecha de expiración (expires) se expresa en número de ciclos de reloj desde el arranque del sistema. La variable global jiffies es mantenida por el *kernel* (su valor se incrementa en cada interrupción de reloj) y contiene siempre el número de ciclos de reloj transcurridos desde el arranque. Además, el campo expires tiene el valor de jiffies cuando el manejador function debe ser invocado con data pasado como parámetro.

- El *kernel* de Linux, de manera interna, implementa timers con el objetivo de provocar esperas cronometradas. En el archivo fuente kernel/sched.c se define varias funciones para la gestión de timers: (1) add_timer(), añade un timer a la lista: explora la lista de timers registrados para insertar un argumento de modo que la lista quede ordenada, y con ello, los primeros elementos de la lista corresponden a los timers a desencadenar en primer lugar. (2) del_timer(), suprimer un timer de la lista, modificando simplemente los encadenamientos para eliminar sin problemas el elemento de la lista. (3) run_timer_list(), esta función es llamada por la cola de tareas (task queue) tq_timer. Para ello, explora la lista de timers, y ejecuta cada timer expirado, es decir, aquel timer cuyo campo expires es menor o igual a la variable global jiffies, llamando a la función asociada (function) con el parámetro especificado en el campo data. Y cada uno de los timers ejecutados, se suprime de la lista.

2.8.3. Colas de Tareas.

- A partir de la versión de la versión 2.0 del *kernel* de Linux se crearon las colas de tareas como una ampliación de BHs que permiten encolar tantas funciones como se quiera para después ser procesadas una tras otra. Aunque se llaman colas de tareas, realmente son "pilas de funciones", ya que son realmente funciones que se ejecutan en orden LIFO. Uno puede crear una cola de tareas utilizando la macro DECLARE_TASK_QUEUE() y encola las tareas utilizando queue_task(), pudiendo ser procesada la cola de tareas por la función run_task_queue(). En lugar de crear nuestra propia cola de tareas, se pueden utilizar las 4 colas de tareas que en Linux, include/linux/tqueue.h, se definen de forma predefinida: (1) tq_timer, se ejecuta tras cada tic de reloj. (2) tq_inmediate, se ejecuta siempre que se evalúan las BH. Por tanto, se puede "sustituir una BH" por una función insertada en esta cola de tareas. (3) tp_scheduler, se ejecuta desde el scheduler antes de modificar el estado de una tarea. (4) tq_disk, utilizada por los drivers de dispositivos cada vez que el VFS (Virtual File System) espera leer algún buffer.
- Las colas de tareas son listas enlazadas de estructuras del tipo tq_struct (include/linux/tqueue.h). Y las funciones más importantes que se pueden aplicar sobre estas colas de tareas son: (1) queue_task(bh_pointer, bh_list), inserta en la cabeza de la lista bh_list la estructura bh_pointer que contiene la dirección de la rutina que queremos que se ejecute en el futuro. (2) run_task_queue(list), recorre la lista list y ejecuta todas las rutinas que hay encoladas: Más en concreto, se extraen todos los datos de las estructuras, se pasa al siguiente elemento de la lista, se marca como libre dicho registro y se llama a la función.

2.8.4. Colas de Espera.

- Una cola de espera es una lista enlazada y circular de descriptores de procesos que se encolan a la espera de algún recurso o evento. Cada descriptor de proceso contiene la dirección de un descriptor de proceso así como un puntero al siguiente elemento en la cola.
- Es importante no confundir las colas de espera con las colas de tareas. **Colas de Tareas** (task queues): (1) utilizadas para postponer la ejecución de una función concreta del *kernel*, (2) utilizadas normalmente desde los manejadores de interrupciones. **Colas de Espera** (wait queues): (1) utilizadas para suspender la ejecución de un proceso hasta que se produzca una cierta condición, (2) utilizadas normalmente en el código de las llamadas al sistema.
- Veamos primero cómo se implementan las colas de espera. En el fichero include/linux/wait.h está la declaración de la estructura de datos struct wait_queue[include/linux/wait.h] con la que se construye la lista. (La estructura *task_struct* es la Tabla de Procesos, donde se guarda toda la información relacionada con cada proceso). Las funciones para (insertar y extraer) elementos de la lista son add_wait_queue(p,wait) y remove_wait_queue(p,wait), y están en include/linux/sched.h.

- La función sleep_on, otras funciones similares y las macros comunes que contienen están en kernel/sched.c y se encargan de suspender la ejecución de procesos. El esquema general lo podemos resumir en los siguientes pasos:
  - Se declara una estructura del tipo wait_queue
  - Se marca como no activo este proceso.
  - Se inserta en la cola correspondiente.
  - Se llama al planificador (función schedule y dentro de schedule se llama a la función switch_to que cambia los registros del procesador por los de otro proceso (cambio de contexto)), para elegir a otro proceso a ejecutar, por lo que la llamada NO RETORNARÁ hasta que vuelva a ponerse en ejecución el proceso actual.
- La función wake_up(p) es una macro que llama a __wake_up(p,mode) que está en kernel/sched.c, que a su vez recorre la lista y despierta a todos los procesos que hay esperando en ella.
  - En __wake_up se maneja la exclusión mutua y se recorre la lista, llamando para cada proceso a wake_up_process
  - En wake_up_process(p) se pone el proceso en estado activo (TASK_RUNNING), se inserta en la lista de procesos activos y se llama a reschedule_idle
  - En reschedule_idle se llama a reschedule_idle_slow, que, llamando a preemption_goodness todas ellas en kernel/sched.c, comprueba si el proceso que despertamos tiene más "prioridad" (goodness) que el que hay en ejecución y, si esto es así, pone a 1 la variable need_resched del procesador actual (esta variable se consulta antes de volver a código de usuario y si su valor es distinto de cero se llama a la función schedule).
- Como acabamos de ver, dentro de sleep_on se queda el proceso bloqueado al llamar a schedule. Cuando el proceso vuelve a ponerse en ejecución, seguirá por donde se quedó (sus registros, incluyendo el contador de programa, se repondrán en la CPU tal y como estaban). Por tanto, ejecutará el código de la macro SLEEP_ON_TAIL, que simplemente llama a remove_wait_queue, que ya hemos visto antes. En definitiva, cuando el proceso se pone en ejecución, él mismo se elimina de la cola de espera en la que estaba. Es importante destacar que sólo cuando otro proceso o una rutina de servicio de una interrupción llame a wake_up podrá continuar la ejecución el proceso que hizo sleep_on.

2.8.5. Semáforos.

- Los semáforos constituyen un mecanismo general de sincronización de procesos además de ser una herramienta para la programación concurrente. Un semáforo no es realmente un mecanismo de comunicación, sino más bien un mecanismo de sincronización de procesos (permite a los procesos compartir recursos (secciones críticas) y sincronizarse). En informática, y más particularmente en el caso de los sistemas operativos, un semáforo se utiliza para controlar el acceso a un recurso. Con los semáforos se pueden construir secciones críticas para proteger datos compartidos, o utilizarlos como mecanismos de sincronización.
- El semáforo y las operaciones relacionadas con él fueron definidas por Dijkstra. Un semáforo comprende un *contador* (un valor entero, para contabilizar el número de procesos bloqueados) y una *cola* (para bloquear procesos, éstas son colas de espera), junto con dos operaciones:
  - **P** (del holandés *Proberen*, probar): Esta operación se utiliza cuando un proceso quiere entrar en una sección crítica (acceder a un recurso) y realiza las siguientes etapas:
    + *Etapa 1*: Probar el valor del contador del semáforo que controla el recurso.
    + *Etapa 2*: Si el valor es positivo, el proceso puede servirse del recurso, y decrementa el valor en 1 para indicar que utiliza una unidad del recurso.
    + *Etapa 3*: Si el valor es nulo, el proceso se duerme hasta que el valor sea de nuevo positivo. Cuando el proceso se despierta vuelve a la *Etapa 1*.
  - **V** (del holandés *Verhogen*, incrementar): Esta operación es simétrica a la anterior, y se utiliza cuando un proceso abandona la sección crítica (liberar el recurso). El valor del contador del semáforo se incrementa y los procesos en espera son despertados.

  - Es claro que si el semáforo tiene asociado una cola (anteriormente hemos supuesto que no tiene asociado ninguna cola, solo el contador), las operaciones son sensiblemente diferentes:
    + **P** (**down**). De forma atómica, se decrementa el contador y, si el resultado es negativo (menor que 0), entonces se suspende el proceso y se inserta en la cola.
    + **V** (**up**). De forma atómica, se incrementa el contador y, si el resultado es menor que 1 (nulo), entonces se despierta a uno de los procesos de la cola.
- Las operaciones **P** y **V** deben de realizarse de manera atómica, es decir, que no deben ser interrumpidas. Las operaciones **P** y **V** se denominan frecuentemente **down** y **up**. Los semáforos se utilizan en el ámbito de los sistemas informáticos principalmente para resolver dos problemas:
  - *La exclusión mutua*: se trata de impedir a los procesos el acceso a un mismo recurso en un mismo instante. Si esto se produjera, el recurso podría encontrarse en un estado de inconsistencia.
  - *El problema de los productores/consumidores*: se trata de permitir la cooperación de dos procesos, uno produce información que el otro utilizará (consumirá). El semáforo se utiliza para prevenir o indicar al consumidor que los datos están a punto.
- Estos problemas se resuelven mediante semáforos binarios. Pero el contador del semáforo puede tomar otros valores positivos. Es el caso, cuando varias unidades del mismo recurso están disponibles, el contador toma entonces el valor del número de unidades del recurso accesibles simultáneamente.
- En include/asm-i386/semaphore.h tenemos la declaración del tipo semáforo, estructura **semaphore**.

```
struct semaphore {
    atomic_t count;
    int waking;
    struct wait_queue *wait;
};
```

- El campo waking se emplea para determinar qué proceso de los bloqueados obtiene el semáforo cuando éste quede libre. Al tomar el semáforo (P): (1) se decrementa count, (2) si es menor que 1, entonces el proceso se marca como suspendido, se inserta en la cola de espera y cede la CPU. Al devolver el semáforo (V) (1) se incremental count, (2) se incremental el campo waking, (3) se despiertan TODOS los procesos bloqueados. De todos los procesos recien desbloqueados, el primero en ejecutarse será el de mayor prioridad: (1) este proceso podrá waking a 0 y considera que ha consguido el semáforo, (2) el resto de procesos, cuando les toque, comprobarán que waking está a 0 y se volverán a bloquear, (3) así, se deja al scheduler la tarea de decidir quien consigue el semáforo.
- La función up(sem) incrementa el contador del semáforo y en caso de ser negativo o cero llama a wakeup que llama a su vez a __up(sem), la cual incrementa en exclusión mutua el campo waking del semáforo (esto lo hace wake_one_more) y despierta a todos los procesos de la cola.

```
void up(struct semaphore *sem)
{
    ++sem->count;
    if (sem->count <= 0)
    {
            unsigned long flags;
            spin_lock_irqsave(&semaphore_wake_lock, flags);
            if (atomic_read(&sem->count) <= 0)
                    sem->waking++;
            spin_unlock_irqsave(&semaphore_wake_lock, flags);
            wake_up(&sem->wait);
    }
}
```

- La función down(sem) decrementa el contador y si el resultado es negativo entonces llama a __down_failed, que llamará a __down. Esta función está construida a base de macros, que al expandirse darían lugar a este código:

```
void __down(struct semaphore * sem)
{
    struct task_struct *tsk = current;
    struct wait_queue wait = { tsk, NULL };
    tsk->state = (TASK_UNINTERRUPTIBLE);
    add_wait_queue(&sem->wait, &wait);
    for (;;)
    {
        if (waking_non_zero(sem))
            break;
        schedule();
        tsk->state = (TASK_UNINTERRUPTIBLE);
    }
    tsk->state = TASK_RUNNING;
    remove_wait_queue(&sem->wait, &wait);
    it_queue(&sem->wait, &wait);
}
```

- Los pasos que se siguen son:
  - Se pone el proceso en estado bloqueado (Como veremos en el tema de planificación, en Linux el estado "bloqueado" presenta dos posibilidades: TASK_INTERRUPTIBLE o TASK_UNINTERRUPTIBLE, que indican respectivamente si una señal puede matar o no al proceso mientras está bloqueado).
  - Se inserta en la cola de espera del semáforo.
  - Se entra en un bucle para asegurar que sólo un proceso pasará el semáforo después de la operación *up*. Se hace uso del campo waking, llamando a la función waking_non_zero, que, de forma atómica, hace lo siguiente: (1) Si waking es mayor que cero le resta uno y devuelve un uno (cierto), con lo cual saldrá del bucle. (2) Si no, directamente devuelve un cero (falso), dando lugar a que se llame a schedule, que pondrá otro proceso en ejecución (el proceso actual sigue en estado "bloqueado" mientras a tsk->state no se le asigne TASK_RUNNING). La idea general es que el primer proceso en despertar sale del bucle y el resto, no. Cuál sea este proceso vendrá determinado por la política de planificación.
  - Se pone el proceso en estado activo.
  - Se quita de la cola del semáforo.

2.8.6. Algunos Aspectos de Implementación en la Sincronización de Procesos.

- El archivo fuente *kernel/sched.c* contiene funciones de servicio que permiten la sincronización de procesos en modo *kernel*. Estas funciones se utilizan en todas las partes del *kernel* cuando un proceso debe suspenderse en espera de un evento (dormir), y cuando debe ser "despertado".
- La función *add_to_runqueue* inserta un descriptor de proceso en la lista de procesos "Listos para ejecutarse". La función *del_from_runqueue* permite suprimir un descriptor de esta lista. Un descriptor de proceso puede colocarse al final de la lista utilizando la función *move_last_runqueue*.
- Las colas de espera *(wait queues)* son manipuladas por las funciones *__add_wait_queue* y *__remove_wait_queue*, declaradas en el archivo de cabecera *<linux/sched.h>*. Se añade un elemento a la cola de espera utilizando *__add_wait_queue*, y se suprime de dicha cola utilizando *__remove_wait_queue*. Las funciones *add_wait_queue* y *remove_wait_queue* llaman respectivamente a *__add_wait_queue* y *__remove_wait_queue* ocultando previamente todas las interrupciones (funciones atómicas).
- La función *wake_up_process* despierta un proceso suspendido (dormido), pone su estado a *TASK_RUNNING* (Listo para su ejecución) y lo inserta en la lista de procesos "Listos para ejecutarse".

- La función *wake_up* permite despertar todos los procesos en espera de un evento en una cola de espera. Explora la cola, y llama a *wake_up_process* para cada proceso cuyo estado sea *TASK_INTERRUPTIBLE* o *TASK_UNINTERRUPTIBLE*. Un tratamiento similar realiza la función *wake_up_interruptible*, pero esta última sólo despierta los procesos cuyo estado es *TASK_INTERRUPTIBLE*.
- La función *__sleep_on* suspende el proceso actual y lo coloca en una cola de espera, modifica el estado del proceso, guarda su descriptor en la cola de espera mediante una llamada a *add_wait_queue*, y provoca un cambio de proceso actual por una llamada a la función *schedule*. Cuando el proceso se despierta, el resto de la función se ejecuta, y el descriptor del proceso se suprime de la cola de espera por una llamada a *remove_wait_queue*. Las funciones *interruptible_sleep_on* y *sleep_on* llaman a *__sleep_on* especificándole que ponga el estado del proceso actual respectivamente a *TASK_INTERRUPTIBLE* (*interruptible_sleep_on*) y *TASK_UNINTERRUPTIBLE* (*sleep_on*).
- La función *__down* permite que el proceso actual se suspenda en espera de un evento sobre un semáforo, se añada el descriptor del proceso actual a la cola de espera del semáforo, su estado pasa al estado conocido como *TASK_UNINTERRUPTIBLE*, y se comprueba el contador del semáforo. Mientras este contador es igual a cero, la función *schedule* se llama para cambiar de proceso actual, y el estado del proceso se pone a *TASK_UNINTERRUPTIBLE*. Cuando el contador del semáforo pasa estrictamente a positivo, el estado del proceso actual se pone a *TASK_RUNNING* y el descriptor del proceso se suprime de la cola de espera del semáforo por una llamada a la función *remove_wait_queue* y se inserta en la lista de procesos "Listos para ejecutarse". El uso de los semáforos se efectúa mediante las funciones *down* y *up* definidas en el archivo de cabecera de Linux <*asm/semaphore.h*>.

2.9. SEÑALES Y FUNCIONES DE TIEMPO.

2.9.1. Concepto de Señal.

Las señales son interrupciones software que pueden ser enviadas a un proceso para informarle de algún evento asíncrono o situación especial. El termino señal se emplea también para referirse al evento. Las formas de generar una señal pueden ser las siguientes:

- Resultado de una excepción hardware. Por ejemplo, cuando un proceso escribe en una zona de memoria no asignada, entonces produce un acceso inválido a una página de memoria que provoca una excepción, que es capturada por el *kernel* generando la señal SIGSEGV hacia el proceso trasgresor.
- Resultado de pulsar Ctrl-C por parte del usuario del terminal. Genera la señal SIGINT, cuya acción predeterminada es terminar le proceso del primer plano de la sesión.
- Resultado de la llamada al sistema *kill* o de la orden del mismo nombre. Permite enviar una señal determinada a un proceso dado.
- Resultado de un evento gestionado por el *kernel*. Por ejemplo, SIGALRM es emitida por el sistema hacia el proceso que la ha solicitado cuando expira una alarma.

Las señales permiten informar a los procesos de la ocurrencia de eventos asíncronos. Los procesos pueden intercambiar señales con la llamada al sistema *kill* o el *kernel* puede enviar señales internamente a los procesos. Aunque existen diversas implementaciones (tipos de señales y la forma de manejarlas), Posix.1 introduce un poco de orden en este tema. También, es necesario diferenciar dos grandes aspectos en las señales: (1) La generación de la señal y, (2) el tratamiento de la señal. El primero se realiza cuando ocurre el evento que se desea señalizar (los eventos se identifican en general con un entero entre 0 y 31, dependiendo de la implementación, a cada entero se le suele asociar una macro que lo define). El segundo aspecto procede cuando el proceso (al que se le ha enviado la señal) reacciona ante ella (la reconoce y la procesa). Como veremos, para que el proceso atienda la señal debe estar en posesión de la CPU, lo cual puede tardar en ocurrir. Durante el tiempo que transcurre entre que la señal se envió y el proceso la reconoce, se dice que la señal está pendiente. El tiempo en que la señal está pendiente depende del estado en que se encuentre el proceso (puede estar bloqueado y puede ser interrumpido, puede estar suspendido, transferido al área de

intercambio *(swap)*, etc.). A la recepción de una señal por el proceso destinatario, éste puede desarrollar alguna de las siguientes acciones (las 5 primeras se especifican como acción por defecto):

- *abort* ⇒ el proceso finaliza y se genera un *core-dump* (volcado de memoria) que se almacena en el archivo *core*. Este volcado puede ser utilizado posteriormente para depurar (seguir la traza) el proceso.
- *exit* ⇒ finalización sin generar volcado de memoria.
- *ignore* ⇒ no procesar la señal.
- *stop* ⇒ suspender el proceso (retirarlo de la competición por la CPU).
- *continue* ⇒ reiniciar el proceso si había sido suspendido o ignorar la señal si no lo fue.
- *procesar* ⇒ la señal indicando una función de usuario para su tratamiento. Esta alternativa puede ser utilizada para todas las señales salvo SIGKILL y SIGSTOP.

Los procesos pueden enviarse señales unos a otros a través de la llamada *kill* y es bastante frecuente que, durante su ejecución, un proceso reciba señales procedentes del *kernel*. Cuando un proceso recibe una señal puede tratarla de tres formas diferentes:

- Ignorar la señal, con lo cual es inmune a la misma.
- Invocar o llamar a la rutina de tratamiento por defecto. Esta rutina no la codifica el programador, sino que la aporta el *kernel*. Según el tipo de señal, la rutina de tratamiento por defecto va a realizar una acción u otra. Por lo general suele provocar la terminación del proceso mediante una llamada a *exit*. Algunas señales no sólo provocan la terminación del proceso, sino que además hacen que el *kernel* genere en el directorio de trabajo actual (CWD) del proceso un archivo llamado *core* que contiene un volcado de memoria del contexto del proceso. Este archivo *core* podrá ser examinado con ayuda del programa depurador (*adb, sdb o gdb*) para determinar que señal provocó la terminación del proceso y en qué punto exacto de su ejecución se produjo. Este mecanismo es muy útil a la hora de depurar programas que contienen errores de manejo de números en coma flotante, instrucciones ilegales, acceso a directorios fuera de rango, etc.
- Invocar o llamar a una rutina propia que se encarga de tratar la señal. Esta rutina es invocada por el *kernel* en el supuesto de que esté montada y será responsabilidad del programador codificarla para que tome las acciones pertinentes como tratamiento de la señal (handler). En estos casos, el programa no va a terminar a menos que la rutina de tratamiento indique lo contrario.

En la Figura 2.15 vemos esquematizada la evolución temporal de un proceso y cómo a lo largo de su ejecución, recibe varias señales procedente del *kernel.*

- La primera señal que recibe no provoca que el proceso cambie el curso de su ejecución, esto es debido a que la acción que está activa es que el proceso ignore la señal.
- El proceso prosigue su ejecución y recibe una segunda señal que le fuerza a entrar en una rutina de tratamiento (handler, función de desviación de la señal que el proceso receptor puede asociar a dicha señal para que realice la acción predeterminada). Esta rutina, después de tratar la señal, puede optar por tres acciones:
  - Restaurar la ejecución del proceso al punto donde se produjo la interrupción.
  - Finalizar el proceso.
  - Restaurar algunos de los estados pasados del proceso (punto de *fallback*) y continuar la ejecución desde ese punto (funciones *setjmp* y *longjmp*).
- El proceso puede también recibir una señal que le fuerce a entrar en la rutina de tratamiento por defecto.

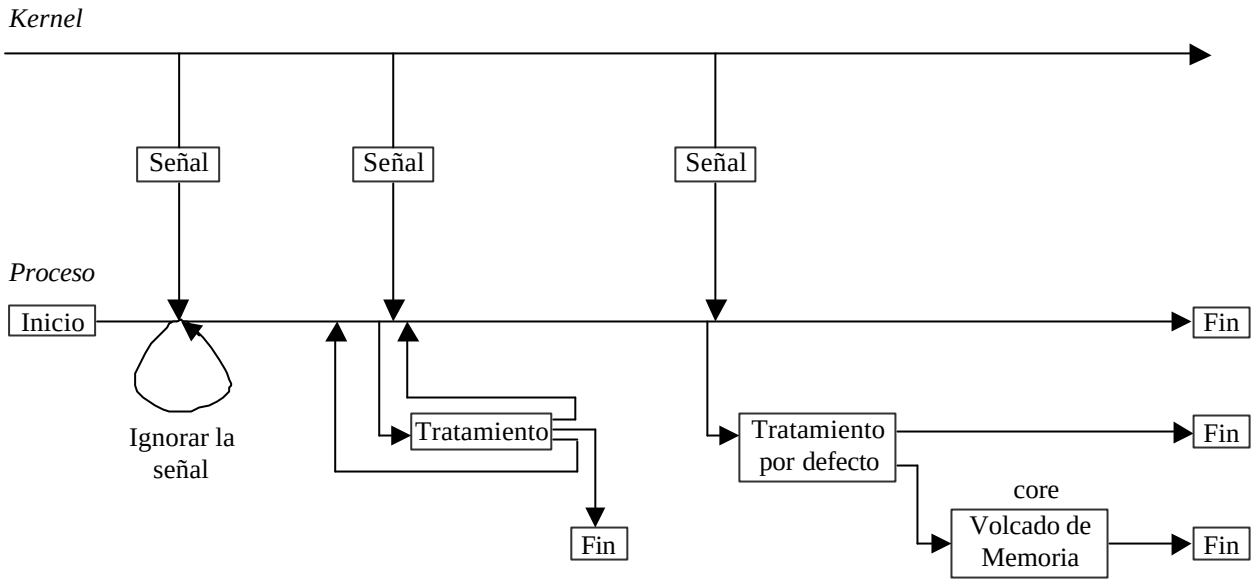

Figura 2.15. Tratamiento de las señales recibidas por un proceso.

Saltos globales. *setjmp(env)* ⇒ función estándar que salva el entorno de la pila en la variable que se le pasa como parámetro *(env)* para un uso posterior de *longjmp*, con el objetivo de hacer que el proceso vuelva a alguno de los estados por los que ha pasado con anterioridad. *setjmp* devuelve el valor 0 en su primera llamada. *longjmp(env, val)* ⇒ función estándar que restaura el entorno guardado en la variable pasada como parámetro a *setjmp(env)*. Después de haberse ejecutado la llamada a *longjmp*, el flujo de la ejecución del programa vuelve al punto donde se hizo la llamada a *setjmp*, pero en este caso *setjmp* devuelve el valor *val* que se pasó como parámetro a *longjmp*. Esta es la forma de averiguar si *setjmp* está saliendo de una llamada para guardar el entorno o de una llamada a *longjmp*. *longjmp* no puede hacer que *setjmp* devuelva 0, ya que en el caso de que *val* valga 0, *setjmp* va a devolver 1. Las funciones *setjmp* y *longjmp* se pueden ver como una forma elaborada de implementar una sentencia *goto* capaz de saltar desde una función a etiquetas que están en la misma o en otra función. Las etiquetas serán los entornos guardados por *setjmp* en la variable pasada como parámetro *(env)*.

2.9.2. Tipos de Señales.

Cada señal (SIGXXXX) tiene asociado un número entero positivo, que es el número intercambiado cuando uno de los procesos envía una señal a otro. En el UNIX System V hay definidas 19 señales, en el 4.3BSD, 30 y en Linux en su versión 2.0 tenía definidas 34. Las 19 señales del UNIX System V las tienen prácticamente todas las versiones de UNIX, y a éstas cada fabricante le incorporan las que considere necesarias. Podemos clasificar a las señales en los siguientes grupos:

- Señales relacionadas con la terminación de procesos.
- Señales relacionadas con las excepciones inducidas por los procesos. Por ejemplo, el intento de acceder fuera del espacio de direcciones virtuales, los errores producidos al manejar números en coma flotante, etc.
- Señales relacionadas con los errores irrecuperables originados en el trascurso de una llamada al sistema.
- Señales originadas desde un proceso que se está ejecutando en modo usuario. Por ejemplo, cuando un proceso envía una señal a otro proceso vía *kill*, cuando un proceso activa un temporizador y se queda a la espera de la señal de alarma, etc.
- Señales relacionadas con la interacción con el terminal. Por ejemplo, pulsar la tecla *break*.
- Señales para ejecutar un programa paso a paso, son utilizadas por los depuradores.
- Señales vinculadas con la acción en la que proceso ha excedido los límites de CPU o de espacio en disco que tiene permitido.
- Señales con las que el mismo proceso solicita (relacionadas con la ocurrencia de un evento).

- Señales vinculadas con la acción en la que el proceso solicita avisos asociados a una cantidad de tiempo transcurrida (alarmas), o cuando un proceso, con capacidad de señalizar a otro, le envía una señal.
- Señales de control de procesos, generadas por proceso ejecutando en el fondo (background) y que intentan realizar operaciones asociadas al terminal.

  En el archivo de cabera *<signal.h>* están definidas las señales que puede manejar el sistema y sus nombres. Las 19 señales de UNIX System V son:
- *SIGHUP*. *Hangup*. Esta señal es enviada cuando un terminal se desconecta de todo proceso del que es terminal de control. También se envía a todos los procesos de un grupo cuando el líder del grupo termina su ejecución. La acción por defecto de esta señal es terminar la ejecución del proceso que la recibe.
- *SIGINT*. *Interrupción*. Se envía a todo proceso asociado con un terminal de control cuando se pulsa la tecla de interrupción. La acción por defecto de esta señal es terminar la ejecución del proceso que la recibe.
- *SIGQUIT*. *Salir*. Similar a *SIGINT*, pero es generada al pulsar la tecla de salida. Su acción por defecto es generar un archivo *core* y terminar el proceso.
- *SIGILL*. *Instrucción ilegal.* Es enviada cuando el hardware detecta una instrucción ilegal. En los programas escritos en C suelen producir este error cuando manejamos punteros a funciones que no han sido correctamente inicializados. Su acción por defecto es generar un archivo *core* y terminar el proceso.
- *SIGTRAP*. *Trace trap*. Es enviada después de ejecutar cada instrucción, cuando el proceso se está ejecutando paso a paso. Su acción por defecto es generar un archivo *core* y terminar el proceso.
- *SIGIOT. I/O trap instruction*. Se envía cuando se da un fallo de hardware. La naturaleza de este fallo depende de la máquina. Es enviada cuando llamamos a la función *abort*, que provoca el suicidio del proceso generando un archivo *core*.
- *SIGEMT. Emulator trap instruction*. También indica un fallo de hardware. Raras veces se utiliza. Su acción por defecto es generar un archivo *core* y terminar el proceso.
- *SIGFPE*. *Error en coma flotante*. Es enviada cuando el hardware detecta un error en coma flotante, como el uso de un número en como flotante con un formato desconocido, errores de *overflow* o *underflow*, etc. Su acción por defecto es generar un archivo *core* y terminar el proceso.
- *SIGKILL*. *Kill*. Esta señal provoca irremediablemente la terminación del proceso. No puede ser ignorada ni modificarse y siempre que se recibe se ejecuta su acción por defecto, que consiste en generar un archivo *core* y terminar el proceso. Esta característica permite al superusuario poder interrumpir o suspender la ejecución de todo proceso.
- *SIGBUS*. *Bus error*. Se produce cuando se da un error de acceso a memoria. Las dos situaciones típicas que las provocan suelen ser intentar acceder a una dirección que físicamente no existe o intentar acceder a una dirección impar, violando así las reglas de alineación que impone el hardware. Su acción por defecto es generar un archivo *core* y terminar el proceso.
- *SIGSEGV*. *Violación de segmento*. Esta señal es enviada a un proceso cuando intenta acceder a datos que se encuentran fuera de su segmento de datos. Su acción por defecto es generar un archivo *core* y terminar el proceso.
- *SIGSYS*. *Argumento erróneo en una llamada al sistema*. No se utiliza.
- *SIGPIPE*. Intento de escritura en una tubería (pipe) de la que no hay nadie leyendo. Esto suele ocurrir cuando el proceso de lectura termina de una forma anormal. Su acción por defecto es terminar el proceso.
- *SIGALRM*. *Alarm clock*. Esta señal es enviada a un proceso cuando alguno de sus temporizadores descendentes llega a cero. Su acción por defecto es terminar el proceso.
- *SIGTERM*. *Finalización software*. Es la señal utilizada para indicarle a un proceso que debe terminar su ejecución. Esta señal no es tan tajante como *SIGKILL* y puede ser ignorada. Lo correcto es que la rutina de tratamiento de esta señal se encargue de tomar las acciones necesarias antes de terminar un proceso (como por ejemplo, borrar los archivos temporales) y llame a la rutina *exit*. Esta señal es enviada a todos los procesos durante el *shutdown* o parada del sistema. Su acción por defecto es terminar el proceso.

- *SIGUSR1*. *Señal número 1 de usuario*. Esta señal está reservada para uso del programador. Ninguna aplicación estándar va a utilizarla y su significado es el que le quiera dar el programador en su aplicación. Su acción por defecto es terminar el proceso.
- *SIGUSR2*. *Señal número 2 de usuario*. Su significado es idéntico al de *SIGUSR1*.
- *SIGCLD* o *SIGCHLD*. *Muerte del proceso hijo*. Esta señal es enviada al proceso padre cuando alguno de sus procesos hijos termina. Esta señal es ignorada por defecto.
- *SIGPWR*. *Fallo de alimentación*. Esta señal tiene diferentes interpretaciones. En algunos sistemas es enviada cuando se detecta un fallo de alimentación y le indica al proceso que dispone tan solo de unos instantes de tiempo antes de que se produzca una caída del sistema. En otros sistemas, esta señal es enviada, después de recuperarse de un fallo de alimentación, a todos aquellos procesos que estaban en ejecución y que se han podido rearrancar. En estos casos, los procesos deben disponer de mecanismos para restaurar las posibles pérdidas producidas durante la caída de la alimentación.

2.9.3. Tratamiento de las Señales.

- El tratamiento de las señales debe cubrir varias etapas: (1) cómo envía el kernel señales a los procesos, (2) cómo manejan las señales los procesos y (3) cómo reacciona el proceso a las señales. Para la primera etapa, el *kernel* activa el bit de señal y activa la entrada del proceso en la Tabla de Procesos (de acuerdo al tipo de señal). Si el proceso esta dormido a una prioridad interrumpible el *kernel* lo despierta, con lo cual finaliza el trabajo de envío de la señal. Observe que el proceso puede "recordar" varias señales a la vez pero no cuantas instancias de la misma señal recibe.
- El *kernel* controla la existencia de una señal cuando el proceso está regresando de modo *kernel* a modo usuario (final de una llamada al sistema), cuando está entrando a Ejecución en modo *kernel* (desde listo pasa a Ejecución modo *kernel*, o desde Listo-Requisado entra a Ejecución modo usuario) y cuando entra al estado de bloqueado (desde Ejecución modo *kernel*). Obsérvese que la señal no tiene efecto instantáneo si el proceso está en modo *kernel*, y si el proceso está en Ejecución en modo usuario y el *kernel* ha ejecutado una interrupción que envía una señal al proceso el *kernel* reconocerá y manejará la señal cuando esté regresando de la interrupción. Es decir, un proceso no ejecuta en modo usuario el manejo de la señal.

```
algoritmo issig                                   // chequeo de recepción de señales
{
    while(bit de señal en la entrada del proceso en la Tabla de Procesos != 0)
    {
        Buscar el número de señal enviada al proceso;
        if (señal es SIGCHLD)
        {
            if (ignorar SIGCHLD)
                Liberar las entradas de los procesos hijos Zombies en la Tabla de Procesos;
            else
                Return TRUE;
        }
        else
        {
            if (no ignorar señal)
                Return TRUE;
        }
    Desactivar bit de señal en la entrada del proceso en la Tabla de Procesos;
    Return FALSE;
}
```

- El algoritmo *issig* se usa para detectar la existencia de señales a procesar (el proceso puede elegir ignorar las señales vía la llamada al sistema *signal*, en cuyo caso desactiva el bit de señal).
- El *kernel* maneja las señales en el contexto del proceso que recibe la señal, por lo tanto el proceso debe estar ejecutando. Se presentan tres situaciones en el manejo de la señal: el proceso finaliza vía *exit*, el proceso ignora la señal o el proceso ejecuta una función de usuario para manejarla. La acción por defecto suele ser llamar a *exit* en modo *kernel*. El uso de acciones específicas se indica mediante *funcion_Old = signal(signum, funcion_New)*, donde *signum* es el número de la señal y funciónNew es la dirección de la función de usuario que se invocará para esta señal. Es posible pasar 0/1 en lugar de la dirección indicando que se ignoren las ocurrencias de esta señal (1) o que se ejecute *exit* al recibir la señal (si el parámetro es 0). En la u-Area se dispondrá de un array con las acciones a tomar para cada señal en el cual el *kernel* almacenará la dirección de la función a ejecutar para cada señal.

```
algoritmo psig                          // manejo de la señal después de reconocerla
{
    Identificar la señal;
    Borrar indicación de señal en la entrada del proceso en la Tabla de Procesos;
    if (se ha indicado ignorar esta señal)
        exit;
    if (se ha especificado una función para manejar esta señal)
    {
        Obtener la dirección del manejador indicado (en la uArea);
        Borrar la dirección del manejador en la uArea;
        Modificar el contexto-de-usuario: crear en la pila un marco artificial para simular la llamada a la
            función que manejará la señal;
        Modificar el contexto-de-sistema escribir la dirección del manejador de la señal en el registro
            PC (Contador de Programa) del contexto salvado;
        Return;
    }
    if (señal involucra un volcado de memoria a disco)
    {
        Crear un archivo core en en directorio actual (CurDir);
        Escribir el contenido del contexto-de-usuario en el archivo;
    }
    exit;
}
```

- Cuando se maneja la señal el *kernel* la identifica y desactiva el bit de indicación de la señal (que se activó al recibir la señal). Si la función de manejo de la señal está a su valor por defecto, el *kernel* puede hacer un volcado de memoria (al archivo *core*) para determinados tipos de señales (en general aquellas que indiquen mal funcionamiento del proceso, a efecto de depuración de programas. Por ejemplo a la señal Ctrl-C no le sigue un volcado porque no ha ocurrido nada anormal en el programa solo que se quiere terminar prematuramente).
- Si el proceso ha recibido una señal que ha decidido ignorar el proceso continúa como si la señal no hubiera llegado. En el caso que el proceso haya especificado que él va a manejar la señal, el *kernel* hace lo siguiente (antes de pasarle el control a la función de usuario): El *kernel* obtiene del contexto de usuario que ha salvado (al entrar a la llamada al sistema) y toma de él el Contador de Programa (PC) y el puntero a la pila (SP) (normalmente limpia el campo del manejador de la señal en la u-Area y lo coloca al valor por defecto). A continuación crea una nueva capa en la pila del usuario, escribiendo los valores del PC y SP (la pila de usuario aparecerá como si el proceso hubiera llamado a una función cualquiera, el manejador de la señal, en el punto que se llamó a la llamada al sistema o donde el *kernel* reconoció la señal. Para acabar el *kernel* cambia el contexto de registros que salvó, modifica el PC para que apunte a la función de manejo y modifica el valor del puntero a la pila de usuario (ha crecido con la nueva capa). Cuando el proceso pase de *kernel* a modo usuario ejecutará su manejador de señal y luego regresará al punto en que se encontraba.

- Observemos que es necesario borrar la dirección del manejador de señal de usuario de la u-Area (si el proceso quiere volverla a usar debe volver a usar *signal* (esto se hace porque el código de usuario ejecuta en modo usuario y podría llegar otra vez la misma señal e invocar el mismo código mientras se está manejando la previa), pero queda un problema que es ... ¿qué hacer si llega otra vez la misma señal antes que el proceso usuario haya restaurado su manejador de señal?. Cierto es que la estrategia aparece apropiada sólo para lo que fue diseñada inicialmente (señales equivalían a situaciones que eran fatales o que debían ser ignoradas). Ahora haría falta una llamada al sistema que permitiera bloquear la recepción de señales de un determinado tipo y que en el desbloqueo el *kernel* enviara las señales pendientes.
- Otra anomalía se puede observar en el manejo de señales que ocurren cuando el proceso duerme en un nivel interrumpible, La señal provoca que el proceso haga un *longjmp* fuera de su *sleep*, regrese a modo usuario y llame al manejador de señal. Cuando el manejador de señal finaliza, el proceso pareciera regresar de una llamada al sistema con un error, indicando que la llamada al sistema ha sido interrumpida. El usuario puede controlar el error en la llamada al sistema y relanzarla, pero no siempre lo hace, así que puede ser mejor que sea el *kernel* quien vuelva a lanzar la llamada al sistema en esta situación.
- También se presentan anomalías cuando un proceso ignora una señal. Si la señal llega cuando el proceso está bloqueado a un nivel interrumpible, el proceso será desbloqueado pero no hará un *longjmp*. Esto es, el *kernel* solo se da cuenta que el proceso ignora la señal cuando lo ha despertado y ejecutado. Una práctica más consistente sería dejar al proceso bloqueado. El problema es que la información sobre ignorar la señal se encuentra en la u-Area que no es accesible cuando llega una señal a un proceso que no está ejecutando (la mayor parte de las veces). La solución de este punto pasa por incluir en la entrada del proceso en la Tabla de Procesos las acciones ante las señales de forma que el *kernel* pudiera controlarlas en la recepción de las mismas. Otra alternativa es volver a dormirse en el algoritmo *sleep* si descubre que no debió ser despertado.
- Finalmente, la señal SIGCHLD tiene el efecto de despertar a un proceso bloqueado a una prioridad interrumpible y recibe un tratamiento especial. En particular, cuando un proceso la recibe desactiva el indicador de señal en la entrada del proceso en la Tabla de Procesos y procede de la siguiente forma:
  - Si se mantiene la acción por defecto, entonces actúa como si la señal no se hubiera recibido.
  - Si el proceso ha especificado un manejador de señal, entonces él invoca a su manejador de señales como en los otros casos.
  - Si el *kernel* debe ignorar la señal, entonces se revisará en la llamada al sistema *wait*.
  - Si un proceso invoca la llamada al sistema *signal* con parámetro SIGCHLD, entonces el *kernel* envía al mismo proceso una señal con *kill(pid, SIGCHLD)* si este proceso tiene algún hijo en estado Zombie (este caso se revisará también en la llamada al sistema *wait*).

2.9.4. Descriptores de Señales.

Ahora vamos a indicar un estudio comparativo de la interfaz de manejo de señales que brindan UNIX System V, 4.3 BSD y el estándar POSIX.1 (norma IEEE POSIX 1003.1-1988) (Linux sigue este último estándar pero incorpora además otras funciones).

Descripción	**System V**	**4.3 BSD**	**POSIX**
Tratamiento de señales (1).	signal()	signal()	signal()
Envío de señales.	kill() o raise()	kill() o raise()	kill() o raise()
En espera de señales.	pause()	sigpause()	sigsuspend() o pause()
Tratamiento de señales (2).	signal()	sigvector() o sigvec()	sigaction() *<desvío de señales>*
Protección de zonas críticas (1).		sigblock()	sigprogmask() o sigpending() *<bloqueo de señales>*
Protección de zonas críticas (2).		sigsetmask()	sigprogmask() o sigpending() *<bloqueo de señales>*
Tratamiento de stack de señales.		sigstack()	
Interrupción de las llamadas al sistema.		siginterrupt()	siginterrupt()
Gestión de alarmas.	alarm()	alarm()	alarm()

2.9.5. Funciones de Tiempo.

En este apartado vamos a dar una introducción al estudio de cómo consultar y controlar los tiempos asociados a un proceso. Si bien UNIX, no es un sistema para aplicaciones en tiempo real, en condiciones de baja carga donde no está congestionado podremos imponerle a nuestros programas que se ejecuten con los tiempos de respuesta adecuados. Hay que tener en cuenta que una respuesta en tiempo real no significa solamente que nuestro programa se ejecute muy rápido (para simular procesos físicos), sino también que seamos capaces de controlar el tiempo para dar la respuesta en el momento exacto y no antes (por ejemplo, un programa para simular un reloj, en donde nuestra respuesta se tiene que dar cada segundo, de nada nos vale darla más deprisa).

A continuación se van a enumerar brevemente los mecanismos para dotar a nuestros programas de una temporización que va a estar impuesta por la naturaleza física del proceso que intentamos simular.

- Lectura de la fecha del sistema. **stime()**.
- Fijar la fecha del sistema. **time()**, **gettimeofday()** y **settimeofday()**.
- Tiempos de ejecución asociados a un proceso. **times()**.
- Temporizadores. **alarm()**, **getitimer()**, **setitimer()** y **localtime()**.

2.9.6. Señales en Linux.

La gestión de señales es un mecanismo existente desde las primeras versiones de UNIX. Permite a los procesos reaccionar a los eventos provocados por ellos mismos o por otros procesos, siendo este mecanismo similar a la gestión de interrupciones lógicas. La semántica de las llamadas al sistema que gestionan las señales en Linux es la de la norma POSIX. Sin embargo, la gestión de las señales existía en los sistemas UNIX mucho antes de la definición de esta norma, por lo que existe un cierto número de llamadas al sistema no POSIX definidas a este efecto. Estas llamadas al sistema aparecieron en las dos clases de sistemas UNIX, System V y BSD. En ocasiones tienen los mismos nombres pero semánticas diferentes. A fin de ser lo más completo posible y permitir la portabilidad de un mayor número de programas, Linux implementas diferentes semánticas. En principio, se utiliza la semántica de System V pero la inclusión del archivo de cabecera *bsd/signal.h* en lugar de *signal.h* en el momento de la compilación de programas y ocasionalmente la inclusión de la librería *libbsd.a* en el momento del linkado (enlazado) les permite respetar la semántica de los sistemas BSD.

Emisión de una señal a un proceso o a un grupo de procesos.
int kill(pid_t pid, int sig);
int killpg(int pgrp, int sig);
int raise(int sig); Esta función es equivalente a *kill(getpid(), sig)*;

Desviación de una señal, modificando el comportamiento predeterminado de una señal.
*void (*signal(int signum, void (*handler) (int))) (int)*;

Espera de una señal, suspendiendo el proceso que llama hasta la llegada de una señal cualquiera.
int pause(void);

Los grupos de señales. Es posible suspender o bloquear varias señales simultáneamente, siendo necesario que las señales sean gesionadas por grupo
*int sigemptyset(sigset_t *set)*;
*int sigfillset(sigset_t *set)*;
*int sigaddset(sigset_t *set, conts int signum)*;
*int sigdelset(sigset_t *set, conts int signum)*;
*int sigismember(conts sigset_t *set, conts int signum)*;

Interrupción de las llamadas al sistema.
int siginterrupt(int sig, int flag);

Bloqueo de las señales. Un proceso puede controlar la llegada de las diferentes señales, pudiendo retrasar su llegada bloqueándolas (se dice entonces que están pendientes). Cada proceso posee un conjunto llamado máscara de señales, que contiene la lista de las señales bloqueadas. Este conjunto puede modificarse mediante la llamada al sistema *sigprocmask*.
*int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)*;
El proceso puede solicitar al sistema su lista de señales pendientes por la llamada al sistema *sigpending* cuyo prototipo es el siguiente:
*int sigpending(sigset_t *set)*;

Desvío de señales. Control del comportamiento del proceso en la recepción de una señal.
*int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)*;

Espera de señales. La llamada al sistema pause no permite esperar una señal particular, el proceso se suspende hasta que llegue cualquier señal. La norma POSIX ha regulado este problema con la introducción de la llamada al sistema *sigsuspend*, que reemplaza temporalmente la máscara de señales del proceso por una máscara de espera y suspende el proceso hasta la llegada de una señal que no pertenezca a la máscara de espera.
*int sigsuspend(const sigset_t *mask)*;

Gestión de las alarmas. Un proceso puede solicitar al sistema que le envíe una señal en un tiempo dado, utilizando para ello la llamada al sistema *alarm.*
long alarm(long seconds);
La llamada al sistema *alarm* es limitada, pues no permite gestionar automáticamente una alarma. La gestión del tiempo es únicamente relativa al tiempo real, pero puede ser interesante gestionar alarmas relativas al tiempo de ejecución del proceso. Los sistemas BSD y System V han resuelto estos problemas introduciendo dos nuevas llamadas al sistema que permiten gestionar automáticamente alarmas periódicas con referencias de tiempo diferentes.
*int getitimer(int which, struct itimerval *value)*;
*int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue)*;

La señal SIGCHLD. Cuando un proceso termina, queda en estado Zombie hasta que un proceso padre tenga conocimiento de su estado de finalización por una de las llamadas al sistema *wait* o *wait4*. El proceso padre es avisado de la terminación de uno de sus hijos por el sistema, que le envíala señal SIGCHLD. La señal se activa únicamente cuando el proceso padre realizar una de las llamadas al sistema citadas anteriormente.
En principio, bajo Linux la señal SIGCHLD se pone a SIG_IGN y el sistema no previene al padre de la terminación de un hijo; el sistema realiza automáticamente la terminación del proceso hijo y éste desaparece de la Tabla de Procesos.
La consideración de la señal SIGCHLD es importante en la escritura de aplicaciones cliente/servidor. En general, un servidor maestro controla la llegada de peticiones y crea un servidor esclavo que responde a ellas. Tras la petición, el servidor esclavo se termina y el servidor maestro verifica que no haya habido problemas por la lectura de su código de terminación. Es necesario leer el estado de terminación, si no existe un riesgo de saturación de la Tabla de Procesos por proceso en estado Zombie.
La información respecto a la gestión de las señales para un proceso se almacenan en la estructura de datos *task_struct* (descriptor de proceso). Los campos de esta estructura relacionados con las señales son los siguientes:
signal: señales en espera, expresadas en forma de cadena de bits.
blocked: señales ocultas, expresadas en forma de cadena de bits.
exit_signal: número de la señal que ha causado la finalización del proceso.
sig: tabla que contiene la dirección de las funciones de desvío de las señales.
La estructura *signal_struct*, que define el tipo de *sig*, se declara en el archivo *linux/sched.h*. Contiene los dos campos siguientes:
int count: contador utilizado por la llamada al sistema clone para referenciar el número de proceso que apuntan a la estructura.
struct sigaction[32] action: tabla de funciones de desvío.

2.10. NOCIONES BÁSICAS DE PLANIFICACIÓN DE PROCESOS EN UNIX.

Planificador de procesos (scheduler) UNIX: tipo *Round Robin con realimentación multinivel* (múltiples colas en un sistema multiusuario).

2.10.1. Algoritmo de Planificación.

- El *kernel* requisa un proceso (restaura contexto) ⇒ cambio de contexto: asignar la CPU a otro proceso.
- Elegir un nuevo proceso ⇒ algoritmo para planificar el mejor proceso: mayor prioridad en la lista de descriptores de procesos "Listos para ejecutarse".
  - Si hay varios procesos en la lista ⇒ se elige el proceso que haya estado "Listo para su ejecución" más tiempo (más tiempo esperando en la lista de descriptores de procesos "Listos para ejecutarse").
  - Si no hay ningún proceso listo para su ejecución:
    + El procesador (CPU) está en estado ocioso (idle) hasta la siguiente interrupción.
    + Después de manejar la interrupción ⇒ el *kernel* busca el proceso para ejecutar.

2.10.2. Parámetros de Planificación.

- Entrada en la tabla de procesos ⇒ campo de *prioridad* para planificar procesos.
- Valor numérico bajo en este campo, indica una prioridad alta.
- Prioridad = *f(uso reciente CPU)* ⇒ Más uso ⇒ menor *prioridad* (valor alto para el campo prioridad).
- Rango de prioridades.
  - Prioridades de usuario por debajo de un umbral.
    + Procesos requisados al volver de modo *kernel* a modo usuario.
  - Prioridades de *kernel* por encima de un umbral.
    + Procesos en el algoritmo *sleep*.
      * Prioridad *kernel* baja (valor alto para el campo *prioridad*) ⇒ despiertan al recibir una señal (wakeup).
      * Prioridad *kernel* alta (valor bajo para el campo *prioridad*) ⇒ continuarán durmiendo (sleep).
- Cada clase de procesos ⇒ varios niveles de prioridad.
- Cada valor posible tiene asociada una cola de procesos (Figura 2.16).

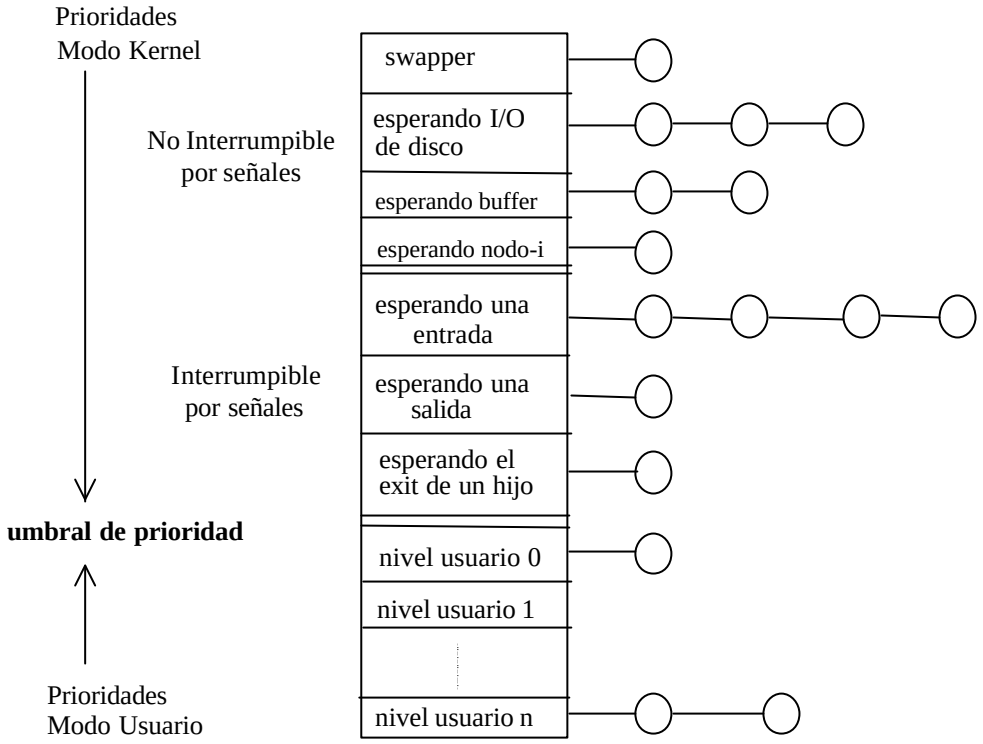

Figura 2.16. Rango de Prioridades de los procesos.

2.10.3. Cálculo y Control de Prioridad de un Proceso.

- **A**: Asigna la prioridad a un proceso a punto de dormir ⇒ Valor fijo dependiente de la razón por la que se fue a dormir.
- **B**: Ajusta la prioridad a un proceso cuando vuelve a modo usuario desde modo *kernel.*
  - Tiene prioridad muy alta.
  - Ha usado valiosísimos recursos del *kernel.*
- **C**: Ajusta las prioridades a procesos en modo usuario en intervalos fijos de tiempo (quantum) y ejecuta el algoritmo de planificación para prevenir la monopolización de la CPU por parte de determinados procesos (Round Robin).
- Reloj puede interrumpir el proceso varias veces durante su quantum de tiempo.
- Cada interrupción ⇒ el manejador de interrupciones incrementa el campo de prioridad en la tabla de procesos que registra el uso reciente de CPU por parte del proceso.
- Cuando se produce el fin del quantum ⇒ calcula el uso reciente de la CPU ajustándolo con:
  - Función de decaimiento ⇒ decaimiento (CPU) = CPU / 2.
- Recalcula la prioridad de procesos en estado "listo para su ejecución".
  - prioridad = (uso_reciente_CPU / 2) + (nivel_base_prioridad_usuario).
- El efecto de recalcular las prioridades ⇒ Movimiento de procesos entre las diferentes colas de prioridad.
- Otras implementaciones del mecanismo de planificación.
  - Variar el tamaño del quantum de tiempo dinámicamente según la carga del sistema.
  - Respuesta más rápida vs. más cambios de contexto ⇒ elegir un tamaño de quantum adecuado.

- *int nice(inc)*. La primitiva *nice* permite modificar la prioridad dinámica del proceso actual. El parámetro *inc* se añade a la prioridad actual. Sólo un proceso privilegiado puede especificar un valor negativo para este parámetro con el objetivo de aumentar su prioridad. En caso de éxito *nice* devuelve el valor 0, si no nice devuelve el valor −1 y la variable *errno* toma el valor EPERM, que indica que el proceso que llama no posee los privilegios para aumentar su prioridad.
- Control de procesos sobre su prioridad de planificación ⇒ *nice (valor)*.
- Cálculo de la prioridad del proceso.
  - prioridad = (uso_reciente_CPU / cte.) + (prioridad_base) + (valor_nice).
- *Nice* incrementa o decrementa (superusuario) *valor_nice* en la tabla de procesos con *valor*. El valor dado a *valor_nice* revalora o degrada el prioridad dinámica de un proceso.
- Se hereda ⇒ El valor de prioridad asignado por *nice* es heredado por los procesos hijo después de realizar la llamada.
- Características.
  - Trabaja únicamente para procesos que están en ejecución.
  - El Proceso1 no puede variar el *valor_nice* del Proceso 2.
  - El administrador decide bajar la prioridad P del proceso que consume mucha CPU ⇒ Matarlo.
- Otras funciones para el control de prioridades de procesos son: *setpriority* y *getpriority*.
  - La llamada al sistema *setpriority* modifica la prioridad de un proceso, de un grupo de procesos o de todos los procesos de usuario.
  - La llamada al sistema *getpriority* permite obtener la prioridad de un proceso, de un grupo de procesos o de todos los procesos de usuario.

- Algoritmo de planificación anterior ⇒ no diferencia clases de usuarios.
- Principio de los **Planificadores de Trato Justo**.
  - Dividir usuarios en grupos de porción justa.
  - Asignación de la CPU proporcional a cada grupo.
  - Restricciones habituales a los miembros de cada grupo.
- Implementación simple.
  - *prioridad_trato_justo* compartida por todos los procesos del grupo.
  - Manejador de interrupciones incrementa el valor de *prioridad_trato_justo* para procesos en ejecución.

2.10.4. Scheduler de Linux.

El *scheduler* es el elemento del *kernel* (subsistema de control de procesos) que decide qué proceso (listo para su ejecución en memoria principal) debe ser ejecutado por el procesador (CPU). El *scheduler* explora la lista de procesos "listos para ejecutarse en memoria" y utiliza varios criterios para elegir el proceso a ejecutar. Es decir, cuando hay más de un proceso "listo para ejecutarse", el sistema operativo debe decidir cuál ejecutará primero, la parte del sistema operativo que toma esta decisión es el *scheduler*, y los algoritmos que utiliza se denominan algoritmos de planificación.
El scheduler tiene que elegir el proceso que más merece ejecutarse entre todos los procesos que se pueden ejecutar en el sistema. Un proceso ejecutable es aquel que en *kernel/sched.c* está esperando solamente a una CPU para ejecutarse. Linux usa un algoritmo para planificar las prioridades razonablemente simple para elegir un proceso entre los procesos que hay en el sistema. Cuando ha elegido un nuevo proceso para ejecutar, el scheduler salva el estado del proceso en curso, los registros específicos del procesador y otros contextos en la estructura de datos task_struct. Luego restaura el estado del nuevo proceso (que también es específico a un procesador) para ejecutarlo y da control del sistema a ese proceso. Para que el scheduler asigne el tiempo de la CPU justamente entre los procesos ejecutables en el sistema, el scheduler mantiene cierta información en la estructura task_struct de cada proceso:

- **policy** (política) Esta es la política de planificación que se aplicará a este proceso. Hay dos tipos de procesos en Linux, normales y de tiempo real. Los procesos de tiempo real tienen una prioridad más alta que todos los otros. Si hay un proceso de tiempo real listo para ejecutarse, siempre se ejecutara primero. Los procesos de tiempo real pueden tener dos tipos de **políticas**: round robin (en círculo) y first in rst out (el primero en llegar es el primero en salir). En la planificación round robin, cada proceso de tiempo real ejecutable se ejecuta por quatums (ticks de tiempo), y en la planificación first in, first out cada proceso ejecutable se ejecuta en el orden que están en la cola de ejecución y el orden no se cambia nunca.
- **priority** (prioridad) Esta es la prioridad que el scheduler dará a este proceso. También es la cantidad de tiempo (en jiffies) que se permitirá ejecutar a este proceso una vez que sea su turno de ejecución. Se puede cambiar la prioridad de un proceso mediante una llamada de sistema y la orden *renice*.
- **rt_priority** (prioridad de tiempo real) Linux soporta procesos de tiempo real y estos tienen una prioridad más alta que todos los otros procesos en el sistema que no son de tiempo real. Este campo permite al scheduler darle a cada proceso de tiempo real una prioridad relativa. La prioridad del proceso de tiempo real se puede alterar mediante llamadas de sistema.
- **counter** (contador) Esta es la cantidad de tiempo (en jiffies) que este se permite ejecutar a este proceso. Se iguala a priority cuando el proceso se ejecuta y se decrementa a cada paso de reloj.

Los campos de una estructura de tareas relevante a planificar incluyen (donde *p* es el puntero a la tabla de procesos y va a ir recorriendo dicha lista):

- **p->need_resched**. Este campo es establecido si *schedule()* debería de ser llamado en la 'siguiente oportunidad'.
- **p->counter**. Número de ticks de reloj que quedan en esta porción de tiempo del scheduler, decrementada por un cronómetro. Cuando este campo se convierte a un valor menor o igual a cero, es reinicializado a 0 y p->need_resched es establecido. Esto también es llamado a veces 'prioridad dinámica' de un proceso porque puede cambiarse a si mismo.
- **p->priority**. La prioridad estática del proceso, sólo cambiada a través de bien conocidas llamadas al sistema como nice, sched_setparam o setpriority.
- **p->rt_priority**. Prioridad en tiempo real.
- **p->policy**. La política de planificación, específica a la clase de planificación que pertenece la tarea. Las tareas pueden cambiar su clase de planificación usando la llamada al sistema sched_setscheduler. Los valores válidos son SCHED_OTHER (proceso UNIX tradicional), SCHED_FIFO (proceso FIFO en tiempo real) y SCHED_RR (proceso en tiempo real round-robin). Uno puede también SCHED_YIELD a alguno de esos valores para significar que el proceso decidió dejar la CPU, por ejemplo llamando a la llamada al sistema sched_yield. Un proceso FIFO en tiempo real funcionará hasta que: (1) se bloquee en una E/S; (2) explícitamente deje la CPU, o (3) es predesocupado por otro proceso de tiempo real con un valor más alto de p->rt_priority. SCHED_RR es el mismo que SCHED FIFO, excepto que cuando su porción de tiempo acaba vuelve al final de la cola de ejecutables.

El scheduler se ejecuta desde distintos puntos dentro del *kernel*. Se ejecuta después de poner el proceso en curso en una cola de espera y también se puede ejecutar al finalizar una llamada de sistema, exactamente antes de que un proceso vuelva al modo usuario después de estar en modo sistema. También puede que el scheduler se ejecute porque el temporizador del sistema haya puesto el contador counter del proceso en curso a cero. Cada vez que el scheduler se ejecuta, hace lo siguiente:

- **Trabajo del *kernel***. El scheduler ejecuta la parte baja de los manejadores y procesos que el scheduler pone en la cola.
- **Proceso en curso**. El proceso en curso tiene que ser procesado antes de seleccionar a otro proceso para ejecutarlo. Si la política de planificación del proceso en curso es round robin entonces el proceso se pone al final de la cola de ejecución. Si la tarea es INTERRUMPIBLE y ha recibido una señal desde la última vez que se puso en la cola, entonces su estado pasa a ser RUNNING (en ejecución). Si el proceso en curso a consumido su tiempo, si estado pasa a ser RUNNING (en ejecución). Si el proceso en curso está RUNNING (en ejecución), permanecerá en ese estado. Los procesos que no estén ni RUNNING (en ejecución) ni sean INTERRUMPIBLEs se quitan de la

cola de ejecución (Listos). Esto significa que no se les considerará para ejecución cuando el scheduler busca un proceso para ejecutar.

- **Selección de un proceso**. El scheduler mira los procesos en la cola de ejecución para buscar el que más se merezca ejecutarse. Si hay algún proceso de tiempo real (aquellos que tienen una política de planificación de tiempo real) entonces estos recibirán un mayor peso que los procesos ordinarios. El peso de un proceso normal es su contador counter pero para un proceso de tiempo real es su contador counter más 1000. Esto quiere decir que si hay algún proceso de tiempo real que se pueda ejecutar en el sistema, estos se ejecutarán antes que cualquier proceso normal. El proceso en curso, que ha consumido parte de su porción de tiempo (se ha decrementado su contador counter) está en desventaja si hay otros procesos con la misma prioridad en el sistema; esto es lo que se desea. Si varios procesos tienen la misma prioridad, se elige el más cercano al principio de la cola. El proceso en curso se pone al final de la cola de ejecución. En un sistema equilibrado con muchos procesos que tienen las mismas prioridades, todos se ejecutarán por turnos. Esto es lo que conoce como planificación round robin. Sin embargo, como los procesos normalmente tiene que esperar a obtener algún recurso, el orden de ejecución tiende a verse alterado.
- **Cambiar procesos**. Si el proceso más merecedor de ejecutarse no es el proceso en curso, entonces hay que suspenderlo y poner el nuevo proceso a ejecutarse. Cuando un proceso se está ejecutando está usando los registros y la memoria física de la CPU y del sistema. Cada vez que el proceso llama a una rutina le pasa sus argumentos en registros y puede poner valores salvados en la pila, tales como la dirección a la que regresar en la rutina que hizo la llamada. Así, cuando el scheduler se ejecuta, se ejecuta en el contexto del proceso en curso. Estará en un modo *kernel* pero aún así el proceso que se ejecuta es el proceso en curso. Cuando este proceso tiene que suspenderse, el estado de la máquina, incluyendo el contador de programa (program counter, PC) y todos los registros del procesador se salvan en la estructura task_struct. A continuación se carga en el procesador el estado del nuevo proceso. Esta operación es dependiente del sistema; diferentes CPUs llevan esta operación a cabo de maneras distintas, pero normalmente el hardware ayuda de alguna manera. El cambio del contexto de los procesos se lleva a cabo al finalizar el scheduler. Por lo tanto, el contexto guardado para el proceso anterior es una imagen instantánea del contexto del hardware del sistema tal y como lo veía ese proceso al final del scheduler. Igualmente, cuando se carga el contexto del nuevo proceso, también será una imagen instantánea de cómo estaban las cosas cuando terminó el scheduler, incluyendo el contador de programa (program counter, PC) de este proceso y los contenidos de los registros. Si el proceso anterior o el nuevo proceso en curso hacen uso de la memoria virtual, entonces habrá que actualizar las entradas en la tabla de páginas del sistema. Una vez más, esta operación es específica de cada arquitectura.

El *scheduler* de Linux proporciona tres políticas de planificación diferentes: una para los procesos "normales" y dos para los procesos en "tiempo real" (interrumpibles o no interrumpibles). A cada proceso se le asocia un tipo de proceso, una prioridad fija y una prioridad variable. El tipo de proceso puede ser: SCHED_FIFO = para un proceso de "tiempo real" no interrumpible, SCHED_RR = para un proceso en "tiempo real" interrumpible, y SCHED_OTHER = para un proceso normal. La política de planificación depende del tipo de procesos contenidos en la lista de procesos "listos para ejecutarse":

- Cuando un proceso de tipo SCHED_FIFO esta listo para ejecutarse, se ejecuta inmediatamente. El *scheduler* da prioridad a la ejecución del proceso de tipo SCHED_FIFO que posea la más alta prioridad, y lo ejecuta. Este proceso es normalmente no interrumpible, es decir, que el proceso posee el procesador y el sistema solo interrumpirá su ejecución en tres casos:
  1. Otro proceso de tipo SCHED_FIFO que posea una prioridad más elevada para al estado de listo para ejecutarse y se ejecuta inmediatamente.
  2. El proceso se suspende en espera de un evento (durmiendo), como por ejemplo, el fin de una operación de entrada/salida.
  3. El proceso abandona voluntariamente el procesador para una llamada a una primitiva *sched_yield*. El proceso pasa a estado de listo para ejecutarse y se ejecutan otros procesos.
- Cuando un proceso de tipo SCHED_RR está listo para ejecutarse, se ejecuta inmediatamente. Su comportamiento es similar al del SCHED_FIFO, con una excepción: cuando el proceso se ejecuta se le atribuye un quantum de tiempo. Cuando este quantum expira, puede elegirse y ejecutarse un proceso de tipo SCHED_FIFO o SCHED_RR de prioridad mayor o igual a la del proceso actual.

- Los procesos de tipo SCHED_OTHER únicamente pueden ejecutarse cuando no existe ningún proceso de "tiempo real" en estado de listo para ejecutarse. El proceso a ejecutar se escoge tras examinar las prioridades dinámicas. La prioridad dinámica de un proceso se basa por una parte en el nivel especificado por el usuario según las llamadas al sistema *nice* y *setpriority*, y por otra parte en una variación calculada por el sistema. Todo proceso que se ejecute durante varios ciclos de reloj disminuye en prioridad y puede así llegar a ser menos prioritario que los procesos que no se ejecutan, cuya prioridad no se ha modificado.

Existen numerosas llamadas al sistema que permiten modificar la política y los parámetros asociados a un proceso, entre ellas destacamos las siguientes:

- *int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param)*. Modifica la política (policy) y los parámetros de planificación (param) asociados al proceso (pid).
- *int sched_getscheduler(pid_t pid)*. Obtiene la política (SCHED_FIFO, SCHED_RR o SCHED_OTHER) de planificación asociada al proceso (pid) o al proceso actual si pid = 0.
- *int sched_setparam(pid_t pid, const struct sched_param *param)* y *int sched_getparam(pid_t pid, struct sched_param *param)*. Establece y obtiene los parámetros de planificación asociados al proceso (pid).
- *int sched_get_priority_min(int policy)* y *int sched_get_priority_max(int policy)*. Obtiene prioridades minima y máxima asociadas a la política de planificación (policy).
- *int sched_rr_get_interval(pid_t pid, struct timespec *interval)*. Devuelve en la estructura "interval" el quantum de tiempo asociado al proceso (pid) que debe ser del tipo SCHED_RR.
- *int sched_yield(void)*. Permite al proceso actual liberar al procesador. Al ejecutar esta primitiva, el proceso actual se coloca al final de la lista de procesos "listos para ejecutarse en memoria" y entonces se ejecuta el *scheduler*. Si no existen otros procesos en el sistema, cambia al proceso actual.

La planificación de procesos *(scheduler)* la implementa la función *schedule* situada en el archivo fuente *kernel/sched.c*. También es preciso mencionar a la función *goodness* que devuelve un valor que indica a *schedule* hasta qué punto el proceso necesita el procesador. Por ello *goodness*, puede devolver: *–1000* si se trata del proceso actual para indicar que no debe seleccionarse; *prioridad estática + 1000* si es un proceso tiempo real; *número de ciclos de reloj que el proceso debe ejecutarse* (counter) si se trata de un proceso normal.

La función *schedule* que implementa la planificación de procesos, empieza por desplazar el proceso actual al final de los procesos listos (llamando a *move_last_runqueue*) si el proceso ha agotado todos sus ciclos (su tiempo). Si el proceso esta en estado = TASK_INTERRUPTIBLE, entonces *schedule* comprueba si ha recibido una señal; si es así estado = TASK_RUNNING con el objetivo de despertarlo. Para ello, *schedule* explora todos los procesos de la Tabla de Procesos, llama a la función *goodness* para escoger el siguiente proceso a ejecutar y al volver de la búsqueda, el proceso escogido se convierte en el proceso actual. Entonces, *get_mmu_context* llama para restaurar el contexto de memoria del proceso y *switch_to* se llama para provocar el cambio de contexto.

La función *schedule* gestiona el tiempo asociado a cada proceso a través del campo *counter* del descriptor del proceso, pudiendo reiniciarlo cuando considere oportuno. El campo *counter* determina el número de ciclos de reloj durante los cuales el proceso debe ejecutarse. El campo *counter* puede ser modificado por:

- *update_process_time*. Esta función es llamada periódicamente por *timer_bh*, que forma parte de la cola de tareas *tq_timer*, activada por el manejador de interrupciones del reloj. Poniendo la variable *need_rsched* a 1 si el proceso ha agotado su quantum.
- *add_to_runqueue*. Esta función añade un proceso a la lista de procesos "listos para ejecutarse en memoria".