Creación de Procesos: Fork

Introducción: Hilos, tareas

Antes de comenzar con fork, repasemos un poco procesos e hilos  . Dos conceptos muy parecidos y relacionados, pero con un conjunto de pequeñas diferencias.

Si queremos que nuestro programa empiece a ejecutar varias cosas **“a la vez”**, tenemos dos opciones. Por un lado podemos crear un nuevo proceso y por otro lado podemos crear uno o varios hilos de ejecución (threads).

En realidad una computadora, salvo que tenga varias CPU’s, no ejecutará varias tareas a la vez esto se refiere a que el sistema operativo, es este caso Linux, irá ejecutando los threads según la política del mismo, siendo lo mas usual ejecutar porciones de código de procesos alternadamente (multitarea), haciendolo tan rápido que dan la sensación de simultaneidad.

Procesos

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

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

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

Cuando ejecutamos un comando en el shell, sus instrucciones se copian en memoria RAM (`¡repasar ciclo de instrucción visto en orga!`) del sistema para ser ejecutadas. Cuando las instrucciones se ejecutaron en su totalidad, el proceso es borrado de la memoria del sistema, dejándola libre para que más programas se puedan ejecutar a la vez.

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

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

Threads

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

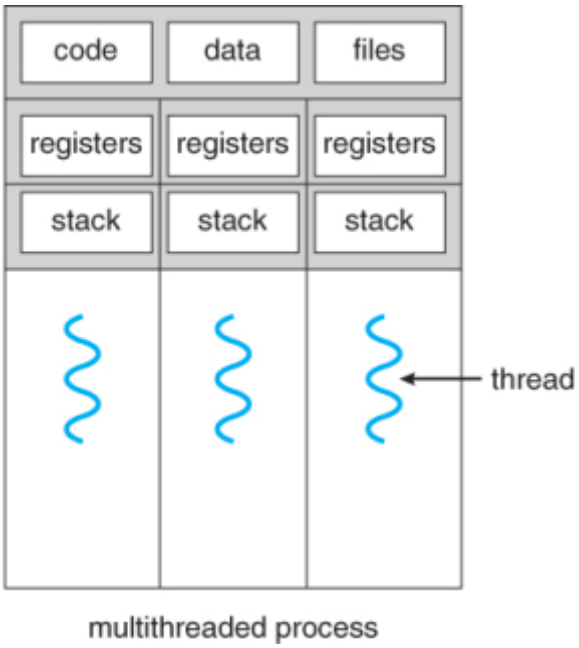
La diferencia más significativa entre los procesos y los hilos, es que los primeros son típicamente independientes, llevan bastante información de estados, e interactúan sólo a través de mecanismos de comunicación dados por el sistema.

Por otra parte, los hilos generalmente comparten la memoria, es decir, acceden a las mismas variables globales o dinámicas, por lo que no necesitan costosos mecanismos de comunicación para sincronizarse (IPC 🤦)). Por ejemplo un hilo podría encarguese de la interfaz gráfica (iconos, botones, ventanas), mientras que otro hace una larga operación internamente. De esta manera el programa responde más ágilmente a la interacción con el usuario.

En sistemas operativos que proveen facilidades para el uso de hilos, es más rápido cambiar de un hilo a otro dentro del mismo proceso, que cambiar de un proceso a otro (recordar qué es Context switch). Es posible que los hilos requieran de operaciones atómicas para impedir que los datos comunes sean cambiados o leídos mientras estén siendo modificados. El descuido de esto puede generar “estancamiento” mmm... me suena a algo que ya vimos 😛 .

La tabla de abajo resume algunas diferencias entre procesos e hilos,

Procesos	**Threads**
Administrador por el SO	Manejados por los procesos
Independientes de otros procesos	Relacionados con otros hilos del mismo proceso
Memoria privada, se necesitan mecanísmos de comunicación para compartir información	Memoria compartida con el resto de los hilos que forman el proceso



multithreaded process

Creación de Procesos: Fork y clone

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

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

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

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

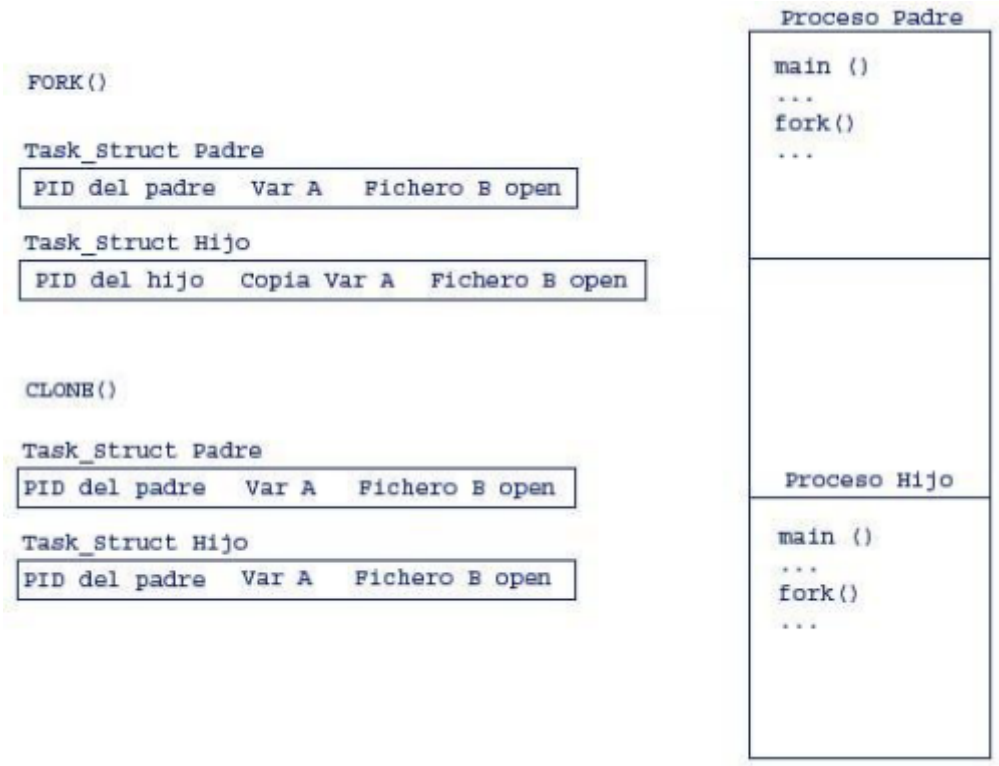
Fork: En el momento de la llamada a fork el proceso hijo:

- Es una copia exacta del padre excepto el PID.
- Tiene las mismas variables y archivos abiertos.
- Las variables son independientes (padre e hijo tienen distintas memorias).
- Los archivos son compartidos (heredan el descriptor).

Clone: Permite especificar qué queremos que compartan padre e hijo.

- Espacio de direccionamiento
- Información de control del sistema de archivos (file system)
- Descriptores de archivos abiertos.
- Gestores de señales o PID.

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



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

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

Recomiendo leer libro: El lenguaje de programación C Brian Kernighan - Dennis Ritchie)

Fork

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

Cuando se hace un fork, se crea un nuevo task_struct a partir del task_struct del proceso padre. Al hijo se le asigna un PID propio y se le copian las variables del proceso padre. Sin embargo, vemos como en la llamada a clone el task_struct del proceso padre se copia y se deja tal cual, por lo que el hijo tendrá el mismo PID que el proceso padre y obtendrá (físicamente) las mismas variables que el proceso padre.

El proceso hijo creado es una copia del padre (mismas instrucciones, misma memoria). Lo normal es que a continuación el hijo ejecute una llamada al sistema exec (Nosotros no lo hicimos, no pasa nada o si?). En cuanto al valor devuelto por el fork, se trata de un valor numérico que depende tanto de si el fork se ha ejecutado correctamente como de si nos encontramos en el proceso padre o en el hijo.

- Si se produce algún error en la ejecución del fork, el valor devuelto es -1
- Si no se produce ningún error y nos encontramos en el proceso hijo, el fork devuelve un 0.
- Si no se produce ningún error y nos encontramos en el proceso padre, el fork devuelve el PID asignado al proceso hijo.

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

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

---

Volver

— *Mariano Vargas*

---

(98)