Autores: Sebastián Sánchez y Mariano Gómez
ssp@aut.alcala.es y mgomez@aut.alcala.es

Procesos en Linux

En el presente artículo realizamos un análisis de los procesos en Linux. Comenzaremos con una descripción general, a continuación nos centraremos en las estructuras de datos y los mecanismos del núcleo y, por último, presentaremos la interfaz de llamadas al sistema empleado en Linux para ejecutar nuevos procesos.

Un *proceso* se puede definir como un programa en ejecución o también como la unidad de trabajo en el sistema en un momento dado. Evidentemente un proceso no es lo mismo que un programa, ya que el primero se puede considerar como un ente dinámico mientras que el segundo es un ente estático (archivo almacenado en una unidad de almacenamiento secundario). Debido a esta característica, varios procesos se pueden derivar de un mismo programa. Es el sistema operativo el encargado de lanzar el programa y convertirlo en proceso.
Los *hilos* se consideran procesos ligeros o especiales debido al hecho de que se derivan de otro proceso que, a su vez, se ha podido derivar de un programa o de otro proceso. Entonces de un mismo proceso se pueden derivar diversos hilos (véase figura 1). Como se ha comentado anteriormente, una vez que un proceso se ha creado a partir de un programa, se podría considerar como un único hilo de ejecución. Ocurre, sin embargo, como veremos más adelante, que este proceso puede derivar bien en otros procesos a su vez o en procesos ligeros denominados hilos (véase figura 1).
En los procesos e hilos se ven involucrados registros máquina (contador de programa, otros registros de la CPU), así como las pilas que contienen datos temporales, tales como parámetros de subrutina, direcciones de retorno y variables.
Se podría considerar que cada uno de los procesos que pueden existir en un sistema que soporte multiprogramación, son entidades separadas con sus propios derechos y responsabilidades. Así por ejemplo, si un proceso, por la razón que sea, termina anormalmente su ejecución, este hecho no debe afectar ni influir a los demás procesos que pasen a tomar la CPU más adelante. Cada proceso se ejecuta en su propio espacio de direccionamiento virtual y no es capaz de interactuar con otros procesos excepto a través de mecanismos de comunicación manejados por el Kernel.
Durante el tiempo que dure la ejecución de un proceso, este hará uso de diferentes recursos del sistema. Usará la CPU para la ejecución de sus propias instrucciones y la memoria del sistema para su propio mantenimiento y almacenamiento de los datos. Abrirá y usará archivos dentro del sistema de archivos proporcionado por el propio sistema operativo (Linux), pudiendo directamente o indirectamente usar los dispositivos físicos del sistema.

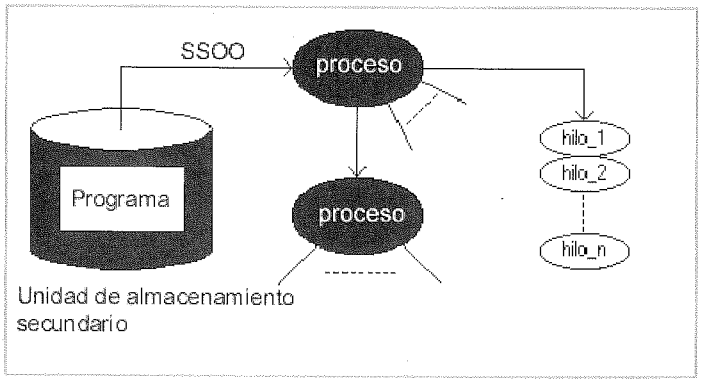

FIGURA 1. PROCESOS E HILOS.

> El sistema operativo se encarga de lanzar el programa y convertirlo en proceso

Linux deberá tener constancia en todo momento de las acciones llevadas a cabo por cada uno de los procesos y de los recursos del sistema que están siendo usados por ellos, de manera que no se produzca ningún conflicto por el hecho de que los recursos disponibles en el sistema sean compartidos por los demás procesos.

PROCESOS EN LINUX

El sistema operativo Linux combina técnicas de multiprogramación y de tiempo compartido para incrementar el rendimiento del hardware a la vez que se favorece a los trabajos interactivos.
Linux mantiene constancia de cada proceso del sistema utilizando una estructura de datos denominada *task_struct*. En ella se almacena toda la información relacionada con el proceso: aspectos de planificación, identificadores, relación con otros procesos, memoria utilizada por el proceso, archivos abiertos, etc. Cada *task_struct* es característica de cada proceso y todas ellas juntas forman lo que se conoce como tabla de control de procesos o tabla de control de tareas. Cada *task_struct* es apuntada por un puntero perteneciente a una matriz de punteros denominada *task*. La cantidad de entradas activas de esta matriz determina el grado de multiprogramación del sistema hasta un máximo definido por la constante *NR_TASKS* que, por defecto, toma el valor 512. La declaración de ambas estructuras de datos se encuentra en el archivo de cabecera *~/include/linux/sched.h*.
Los procesos en Linux no son entes estáticos, sino que van cambiando su estado continuamente en función de sus necesidades. Por ejemplo, cuando desde un programa en ejecución se nos pide que pulsemos una tecla, el programa se debe quedar esperando sin hacer nada hasta que la tecla correspondiente sea pulsada. En este caso, mientras no introduzcamos ningún dato por el teclado, no tiene sentido que el proceso consuma tiempo del procesador. Es más interesante bloquear al proceso y reanudarlo cuando se detecte una interrupción del teclado que nos indique la llegada de la información necesaria. Debido a esta circunstancia, cada proceso a lo largo de su vida va pasando por diferentes estados. En el campo *state* de la estructura *task_struct* se mantiene el estado de cada proceso.
Task_running: en Linux se dice que un proceso se encuentra estado activo cuando se está ejecutando o cuando está listo para ejecutarse. En este segundo caso, el proceso no dispondrá del procesador por estar éste asignado a otra tarea de mayor prioridad.
Task_interrumpible: el proceso está esperando por la llegada de una señal. Cuando la señal llegue el proceso pasará al estado activo.
Task_ininterrumpible: el proceso está esperando por algún recurso hardware.
Task_stopped: el proceso ha sido detenido momentáneamente.
Task_zombie: en este estado quedan los procesos que han finalizado, pero no se ha eliminado su entrada de la tabla de control de tareas.

PLANIFICACIÓN

La planificación hace referencia al hecho de cómo se reparte el tiempo de procesador entre las distintas tareas activas en el sistema. El planificador es la parte del sistema operativo encargada de determinar cuál ha de ser el siguiente proceso que debe tomar la CPU. El método empleado en Linux es relativamente simple y equitativo, evitando posibles problemas de inanición, es decir, dando a todas las aplicaciones la oportunidad de ejecutarse cada ciertos intervalos de tiempo. Si una aplicación no libera voluntariamente el procesador, al cabo de 200 ms, la CPU es requisada y asignada a otra aplicación activa. Esto evita que una aplicación pueda adueñarse del procesador y bloquear el sistema completo. En este punto, el planificador guarda el estado de la tarea que se estaba ejecutando, sus registros y su información de estado, en la estructura *task_struct* y recupera el estado de la nueva tarea a la que se le asigna el procesador.

Para establecer correctamente las políticas de planificación, el planificador mantiene en la estructura de control de cada tarea información que permite determinar cuál es la siguiente tarea que utilizará el procesador. Linux establece diferencias entre procesos ordinarios y procesos de tiempo real. Estos últimos siempre se ejecutarán antes que los procesos ordinarios. Para diferenciar entre ambos tipos de procesos, en la estructura *task_struct* se mantiene un campo denominado *policy*. Dentro de los procesos de tiempo real los puede haber con mayor o menor prioridad, para diferenciar unos de otros se emplea el campo *rt_priority* que puede modificarse mediante llamadas al sistema.

Siempre que existan procesos de tiempo real que necesitan la CPU, serán ejecutados antes que los procesos ordinarios. Si todos los procesos activos son procesos ordinarios, Linux emplea el campo *priority* para determinar cuál es el siguiente proceso que tomará la CPU. Este campo almacena la cantidad de pulsos de reloj (*jiffies*) que el proceso tiene asignados para su ejecución. Con cada pulso de reloj este valor se decrementa y se almacena en la variable *counter*.

La rutina encargada de determinar la política de planificación es la función *goodness* incluida en el archivo *~/kernel/sched.c*. El valor devuelto por esta función es utilizado para este propósito. Cuanto mayor sea este valor, mayor es la prioridad del proceso. Si el valor devuelto es 1000, entonces el proceso tiene asignado otro procesador y no es bueno elegirlo como candidato. Para los procesos de tiempo real devuelve el valor *rt_priority+1000* y para los procesos ordinarios devuelve el valor *counter* que indica el número de ciclos de reloj durante los cuales debe ejecutarse el proceso. Si existen varios procesos con la misma prioridad, se ejecutará primero el que se encuentre más cerca de la cabeza de la lista de procesos listos. Una vez que el proceso finaliza su cuanto de tiempo, se coloca al final de la cola y se le asigna el procesador al siguiente proceso. A este método de planificación se le conoce como planificación *Round Robin*. La función *schedule* del módulo *~/kernel/sched.c* es la que define el mecanismo de planificación, dicho de otro modo, esta función no toma decisiones, simplemente ejecuta las tareas tal y como se lo indica la función *goodness*. Las operaciones realizadas por la rutina *schedule* se enumeran a continuación:

- Primero comprueba si la llamada es correcta, si no es así, simplemente retorna.
- Seguidamente se invoca a la rutina *do_botton_half*, la cual lleva a cabo una lista de operaciones que deben ser realizadas dentro del núcleo para un correcto funcionamiento del sistema. Este tipo de operaciones tiene que ver con los manejadores de dispositivo y las interrupciones. Cuando se produce una interrupción, el procesador deja de realizar la operación actual y pasa a ejecutar código del manejador. La rutina de interrupción debe retornar rápidamente por motivos de eficiencia. Por este motivo, los manejadores pueden encolar operaciones que se llevarán a cabo en los momentos que el núcleo considere oportunos y no asíncronamente como ocurre con las interrupciones. Como ejemplo de este tipo de operaciones tenemos la *timer_bh*, que actualiza la hora del sistema; la *serial_bh*, que está relacionada con las líneas serie o la *tqueue_bh* que es la cola de temporizadores.

Linux combina técnicas de multiprogramación y de tiempo compartido para incrementar el rendimiento del hardware

- En este punto, en caso de haberse agotado los ciclos de reloj asociados al proceso actual y si la política de planificación es *Round Robin* (*sched_rr*), éste es movido al final de la cola de procesos listos invocando a la rutina *move_last_runqueue*.
- A continuación, si el estado del proceso actual es interrumpible (*task_interrumpible*), se analiza si se ha recibido una señal desde la última vez que se planificó. Si éste es el caso, el proceso pasará al estado activo (*task_running*).
- Finalmente, por cada uno de los procesos activos se invoca a la rutina *goodness* para determinar cuál de los procesos listos es el siguiente que debe ser ejecutado. Una vez definido el candidato idóneo, éste pasa a ser el proceso actual. El control se le pasa al mismo restaurando su contexto de memoria mediante la rutina *get_mmu_context* e invocando finalmente a la rutina *switch_to*.

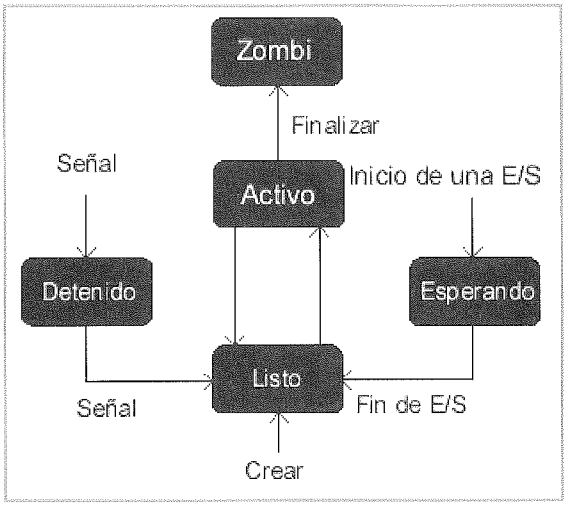

FIGURA 2. DIAGRAMA DE ESTADOS DE LOS PROCESOS.

Linux tiene soporte para sistemas con varios procesadores en configuración simétrica (*SMP o Symmetric Multi-Processing*). En este caso se puede repartir la carga entre las distintas CPUs con objeto de aumentar el rendimiento global del sistema. En este tipo de sistemas no tenemos un único proceso en ejecución, podremos tener tantos como procesadores tenga el sistema. Contemplando la posibilidad de trabajar en un sistema SMP, dentro de la *task_struct* de cada proceso se mantiene información relacionada con cada procesador específico.

IDENTIFICADORES DE UN PROCESO

Linux asocia a cada proceso del sistema un identificador de proceso denominado PID (*process identifier*) y otro que identifica al proceso padre denominado PPID (*parent process identifier*). Aparte de estos dos identificadores existen otros adicionales que permiten determinar los derechos de acceso del propio proceso a los recursos del sistema.

La lista de control de acceso de cada recurso en Linux determina los derechos que tiene el propietario del recurso, los derechos del grupo al cual pertenece el propietario y los derechos por defecto. Para ello, cada recurso del sistema lleva asociados dos identificadores que permiten determinar quién es su propietario y cuál es su grupo. Los campos de identificación del proceso permiten conocer quién ha lanzado el proceso (*uid* o *user identifier*) y a qué grupo pertenece el propietario del mismo (*gid* o *group identifier*). Comparando estos identificadores con los identificadores del recurso, y acorde con la lista de control de acceso (ACL) del propio recurso, el sistema Linux establece los derechos de acceso.

Existen por cada proceso cuatro pares de identificadores de usuario y de grupo respectivamente, los cuales se describen a continuación:

- *uid y gid*: son los identificadores de usuario y de grupo del propietario del proceso. Ambos se almacenan en la entrada correspondiente al usuario en los campos 3 y 4 del archivo */etc/passwd* respectivamente.

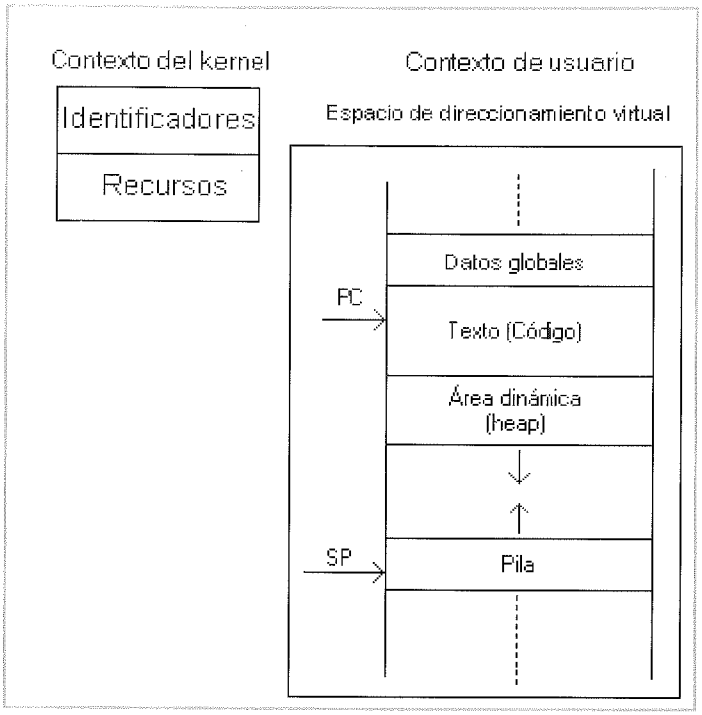

FIGURA 3. REPRESENTACIÓN DEL ESPACIO DE MEMORIA VIRTUAL DE UN PROCESO.

- *uid y gid* efectivos (*euid y egid*): son dos identificadores que toman ciertos programas durante su ejecución. Dichos identificadores no coinciden con los que se les asigna normalmente a cualquier programa lanzado por un determinado usuario, sino que coinciden con los identificadores del propietario del programa. Veamos un ejemplo que aclare el concepto anterior. Es bien sabido que el archivo de contraseñas, */etc/passwd*, es un archivo de sólo lectura para los usuarios ordinarios. Si embargo, en este archivo es donde se almacenan las contraseñas encriptadas de cada usuario del sistema.

Linux establece diferencias entre procesos ordinarios y procesos de tiempo real

Entonces, ¿cómo es posible que un usuario pueda cambiar su clave si ello implica una modificación del archivo */etc/passwd*? La respuesta es bien sencilla, el programa que permite modificar nuestra clave (*passwd*) es un programa que pertenece al administrador del sistema y que tiene activo su *setuid*. Esto quiere decir que cuando cualquier usuario ejecute este programa, el proceso correspondiente llevará asociados dos identificadores de usuario y de grupo efectivos que coinciden con los identificadores del propietario del programa, en este caso, el administrador. De este modo, a todos los efectos, la orden *passwd* se ejecuta como si la hubiese lanzado el propio administrador.

- *uid y gid* de sistema de archivos (*fsuid y fsgid*): son dos identificadores utilizados cuando trabajamos con sistemas de archivos en red como puede ser NFS (*Network File System*). En este caso, el servidor NFS debe comportarse a efectos de derechos como si fuese el programa de usuario, pero no se le puede asignar al servidor los identificadores del usuario, ya que de este modo alguien podría *matarlo* maliciosamente.
- *uid y gid* de guarda (*suid y sgid*): se emplean para almacenar los *uid y gid* originales cuando un proceso pide modificar sus identificadores a través de una llamada al sistema.

ARCHIVOS

Dentro de la estructura de control de cada proceso se mantiene información relacionada con el sistema de archivos y con los archivos abiertos por el proceso. Para ello, se emplean los campos *fs* y *files* de la estructura *task_struct*, respectivamente. El primero es un puntero a una estructura *fs_struct* que contiene, a su vez, punteros al *inode* raíz y al *inode* correspondiente al directorio actual de trabajo del proceso. El segundo es un puntero a una estructura *files_struct* que contiene información acerca de los archivos abiertos por el proceso en cada instante. El descriptor no es más que un índice dentro de esta tabla. Es bien sabido, que los tres primeros descriptores, fd[0], fd[1] y fd[2] identifican a los archivos estándar de entrada, de salida y de errores respectivamente. Cada *fd[x]* apunta a una estructura de tipo *file* que contiene información relacionada con el archivo cuyo descriptor es *x*. Dentro de esta estructura, el campo *f_inode* apunta al *inode* correspondiente al archivo, el cual puede ser un archivo ordinario o un dispositivo. Otro campo que merece especial mención es *f_pos*, en él se mantiene el puntero de lectura-escritura del archivo. De este modo evitamos colocar en las llamadas al sistema *read* y *write* este puntero de forma explícita. Este puntero puede ser modificado utilizando la llamada *lseek*.

PROGRAMAS Y PROCESOS

En las secciones de introducción y entidades de ejecución se vieron las diferencias entre lo que es un programa y un proceso. Podíamos decir que un proceso es un programa en ejecución. Insistimos en que un programa no es un proceso en sí mismo; un programa es una entidad pasiva o estática mientras que un proceso es una entidad activa o dinámica. La ejecución de un proceso tiene que progresar de una manera secuencial. Esto quiere decir que en todo momento se ejecuta como máximo una sola instrucción por cuenta del proceso. Así, aunque pueda haber dos procesos asociados con el mismo programa, se consideran, sin embargo, dos secuencias de ejecución independientes.

ESPACIO DE DIRECCIONAMIENTO

En este punto se va a tratar de explicar o comentar algunos aspectos referentes al espacio de direccionamiento usado por los programas y procesos. De lo comentado en los puntos anteriores, es evidente deducir que para los programas que posteriormente se pueden convertir en procesos, sería más conveniente hablar de espacio de almacenamiento en lugar de direccionamiento. Esto es así, porque al considerarse entidades estáticas, están almacenados en unidades de almacenamiento secundarias (disco duro, disquete, CD-ROM, etc.), por lo que ocuparán un espacio físico determinado dentro de dicha unidad.
Sin embargo, cuando hablamos de procesos, lo correcto es referirse a la zona de memoria en donde está cargado debido a que es código que está siendo ejecutado por el procesador y, por lo tanto, deberá estar ubicado en memoria principal y no en la unidad de almacenamiento secundario. Debido a que el código cargado en memoria principal es aquél que se ejecuta por el microprocesador, es necesario que exista algún subsistema encargado de gestionar este espacio de direccionamiento. Este subsistema es una de las partes más importantes de cualquier sistema operativo. Y ya que el sistema operativo se va a encargar de la gestión de memoria, uno de los objetivos planteados a la hora de diseñar el subsistema software anterior, es tratar de lanzar programas a la CPU que tengan un tamaño mayor que la propia memoria física del sistema. La solución al objetivo marcado se basa en la utilización de memoria virtual, es decir, un sistema basado en memoria virtual se comporta como si tuviera un tamaño de memoria principal mayor que el que existe físicamente. Por lo tanto, en un sistema basado en multiprogramación, cada proceso del sistema va a tener su propio espacio direccionamiento virtual y ningún proceso va a poder interferir con otro a través de su espacio virtual. En este caso, la forma de comunicarse varios procesos entre sí, es usando memoria virtual compartida. El campo *mm* de la estructura *task_struct* contiene información relacionada con la memoria virtual de cada proceso en Linux. Este campo apunta a una estructura *mm_struct*, la cual contiene información acerca de la memoria vinculada al proceso.
A continuación, en la figura 3 se muestra un ejemplo de una posible distribución del espacio de memoria virtual asociado a un proceso, en donde se indica la memoria asignada y los recursos usados por el proceso. Existen las siguientes áreas o zonas de memoria:

- Area de sólo-lectura en donde se almacenan las instrucciones que componen el programa.

- Area de lectura-escritura para el almacenamiento de los datos o variables globales.
- Área de *heap* para la asignación de memoria dinámica.
- Una pila sobre la que se van almacenando las variables locales

Además de la información anterior, el sistema operativo necesita tener conocimiento de:

- Contador de programa (PC), que contiene la dirección de la instrucción que está siendo actualmente ejecutada.
- Puntero de pila (SP), que apunta a la dirección, a partir de la cual se salvarán posibles variables.
- Recursos usados por el propio proceso, tales como archivos abiertos, señales, *sockets*, etc.
  Identificadores de proceso.

Esta última información, el sistema operativo la guarda también en memoria principal, en la estructura *task_struct*.

CREACIÓN DE PROCESOS (FORK ())

El sistema operativo Linux va a tratar a los diferentes procesos de forma jerárquica, existiendo por lo tanto procesos denominados hijos y procesos padres. Otra característica de Linux, es la manera en la que crea los procesos. Todos los procesos que en un momento dado puedan existir en el sistema, se han ido derivando o creando a partir de otro (padre) situado en un nivel jerárquico por encima del creado (hijo). Por lo tanto, podemos decir que todos los procesos se han ido derivando de un único proceso padre, que el sistema operativo lanzó en su inicialización.

Habría que distinguir entre los procesos creados por programa y los procesos que el usuario lanza al sistema operativo a través de la consola para su posterior ejecución. Tanto en un caso como en otro, se van a considerar ambos procesos como hijos de aquel proceso inicial que el sistema operativo lanzó en su inicialización, con la diferencia de que los procesos que el usuario lanza se pueden considerar padres independientes y los procesos que se crean por programa se consideran hijos, a su vez, del proceso que desencadenó su creación.

La llamada al sistema *fork* permite crear procesos por programa. Esta función o llamada al sistema permite crear procesos hijos que se diferencian del padre en su identificador de proceso. Una vez que se invoca esta función, el sistema duplica en memoria el contenido del espacio virtual asignado al proceso padre en otro espacio para el proceso hijo, de tal manera, que ambos espacios virtuales sean independientes. Esta función tiene tres posibles valores de retorno: si la ejecución es correcta, el identificador del proceso creado (hijo) se retorna al proceso padre y 0 al proceso hijo. En caso de fallo, -1 se retorna al padre y no se crea ningún proceso hijo. A continuación, se muestra un fragmento de código en lenguaje C en donde se ve el uso de *fork()*.

```
...
main ()
...
{
int id, a = 5;
if ( ( pid = fork () )
== -1 ) {
        perror ( "" ),
        exit ( 1 );
} /* if */
else if ( pid != 0 ) {
        /* proceso
padre */
        a = a + 1;
} /* else if */
else {
        /* proceso hijo
*/
        a = a * 2;
} /* else */
} /* main */
```

Una vez este programa sea convertido en un proceso por el sistema operativo, en el momento en que se ejecute la línea que contiene la función *fork()*, se bifurca un nuevo proceso (hijo). El código que ejecutará el proceso padre será el contenido en la instrucción *else if*, cuya condición es distinta de cero, queriendo decir esto que el identificador del proceso hijo ha sido retornado por la función *fork* al proceso padre. Por el contrario, el código del hijo será el especificado en la instrucción *else*. Hasta antes de ejecutarse la función *fork*, la variable *a* era igual a 5. Cuando los dos procesos han sido creados, la variable *a* va a ser vista por ambos pero con la diferencia de que al tener éstos distinto espacio de direccionamiento virtual, la variable será tratada de forma independiente por cada uno de ellos. Es decir, las modificaciones que cualquiera de los dos procesos haga sobre la variable no serán vistas por el otro. Por lo tanto, cuando el proceso padre termine de ejecutar la instrucción $a = a + 1$, la variable *a* será igual a *6*. Y cuando el proceso hijo ejecute la instrucción $a = a * 2$, el proceso hijo va a ver dicha variable con el valor de *10*.

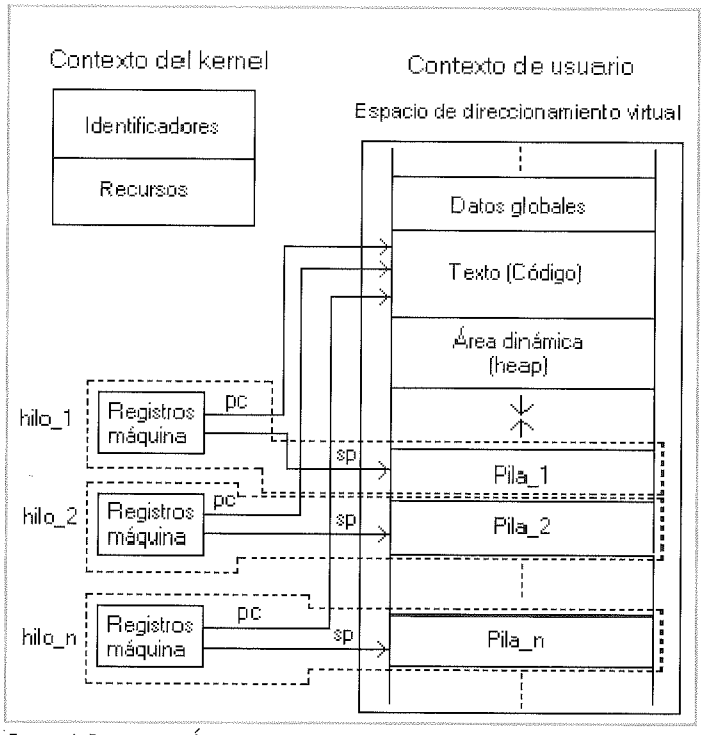

FIGURA 4. REPRESENTACIÓN DEL ESPACIO DE MEMORIA VIRTUAL DE UNA TAREA CON N HILOS.

PROCESOS E HILOS

Anteriormente se describió y se diferenció entre lo que es un proceso y un hilo. Veíamos que un hilo podía ser considerado como un proceso especial y debido a esto, un proceso tradicional puede ser considerado como un solo hilo de ejecución. El objetivo principal de los hilos o *threads* es compartir recursos entre ellos de forma cómoda. Cada hilo va a compartir con los demás hilos cooperantes, código, datos y recursos del sistema operativo. En este punto, vamos a introducir un concepto muy usado a la hora de programar hilos y denominado *tarea*. Una *tarea* se define como una entidad que no tiene capacidad de ejecución y que solamente posee recursos. Por sí sola no es capaz de hacer nada. En lugar de decir que un proceso tradicional se puede considerar como un solo hilo de ejecución, sería más correcto decir que un proceso tradicional está compuesto de una tarea, la cual tiene un único hilo de ejecución.

Por cada proceso existen cuatro pares de identificadores de usuario y de grupo respectivamente

Así, una tarea puede tener varios hilos. La conmutación de un hilo a otro dentro de la misma tarea requiere un coste mínimo ya que sólo es necesario salvar los registros y conmutar la pila. No es necesario ningún manejo de memoria, ya que parte de la memoria ocupada por la tarea es compartida entre todos los hilos de la misma, y por lo tanto la carga que sufre el sistema por el hecho de usar hilos es mínima.

ESPACIO DE DIRECCIONAMIENTO

Los hilos al ser considerados también procesos aunque *especiales*, van a tener al igual que aquéllos un espacio de direccionamiento virtual asociado, con la diferencia de que los hilos que pertenecen a la misma tarea van a tener el mismo espacio virtual. Esta es la razón por la que los hilos cooperantes, es decir, aquéllos que pertenecen a una misma tarea, pueden compartir código, datos y recursos del sistema.

A continuación, en la figura 4 se muestra un ejemplo de una posible distribución del espacio de memoria virtual asociado a una tarea que se compone de *n* hilos. Existen las siguientes áreas o zonas de memoria:

- Area de sólo-lectura en donde se almacenan las instrucciones que componen el programa.
- Area de lectura-escritura para el almacenamiento de los datos o variables globales. Estas variables son compartidas por los diferentes hilos
- Area de *heap* para la asignación de memoria dinámica. Esta área también es compartida por los hilos cooperantes de que consta la tarea.
- Una pila por cada uno de los hilos.

Un programa es una entidad pasiva y estática mientras que un proceso es activo y dinámico

En este caso cada hilo va a tener una copia de sus propios registros máquina (contador de programa, puntero de pila, etc.).
Utilizando hilos, el flujo de un programa puede seguir varios caminos al igual que usando procesos pero, debido al hecho de que los datos globales y heap se comparten, los hilos que tengan necesidad de acceder a estos datos lo deberán llevar a cabo de una forma controlada.

CREACIÓN DE HILOS

En Linux existe una llamada al sistema denominada *clone*, que aunque su objetivo es crear procesos, dependiendo de los argumentos que le pasemos, podremos crear los denominados hilos. Esta función tiene una sintaxis como la que se muestra a continuación:

```
pid_t clone(void *sp, unsigned long flags))
```

Esta función es una alternativa a *fork* pero con más opciones, de manera que la llamada al sistema *fork* es equivalente a:

```
clone(0, SIGCLDICOPYVM)
```

Si *sp* es distinto de cero, el proceso hijo creado usa *sp* como su puntero de pila. SIGCLD, es el *flag* que determina la señal que se enviará al proceso padre cuando el hijo termine.
Si el *flag* COPYVM se considera como en el caso anterior, el espacio virtual del proceso hijo es copiado del espacio virtual de proceso padre. Si no se especifica el hijo, comparte el mismo espacio que el padre, y tanto el padre como el hijo pueden escribir sobre los mismos datos.
También existe otro *flag*, denominado COPYFD que si se especifica, los descriptores de archivos del proceso hijo serán copias de los descriptores del padre. Y por el contrario, si no se especifica, los descriptores del hijo son compartidos con el padre.
Según lo anterior, si queremos crear un hilo dentro del código de un proceso usando la llamada al sistema clone, deberemos usar:

```
clone (sp)
```

donde *sp*, apunta a la primera posición de la pila que usará el hilo creado.

Esta función retorna el identificador del proceso hijo o hilo creado al padre y 0 al proceso hijo o hilo creado. En caso de fallo, se retorna –1 al proceso padre y no se crea ningún proceso o hilo. Veamos a continuación, cómo mediante el uso de la función *clone()* podemos crear hilos:

```
...
main () {
...
int id, a = 5;
...
if ( ( pid = clone ( 0 ) ) == -1 ) {
perror ( "" ),
exit ( 1 );
} /* if */
else if ( pid != 0 ) {
/* hilo padre */
a = a + 1;
} /* else if */
else {
/* hilo hijo */
a = a * 2;
} /* else */
} /* main */
```

En este caso, la variable *a* es accesible por ambos hilos, por lo que si uno la modifica al otro le afectará el cambio. Así, en un sistema que soporte multiprogramación, posibles resultados para la variable *a*, podrían ser los siguientes, dependiendo del orden de ejecución:

a=12, a =11

BIBLIOTECA ESTÁNDAR POSIX THREADS

POSIX Threads es una biblioteca estándar que simplifica la creación, seguridad y portabilidad de aplicaciones multihilo, para diferentes entornos.

Linux trata a los procesos de forma jerárquica existiendo procesos padre y procesos hijo

A continuación, se muestra un ejemplo que utiliza algunas de las funciones existentes para la creación y gestión de hilos desde un programa principal. Estas funciones o llamadas al sistema están declaradas dentro del fichero de cabecera *<pthreads.h>*:

```
#include <pthreads.h>
void rutina_1 ( int *dato ) {
dato = dato + 1;
} /* rutina_1 */

void rutina_2 ( int *dato ) {
dato = dato * 2;
} /* rutina_2 */

main () {
pthread_t  th1, th2;
phread_attr_t          attr;
int          a = 5;
if (pthread_attr_create (&attr) == -1) {
perror ( "" );
} /* if */

if (phread_create (&th1, attr, rutina_1, &a)==-
1) {
perror ( "" );
} /* if */

if (phread_create (&th2, attr, rutina_2, &a)==-
1) {
perror ( "" );
} /* if */

sleep ( 10 );
printf ( "Salida del hilo principal\n" );
} /* main */
```

El programa anterior tiene un hilo principal (función *main*) que haciendo dos llamadas a la función *pthread_create* crea dos hilos. Como se ha comentado anteriormente, estos hilos pertenecen a una única tarea (hilo principal) y por esta razón van a compartir el segmento de datos. En este ejemplo, será la variable *a* a la que accedan simultáneamente ambos hilos. Veamos a continuación, los pasos seguidos por el hilo principal, para la creación de los hilos:

El objetivo principal de los hilos es compartir recursos entre ellos de forma cómoda

Declaración de las variables *th1* y *th2* de tipo *pthread_t*. Estas variables se utilizan como identificadores de hilo para la función *pthread_create*.
Declaración de la variable *attr* de tipo *pthread_attr_t*. Es una variable en donde se almacenarán los atributos o características del hilo que se creará posteriormente.
Llamada a la función *pthread_attr_create*. Con esta llamada, se almacena en la variable *attr*, los atributos que el sistema operativo asigne por defecto.
Llamada a la función *pthread_create*. Esta función crea un hilo con los atributos especificados en *attr*. Una vez creado el hilo, se ejecutará su manejador correspondiente, bien rutina_1 para el hilo 1 o rutina_2 para el hilo 2. Por último, a esta función se le pasa también un puntero a la variable a para que el manejador pueda acceder a dicha variable.
Como ocurría al usar la función *clone*, los dos hilos van a acceder a la variable a de una manera no controlada y como consecuencia de ello, se producirán resultados diferentes, por el hecho de que ambos hilos no se sincronizan en su acceso a la variable.
Por lo tanto, cuando se trabaje con hilos, es muy importante utilizar mecanismos de sincronización para el caso en que dos o más hilos tengan la necesidad de compartir datos. La biblioteca *<pthreads.h>* también incorpora algunas funciones para gestionar esta sincronización.