La gestión de procesos

Teodor Jové Lagunas
Josep Lluís Marzo i Lázaro
Enric Morancho Llena
José Ramón Herrero Zaragoza

PID_00214802

Universitat Oberta de Catalunya

Índice

Introducción

Este módulo didáctico se centra en el **estudio de la vida de los procesos** desde el punto de vista de las llamadas al sistema que permiten manipularlos. Así pues, en lugar de ver la creación de ficheros ejecutables y los mecanismos que tiene el sistema operativo para comunicarse con los programas (las llamadas al sistema), como ya hemos hecho en esta asignatura, nos centraremos en el **estudio de las operaciones que permiten gestionar los procesos**, concretamente en las que permiten crearlos, destruirlos y modificarlos. Antes de nada, sin embargo, deberemos detenernos brevemente en el concepto de proceso, ya visto en esta asignatura, y en la gestión interna de los procesos que lleva a cabo el sistema operativo (SO).

Objetivos

Los materiales didácticos de este módulo contienen las herramientas necesarias para alcanzar los objetivos siguientes:

1. Entender el concepto de proceso y de flujo de ejecución (*thread*) como objetos gestionados por el SO.

2. Conocer las diferentes partes que componen un proceso y ver cómo éste se representa en el interior del sistema operativo.

3. Entender los estados en los que puede estar un proceso y los motivos por los que un proceso puede cambiar de estado.

4. Saber qué es el ciclo de vida de los procesos y conocer las relaciones entre procesos.

5. Comprender el concepto de herencia y ver qué repercusiones tiene la herencia en la creación de procesos.

6. Conocer las diferentes posibilidades que nos ofrece el hecho de poder cambiar algunos de los elementos que componen los procesos, sobre todo en relación con los redireccionamientos.

7. Conocer las llamadas básicas de gestión de procesos de UNIX y de Windows.

8. Conocer el funcionamiento básico de los *threads* POSIX (*pthreads*).

1. El proceso: un vistazo desde el interior del sistema

Un **proceso** es básicamente un entorno formado por todos los recursos necesarios para poder ejecutar programas. Desde el punto de vista del SO, un proceso es un objeto más que se debe gestionar y al que se ha de dar servicio.

Para gestionar los procesos y darles servicios, el sistema operativo debe proporcionar herramientas o tiene que llevar a cabo acciones que permitan conseguir los objetivos siguientes:

- Crear y eliminar procesos.
- Garantizar que los procesos dispongan de los recursos necesarios para avanzar en su ejecución.
- Actuar en casos excepcionales[1] durante la ejecución del proceso.
- Proporcionar los mecanismos necesarios para que los procesos se comuniquen, ya sea para intercambiar información o para sincronizarse durante la ejecución.
- Mantener estadísticas sobre el funcionamiento de los procesos.
- Temporalizar la ejecución de un proceso: hacer que un proceso se ejecute cada cierto tiempo.
- Otros servicios misceláneos.

(1)Consideramos casos excepcionales sucesos como las interrupciones, los errores, etc.

En este apartado haremos una introducción a la gestión que el SO lleva a cabo sobre los procesos. Veremos las estructuras de datos que representan internamente un proceso y cómo varios procesos pueden ejecutarse de manera concurrente en un único procesador.

1.1. Los elementos de un proceso y su representación

Tal como hemos visto, el SO construye los procesos de acuerdo con un conjunto de elementos que son necesarios para la ejecución de un programa. Para poder gestionar este conjunto de recursos como un todo, el sistema reúne in-

(2)El término *process control block* o PCB es el equivalente inglés del término *bloque de control de procesos*.

formación de todos ellos en una estructura de datos denominada **bloque de control de procesos**[2] o **PCB**. A cada proceso le corresponde su propio PCB. Los campos más importantes que configuran los PCB son los siguientes:

1) El identificador del proceso (*PID*[3]). Es el código único que identifica de manera biunívoca cada uno de los diferentes procesos que hay en ejecución en el sistema y que, por lo tanto, debe ser distinguido de manera individual. Este identificador normalmente es numérico y es único durante toda la vida del SO.

(3)PID es el acrónimo del término inglés *process identifier*.

2) El estado del proceso. Este campo indica el estado del proceso en el momento actual, dentro de unas posibilidades determinadas (*run*, *ready*, *wait* –*blocked*–, etc.).

Ved también

Podéis ver los estados de un proceso y su significado en el subapartado 1.3 de este módulo didáctico.

3) El contador del programa. Este campo es fundamental porque "señala" la instrucción que estaba a punto de ser ejecutada justo en el momento en el que se produce una interrupción. Cuando el proceso puede continuar, lo hace exactamente en este punto.

4) Los registros arquitectónicos de la UCP. Son registros utilizados por todos los procesos. Por lo tanto, después de una interrupción no basta con continuar la ejecución del proceso en el punto donde se dejó, sino que también se debe encontrar un entorno idéntico al que se tenía antes de producirse la interrupción. Para tener esta posibilidad, también hemos de guardar el valor de los registros.

5) El estado de la memoria. Es difícil generalizar sobre lo que necesita cada sistema operativo para tener toda la información relativa a la memoria de cada proceso, pero podemos dar algunas ideas, como la cantidad de memoria asignada, el lugar donde se encuentra, el tipo de gestión que se hace de él, las protecciones de cada parte, las comparticiones, etc.

6) Contabilidad y estadísticas. El sistema operativo obtiene para cada proceso una serie de información relativa al comportamiento de cada usuario. Esta información tiene un gran valor para los administradores de sistema, pero no lo tiene mucho para los usuarios o los programadores.

7) El estado de los dispositivos de entrada/salida. Los dispositivos de entrada/salida asignados, las solicitudes pendientes, los ficheros abiertos, etc. también forman parte del entorno de ejecución.

8) El dominio de protección. Este campo contiene información de los dominios a los que pertenece el proceso y de los derechos que tienen asociados.

Ved también

Podéis ver los dominios de protección en el subapartado 4.1 del módulo didáctico "El sistema de ficheros".

9) La planificación de la UCP. En este campo se agrupa información relativa sobre cómo el proceso accede al procesador en concurrencia con los otros procesos.

10) Otras informaciones. Cada sistema operativo mantiene otras informaciones particulares en función de diferentes aspectos, como el fabricante del sistema operativo, el tipo de orientación[4] y también del tipo de explotación[5].

[4] El trabajo en tiempo real, las comunicaciones, etc.

[5] Uso privado, uso público, etc.

De acuerdo con los PCB, el sistema gestiona la ejecución de los programas contenidos en la memoria de los procesos. En los subapartados siguientes analizamos los principales componentes de esta gestión teniendo en cuenta las necesidades de un sistema multiprogramado.

1.2. La ejecución concurrente

En la figura 1 podemos ver la ejecución concurrente de un conjunto de procesos (P1, P2 y P3) sobre un sistema monoprocesador multiprogramado. En función de la escala de tiempo con la que examinemos la evolución de los procesos en el sistema podemos tener visiones diferentes sobre ellos.

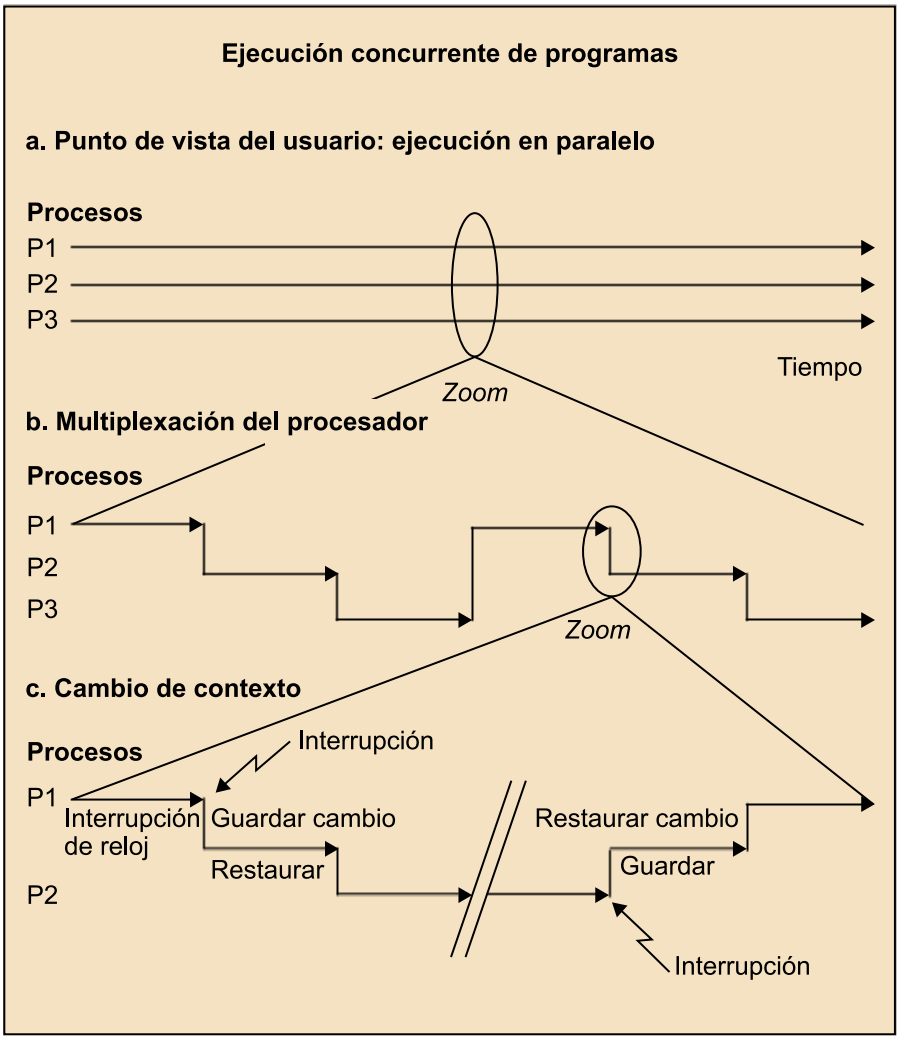


Figura 1

1) La primera escala de tiempo, que se ve en la figura 1a, corresponde al punto de vista del usuario. Aquí la ejecución de los tres procesos parece que se efectúa en paralelo, aparentemente cada proceso utiliza siempre el procesador. Ésta es la impresión que se da en los sistemas multiusuario: que cada usuario tiene una buena interactividad con el sistema y que el usuario se encuentra solo trabajando delante de la máquina. Pero, en realidad, en un sistema que trabaja en modalidad de tiempo compartido eso no es cierto.

2) Si observamos el comportamiento de los procesos a una escala más pequeña de tiempo, vemos que hay una multiplexación del procesador en el tiempo (podéis ver la figura 1 b). Ampliando la figura 1a (haciendo un *zoom)*, podemos observar que no hay una ejecución real en paralelo de los procesos; sólo uno de ellos está en ejecución real. Para conseguir el efecto de ejecución concurrente se realiza una conmutación entre los procesos que se reparten el tiempo del procesador. Estas conmutaciones se denominan **cambio de contexto**.

3) Finalmente, ampliando más la imagen, es decir, a una escala de tiempo todavía más pequeña, podemos ver el detalle de las transiciones entre procesos o cambio de contexto (podéis ver la figura 1c). Podemos observar que debe aparecer un fragmento de código nuevo para gestionar la interrupción que provoca el cambio. Para hacerlo, se han de llevar a cabo las operaciones siguientes:

- Guardar el estado del procesador tal como lo tenía el proceso P1 en el instante en el que se ha producido la interrupción sobre su PCB.
- Localizar el PCB del proceso P2.
- Restaurar el estado del procesador guardado en el PCB del segundo proceso.

El estado del procesador en un instante concreto está formado por los valores contenidos en cada uno de los registros del lenguaje máquina, por el puntero de pila, por el contador de programa y, en general, por toda la información que configura un proceso. Los valores concretos de este conjunto de registros en un instante de la vida de un proceso se denominan **contexto del proceso**.

1.3. Los estados de un proceso

En un sistema multiprogramado, con muchos procesos y un procesador, como describimos en el subapartado anterior, en un momento determinado sólo puede haber un proceso en ejecución; el resto de los procesos pueden estar esperando su turno para acceder al procesador o pueden estar esperando la finalización de una operación de entrada/salida. Esta diversidad de situaciones se puede representar con un diagrama de estados como el de la figura 2, donde los nodos representan los estados en los que los procesos pueden estar, y los arcos son las acciones que hacen que un proceso cambie de estado.

Cuando el sistema acaba de crear un proceso, este proceso se encuentra en el estado inicial (1), el **estado** *ready*. El sistema puede salir de este estado por las dos causas siguientes:

El estado ready

Ready significa: "Lo tengo todo a punto y estoy preparado para recibir la CPU y trabajar".

1) Un acontecimiento externo al propio proceso, que puede ser debido a la acción de otro proceso mediante una señal de software, provoca la finalización del proceso (2).

2) El sistema operativo le asigna la UCP (3) y el proceso empieza a ejecutar sus instrucciones y pasa al **estado** *run*.

Del estado *run* se puede salir por tres motivos:

1) El proceso ejecuta la última línea de código y acaba (4).

2) El proceso ha de esperar un acontecimiento externo. El ejemplo más normal es cuando se pide una operación de entrada/salida[6] (5) y el proceso espera que sea servida. En este caso, el proceso pasa al **estado** *wait* (*blocked*) y se espera hasta que la petición se haya servido.

[6]Por ejemplo, una entrada de información por teclado.

3) Cuando el sistema operativo trabaja en la modalidad de tiempo compartido, si el proceso supera la cuota máxima de tiempo de uso de UCP que tiene asignada (6), deja el procesador y vuelve al estado *ready*.

Ved también

Podéis ver la ejecución de procesos en la modalidad de tiempo compartido en el subapartado 2.3 del módulo didáctico *Introducción a los sistemas operativos*.

Cuando un proceso sale del estado *run* para pasar a *ready* o *wait*, se produce un cambio de contexto, tal como hemos mencionado en el subapartado anterior.

Diagrama de estados

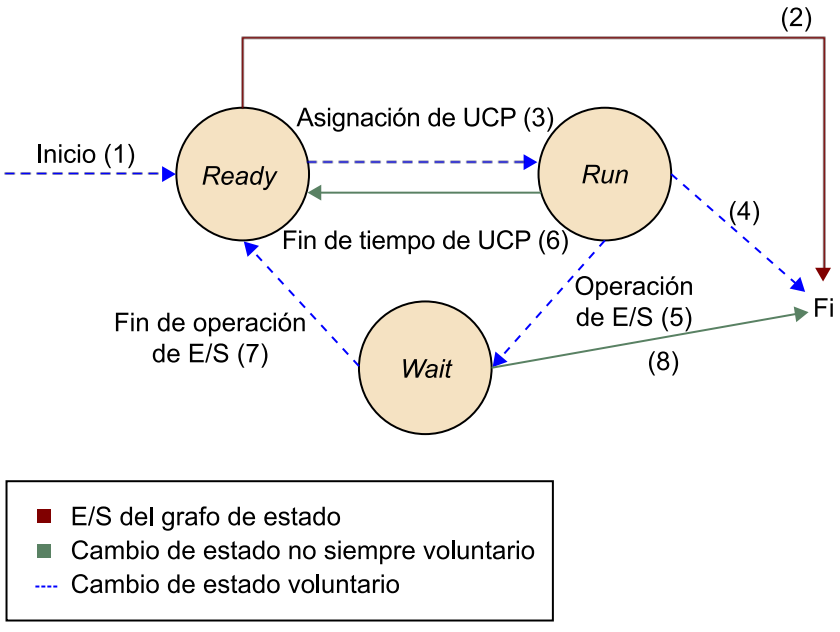


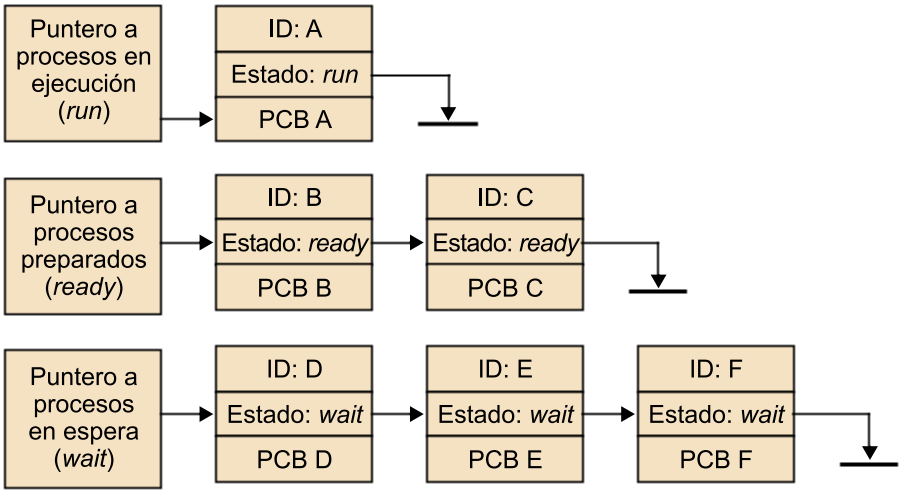
Figura 2

Finalmente, los procesos tienen dos destinos posibles cuando salen del estado *wait*:

1) Uno, hacia el estado *ready*, que es cuando finaliza la operación por la que esperaban (7). Dicho de otra manera, cuando el proceso tiene bastante información y puede continuar. Por ejemplo, si el proceso estaba pendiente de una operación de entrada/salida y ya ha llegado la información esperada porque el usuario ha pulsado una tecla.

2) Otro, hacia la finalización del proceso (8), debido a un acontecimiento externo al propio proceso, como sucedía en el estado *ready*.

Para saber qué hace cada uno de los procesos y así poder controlar sus recursos, el sistema operativo mantiene unas colas de procesos en función de su estado. Así, el sistema tiene una cola de procesos preparados (*ready*) y una de procesos en estado de espera (*wait*). No podemos hablar de una cola de procesos en ejecución, ya que en entornos monoprocesadores sólo hay un proceso en ejecución. De esta manera el procedimiento del núcleo del SO encargado de la planificación del procesador puede examinar la lista de procesos preparados con el fin de asignar el procesador al proceso que más convenga.



ID: identificador de procesos

Figura 3

2. El ciclo de vida de un proceso

Como hemos visto en esta asignatura, los procesos son uno más de los objetos que gestiona el SO. A diferencia de muchos otros objetos, los procesos son dinámicos y suelen tener un tiempo de vida limitado[7].

(7)Los procesos se crean, interactúan con el sistema y mueren.

En este apartado analizaremos el ciclo de vida de los procesos y las operaciones con las que se relacionan.

Las **diferentes etapas de la vida de un proceso** son las siguientes:

1) Creación, **nacimiento** o **inicio**. En esta etapa se asignan y se inicializan los recursos necesarios para crear un proceso nuevo.

2) Desarrollo. Una vez creados, los procesos evolucionan a partir de la ejecución del programa que contienen. Este desarrollo puede llevarlos a modificar los recursos con los que se han constituido inicialmente.

3) Destrucción, **muerte** o **finalización**. Una vez acabado el trabajo que especifica la aplicación que se ejecuta en el marco del proceso, el SO destruye el proceso y libera los recursos que se le habían asignado.

En este apartado analizaremos el ciclo de vida del proceso. Para hacerlo, estudiaremos primero los procesos de creación y destrucción de un proceso, a continuación las relaciones que hay entre estas dos acciones y, finalmente, veremos algunas de las modificaciones que se pueden hacer sobre el entorno que constituye un proceso durante su existencia.

2.1. La creación y la destrucción de procesos

Los procesos son elementos dinámicos que se crean, operan durante un intervalo de tiempo y se destruyen. El sistema operativo es el encargado de proporcionar el conjunto de llamadas necesarias para llevar a cabo todas estas acciones. En este subapartado analizaremos las operaciones de creación y destrucción de procesos.

2.1.1. Creación de procesos

La creación de un proceso nuevo es el resultado de la ejecución de una llamada al sistema del tipo *crear_proceso*, que es invocada, como todas las llamadas, por un proceso ya existente.

La ejecución de la llamada *crear_proceso* supone la **creación de un entorno de ejecución** que contiene los elementos siguientes:

- La memoria donde residirá el código del programa, los datos con los que operará el proceso y la pila utilizada para pasar parámetros o guardar variables locales a las subrutinas.

- El punto de entrada (dirección inicial) desde donde se ejecutará el programa que contenga la memoria.

- El entorno de entrada/salida con el que el proceso se comunicará con el exterior.

- Los atributos relacionados con los dominios de protección con los que el sistema operativo verificará la legalidad de las operaciones que quiera efectuar.

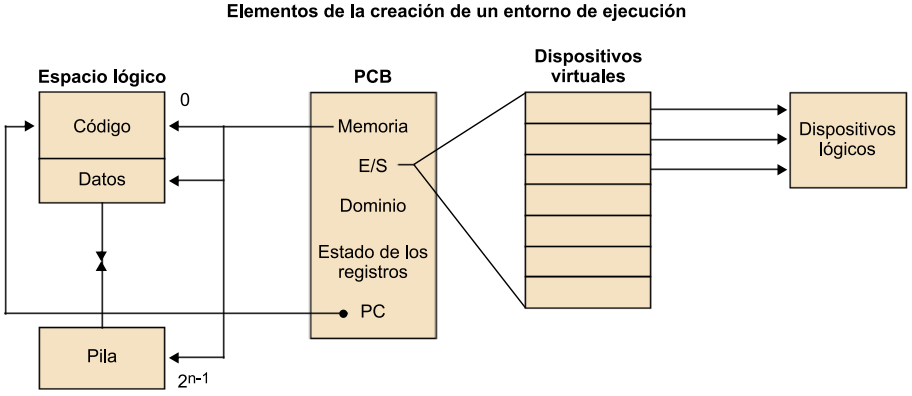


Figura 4

Cada uno de estos elementos del entorno se debe especificar con el sistema para que éste pueda crear un proceso nuevo. La especificación de estos elementos se puede llevar a cabo de dos maneras:

a) De manera explícita, con los parámetros de la llamada al sistema que crea el proceso.

b) De manera implícita, haciendo que el sistema tome unos valores por defecto.

Normalmente, los sistemas combinan las dos alternativas: obligan a especificar algunos de estos elementos mediante los parámetros de la llamada al sistema y dejan otros con valores por defecto.

Otro aspecto que hay que tener en cuenta es la relación que existe entre el entorno desde donde se ejecuta la llamada al sistema que dará lugar al nuevo proceso y el entorno nuevo que se creará. Para ver esta relación mejor partimos del hecho, ya mencionado, de que los procesos son creados por el sistema operativo a petición de otros procesos. Esta situación provoca que los procesos se puedan mirar desde el punto de vista de la descendencia, en la que los procesos tienen relaciones de parentesco como la de padre e hijo. En este ámbito podemos hablar de los conceptos de **herencia entre procesos** y de **sincronización entre procesos padre e hijo**.

Ved también

Podéis ver la herencia entre procesos en el subapartado 2.2 y la sincronización entre procesos en el subapartado 2.3 de este módulo didáctico.

Una vez creado el proceso, el sistema le da un nombre (generalmente un número dentro de un espacio lineal) con el que podrá ser referenciado en acciones de control y manipulación, tanto desde otros procesos como directamente desde el SO. Este nombre debe ser único para cada proceso, no sólo durante la vida del proceso al que haga referencia, sino durante toda la vida del sistema.

Ved también

Podéis ver los espacios lineales en el subapartado 3.2 del módulo didáctico "El sistema de ficheros".

La finalidad del hecho de tener un nombre único para cada proceso durante toda la vida del sistema es evitar confusiones y malos funcionamientos del SO a causa de la reutilización de nombres. Por ejemplo, imaginemos dos procesos (proceso A y proceso B) que colaboran y se sincronizan mediante señales gracias al hecho de que conocen sus identificadores. Si uno (el proceso B) acaba de manera imprevista y su identificador es reutilizado para crear otro proceso, entonces el proceso A se está sincronizando con un proceso que tiene un identificador B, que es el proceso con el que se había previsto una comunicación. El resultado puede ser que ninguno de los dos procesos funcione correctamente o, lo que es más problemático, que se haya abierto un agujero en la protección del sistema.

Ved también

Podéis ver los identificadores de los procesos en el subapartado 1.1 del módulo didáctico "Comunicación y sincronización".

Así, una posible estructura de la llamada *crear_proceso* podría ser:

Id_proceso = *crear_proceso*(*entorno_memoria*,*nombre_programa*, *punto_ejecución*,*entorno_E/S*, *entorno_dominio*).

Debemos comentar que en los sistemas en los que el hijo que se crea es una copia del padre*, el valor de retorno de *crear_proceso* debe ser diferente en función de si el proceso es el padre o es el hijo. Esto permite diferenciar el comportamiento del padre y del hijo, que estarán ejecutándose justo después de la llamada *crear_proceso*.

Ved también

Podéis ver los diferentes valores devueltos para la operación *crear_proceso* según el tipo de proceso en el caso UNIX en el apartado 4 de este módulo didáctico.

Puede resultarle útil a un proceso encontrar información sobre sí mismo. Por ello existe una llamada que devuelve el identificador del proceso que la invoca. Esta llamada es:

Id_proceso = quiensoy()

2.1.2. Destrucción de procesos

La destrucción de un proceso implica la destrucción del entorno que lo constituye y la liberación de los recursos que tenía asignados.

La destrucción de un proceso por parte del sistema puede ser consecuencia de alguna de las situaciones siguientes:

a) La ejecución de la llamada al sistema *destruir_proceso* específica para este motivo. En la mayoría de los sistemas esta llamada provoca la destrucción del proceso que la invoca, y no puede ser dirigida a otros procesos.

b) El mal funcionamiento del proceso destruido. Si el sistema operativo detecta que un proceso no funciona correctamente y efectúa operaciones no permitidas, lo destruye. Estas operaciones suelen estar asociadas a las excepciones provocadas por acciones como: acceder a posiciones de memoria que no se tienen asignadas, ejecutar instrucciones privilegiadas o efectuar operaciones aritméticas incorrectas como por ejemplo, una división por cero, etc.

c) El efecto lateral de la ejecución, por parte de otro proceso[8], de una llamada al sistema diferente de la llamada *destruir_proceso*, que provoca una excepción sobre el proceso que es destruido.

[(8)]El proceso que la invoca ha de tener derecho para provocar la destrucción del otro proceso.

Ahora nos centraremos exclusivamente en el primer punto: la destrucción de un proceso como resultado de la ejecución de la llamada al sistema *destruir_proceso*. Las dos últimas situaciones las veremos más adelante.

Ved también

Podéis ver la destrucción de procesos como efecto derivado de la ejecución de otros procesos o de las llamadas al sistema pedidas por otros procesos en el subapartado 3.2.1 del módulo didáctico "Comunicación y sincronización".

Todo proceso, cuando finaliza sin incidentes la ejecución del programa que almacena, debe ser destruido. Esta destrucción sólo puede ser el resultado de la ejecución de la llamada *destruir_proceso*. A fin de que se lleve a cabo la destrucción, el compilador inserta automáticamente, de manera transparente para el programador, la llamada *destruir_proceso* al sistema como última instrucción del programa. De manera adicional, el programador puede incluir solicitudes a la llamada *destruir_proceso* para provocar la finalización del proceso en situaciones controladas por el programa.

La llamada para destruir un proceso podría ser:

Estado = destruir_proceso(*id_proceso*).

Si esta llamada funciona correctamente, no volverá a aparecer, ya que provocará la destrucción del propio proceso que la invoca.

2.2. La herencia entre procesos

La herencia entre procesos es la relación que existe entre los diferentes elementos que configuran el entorno del proceso padre y los que configuran el entorno del proceso hijo.

En concreto, tenemos **tres tipos de herencia**:

1) Compartición: el proceso padre y el proceso hijo comparten un mismo elemento. Por lo tanto, las manipulaciones que se hagan de este elemento afectarán a los dos procesos de la misma manera.

2) Copia: el SO crea los elementos que configuran el entorno del proceso hijo como una copia de los elementos del proceso padre en el momento de invocar la operación de creación. A partir del momento en el que el proceso hijo ha sido creado, los dos entornos tienen evoluciones diferentes.

3) Valores nuevos: en este caso el SO crea los elementos del proceso hijo de nuevo y lo hace sin tener en cuenta los del proceso padre.

Cada elemento que configura el entorno puede tener una herencia diferente que puede venir predeterminada por el sistema o puede ser establecida mediante los parámetros de la llamada de creación al sistema. A continuación analizamos lo que representan estas posibilidades con respecto a cada uno de los elementos del entorno:

1) La memoria y su contenido. La memoria, tal como hemos visto anteriormente, se puede organizar por segmentos. Para simplificar la exposición nos centraremos en tres segmentos: el código, los datos y la pila. Los atributos de herencia pueden ser diferentes para cada segmento. El código y los datos pueden tener cualquiera de los tres atributos de herencia, mientras que la pila sólo puede tener herencia de copia o de valores nuevos, pero no compartida, ya que refleja el estado de llamada a procedimientos y a variables locales de cada proceso. Esto último se debe al hecho de que la pila es necesaria para garantizar que los dos procesos (padre e hijo) evolucionen de manera independiente. Así pues, las herencias posibles para la memoria son las siguientes:

Ved también

Podéis ver la segmentación de la memoria en el subapartado 3.1 del módulo didáctico "La gestión de la memoria".

a) Compartición: los procesos hijo y padre comparten el mismo segmento de memoria física. Ésta es la situación más conveniente para segmentos de código y, en general, para segmentos que contengan información que sólo ha de ser leída. En caso de compartición de un segmento de datos, cualquier modificación que haga uno de los dos procesos alterará el estado de la memoria. Esta situación permite la colaboración de los procesos mediante la compartición de información.

Ved también

Podéis ver la problemática asociada a la compartición de la información en el apartado 2 del módulo didáctico "Comunicación y sincronización".

b) Copia: los segmentos del proceso hijo son una copia exacta de los del padre en el instante de la creación. Cuando se crea un proceso por duplicación del proceso padre, se deben copiar, como mínimo, todos los segmentos que durante la ejecución independiente del proceso padre y del hijo pueden ser modificados. Éstos son los segmentos de datos y de pila. Otro caso es cuando se crea un proceso por compartición y padre e hijo comparten código y datos. En esta situación, el segmento de pila no puede ser compartido y ha de ser copiado. El motivo, tal como se ha expuesto anteriormente, es que la pila refleja el estado de las llamadas a procedimientos y a variables locales de cada proceso que son fruto de la ejecución independiente de cada proceso.

c) Valores nuevos: en este caso se carga un fichero ejecutable que definirá de nuevo el contenido de la memoria.

2) El punto de ejecución dentro de la memoria. El punto de inicio de ejecución del programa depende del contenido de la memoria, en concreto del contenido del segmento de código. En caso de copiar o compartir el código con el proceso padre, el punto de ejecución puede ser o bien el mismo que el del proceso padre o bien una dirección que se introduce como parámetro de la operación de creación. En caso de cargar en la memoria un programa nuevo, el punto inicial de ejecución ya es especificado en el fichero ejecutable.

Ved también

Podéis ver la coincidencia de los puntos de ejecución de los procesos padre e hijo en el apartado 4 de este módulo didáctico.

Así pues, las herencias posibles son las siguientes:

a) Copia: el punto de ejecución del proceso padre se copia en el hijo. Este caso sólo es posible si se comparten o se copian los segmentos de código y de datos. Para poder distinguir el proceso padre del proceso hijo lo más habitual es que los dos procesos reciban valores de retorno diferentes de la llamada de creación al sistema.

b) Valores nuevos: el punto de ejecución es definido por los parámetros de la llamada al SO o por el fichero ejecutable que definirá el contenido de la memoria.

El entorno de compartición no es posible, ya que los procesos padre e hijo tienen ejecuciones independientes.

3) El entorno de entrada/salida. La herencia del entorno de entrada/salida es independiente de la de memoria. El concepto de entorno de entrada/salida hace referencia a las sesiones de acceso a dispositivos que el proceso encontrará abiertas de manera implícita.

El entorno de entrada/salida puede tener las tres modalidades de herencia siguientes:

a) Compartición: el proceso padre comparte con el proceso hijo las sesiones de trabajo con los dispositivos que tenga abiertos en el momento de la creación. Las modificaciones que se hagan sobre estas sesiones de trabajo con uno de los dos procesos también afectarán al otro. Por ejemplo, en caso de una sesión de acceso secuencial, los dos procesos compartirán un único puntero de acceso. Las sesiones de acceso a los dispositivos que abran a partir del momento de la creación serán independientes en cada proceso.

b) Copia: el proceso hijo encuentra abiertas las mismas sesiones de acceso a los dispositivos que tenía abiertas el padre, pero con la diferencia de que las acciones que se efectúen en estas sesiones no afectarán a las del otro.

c) Valores nuevos: en este tercer caso el proceso hijo encuentra abiertas un conjunto de sesiones de acceso a los dispositivos que se especifican como parámetros de la llamada al sistema.

4) El dominio de protección. El dominio de protección que hereda un proceso está formado por los atributos de los dominios a los que pertenecerá por sus derechos asociados. En el caso de un sistema con protecciones basadas en *capabilities*, la herencia también incluye la lista de *capabilities* inicial del proceso nuevo. Las *capabilities* tienen un comportamiento análogo al de los dispositivos virtuales.

Ved también

Podéis ver las *capabilities* en el subapartado 4.4 del módulo didáctico "El sistema de ficheros".

Por lo tanto, en la descripción siguiente sólo haremos referencia a los atributos de dominio. Consideraremos que éstos no pueden ser copiados, de manera que el dominio de protección puede tener las dos modalidades de herencia siguientes:

a) Compartición: el proceso hijo pertenece al mismo dominio que el proceso padre o, lo que generalmente es lo mismo, pertenecen al mismo usuario. Ésta es la situación más habitual, e implica que las acciones del proceso hijo son imputables al mismo usuario que las del proceso padre.

b) Valores nuevos: a veces, sin embargo, el proceso hijo necesita unos privilegios diferentes de los que tiene asociados el usuario al que pertenece el proceso padre. En este caso, y bajo condiciones controladas de protección, se puede especificar un nuevo dominio para el proceso hijo. El cambio de dominio suele ir acompañado de un cambio de programa. Para mantener el sistema protegido, el programa nuevo debe haber sido escrito por el usuario que configura el nuevo dominio, o ser de su total confianza.

2.3. La sincronización y el estado de finalización en la destrucción de procesos

Hasta ahora hemos analizado el mecanismo de creación y destrucción de procesos básicamente desde la óptica del proceso creado, y no nos hemos fijado en las acciones que puede efectuar el proceso padre como consecuencia de la creación de un hijo. Los procesos padre deben sincronizar su ejecución con la finalización de los procesos que han creado y, al mismo tiempo, necesitan conocer el estado en el que se ha producido esta finalización.

Ved también

Podéis ver el estudio en detalle y desde un punto de vista más general del concepto de sincronización en el módulo didáctico "Comunicación y sincronización".

2.3.1. La sincronización

Un ejemplo en el que aparece la necesidad de sincronización entre procesos padre e hijo lo encontramos cuando exploramos las posibles modalidades de ejecución de los comandos: de primer plano, de fondo y diferidos. Si nos fijamos en las dos primeras modalidades se puede identificar un proceso padre, que es el que ejecuta el intérprete de comandos, y unos procesos hijos, que son los que ejecutan los comandos. Así pues, podemos ejecutar los comandos de las tres maneras siguientes:

1) En la **modalidad de primer plano** (*foreground*) el intérprete de comandos espera que finalice el comando antes de pedir un nuevo comando para ejecutar. En esta situación, el proceso padre debe esperar a que el proceso hijo finalice y para conseguirlo el SO tiene que proporcionar herramientas para congelar la ejecución del proceso padre hasta que el proceso hijo sea destruido.

2) En la **modalidad de ejecución de segundo plano** (*background*) el intérprete de comandos no espera a que finalice el comando, sino que inmediatamente después de crear el proceso que ejecutará el comando continúa adelante y pide un comando nuevo. En esta otra situación, el proceso padre sólo necesita que el sistema lo retenga mientras crea el nuevo proceso con el fin de devolverle el identificador del proceso si la creación se ha ejecutado correctamente o, en caso contrario, devolverle un error.

3) Una tercera posibilidad es la **modalidad mixta**, que consiste en una combinación de las dos modalidades anteriores. Entonces un proceso padre crea un proceso hijo en modalidad de segundo plano y, a partir de un cierto instante de su ejecución, decide esperar que finalice uno de sus procesos hijos. El SO debe proporcionar llamadas específicas de sincronización.

La figura siguiente muestra los tres modelos de ejecución:

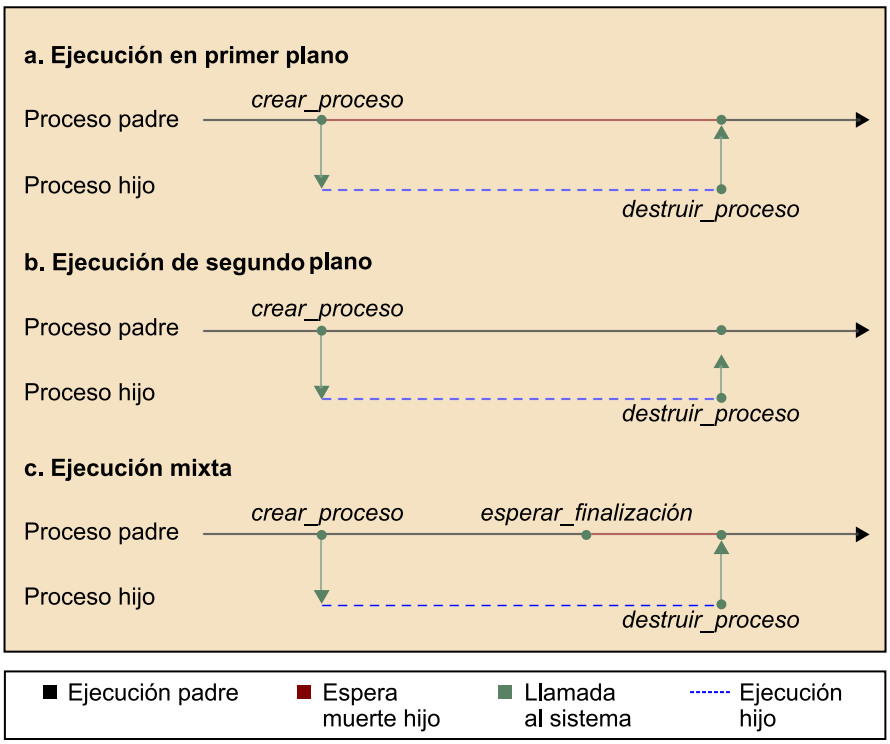


Figura 5

Como consecuencia de la sincronización entre los procesos padre e hijo, el sistema puede ofrecer los siguientes modelos de llamadas al sistema:

a) Introducir un parámetro, *modo_ejecución*, que indique si el proceso padre ha de esperar la destrucción del hijo o no.

- Id_proceso = *crear_proceso*(*modo_ejecución*,*entorno_memoria*, *nombre_programa*,*punto_*ejecución,*entorno_E*/S,*entorno_dominio*).
- Estado = destruir_proceso(*id_proceso*).

b) Hay que conseguir que para crear nunca sea necesaria la finalización de un proceso y se pueda proporcionar una llamada específica que permita llevar a cabo esta función.

- Id_proceso = *crear_proceso*(*entorno_memoria*,*nombre_programa*, *punto_ejecución*,entorno_*E*/S,*entorno_dominio*).
- Estado = destruir_proceso(*id_proceso*).
- Estado = esperar_finalización(*id_proceso*).

El parámetro de la llamada *esperar_finalización* puede ser de entrada o de salida. En el primer caso, sirve para indicar el proceso concreto con el que se quiere sincronizar. En el segundo caso, puede servir para esperar a cualquiera de los hijos. La llamada esperará hasta que alguno de los hijos del proceso que la in-

(9)En algunos sistemas también se incluye el caso en el que el proceso hijo, a pesar de estar todavía vivo, ha sido detenido por algún motivo.

voca haya terminado[9]. En el momento en el que un proceso hijo haya acabado la llamada a sistema le devolverá el control al proceso padre, que recibirá el identificador del proceso hijo terminado con su estado de finalización.

2.3.2. El estado de finalización de los procesos

Tal como hemos avanzado, para el proceso padre puede ser interesante conocer el punto en el que la aplicación ha finalizado o el motivo por el que ha finalizado. Con este fin, la llamada *destruir_proceso* suele tener un parámetro que el programador puede utilizar para notificar al proceso padre el motivo de la finalización o, lo que es lo mismo, el punto del algoritmo donde se ha decidido finalizar el proceso.

La llamada *destruir_proceso* quedaría de la manera siguiente:

Estado = destruir_proceso(*id_proceso*,*estado_finalización*).

Además, tal como hemos visto anteriormente, la ejecución de un proceso puede finalizar como consecuencia de acciones no previstas por el programador: por errores de ejecución o por efectos laterales de la ejecución de otras llamadas al sistema. En estos casos el sistema operativo es el encargado de codificar los motivos de la finalización con el fin de notificarlo al proceso padre.

El sistema operativo debe proporcionar una llamada que permita a los procesos padres recuperar los parámetros de finalización de sus procesos hijos. Esta llamada suele ser la misma que les permite sincronizarse con la finalización del proceso hijo: *esperar_finalización*.

2.4. Los cambios en el entorno de ejecución

Una vez ha sido creado un proceso, el SO inicia la ejecución del código que este proceso contiene. Como resultado de esta ejecución, el entorno puede evolucionar y cambiar, y los cambios pueden afectar a cualquiera de los elementos que configuran el proceso: la memoria, el contenido de la memoria, el entorno de entrada/salida o el dominio de protección. Para todos ellos, el SO debe proporcionar llamadas que permitan modificarlos.

En esta asignatura ya hemos estudiado las llamadas que hacen referencia a los dispositivos, a los ficheros y a la protección. En este subapartado nos centraremos especialmente en aquellas que modifican el contenido y la estructura del espacio de memoria en el marco del cambio de imagen, y profundizaremos en aquéllas otras que cambian el entorno de las entradas/salidas en el marco de los redireccionamientos o, en otras palabras, en el marco de la asignación implícita de dispositivos virtuales a lógicos.

Ved también

Podéis encontrar las llamadas al sistema relacionadas con los dispositivos, los ficheros y la protección en los módulos didácticos "Entrada/salida" y "El sistema de ficheros".

2.4.1. Cambio de imagen

El sistema operativo ha de permitir que los procesos carguen nuevos programas a la memoria con el fin de ser ejecutados. Esta acción se puede llevar a cabo de dos modos posibles:

a) Al mismo tiempo que se crea un nuevo proceso. Por ejemplo, en el sistema operativo Windows se cambia la imagen en el momento en el que se crea un nuevo proceso.

b) A posteriori, una vez creado el proceso, mediante una llamada específica (*cargar_imagen*). Por ejemplo, en el sistema operativo UNIX sólo se cambia la imagen si se invoca explícitamente una llamada a sistema para cargar una nueva imagen. En el momento en el que se crea un nuevo proceso no se puede modificar la imagen, sino que el sistema realiza una copia de la imagen del proceso padre.

En función del SO, se ofrecerán combinaciones diferentes de estas dos posibilidades. Aquí enfocaremos la explicación partiendo de un SO que sólo ofrece la segunda posibilidad. Consideraremos, pues, que los procesos nuevos inicialmente siempre ejecutan el mismo código que el proceso padre, ya sea porque lo comparten, o porque se ha copiado. A posteriori, una vez iniciada la ejecución del nuevo proceso, éste decidirá si debe cargar un nuevo código o no.

Éste es el caso del funcionamiento del intérprete de comandos, que, como veremos más adelante, primeramente obtendría por la entrada estándar la orden que debería llevar a cabo, analizaría si había algún ejecutable que lo pudiera hacer y, en este caso, crearía un nuevo proceso igual. Este nuevo proceso, al ejecutarse, cargaría la aplicación que da servicio al comando recibido.

Ved también

Podéis ver el funcionamiento del intérprete de comandos en el subapartado 4.2 de este módulo didáctico.

La carga de un nuevo ejecutable provoca la reconfiguración total del espacio lógico del proceso sobre el que se efectúa. Por lo tanto, todos los valores de variables y constantes, los procedimientos y las funciones que se encontraban dentro del espacio lógico del proceso antes de la carga desaparecen. Con el fin de que el código cargado pueda utilizar información elaborada por el código anterior a la carga, debe utilizar mecanismos o dispositivos de almacenamiento que sirvan de puente entre los dos. Una manera sencilla de hacerlo es utilizar el sistema de ficheros o, en general, un dispositivo de almacenamiento. No obstante, los SO ofrecen la posibilidad de pasar información a través de ellos en forma de parámetros de la llamada *cargar_imagen*, los cuales son recibidos por el nuevo programa como parámetros de entrada de la función principal.

El paso de código en C

Si se utiliza el lenguaje C, el nuevo programa recibirá la información del código anterior a su carga como parámetros de la función main:

```
main(argc,argv,env)
int argc;
char **argv,**env;
```

Hemos de advertir que la llamada *cargar_imagen* sólo afecta a la estructura y al contenido de la memoria, y también al contador de programa, que nos indica la próxima instrucción que hay que ejecutar. El resto de los elementos que configuran el entorno del proceso no tienen por qué verse afectados, de manera que el nuevo ejecutable debe encontrar el mismo entorno de entrada/salida y el mismo entorno de protección. No obstante, esta afirmación puede variar en función de si el SO asigna otras funciones a la llamada *cargar_imagen*.

Ved también

Podéis ver las funciones de la llamada *cargar_imagen* en el caso de UNIX en el subapartado 4.4 de este módulo didáctico.

La llamada *cargar_imagen* podría tener la siguiente forma:

Estado = cargar_imagen(*nombre_ejecutable*,*parámetros*).

Esta llamada devuelve un valor sólo en caso de error, ya que si tiene éxito, todo el código y los datos del programa que la contenían habrán desaparecido de la memoria.

2.4.2. Redireccionamientos de entrada/salida

La manera de introducir modificaciones en general en el entorno de entrada/salida de un proceso consiste en abrir y cerrar sesiones de acceso a los dispositivos. En este subapartado nos fijaremos en el caso concreto de cómo se pueden realizar los redireccionamientos de las entradas/salidas. Como hemos visto, los redireccionamientos de las entradas/salidas se basan en la asignación de dispositivos virtuales estándar y en dispositivos lógicos. Para conseguir la independencia y la portabilidad de las aplicaciones, esta asignación se debe llevar a cabo con anterioridad a la ejecución de los programas que configuran las aplicaciones y, por consiguiente, de manera transparente para ellos. Este hecho es el que hemos denominado **asignación implícita de los dispositivos virtuales**.

Ved también

Podéis consultar las sesiones de acceso en el entorno de entrada/salida en el subapartado 5.2 del módulo didáctico "Entrada/salida".

El sistema puede hacer esta asignación de las tres maneras siguientes:

a) El proceso hijo puede heredar del proceso padre los dispositivos virtuales que éste tenga abiertos.

b) El proceso padre especifica en los parámetros del llamada *crear_proceso* qué dispositivos lógicos se deben asignar a los dispositivos virtuales del hijo.

c) El proceso hijo modifica su entorno de entrada/salida una vez creado y antes de cargar un programa nuevo.

La última de estas opciones es la que nos interesa en este subapartado.

3. Flujos de ejecución

Un **hilo** o **flujo de ejecución** (***thread***) *e*s la mínima unidad de planificación del sistema operativo.

Un flujo forma parte de un proceso y disfruta de los recursos asignados al proceso. Un proceso tiene como mínimo un flujo, pero en los sistemas operativos actuales un proceso puede tener más de un flujo de ejecución.

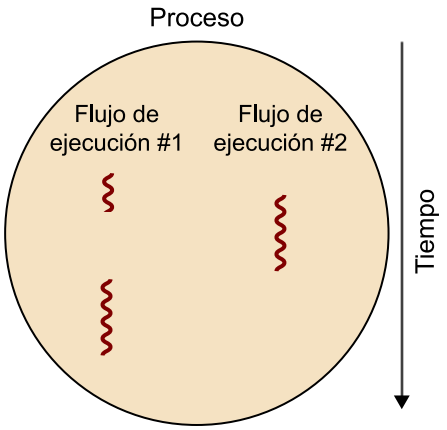


Figura 7

Los diferentes hilos que forman parte de un mismo proceso comparten la mayoría de los recursos del proceso, como el espacio de memoria, los archivos abiertos, los permisos, el directorio de trabajo, el identificador de proceso, etc. En cambio, cada hilo tiene su propia pila de ejecución, puede estar ejecutando diferentes instrucciones (cada hilo tiene su propio registro contador de programa o PC), se ejecuta a una determinada velocidad y tiene su propio estado de ejecución. Por lo tanto, todos los flujos de un proceso disponen del mismo código y datos, pero en un momento determinado pueden estar ejecutando partes diferentes del código y/o trabajar con datos diferentes.

Puede haber varios motivos por los que sea interesante diseñar aplicaciones con distintos flujos (*multithreaded*):

- Programación más modular, encapsulando tareas.
- Explotar el paralelismo disponible en las máquinas con memoria compartida por más de un procesador (*multicore* o *multiprocesador*).
- Hacer Entrada/Salida paralela, dedicando flujos a realizar la E/S.
- Hacer servidores concurrentes, haciendo que cada flujo atienda una petición de servicio.

Cabe señalar que si disponemos de varios procesadores, los flujos se podrán ejecutar simultáneamente. Si en cambio sólo disponemos de un procesador o, en general, tenemos menos procesadores que flujos, entonces hay que repartir el tiempo del procesador entre los diferentes flujos, multiplexando el uso del procesador en el tiempo.

Aunque los flujos comparten todos los recursos del proceso, es importante que haya alguna información propia de cada flujo para garantizar un funcionamiento correcto. Por ejemplo:

1) Podemos estar interesados en realizar una acción determinada sobre un flujo concreto.

2) Cada flujo puede estar ejecutando una parte del código diferente.

3) Aunque llamen a una misma rutina, cada uno necesitará guardar los parámetros y las variables locales en una zona de memoria diferente que haga las funciones de pila.

4) Cuando se produzcan cambios de contexto entre flujos, hay que garantizar que se conserva el estado del flujo para poder continuar con su ejecución posteriormente.

5) Hay que diferenciar las condiciones de error producidas por llamadas a sistema realizadas por diferentes flujos dentro de un mismo proceso.

6) Opcionalmente, es interesante controlar la planificación de los *threads*, por ejemplo priorizando un flujo sobre otros.

Por todo esto cada flujo tiene asociado:

1) Un identificador.

2) Un puntero a la siguiente instrucción que hay que ejecutar.

3) Un puntero a la cima de la pila.

4) El estado de los registros del procesador.

5) Puede haber información local en un flujo. Eso puede ser útil en determinadas circunstancias. Un ejemplo de ello es la definición de la variable `errno` cuando una llamada a sistema ha producido un error.

6) Puede haber información de planificación específica para cada flujo y usada por el planificador de flujos.

Los flujos de un mismo proceso comparten la mayoría de los recursos. Por ello, el cambio de contexto entre flujos de un mismo proceso es menos costoso que el cambio de contexto entre flujos de procesos diferentes.

Los hilos de ejecución también son conocidos como **procesos ligeros** porque consumen menos recursos de sistema que los procesos.

Posibles utilidades de los flujos

- Aplicaciones gráficas: un flujo se encarga de la gestión de la interfaz gráfica de usuario mientras otro realiza las operaciones de cálculo.
- Aplicaciones cliente/servidor: el servidor crea múltiples flujos con el fin de dar servicio a múltiples clientes simultáneamente.

4. Ejemplos de gestión de procesos

4.1. La gestión de procesos en UNIX

En UNIX un proceso es un entorno de ejecución identificado por un número denominado **PID**, que es el **identificador del proceso** y es único durante toda la vida del SO.

Los **principales elementos que constituyen un proceso de UNIX** son:

1) El **espacio de memoria**, formado por tres segmentos lógicos: uno de código[10], uno de datos y uno de pila. El segmento de código puede ser compartido por otros procesos. Entre el segmento de datos y el de pila hay una porción de espacio lógico no asignada al proceso. Cuando conviene, el proceso puede aumentar el segmento de datos con la llamada rutina de biblioteca `malloc` (vista en el módulo 3).

[10]El segmento de código también se denomina *segmento de texto*.

2) El **entorno de entrada/salida**, que está formado por una tabla de *file descriptors* (dispositivos virtuales), local en cada proceso. Cada entrada de esta tabla apunta a otra tabla, que en este caso es global para el sistema, y contiene los punteros de lectura secuencial, el modo de acceso y un puntero a la estructura del SO que gestiona el dispositivo lógico asociado a un *file descriptor*. Finalmente, el entorno de entrada/salida se complementa con la información de cuál es el directorio de trabajo actual.

Ved también

Podéis ver los *file descriptors* en el subapartado 7.1 del módulo didáctico "Entrada/salida".

3) El **UID** y el **GID**, que corresponden al **número de identificador del usuario** y al **número de identificador del grupo del usuario**, hacen referencia al usuario y al grupo al que pertenecen los procesos dentro del dominio de protección.

UID y GID

Son, respectivamente, los acrónimos de los términos ingleses *User IDentifier* y *Group IDentifier*.

4) El **estado de los registros del procesador**, que refleja en qué estado se encuentra la ejecución del programa que contiene el proceso.

5) La **información estadística**, que recoge informaciones como el tiempo consumido de UCP, el volumen consumido de memoria o el número de procesos hijos generados.

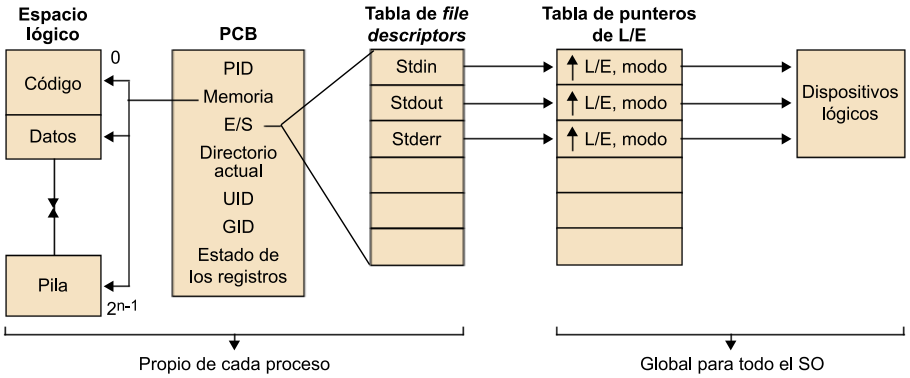


Figura 8

4.1.1. La creación y la destrucción de procesos

Las correspondencias entre las llamadas de UNIX y las vistas en este módulo son:

Operación	Llamadas de UNIX
Crear_proceso	`fork`
Destruir_proceso	`exit`
Esperar_finalización	`wait`
Cargar_imagen	`exec` (`execl`, `execlp`, `execle`, `execv`, `execvp`, `execve`)
Quiensoy	`getpid`

UNIX ha optado por tratar el tema de la creación y la destrucción de procesos de la manera más sencilla posible. Las llamadas que ofrecen tienen el mínimo número posible de parámetros, y crean y destruyen los procesos a partir de una definición implícita en el sistema. A posteriori, y definiéndolo por programa, el usuario puede cambiar el entorno de ejecución y hacer el proceso a su medida. A continuación veremos cómo actúa cada una de las llamadas vistas en este módulo en el sistema operativo UNIX:

Creación y destrucción de procesos

En UNIX, la creación y destrucción de procesos se ajusta a la filosofía general del sistema de ofrecer herramientas de utilización sencilla que permitan desarrollar cualquier política que se necesite.

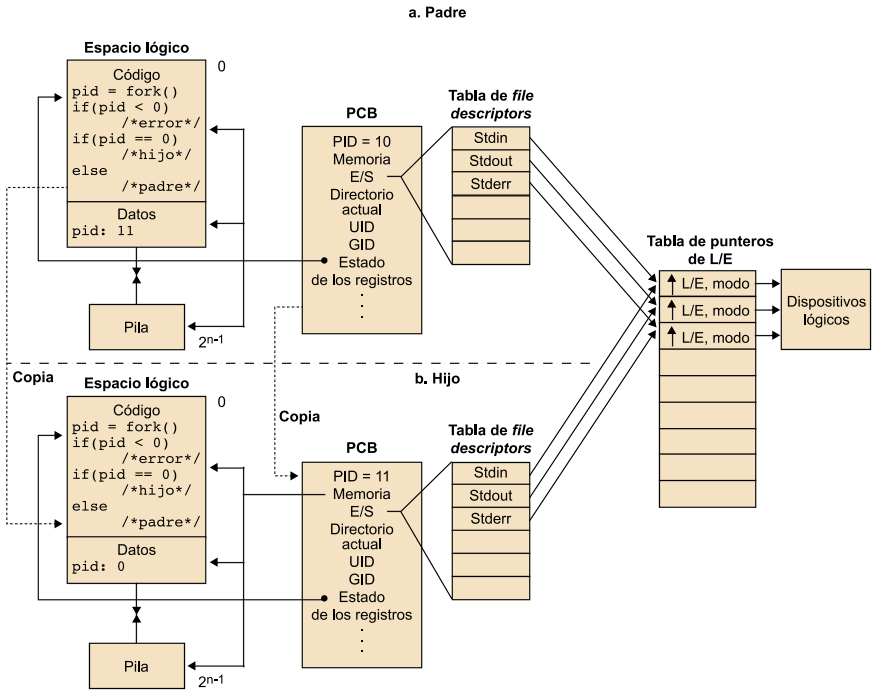


Figura 9

1) La **llamada `fork`** no tiene ningún parámetro y crea un nuevo proceso hijo que es un duplicado del proceso padre. En general, el PCB del hijo es una copia del PCB del padre; el espacio de memoria del proceso hijo es una copia del espacio del padre y las entradas de la tabla de *files descriptors* apuntan a las mismas entradas de la tabla de punteros de lectura/escritura. El proceso hijo pertenece al mismo usuario y grupo de usuarios que el padre. Los dos procesos continúan la ejecución de manera independiente en el punto posterior a la llamada `fork` en el sistema.

Normalmente después de hacer la llamada `fork` el proceso hijo ha de ejecutar un código diferente del código del padre. Para poder distinguirse, la llamada fork devuelve valores diferentes en función de si es el proceso padre o el hijo: el hijo recibe un valor 0 y el padre recibe el identificador (PID) del hijo.

Durante la creación de un nuevo proceso, UNIX controla los aspectos importantes que afectan al conjunto del sistema, como la disponibilidad de recursos[11]. Un ejemplo concreto de este control es el hecho de que no permite que un usuario normal (no procesos de sistema) ocupe la última entrada de la tabla de procesos, la última área de memoria libre, etc. Si eso sucediera, el sistema quizá no se podría recuperar de situaciones en las que sería necesario poner en marcha algún proceso del sistema de manera urgente. Por ejemplo, si se tuviera que parar rápidamente el sistema, podría ser catastrófico no poder poner en marcha el proceso de parada del sistema (*shutdown*).

[11] Los recursos del sistema son la memoria, las entradas en la tabla de PCB, etc.

2) La **llamada `exit`** destruye un proceso. La destrucción de un proceso mediante esta llamada es idéntica a la que hemos explicado en el caso de la creación y la destrucción de procesos. El proceso que la ejecuta es destruido por el sistema. Con el fin de notificar el estado de finalización se pasa al sistema operativo un parámetro que el proceso padre podrá recoger mediante la llamada `wait`.

Ved también

Podéis ver la destrucción de procesos en el subapartado 2.1 de este módulo didáctico.

3) La **llamada `wait`** bloquea[12] el proceso que la ha llamado hasta que alguno de sus procesos hijos haya sido destruido. Cuando esto sucede, el sistema le devuelve el PID del proceso destruido y, como parámetro de salida, el estado en el que ha finalizado. Combinando las tres llamadas presentadas en este subapartado, el intérprete de comandos puede ejecutar comandos en las modalidades de primer plano y de fondo (podéis ver la figura 10).

(12)Existen otras variantes, como waitpid y waitid, que permiten un control más fino sobre el proceso al que se quiere esperar o los tipos de cambios de estado que se desean monitorizar.

UNIX guarda el estado de finalización de los procesos destruidos en su PCB y espera que su proceso padre lo recoja mediante la llamada `wait`. Por este motivo, UNIX no libera el PCB de un proceso en el momento de su destrucción[13].

(13)Cuando se destruye un proceso se liberan todos los recursos excepto el PCB: se libera memoria, se cierran los canales de entrada/salida, etc.

Los procesos esperan en un estado especial denominado *zombie* hasta que se ejecuta la llamada `wait` que permitirá eliminarlos definitivamente. Para evitar la acumulación de procesos en estado *zombie* se han previsto dos mecanismos:

1) Cada usuario sólo puede tener en un instante determinado un cierto número de procesos activos, incluidos los *zombies*.

2) Los procesos que son destruidos más tarde que sus procesos padre pasan a ser considerados hijos del primer proceso del sistema[14], que es el encargado de recoger su estado de finalización.

(14)El primer proceso del sistema se llama *init* y tiene un PID igual a uno.

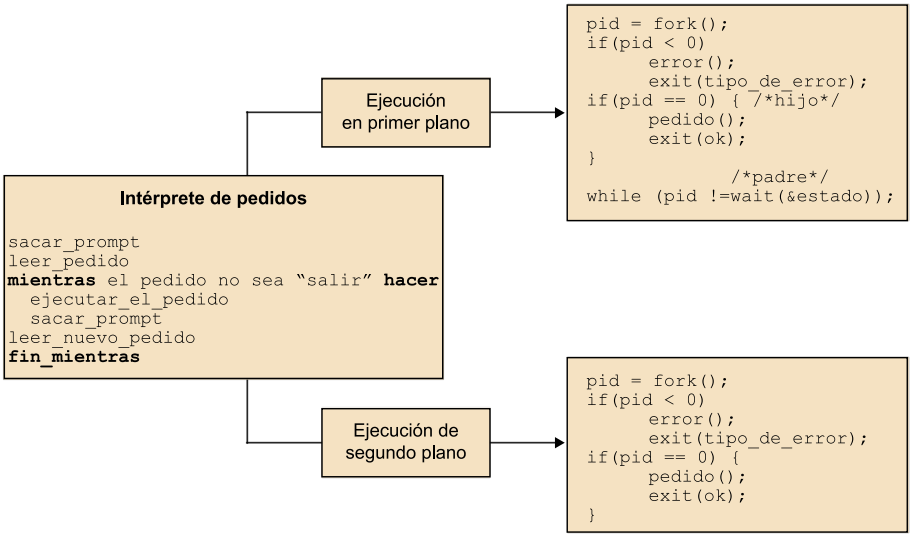


Figura 10

4.1.2. Los cambios del entorno que configura un proceso

En UNIX hemos visto que cualquier cambio que se quiera hacer en un proceso se debe llevar a cabo una vez éste se haya creado. El cambio de ejecutable y los cambios en las entradas/salidas hechos a posteriori de la llamada fork permiten controlar por programa y, de manera flexible, tener acceso a un abanico muy amplio de posibilidades que difícilmente se podrían haber previsto a priori desde el SO. Para ilustrar este hecho nos fijaremos en el intérprete de comandos de UNIX (*shell*) y en los redireccionamientos que permite. Antes, sin embargo, analizaremos los puntales de esta flexibilidad, que son los elementos siguientes:

- La **llamada `fork`**, que ya hemos visto.
- La **llamada `exec`** de cambio de ejecutable.
- La **llamada `dup`** de entrada/salida, que permite manipular los *file descriptors*.

A continuación veremos cómo los dos últimos elementos nos permiten modificar el entorno que configura un proceso.

1) El cambio de ejecutable

La llamada exec al sistema de UNIX permite cambiar la imagen o el ejecutable que contiene un proceso. De hecho, hay toda una familia de funciones que se suelen referenciar bajo el nombre exec, que son diferentes formas de invocar la llamada a sistema execve. En concreto, las funciones son: `execl`, `execlp`, `execle`, `execv`, `execvp`.

El cambio que tiene lugar mediante la llamada `exec` no afectó a los elementos que configuran el proceso, como el entorno de entrada/salida. En general, sólo afecta a la programación de las señales y al dominio de protección si el fichero ejecutable que se debe cargar tiene activo el bit *setuid* o el *setgid*. Los bits *setuid* y *setgid* hacen que durante la ejecución del programa el proceso que lo ha cargado cambie respectivamente al dominio del usuario propietario, o al del grupo del usuario propietario del fichero ejecutable. Estos derechos permiten construir aplicaciones que acceden de manera controlada a bases de datos o a ficheros en general sobre los que no se quiere dar un derecho de escritura generalizado.

Ved también

Podéis ver la programación de las señales en el subapartado 3.2 del módulo didáctico "Comunicación y sincronización" y el cambio de dominio de usuario por parte del proceso en el subapartado 5.1 del módulo didáctico "El sistema de ficheros".

2) La manipulación de los *file descriptors*

La llamada `dup` permite duplicar el valor de una entrada concreta de la tabla de *file descriptors* en la primera posición libre que se encuentre dentro de la tabla.

Con esta operación se pueden modificar fácilmente los valores asociados a los *file descriptors* estándares.

Veamos un ejemplo de uso de la llamada `dup` en combinación con la llamada `exec`:

```
int estado, descFichero, pid;
...
pid = fork();
switch(pid) {
case 0:     /* Código del hijo. */
     descFichero = open("fitxer",O_RDONLY);
     if (descFichero == -1) {
     /*tratamiento del error*/
     }
     ...
     close(0);
     dup(descFichero);
     close(descFichero);
     ...
     execl("/bin/comando","comando",0);
     ...
     /*tratamiento del error*/
     ...
case -1:    /* La llamada fork ha fallado. */
     ...
     /*tratamiento del error*/
     ...
default:    /* Código del padre. */
     while(pid != wait(&estat));
}
...
```

El código anterior podría ser el que ejecuta el intérprete de comandos dada la orden siguiente: `$ comando < fichero`.

En este caso sólo se muestra el trozo de código que, una vez leída la orden desde el terminal, ejecuta el comando. Esta ejecución se efectúa en la modalidad de primer plano. Ahora veremos los diferentes pasos de esta ejecución.

En primer lugar, el intérprete crea el proceso hijo que tendrá que ejecutar el comando. Una vez creada se debe realizar el redireccionamiento de la entrada estándar a fin de que cuando se cargue el programa comando se encuentre el redireccionamiento hecho. Para hacerlo, se crea un *file descriptor* vinculado a `fichero` con la llamada `open (descFichero)`. Con open se ocupa la primera entrada libre de la tabla de los *file descriptors*.

Las figuras siguientes ilustran la evolución de las tablas internas del sistema durante este proceso:

a) Situación después de abrir `fichero`:

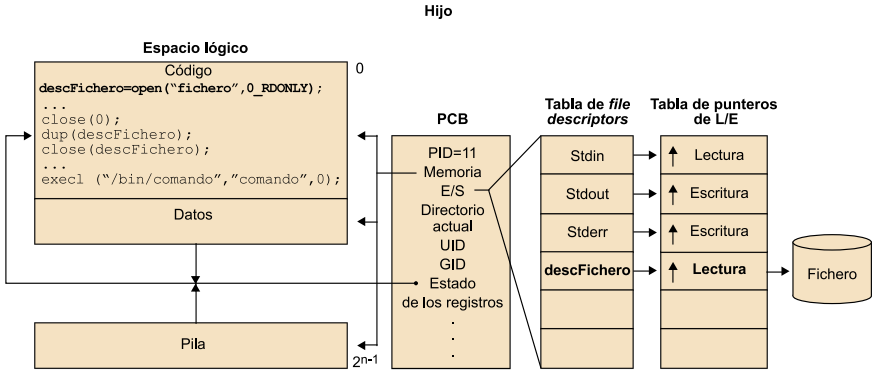


Figura 11

Para redireccionar la entrada estándar desde `fichero` se debe conseguir que el *file descriptor* 0 esté asociado al fichero. Para hacerlo, desasignamos el dispositivo lógico asociado al *file descriptor* 0 y, mediante la llamada `dup`, lo volvemos a asignar copiando sobre su entrada en la tabla de los *file descriptors* la entrada asociada a `descFichero`. Así, los dos *file descriptors* hacen referencia a la misma sesión de trabajo abierta sobre `fichero`.

b) Situación después de haber ejecutado la llamada `dup`:

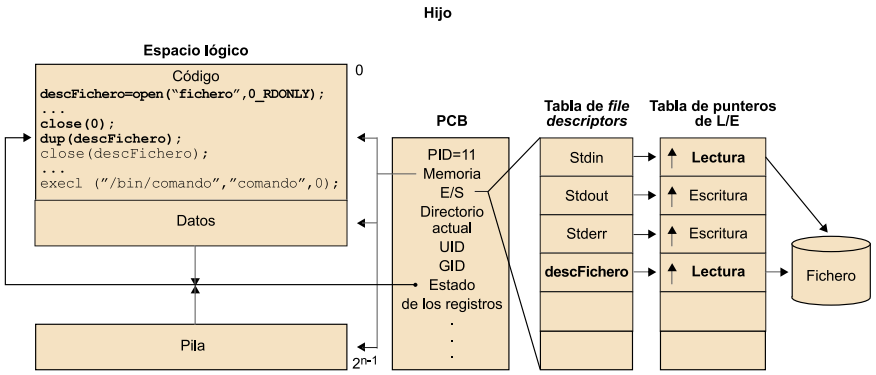


Figura 12

Falta eliminar, mediante la llamada close, el *file descriptor* descFichero para conseguir el entorno de entrada/salida que debe encontrar el comando que hay que ejecutar.

Después, el proceso hijo puede cargar el nuevo ejecutable mediante la llamada `exec`[15].

c) Situación después de haber cargado el ejecutable del comando:

[15] En el ejemplo se utiliza execl, que es una de las diferentes formas de esta llamada.

Ved también

En los recursos del Aula disponéis de material adicional con ejemplos de programas UNIX, que utilizan estas llamadas al sistema.

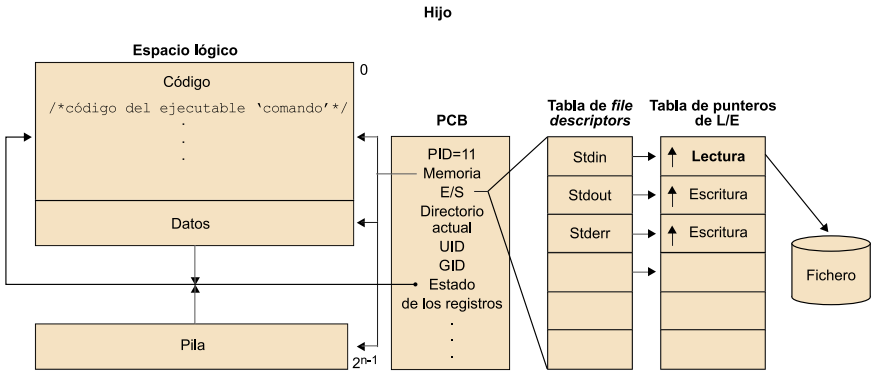


Figura 13

4.1.3. La jerarquía de procesos en UNIX

El **proceso** *init* con PID igual a uno es el primer proceso que se crea en un sistema UNIX, y es el antecesor de todos los procesos que se crearán a continuación. *Init* es el único proceso que no se crea con la llamada `fork` al sistema, ya que lo crea directamente el núcleo durante la inicialización del sistema.

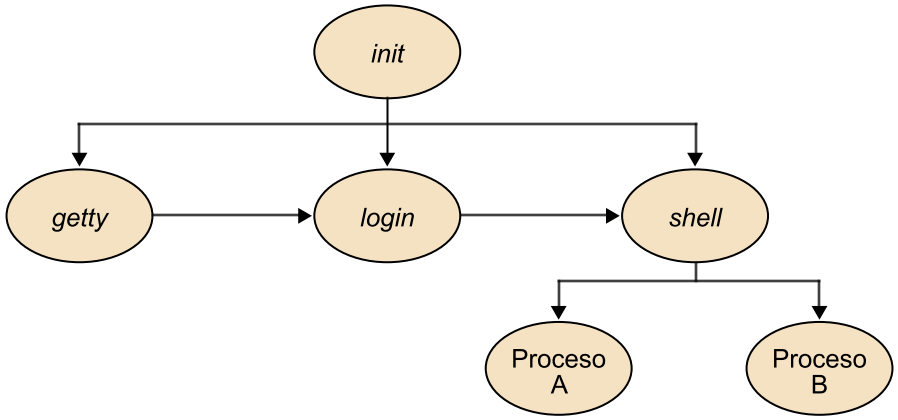


Figura 14

Las principales funciones del proceso *init* son las de inicializar todos los procesos de sistema, como los procesos servidores de red, poner en marcha un proceso *getty* para cada terminal del sistema y, finalmente, liberar los procesos *zombie* que ya no tienen el proceso padre.

El **proceso** *getty* es el encargado de esperar a que se pongan en marcha los terminales. Cuando un terminal es puesto en marcha, el proceso que ejecuta el programa *getty* carga el **programa** *login* con el fin de autentificar el nuevo usuario. Una vez verificada la identidad del usuario, el programa *login* carga el **intérprete de comandos** (*shell*) e inicia una sección de trabajo con el usuario. Cuando el usuario acaba la sesión de trabajo el proceso shell se destruye, e *init* crea un nuevo proceso *getty* que lo sustituye.

4.2. La gestión de procesos en Windows

Los elementos que constituyen un proceso son los que ya hemos visto anteriormente y no requieren nuevas explicaciones. Nos centraremos en este subapartado en algunos aspectos particulares de Windows con respecto a la gestión de procesos y su entorno.

4.2.1. La creación y la destrucción de procesos

Lo más destacable con respecto a la gestión de procesos en Windows es que este sistema sigue un modelo explícito de paso de parámetros. Es decir, en el momento en el que se realiza la llamada para pedir la creación de un proceso hay que especificar explícitamente los elementos que formarán el entorno de ejecución del nuevo proceso. El nuevo proceso hijo no será por lo tanto una copia del padre, como sucede en UNIX. Una consecuencia de eso es que la llamada de Windows para invocar la creación de un proceso (`CreateProcess`) tiene 9 parámetros. Recordemos que la llamada equivalente de UNIX (`fork`) no recibe ni un solo parámetro, ya que en el caso de UNIX el proceso hijo es, en el momento de su creación, una copia idéntica del padre. Algunas de las llamadas relacionadas con la gestión de procesos se muestran en la tabla siguiente:

Operación	Llamadas Windows
Crear_proceso + Cargar_imagen	CreateProcess
Destruir_proceso	ExitProcess
Esperar_finalización	WaitForSingleObject o WaitForMultipleObjects
Quiensoy	GetCurrentProcessId

Web recomendada

Podéis encontrar más información en la web de ayuda a los desarrolladores de Microsoft (MSDN), en la parte de procesos y flujos. Actualmente la URL es http://msdn.microsoft.com/en-us/library/ms684841.

En este apartado mostramos algunos ejemplos o partes de ejemplos extraídos de la web MSDN que ilustran la creación de procesos, la sincronización y la redirección de la E/S. En los ejemplos observamos cómo se pueden ajustar objetos mediante sus manejadores (*handles*).

El primer ejemplo muestra cómo se puede crear un proceso en Windows y cómo se puede sincronizar con la finalización del hijo. Se puede observar cómo se inicializan las estructuras de datos de tipo `STARTUPINFO` y `PROCESS_INFORMATION` y cómo se hacen los llamadas `CreateProcess` y `WaitForSingleObject`. El proceso hijo ejecutará el programa que se indique como parámetro al ejecutar el proceso padre.

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>


void _tmain( int argc, TCHAR *argv[])
```

```
{
  STARTUPINFO si;
  PROCESS_INFORMATION pi;

  ZeroMemory( &si, sizeof(si));
  si.cb = sizeof(si);
  ZeroMemory( &pi, sizeof(pi));

  if( argc != 2)
  {
    printf("Usage: %s [cmdline]\n, argv[0]);
    return;
  }

  // Start the child process.
  if( !CreateProcess( NULL // No module name (use command line)
    argv[1], // Command line
    NULL // Process handle not inheritable
    NULL // Thread handle not inheritable
    FALSE // Set handle inheritance to FALSE
    0 // No creation flags
    NULL // Use parent's environment block
    NULL // Use parent's starting directory
    &si // Pointer to STARTUPINFO structure
    &pi) // Pointer to PROCESS_INFORMATION structure
  )
  {
    printf( "CreateProcess failed (%d).\n, GetLastError());
    return;
  }

  // Wait until child process exits.
  WaitForSingleObject( pi.hProcess, INFINITE);

  // Close process and thread handles.
  CloseHandle( pi.hProcess);
  CloseHandle( pi.hThread);

}
```

4.2.2. Los cambios del entorno que configura un proceso

El siguiente ejemplo muestra cómo se puede crear un proceso en Windows y cómo se puede realizar la redirección de las entradas y las salidas por medio de los manejadores (*handles*). Mostramos sólo algunos trozos que ilustran cómo se puede realizar la redirección. El ejemplo usa *pipes*. Aunque estudiaremos con detalle las *pipes* en el último módulo del curso, hemos visto ya su funcio-

namiento básico al hablar del intérprete de comandos y la creación de filtros conectando la salida de unos comandos con la entrada de otros comandos. El ejemplo completo se puede encontrar en la web de ayuda a los desarrolladores de Microsoft bajo el concepto "Creating a Child Process with Redirected Input and Output".

El código del padre contiene:

```
...
HANDLE g_hChildStd_IN_Rd = NULL;
HANDLE g_hChildStd_IN_Wr = NULL;
HANDLE g_hChildStd_OUT_Rd = NULL;
HANDLE g_hChildStd_OUT_Wr = NULL;
...
int _tmain(int argc, TCHAR *argv[])
{

  SECURITY_ATTRIBUTES saAttr;

  printf("\n->Start of parent execution.\n");

  // Set the bInheritHandle flag so pipe handles are inherited.
  saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
  saAttr.bInheritHandle = TRUE;
  saAttr.lpSecurityDescriptor = NULL;

  // Create a pipe for the child process's STDOUT.
  ¡if (! CreatePipe(&g_hChildStd_OUT_Rd, &g_hChildStd_OUT_Wr, &saAttr, 0))
    ErrorExit(TEXT("StdoutRd CreatePipe"));

  // Ensure the read handle to the pipe for STDOUT is not inherited.
  ¡if (! SetHandleInformation(g_hChildStd_OUT_Rd, HANDLE_FLAG_INHERIT, 0))
    ErrorExit(TEXT("Stdout SetHandleInformation"));

  // Create a pipe for the child process's STDIN.
  ¡if (! CreatePipe(&g_hChildStd_IN_Rd, &g_hChildStd_IN_Wr, &saAttr, 0))
    ErrorExit(TEXT("Stdin CreatePipe"));

  // Ensure the write handle to the pipe for STDIN is not inherited.
  ¡if (! SetHandleInformation(g_hChildStd_IN_Wr, HANDLE_FLAG_INHERIT, 0))
    ErrorExit(TEXT("Stdin SetHandleInformation"));

  // Create the child process.
  CreateChildProcess();

...
}
```

```
void CreateChildProcess()
// Create a child process that uses the previously created pipes for STDIN and STDOUT.
{
  TCHAR szCmdline[]=TEXT("child");
  PROCESS_INFORMATION piProcInfo;
  STARTUPINFO siStartInfo;
  BOOL bSuccess = FALSE;

  // Set up members of the PROCESS_INFORMATION structure.
  ZeroMemory( &piProcInfo, sizeof(PROCESS_INFORMATION));

  // Set up members of the STARTUPINFO structure.
  // This structure specifies the STDIN and STDOUT handles for redirection.
  ZeroMemory( &siStartInfo, sizeof(STARTUPINFO));
  siStartInfo.cb = sizeof(STARTUPINFO);
  siStartInfo.hStdError = g_hChildStd_OUT_Wr;
  siStartInfo.hStdOutput = g_hChildStd_OUT_Wr;
  siStartInfo.hStdInput = g_hChildStd_IN_Rd;
  siStartInfo.dwFlags |= STARTF_USESTDHANDLES;

  // Create the child process.
  bSuccess = CreateProcess(NULL,
    szCmdline // command line
    NULL // process security attributes
    NULL // primary thread security attributes
    TRUE // handles are inherited
    0 // creation flags
    NULL // use parent's environment
    NULL // use parent's current directory
    &siStartInfo // STARTUPINFO pointer
    &piProcInfo); // receives PROCESS_INFORMATION

  // If an error occurs, exit the application.
  ¡if (! bSuccess)
    ErrorExit(TEXT("CreateProcess"));
  else
  {
    // Close handles to the child process and its primary thread.
    CloseHandle(piProcInfo.hProcess);
    CloseHandle(piProcInfo.hThread);
  }
}
```

El código del hijo es:

```
#include <windows.h>
```

```
#include <stdio.h>

#define BUFSIZE 4096

int main(void)
{
  CHAR chBuf[BUFSIZE];
  DWORD dwRead, dwWritten;
  HANDLE hStdin, hStdout;
  BOOL bSuccess;

  hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
  hStdin = GetStdHandle(STD_INPUT_HANDLE);
  if (
     (hStdout == INVALID_HANDLE_VALUE) ||
     (hStdin == INVALID_HANDLE_VALUE)
     )
     ExitProcess(1);

  // Send something to this process's stdout using printf.
  printf("\n ** This is en message from the child process. ** \n");

  // This simple algorithm uses the existence of the pipes to control execution.
  // It relies on the pipe buffers to ensure that no data is lost.
  // Larger applications would use more advanced process control.

for (;;)
  {
   // Read from standard input and stop on error or no data.
   bSuccess = ReadFile(hStdin, chBuf, BUFSIZE, &dwRead, NULL);

   ¡if (! bSuccess || dwRead == 0)
   break;

   // Write to standard output and stop on error.
   bSuccess = WriteFile(hStdout, chBuf, dwRead, &dwWritten, NULL);

   ¡if (! bSuccess)
    break;
  }
  return 0;
}
```

4.2.3. Jerarquía de procesos en Windows

En Windows los procesos no tienen una relación jerárquica, sino que trata todos los procesos como si pertenecieran a una misma generación. Para sincronizarse con la finalización de un proceso sólo hay que tener el manejador y el identificador del proceso. Como el proceso padre tiene estas informaciones se puede mantener o simular una relación jerárquica si se desea.

Con respecto a los procesos de sistema, hay todo un conjunto de procesos que implementan diferentes servicios. Hay que destacar como procesos de especial relevancia y que no pueden estar desempleados: el proceso nulo, denominado *System Idle Process*, que tiene el PID 0 que se ejecuta cuando no hay nada a hacer; y un proceso denominado *System* **con PID 4, que es el núcleo del sistema operativo**.

5. *Pthreads*

Hoy día los sistemas operativos suelen proporcionar directamente implementaciones de flujos. Varios lenguajes de programación también facilitan la programación si los tienen integrados en su sintaxis. No obstante, aquí presentamos los flujos definidos por el estándar POSIX, denominados *pthreads*.

La ventaja de realizar implementaciones con *pthreads* radica en la gran portabilidad que se puede obtener, dada la gran implantación de estos flujos en la práctica totalidad de los sistemas.

Existen muchas primitivas relacionadas con la gestión de los *pthreads*, pero aquí nos centraremos en las más básicas. Lo primero que hemos de conocer son las primitivas que permiten crear (`pthread_create`), finalizar (`pthread_exit`) y sincronizar (`pthread_join`) pthreads. Una llamada a `pthread_join` bloqueará al `pthread` que la haga hasta que el pthread con el que se pide sincronizar no haya acabado. Además, la llamada le devolverá información sobre el estado de finalización que el pthread haya indicado al hacer la llamada `pthread_exit`. Si no se quiere esperar a un pthread, se puede indicar por medio de la primitiva `pthread_detach`.

Además, se puede conseguir información sobre el identificador del *pthread* utilizando la primitiva `pthread_self`.

Primitiva	Descripción
`pthread_create`	Creación de un pthread
`pthread_exit`	Terminación de un pthread
`pthread_join`	Espera la finalización de un pthread
`pthread_detach`	Indica que no se querrá esperar al pthread
`pthread_self`	Retornar el identificador del pthread

Hay que guardar el valor de retorno indicado al hacer un `pthread_exit` hasta que se haga un `pthread_join` o un `pthread_detach`. Por ello, la estructura de datos que representa al `pthread` no será liberada hasta después de ejecutar la llamada que llegue más tarde de la combinación `pthread_exit` y `pthread_join` en caso de que queramos una sincronización, o `pthread_exit` y `pthread_detach` en caso de que no queramos una sincronización. Hemos de ser conscientes de que, como con cualquier otro re-

curso, es necesario que se libere la estructura de datos del pthread cuando deje de necesitarse. Por ello, como diseñadores de programas que usan pthreads hemos de tener en cuenta estos aspectos.

Como los `pthreads` de un proceso comparten memoria, se pueden comunicar rápidamente a través de ella. Pero eso supone la necesidad de utilizar mecanismos de sincronización y exclusión mutua. Es conveniente conocer como mínimo algunas primitivas que permiten proteger la modificación de estructuras de datos de manera concurrente y la implementación de *regiones críticas*. Eso se puede hacer forzando el acceso a las regiones críticas en exclusión mutua (*mutual exclusion* o *mutex*). Para ello existen unas variables de tipo `pthread_mutex_t` sobre las que se puede pedir el acceso de manera exclusiva en un momento determinado utilizando la primitiva `pthread_mutex_lock` (adquirir el *lock* sobre el mutex). Una vez la llamada retorna, tenemos la garantía de que el `pthread` que ha hecho la llamada es el único que tiene derecho a realizar las acciones que queremos proteger con esta variable de mutex y puede entrar en la región crítica y ejecutar las operaciones que ésta contiene. Al acabar estas operaciones, hay que liberar la variable de *mutex* utilizando `pthread_mutex_unlock` con el fin de permitir el acceso a la región crítica a otros pthreads que estén esperando.

Ved también

En el módulo *7*, trataremos a fondo el problema de la sincronización y de la exclusión mutua.

Primitiva	Descripción
`pthread_mutex_lock`	Pedir acceso en exclusividad al recurso mutex
`pthread_mutex_unlock`	Liberar recurso mutex

A continuación mostramos un ejemplo sencillo disponible en múltiples sitios de Internet que ilustra el uso de las llamadas básicas para conseguir crear 10 `pthreads` que trabajen de modo concurrente, pero modifiquen una variable compartida en exclusión mutua, garantizando que el resultado final de un contador sea equivalente a una ejecución en serie.

```
#include <stdio.h>
#include <pthread.h>

#define NTHREADS 10

void *thread_function();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

main()
{
   pthread_t thread_id[NTHREADS];
   int y, j;
   for(i=0; y < NTHREADS; i++)
   {
```

```
        pthread_create( &thread_id[i], NULL, &thread_function, NULL);
    }
    for(j=0; j < NTHREADS; j++)
    {
        pthread_join( thread_id[j], NULL);
    }
    /* Now that all threads are complete I can print the final result. */
    /* Without the join I could be printing en value before all the threads */
    /* have been completed. */
    printf("Final counter value: %d\n", counter);
}

void *thread_function()
{
    printf("Thread number %ld\n", pthread_self());
    pthread_mutex_lock( &mutex1);
    counter++;
    pthread_mutex_unlock( &mutex1);
}
```

Resumen

En este módulo didáctico hemos analizado la **gestión de procesos** que hace el SO partiendo de la definición de proceso que ya habíamos visto en esta asignatura. Lo hemos hecho siguiendo los pasos siguientes:

Ved también

Podéis ver la definición de proceso en el subapartado 1.2 del módulo didáctico "El sistema operativo: una máquina virtual".

a) En primer lugar, para entender mejor el concepto de proceso, hemos analizado de manera superficial la **representación interna del proceso en el SO** y la **gestión de los procesos en un sistema multiprogramado de tiempo compartido**. Los principales conceptos que hemos presentado son los siguientes:

- El **bloque de control de procesos**, como estructura básica que configura un proceso.
- La **ejecución concurrente**, mediante la multiplexación del procesador en tiempo.
- El **estado de los procesos** (*run*, *ready* y *wait*).
- El **cambio de contexto** como mecanismo para pasar un proceso del estado *run* a *ready* o *wait*, o viceversa.

b) En segundo lugar, hemos analizado desde la óptica del usuario las **llamadas que permiten manipular los procesos**, es decir, que permiten crearlos, modificarlos y destruirlos. Como principales conceptos relacionados con estas llamadas podemos destacar los siguientes:

- La **herencia entre los procesos padre e hijo** en el momento de la creación de un nuevo proceso.
- La **sincronización**, para ver cómo el proceso padre sincroniza la creación y la destrucción de un proceso hijo.
- Los **cambios de ejecutable** y los **redireccionamientos**, como principales cambios del entorno que configura un proceso.

A continuación hemos descrito los flujos de ejecución (*threads*) que permiten la ejecución concurrente del código de un proceso. Hemos visto las características de los flujos que comparten la mayoría de los recursos del proceso al que pertenecen, a pesar de requerir algunos recursos propios de cada thread para garantizar un funcionamiento correcto. Hemos visto que el cambio de contexto entre flujos de un mismo proceso es menos costoso que el cambio entre flujos de diferentes procesos. Utilizando flujos se puede facilitar la programación y mejorar la eficiencia de muchos programas.

La figura siguiente muestra un mapa conceptual con los principales conceptos explicados en este módulo didáctico y sus relaciones:

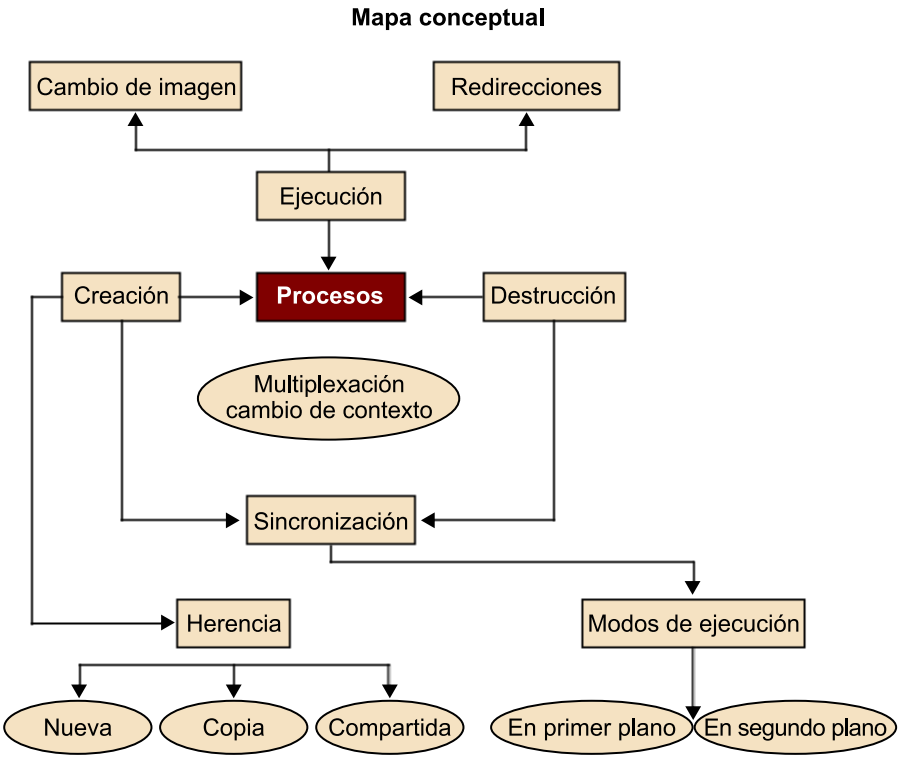


Figura 15

Después de ver los SO en general, hemos hecho una confrontación de los conceptos anteriores con el sistema UNIX. Las llamadas al sistema de gestión de procesos siguen una filosofía de diseño basada en la sencillez y la flexibilidad que permite crear y modificar los procesos con toda libertad desde el programa. Una consecuencia de este hecho es la gran cantidad de posibilidades que puede ofrecer el intérprete de comandos (*shell*) con respecto a modos de ejecución y de redireccionamiento de las entradas y salidas.

A continuación hemos destacado las diferencias que encontramos en el sistema Windows relativas a la gestión de procesos, la herencia de recursos y la jerarquía de procesos.

Finalmente, hemos introducido la funcionalidad básica del paquete de flujos del estándar POSIX (*pthreads*).

Actividades

1. Estudiad el comando `ps` de UNIX. Mirad qué campos presenta y qué significado tienen. Analizad qué procesos tenéis en marcha en el sistema y en qué estado se encuentran.

2. LINUX permite ver los recursos asignados a los procesos como ficheros que cuelgan del directorio `/proc`. Mirad en el manual de UNIX la entrada asociada a este directorio y analizad qué se puede encontrar en él.

3. Analizad las diferentes posibilidades de ejecución de comandos y de redireccionamientos que nos ofrece el intérprete de comandos (*shell*) de UNIX y pensad cómo se podrían hacer desde las llamadas al sistema.

4. Estudiad el modo de crear procesos en Windows y el modo de redireccionar las entradas y las salidas estándar.

5. Cread un programa que cree varios `pthreads` que ejecuten una rutina, investigando cómo se puede pasar parámetros al `pthread` en el momento de hacer el `pthread_create`. Debéis aseguraros de que la información pasada por parámetro seguirá estando disponible en el momento en el que el pthread intente acceder, sin suponer un problema sobre la velocidad ni el orden en el que se ejecutan los *pthreads*.

Ejercicios de autoevaluación

1. Dentro del diagrama de estados de los procesos que se muestra en el subapartado de estados de un proceso, ¿dónde colocaríais el estado *zombie* de UNIX? Justificad la respuesta.

2. ¿Cómo se podrían redireccionar las entradas/salidas de los procesos hijos en un sistema en el que la llamada *crear_proceso* fuerza el cambio de imagen? Justificad la respuesta.

3. Después de ejecutar "primero", ¿qué salidas obtendremos y por qué canal saldrán? Justificad la respuesta.

Primero

```
main() {
        char a;

        a = 'A';
        if (write(1,&a,1) == -1) {
                /*error*/
                write(2,"error",5);
        }
        close(1);
        execl("segundo","segundo",0);
        if (write(2,&a,1) == -1) {
                /*error*/
                exit(1);
        }
}
```

Segundo

```
main() {
        char a;

        if (write(1,&a,1) == -1) {
                /*error*/
                write(2,"error",5);
        }
        if (write(2,&a,1) == -1) {
                /*error*/
                exit(1);
        }
        a = 'B';
        if (write(2,&a,1) == -1) {
                /*error*/
                exit(1);
        }
}
```

Selección

4. Cuando se lleva a cabo un cambio de contexto entre dos procesos, el sistema debe guardar los valores que configuran el estado del proceso que deja el procesador con el fin de poder reemprender su ejecución más adelante. Decid cuáles de los elementos siguientes que configuran un proceso deben ser guardados explícitamente en el momento del cambio:

a) El contador de programa.

b) Los segmentos de memoria.

c) Los dispositivos virtuales.

d) El identificador del proceso.

e) Los registros del procesador.

Justificad la respuesta.

5. Cuál de los resultados siguientes creéis que producirá la ejecución del programa siguiente:

```
main() {
   int fd, pid;
   char buff[2];

   fd = open("fichero",O_RDONLY);
   if (read(fd,buff,2) == -1){
      /*error*/
      exit(1);
   }
   write(1,buff,2);
   pid = fork();
   switch(pid){
      case 0: if (read(fd,buff, 1) == -1){
                 /*error o fin de fichero*/
                 exit(1);
              }
              write(1,buff,1);
              exit(0);

      default: if (read(fd,buff,1) == -1){
                 /*error o fin de fichero*/
                 exit(1);
              }
              write(1,buff,1);
              exit(0);
   }
}
```

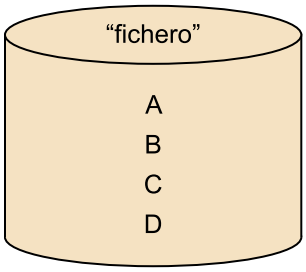


Figura 17

a) ABCD.

b) AABBCCDD.

c) ABCCDD.

d) ABDC.

e) Las opciones **a** o **d** indistintamente.

Justificad la respuesta.

6. Decid cuál de los resultados siguientes creéis que producirá la ejecución del programa que presentamos a continuación:

```
main() {
          int num, pid;

          num = 3;
          pid = fork();
          switch(pid) {
                       case 0:
                                num = num + 1;
                                printf("hijo: num = %d\n",num);
                       default:
                                num = num + 2;
                                printf("padre: num = %d\n",num);
          }
}
```

a) "Hijo: num = 4", "padre: num = 6".

b) "Hijo: num = 4", "padre: num = 5".

c) "Hijo: num = 1", "padre: num = 2".

d) "Hijo: num = 4", "padre: num = 5".

e) El valor de *num* no está definido en el hijo y el padre efectúa la salida: "padre: num = 5".

Justificad la respuesta.

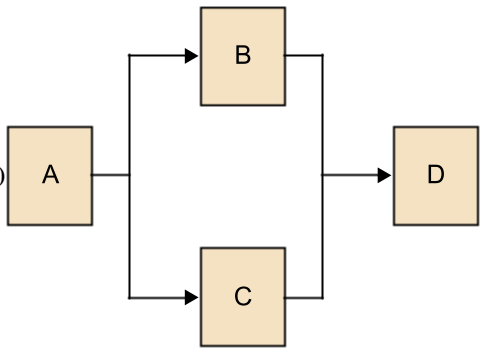
7. ¿Cuál de los diagramas de tiempo es el que representa la ejecución del código siguiente?

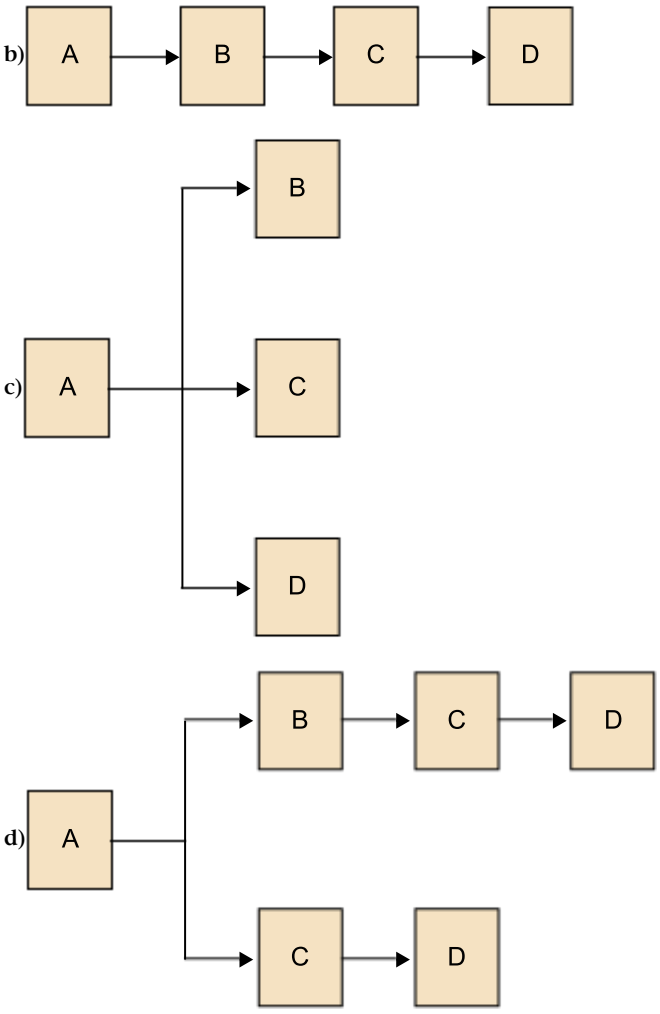
Justificad la respuesta.

```
main() {
          int pid1, pid2, estado;

          A
          pid1 = fork();
          switch(pid1) {
                       case 0:
                                B
                       default:
                                pid2 = fork();
                                switch(pid2) {
                                             case 0:
                                                      C
                                             default:
                                                      while (pid1 != wait(&estado));
                                                      while (pid2 != wait(&estado));
                                                      D
                                }
         }
}
```

a)




b)
A
B
C
D
c)
A
B
C
D
d)
A
B
C
D
C
D

Solucionario

Ejercicios de autoevaluación

1. El estado *zombie* es previo a la destrucción total del proceso mientras éste espera que el proceso padre, o en su defecto el proceso *init*, recoja el parámetro de finalización. Una vez recogido se acaban de liberar los recursos del proceso.

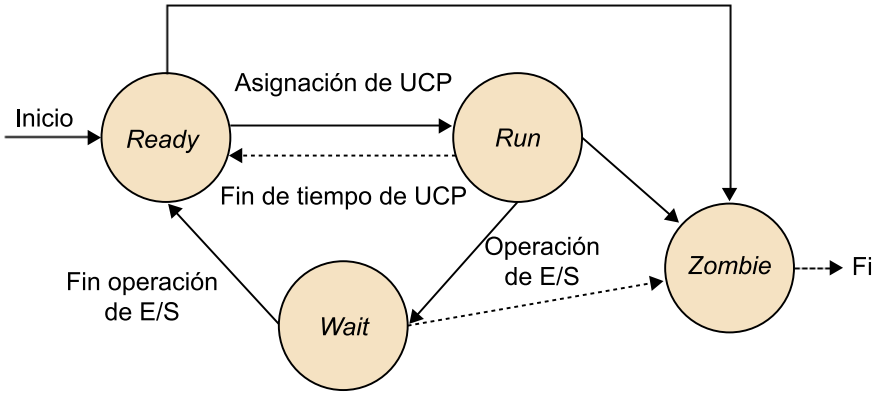


Figura 16

2. Como la llamada *crear_proceso* obliga a cargar un nuevo ejecutable, es imposible que el hijo herede el código del proceso padre, por lo que no se puede especificar mediante el programa cómo se debe cambiar el entorno de entrada/salida del proceso hijo. En estas condiciones, con los parámetros de la llamada *crear_proceso* se debe poder especificar los dispositivos lógicos que se quieren asociar a los dispositivos virtuales estándares del proceso hijo. Por ejemplo:

crear_proceso(*nombre_ejeecutable*,*fichero1*,*fichero2*,*fichero3*,*otros parámetros*...)

donde *fichero1* corresponde a la entrada estándar, *fichero2* a la salida estándar y *fichero3* a la salida de error. Si se quisiera hacer más flexible para permitir redireccionar cualquier dispositivo con cualquier modo de acceso, habría que incluir más parámetros.

3. La salida se efectuará de la manera siguiente:

a) El ejecutable "primero" escribe "A" para la salida estándar.

b) Se cierra la salida estándar.

c) Se cambia de ejecutable. Se carga "segundo" y todo el código que el ejecutable "primero" tiene por debajo de la llamada exec desaparece, y no se ejecutará más.

d) El ejecutable "segundo" escribe "error" para la salida de error estándar, ya que está cerrada, tal como hemos comentado en el punto **b**.

e) El ejecutable "segundo" escribe un carácter indeterminado para la salida de error estándar, ya que la variable a del ejecutable "segundo" se ha creado en el momento de su carga, y no ha sido inicializada.

f) El ejecutable "segundo" escribe" B" para la salida de error estándar.

4. Se han de guardar el contador de programa (**a**), que indica la próxima instrucción que se tiene que ejecutar, y el valor de los registros del procesador (**e**), que contienen, básicamente, valores parciales de los cálculos que se están llevando a cabo y el puntero a la última posición de la pila. Se deben guardar todos aquellos elementos dinámicos que configuran el punto de ejecución donde se encuentra el programa que contiene el proceso. El resto de los elementos, aunque configuran el entorno de ejecución, ya están guardados en el PCB, de manera que no hay que volver a guardarlos.

5. La respuesta correcta es la **e**. Los procesos padre e hijo comparten los punteros de lectura/escritura de los ficheros que el padre tiene abiertos en el momento de la creación del hijo. Así, las operaciones de lectura que haga cualquiera de los dos modifican el mismo puntero. Por otra parte, los dos procesos se ejecutan de manera concurrente y, por lo tanto, no podemos saber cuál de los dos efectuará primero la operación de lectura. Por este motivo se pueden dar las dos salidas.

6. La solución correcta es la **b**. La herencia de la llamada fork es de copia. Por lo tanto, el proceso hijo tendrá, una vez creado, una copia del mismo código y los mismos datos que el padre. A partir del momento de la creación, cada uno de ellos evolucionará independientemente y con sus variables.

7. La respuesta correcta es la **a**. El proceso padre ejecuta el código A y después crea un proceso hijo que ejecuta el código B. A continuación, el padre crea un segundo proceso hijo que ejecuta de manera concurrente el código C. Una vez creados los dos hijos, el proceso padre espera a que sus dos procesos hijos hayan acabado, y ejecuta D.

Glosario

bloque de control de procesos (PCB) *m* Estructura de datos que contiene la información del entorno de cada proceso necesaria para que el sistema operativo pueda gestionar la ejecución concurrente de un conjunto de procesos.

cambio de contexto *m* Técnica que, mediante la multiplexación del tiempo de procesador, consigue la ejecución concurrente de todos los procesos preparados para utilizarlo. Requiere guardar el estado de los procesos o flujos con el fin de restaurarlo cuando se quiera continuar ejecutándolos.

estado de un proceso *m* Estado que se asigna a cada proceso para controlar sus cambios de modo de ejecución a lo largo de su existencia al sistema. Sólo hay un conjunto limitado de estados posibles y las acciones que pueden dar lugar a transiciones entre los estados también están definidas.

hilo o flujo de ejecución *(thread)* *m* Unidad mínima de planificación del sistema operativo. Un flujo forma parte de un proceso y disfruta de los recursos asignados al proceso. Un proceso tiene como mínimo un flujo, pero en los sistemas operativos actuales puede tener más de uno. Los diferentes hilos que forman parte de un mismo proceso comparten ciertos recursos, como el espacio de memoria, los archivos abiertos, los permisos, el directorio de trabajo, el identificador de proceso, etc. En cambio, cada hilo tiene su propia pila de ejecución, puede estar ejecutando diferentes instrucciones (cada hilo tiene su propio registro contador de programa o PC), se ejecutan a diferentes velocidades y tienen su propio estado de ejecución. Los hilos de ejecución también son conocidos como **procesos ligeros** por el hecho de que los hilos de ejecución consumen menos recursos de sistema que los procesos.

cuota (*quantum*) *f* Tiempo máximo de UCP que puede utilizar un proceso de manera continua. Los sistemas operativos que trabajan en la modalidad de tiempo compartido hacen uso de la cuota para hacer cambio de contexto y dar el control del procesador a un nuevo proceso.

Bibliografía

Bibliografía básica

Silberschatz, A.; Galvin, P.; Gagne G. (2008). *Operating Systems Concepts* (8.ª edición). John Wiley & Sons.

Tanembaum, A. (2009). *Modern Operating Systems*. Prentice-Hall.

Bibliografía complementaria

Kernighan, B.; Pike, R. (1987). *El entorno de programación UNIX*. México: Prentice Hall Hispanoamericana.

Robbins, K.; Robbins, S. (2000). *UNIX: programación práctica*. Prentice Hall.

Documentación disponible sobre llamadas al sistema UNIX en los recursos del Aula.