UF2: Procesos e hilos

1. Programación multiproceso

1.1. Caracterización de la programación concurrente, paralela y distribuida

La **concurrencia** es la propiedad por la cual los sistemas tienen la capacidad de ejecutar diferentes procesos en un mismo tiempo.

Un **proceso** es el conjunto de instrucciones que se van a ejecutar.

La programación concurrente es la que se encarga de ejecutar las tareas simultáneamente. Hay que tener en cuenta que toda ejecución simultanea debe tener unas características especiales, como, por ejemplo, la secuencia de ejecución, las comunicaciones entre los procesos y el acceso coordinado a los recursos.

Ventajas de la programación concurrente:

- Facilita el diseño orientado a objetos.
- Posibilita la compartición de recursos.
- Facilita la programación de aplicaciones en tiempo real.
- Reduce los tiempos de ejecución.

Cada programa se compone de diferentes instrucciones que pueden ser ejecutadas o no simultáneamente. **Bernstein** dividió las instrucciones en dos tipos, dependiendo de la operación realizada sobre ellas: de **lectura** y de **salida**.

Para que dos instrucciones se puedan ejecutar concurrentemente, es necesario que se cumplan estas tres **condiciones**:

- $L(S_i) \cap E(S_j) = \emptyset$
- $E(S_i) \cap L(S_j) = \emptyset$
- $E(S_i) \cap E(S_j) = \emptyset$

Veamos un **ejemplo**:

A partir del siguiente conjunto de instrucciones, hay que indicar las que se pueden ejecutar concurrentemente:

Instrucción 1 (I1) → a = x + z
Instrucción 2 (I2) → x = b – 10
Instrucción 3 (I3) → y = b + c

En primer lugar es necesario formar 2 conjuntos de instrucciones:

- **Conjunto de lectura** → Formado por instrucciones que cuentan con variables a las que se accede en modo lectura durante su ejecución.

  L(I1) = {x,z}
  L(I2) = {b}
  L(I3) = {b,c}

- **Conjunto de escritura** →Formado por instrucciones que cuentan con variables a las que se accede en modo escritura durante su ejecución.

  E(I1) = {a}
  E(I2) = {x}
  E(I3) = {y}

Para que dos conjuntos se puedan ejecutar concurrentemente se deben cumplir estas 3 condiciones:

- $L(Si) \cap E(Sj) = \emptyset$
- $E(Si) \cap L(Sj) = \emptyset$
- $E(Si) \cap E(Sj) = \emptyset$

Aplicando estas tres condiciones a cada pareja de instrucciones podemos descubrir cuales se pueden ejecutar concurrentemente y cuáles no.

Conjunto de I1 e I2→ $L(S1) \cap E(S2) \neq \emptyset$

$E(S1) \cap L(S2) = \emptyset$

$E(S1) \cap E(S2) = \emptyset$

Conjunto de I1 e I3→ $L(S1) \cap E(S3) = \emptyset$

$E(S1) \cap L(S3) = \emptyset$

$E(S1) \cap E(S3) = \emptyset$

Conjunto de I2 e I3→ $L(S2) \cap E(S3) = \emptyset$

$E(S2) \cap L(S3) = \emptyset$

$E(S2) \cap E(S3) = \emptyset$

Por lo tanto, las instrucciones que se pueden ejecutar concurrentemente son:

I1 e I3
I2 e I3

- **Problemas de la programación concurrente**

- **Exclusión mutua:** es el resultado de que dos procesos intenten acceder a la misma variable. Esto puede producir inconsistencia, puesto que un proceso puede estar actualizando una variable, mientras que otro proceso está leyendo la misma. Para evitar este problema, en la programación concurrente se utiliza la región crítica, que controla el número de procesos que se encuentra utilizando el recurso.

- **Abrazo mortal:** dos procesos se quedan bloqueados porque están esperando a los recursos que tiene el otro proceso. También es denominado DeadLock o interbloqueo.
- **Inanición:** un proceso se queda esperando a un recurso compartido, que siempre se le deniega. Sin este recurso el proceso no puede finalizar.

1.2. Identificación de las diferencias entre los paradigmas de programación paralela y distribuida

- **Programación paralela**

Es una técnica de programación en la que **muchas instrucciones se ejecutan simultáneamente**. Este tipo de programación permite dar solución a problemas de grandes dimensiones, para ello, divide en pequeños problemas uno más grande. Estas partes se pueden resolver de forma paralela, por lo que podrán ser ejecutadas a la vez que otras. Esta técnica se utiliza en equipos multiprocesadores en los que cada uno de ellos es el encargado de resolver una tarea. Todos los procesos que se encuentran en ejecución deben estar comunicados entre sí.

Ventajas de la programación paralela:

- Permite la ejecución de tareas de manera simultánea.
- Permite resolver problemas complejos.
- Disminuye el tiempo en ejecución.

Inconvenientes de la programación paralela:

- Mayor dificultad en la programación.
- Mayor complejidad en el acceso a los datos.

Existen diferentes mecanismos de intercambio de información, denominados **modelos**. Según su forma se pueden **clasificar** en:

- **Modelo de memoria compartida**

Este tipo de memoria permite ser accedida por múltiples programas, por lo que es un modo eficaz de transferencia de datos. Esto se consigue creando un espacio de acceso común por parte de cada uno de los procesos en la memoria RAM.

Hay que tener en cuenta que uno de los mayores problemas se da cuando existe un punto de acceso en común, como son las **secciones críticas**.

Es necesario establecer una serie de controles dentro de un punto de acceso a datos común, puesto que estos procesos intentarán realizar lectura y escritura de datos a la vez, dando lugar a datos incoherentes. **Existen diversos mecanismos de control** como son: semáforos, tuberías, monitores, etc. Estos permitirán el acceso de los hilos de manera individual, asegurando que las operaciones realizadas son de forma **atómica**, es decir, hasta que un hilo no termina, otro no puede acceder al mismo espacio de memoria.

- **Modelo de paso de mensaje**

Es el mecanismo más utilizado en la programación orientada a objetos. Cada proceso tiene definidas sus funciones y métodos, por lo que cada uno de ellos ejecuta individualmente las tareas y, si necesita datos de otro, realiza una petición de los resultados que necesita al propietario.

El principal inconveniente es que todos los procesos deben tener implementados métodos que puedan interpretar este intercambio de mensajes.

- **Programación distribuida**

Este tipo de técnica de programación permite que un conjunto de máquinas separadas físicamente, que están interconectadas entre sí por una red de comunicaciones, trabaje como una sola para dar solución a un problema. Cada una procesa de manera independiente una parte del total. Este conjunto de máquinas que operan interconectadas se conoce como **grid**.

Esta programación se utiliza en la computación de altas prestaciones, en la que las máquinas son configuradas específicamente para cada tipo de escenario y se dan necesidades de computaciones de manera remota. Está enfocada a sistemas distribuidos cliente-servidor que requieren de escalabilidad y con una gran capacidad de tolerancia a fallos, puesto que cada uno opera de manera independiente y en caso de fallo no repercute en el de los demás. La comunicación entre estas máquinas se realiza mediante la llamada de procedimientos remotos (**RPC**), invocación remota de objetos (**RMI**) y **sockets**.

Características de los sistemas distribuidos:

- **Capacidad de balanceo:** las propias máquinas son capaces de realizar una asignación de recursos concreta para cada proceso, de manera que otro proceso pueda hacer uso de una mayor cantidad de recursos si lo requiere.
- **Alta disponibilidad:** es una de las características más usadas, que permite a este tipo de programación una flexibilidad enorme, en caso de que haya un fallo automáticamente los servicios son asignados en otros servidores.

De la misma manera que si la previsión de recursos necesarios se queda justa, es posible que de manera automática se añada una nueva máquina con las mismas características, para poder hacer frente a la computación de distintos procesos.

Por el contrario, este tipo de programación también cuenta con algunos inconvenientes como puede ser un aumento de la complejidad, ya que es necesario un determinado tipo de software. Asimismo, se incrementa el riesgo de fallos en la seguridad, como pueden ser los ataques de denegación de servicios con sobrecargas de peticiones remotas al servidor.

Ventajas de la programación distribuida:

- Permite escalabilidad, capacidad de crecimiento.
- Permite la compartición de recursos y datos.
- Mayor flexibilidad.
- Alta disponibilidad.

Inconvenientes de la programación distribuida:

- Pérdida de mensajes.
- Está expuesta a diferentes ataques para vulnerar su seguridad.

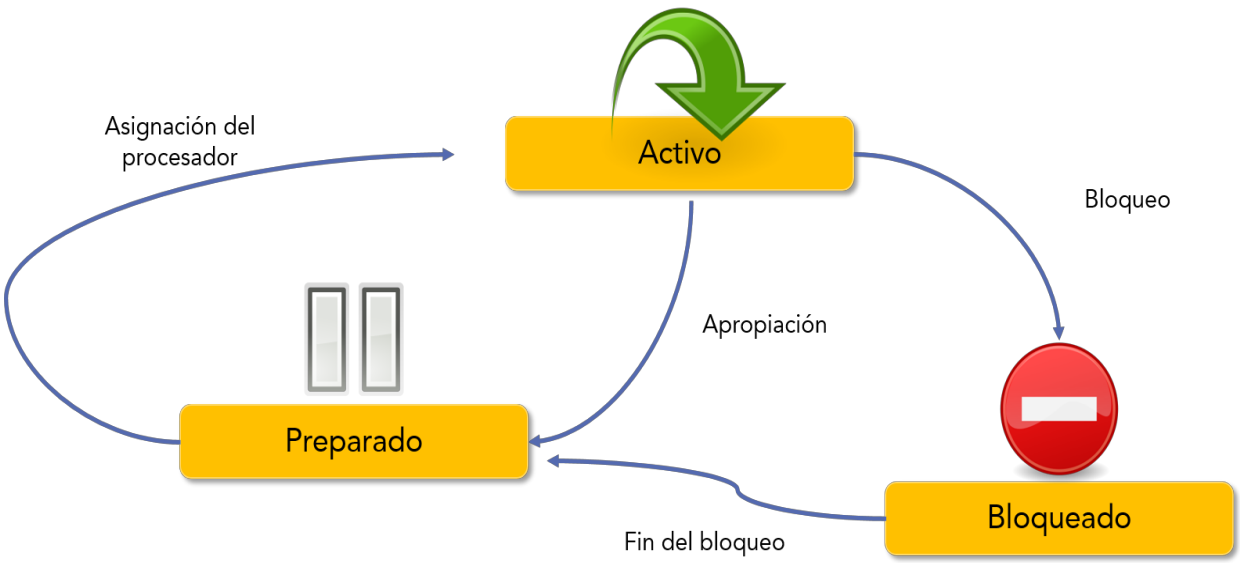
1.3. Identificación de los estados de un proceso

En primer lugar, es necesario definir qué es un **proceso**. Un proceso es un **conjunto de instrucciones que se van a ejecutar dentro de la CPU**.

Todo proceso está caracterizado por un indicador que define la situación o estado en el que se encuentra. Al menos existen **tres estados** diferentes:

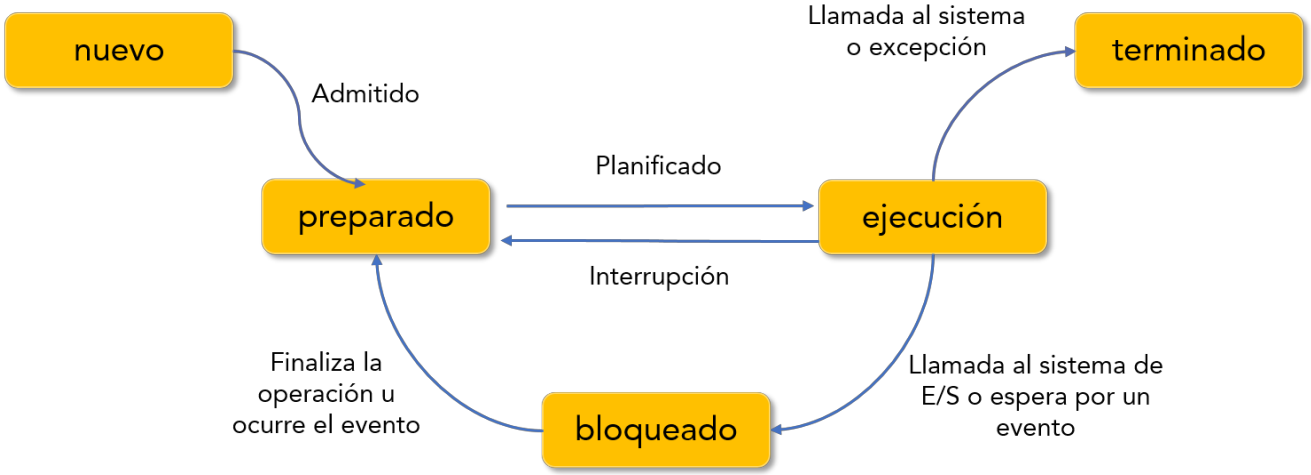
- **Activo o en ejecución:** aquellos procesos que han sido asignados para ejecutarse en el procesador.
- **Bloqueado:** aquellos procesos que han interrumpido su ejecución, y, por lo tanto, se encuentran actualmente a la espera de que termine la operación que los ha dejado bloqueados.
- **Preparado:** aquellos procesos que se encuentran disponibles para entrar a la CPU y ejecutarse.

Durante la vida de un proceso, se realizan transiciones entre los diferentes estados. En el siguiente diagrama se muestran las distintas **transiciones permitidas**:



- **Transición bloqueo:** cuando un proceso se está ejecutando y produce una llamada al sistema, debe quedarse como bloqueado para evitar consumir CPU. Por ejemplo, cuando un proceso requiere información y pide que se lean datos. En este caso, debe esperar a que se complete esa operación.
- **Transición apropiación:** cuando un proceso se encuentra en ejecución y el gestor de procesos indica que debe dejar de ejecutarse. En este caso, el proceso sale de la CPU hasta que puede volver a estar activo. En el próximo apartado se verán las diferentes técnicas por las cuales un proceso realiza esta transición.
- **Transición asignación:** cuando un proceso entra a ejecutarse en la CPU.
- **Transición fin de bloque:** cuando un proceso está esperando a que acabe la operación por la cual ha pasado a estar en el estado de bloqueo y pasa directamente a seguir su ejecución.

En **Unix** existen más estados para los procesos que los que se han explicado anteriormente. En este diagrama se pueden observar las transiciones entre ellos:



Cabe destacar que existen varios **estados nuevos**, como pueden ser:

- **Nuevo:** aquellos procesos que aún no han sido elegidos para iniciar su procesamiento.
- **Terminado:** aquellos procesos que han finalizado su ejecución.
- **Zombies:** aquellos procesos que han finalizado su ejecución, pero no han liberado los recursos que han utilizado.

El sistema operativo debe agrupar la información de todos los procesos del sistema. Esta información se encuentra en el bloque de control del proceso **(BCP),** que se crea a la vez que el proceso. Cuando el proceso es eliminado se borra también toda esta información.

Esta información se refiere al identificador del proceso, su estado, prioridad, recursos y permisos asignados.

1.4. Ejecutables. Procesos. Servicios

- **Ejecutables**

Son archivos binarios que contienen un conjunto de instrucciones en código fuente que el compilador ha traducido a lenguaje máquina. Este tipo de archivos suele contener instrucciones de llamada a las funciones del sistema operativo.

Este tipo de archivos son incomprensibles para el ser humano. Algunos de los más comunes son **.bat, .com, .exe y .bin**.

- **Procesos**

Conjunto de instrucciones que ejecutará el microprocesador, es lo que se entiende como un **programa en ejecución**. Los procesos son gestionados por el sistema operativo, que es el encargado de crearlos y destruirlos. Requieren de unos recursos de memoria que reservan para su ejecución.

En los sistemas operativos multihilo es posible crear hilos y procesos. La diferencia reside en que un proceso solamente puede crear hilos para sí mismo, y en que dichos hilos comparten toda la memoria reservada para el proceso.

Los procesos están caracterizados por su estado de ejecución en un momento determinado. Estos procesos son el valor de los registros de la CPU para dicho programa.

- **Servicios**

Es un tipo de proceso informático que posee unas características especiales, ya que se ejecutan en segundo plano y no son controlados por el usuario. Pueden permanecer de manera continuada en ejecución dentro del sistema y carecen de interfaz gráfica.

La mayor parte de estos servicios están destinados a realizar tareas de mantenimiento del sistema operativo de forma periódica.

1.5. Caracterización de los hilos y relación con los procesos

El término **hilo** es un concepto reciente y que se podría definir como una **ejecución que forma parte de un proceso**.

Un proceso puede contener un hilo o múltiples hilos de ejecución. Un sistema operativo **monohilo** (un solo hilo de ejecución por proceso) constará de una pila de proceso y una pila de núcleo. En un sistema **multihilo** cada hilo consta de su propia pila de hilo, un bloque de control por cada hilo y una pila del núcleo.

Un hilo posee las siguientes **características**:

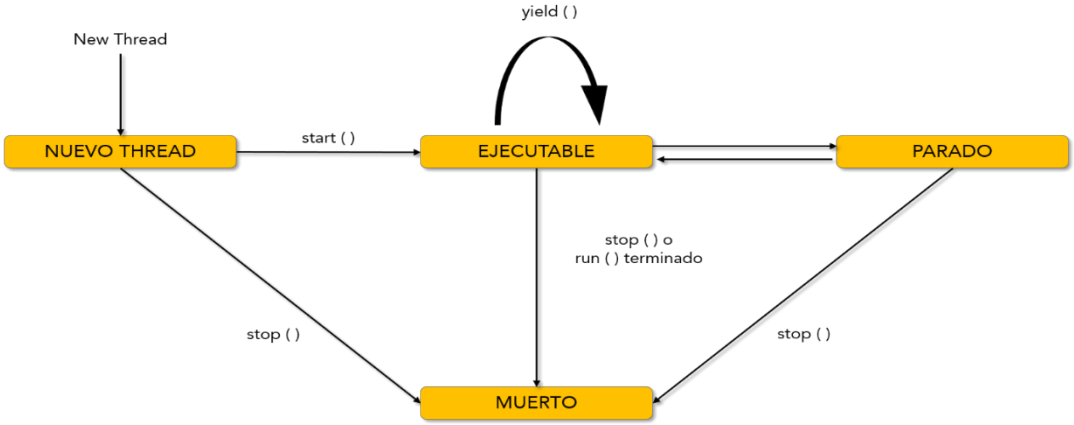
- Contador de programa.
- Juego de registros.
- Espacio de pila.

Los hilos dentro de una misma aplicación disponen de las siguientes **zonas comunes**:

- Sección de código.
- Sección de datos.
- Los recursos del sistema operativo.

Un hilo, al igual que un proceso, está caracterizado además por su estado. Los estados en los que se puede encontrar un hilo son:

- Ejecución.
- Preparado.
- Bloqueado.



Para realizar la transición de un estado a otro existen **cuatro operaciones** básicas:

- **Creación:** cuando se crea un nuevo proceso automáticamente se crea un hilo asociado a dicho proceso.
- **Bloqueo:** un hilo debe esperar a que un evento indique su ejecución. De esta forma el procesador ejecutará otro proceso que esté preparado.
- **Desbloqueo:** evento que indica la salida de un proceso del estado de bloqueo y pasa a la cola de preparados.
- **Terminación:** cuando un hilo finaliza, se liberan los recursos que estaba usando.

En la mayor parte de los casos, los hilos realizan las mismas operaciones que los procesos. **La existencia de un hilo está vinculada a un proceso**. Los hilos permiten que la comunicación entre los distintos procesos sea más rápida, además, se tarda un tiempo menor en crear un nuevo hilo que un nuevo proceso.

1.6. Programación de aplicaciones multiproceso

A la hora de ejecutar una aplicación, el sistema operativo se encarga de asignar en la CPU las distintas tareas que se deben realizar. En muchas ocasiones existen aplicaciones que necesitan que estas tareas se realicen de forma simultánea. De esta necesidad sale el concepto de **multiproceso**, que consiste en el **uso de dos o más procesadores para la ejecución de varios procesos**.

Gracias a esto es posible **mantener en ejecución** distintos programas a la vez.

Existen diferentes mecanismos a la hora de programar este tipo de aplicaciones multiproceso y el más utilizado es la realización, en segundo plano, de tareas que no se encuentran en la memoria principal. Esto permite asignar un mayor número de recursos a aquellos procesos que lo necesitan.

Estas tareas que no se están ejecutando en primer plano suelen ser asignadas a subprocesos del sistema operativo que son iniciados y detenidos de forma controlada. Esto aumenta la velocidad de lectura y escritura de datos para generar una respuesta a la solicitud de un usuario.

1.7. Sincronización y comunicación entre procesos

Tal y como se ha explicado en el apartado anterior, es posible trabajar con procesos que se ejecutan de manera concurrente. Esto ofrece una **mayor velocidad y mejora la comunicación** con la interfaz de usuario. Es muy importante tener en cuenta que se trata de operaciones con datos que acceden, en muchas ocasiones, a los mismos espacios de memoria. Es necesario que exista una comunicación entre procesos para no bloquear el acceso de otros.

La **sincronización entre procesos** es fundamental para conseguir una correcta funcionalidad y para ello se suele hacer uso de señales que definen el estado que tendrá cada uno de esos procesos durante un periodo de tiempo determinado. Estas señales son enviadas de forma independiente a cada proceso y cada señal define un comportamiento.

Las más **comunes** son:

- **SIGKILL:** se usa para terminar con un proceso.
- **SLEEP:** suspende el proceso durante una cantidad de segundos.
- **KILL:** envía una señal.
- **SIGINT:** se envía una señal a todos los procesos cuando se pulsan las teclas Ctrl+C. Se utiliza para interrumpir la ejecución de un programa.

La comunicación entre procesos se consigue utilizando los mecanismos ya detallados en apartados anteriores y los más importantes son: **tuberías, paso de mensajes, monitores o buzones**, entre otros. Estos asegurarán que el intercambio de datos sea correcto y no existan problemas en la persistencia de datos. Estos procesos se pueden bloquear a la espera de eventos y desbloquear de la misma manera.

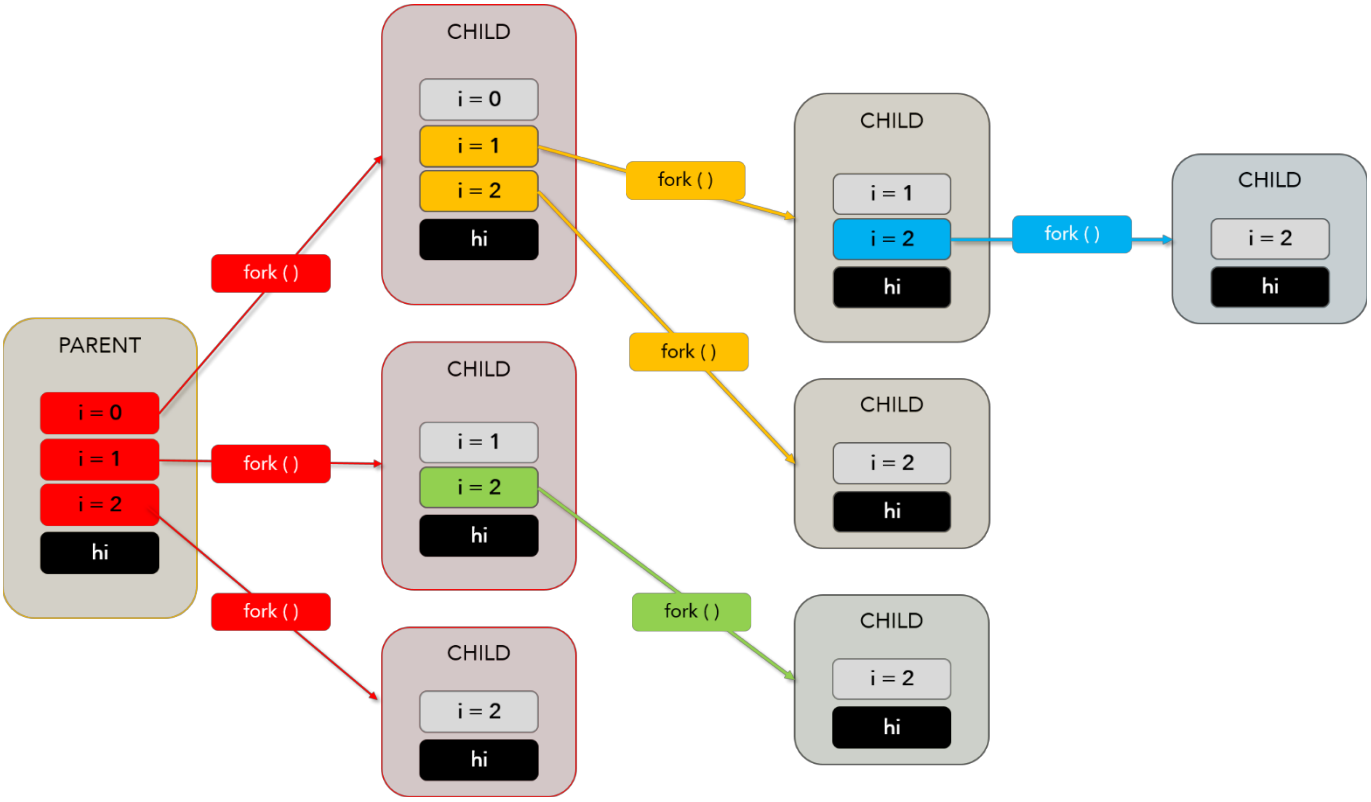
Los **hilos** agilizan las tareas que tienen los procesos, por lo que es importante considerar la posibilidad de que un hilo permanezca bloqueado de forma continuada. Como los hilos no son independientes, sino que **forman parte todos del mismo proceso**, el proceso podrá llegar a bloquearse por completo, así como todos los hilos que componen ese proceso.

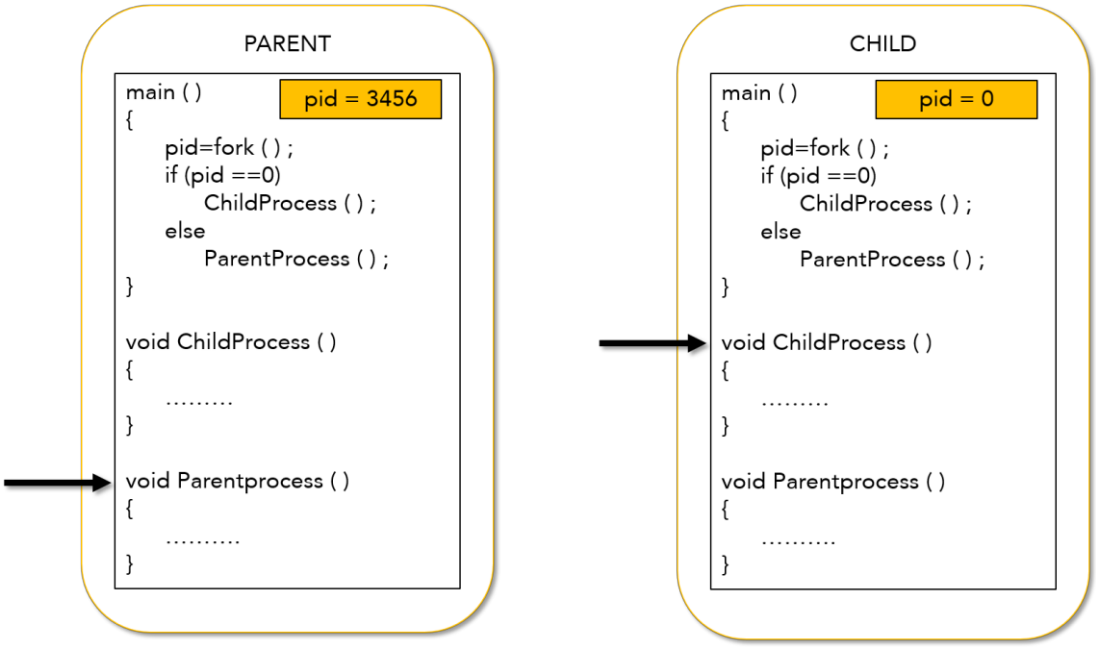
1.8. Gestión de procesos y desarrollo de aplicaciones con fines de computación paralela

La **gestión de procesos** está a cargo del sistema operativo, al igual que la creación de los mismos, en función de las acciones ejecutadas por el usuario.

Un nuevo proceso creado es el encargado de solicitar la creación de otros procesos. Estos nuevos procesos creados dependen del proceso inicial que los creó. Estos conjuntos de procesos dependientes creados reciben el nombre de **árbol de procesos**. Se caracterizan por tener un identificador único y un nombre que los distingue de los demás. El proceso creador se denomina **Padre** y el proceso creado **Hijo**. De la misma forma, estos procesos hijos pueden crear a su vez nuevos hijos. Estos procesos comparten los recursos de la CPU.

Las funciones que permiten crear un proceso son **createProcess()** para Windows y **fork()** para Linux. Son creadas en el mismo espacio de direcciones y se comunican entre sí mediante lectura y escritura.





Un proceso padre puede finalizar la ejecución de un proceso hijo en cualquier momento, aunque por lo general, es el hijo quien informa al proceso padre de que ha terminado su ejecución solicitando su terminación.

Es importante comprobar que los procesos han terminado de una forma ordenada, es decir, que los procesos hijos finalizan antes de hacerlo el padre. Para obligar a los procesos hijos a finalizar su ejecución se puede hacer uso de la orden **destroy()** que realizará una terminación en cascada de todos los procesos hijos.

En lenguajes como java ya existe un mecanismo automático denominado **garbage collector** (recolector de basura) que se encarga de liberar los recursos cuando un proceso finaliza. En otros lenguajes, un proceso que finaliza su ejecución de forma correcta liberará sus recursos al finalizar su ejecución por medio de la operación **exit()**.

El control y gestión ordenado de estos procesos es muy importante en la computación paralela. El desarrollo de aplicaciones que gestionan las vías de comunicación entre distintos procesos de forma eficiente permite incrementar mucho la productividad de un software. **Es necesario separar qué tareas requieren de un mayor número de recursos y cuáles se pueden realizar en segundo plano, permitiendo una comunicación fluida con la interfaz de usuario**. De esta forma se evita una gran cantidad de tiempo de espera por la respuesta de algunas de las peticiones generadas por los usuarios durante el uso de la aplicación.

1.9. Depuración y documentación de aplicaciones

En Linux existe el comando **ps**. Su función es mostrar la información de los procesos que se encuentran activos en el sistema.

```
prueba@PRINCIPAL:~$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
prueba      3646    3640  0 12:58 pts/11   00:00:00 bash
prueba      4178    3646  0 13:12 pts/11   00:00:00 ps -f
```