2º de Grado en Ingeniería Informática
Sistemas Operativos

TEMA 2 – PROCESOS e HILOS

Bibliografía

El contenido de este documento se ha elaborado principalmente a partir de las siguientes referencias bibliográficas, y con propósito meramente académico y no lucrativo:

- W. Stallings. *Sistemas operativos, 5ª edición*. Prentice Hall, Madrid, 2005.
- A. S. Tanenbaum. *Sistemas operativos modernos, 3a edición*. Prentice Hall, Madrid, 2009.
- A. Silberschatz, G. Gagne, P. B. Galvin. *Fundamentos de sistemas operativos, séptima edición*. McGraw-Hill, 2005.
- A. Mclver, I. M. Flynn. *Sistemas operativos, 6ª edición*. Cengage Learning, 2011.
- J. A. Alamansa, M. A. Canto Diaz, J. M. de la Cruz García, S. Dormido Bencomo, C. Mañoso Hierro. *Sistemas operativos, teoría y problemas*. Editoral Sanz y Torres, S.L, 2002.
- F. Pérez, J. Carretero, F. García. *Problemas de sistemas operativos: de la base al diseño, 2ª edición*. McGraw-Hill. 2003.
- S. Candela, C. Rubén, A. Quesada, F. J. Santana, J. M. Santos. *Fundamentos de Sistemas Operativos, teoría y ejercicios resueltos*. Paraninfo, 2005.
- J. Aranda, M. A. Canto, J. M. de la Cruz, S. Dormido, C. Mañoso. *Sistemas Operativos: Teoría y problemas*. Sanz y Torres S.L, 2002.
- J. Carretero, F. García, P. de Miguel, F. Pérez, *Sistemas Operativos: Una visión aplicada*. Mc Graw Hill, 2001.

1 Procesos

Un **programa o tarea** es una unidad inactiva, como un archivo almacenado en un disco. Si hablásemos en términos de objetos, un programa sería un objeto y un proceso sería una instancia de ese objeto. Por tanto, **un programa no es un proceso**.

Una vez hecha la distinción entre proceso y programa, hay que decir que se han dado muchas **definiciones del término proceso**, incluyendo:

- Un programa en ejecución.
- Una instancia de un programa ejecutándose en un computador.
- La entidad que se puede asignar o ejecutar en un procesador.
- Una unidad de actividad cargada en memoria principal y caracterizada por un solo hilo secuencial de ejecución, un estado actual, y un conjunto de recursos del sistema asociados.

Esta última definición es la más completa, ya que tiene en cuenta lo que denominaremos como contexto de un proceso.

Para una mejor comprensión, y teniendo en cuenta los conceptos que ya ha adquirido del tema 1, lea la siguiente historia:

Ha pedido por internet uno de sus juguetes preferidos. El paquete le llega a casa, lo abre y saca un manual de ensamblado y las piezas que conforman el juguete. Armado con las instrucciones del manual y toneladas de paciencia, se embarca en esta tarea: Leer las instrucciones, reunir las herramientas necesarias, seguir cada uno de los pasos que se indican y terminar con el producto final.

Según el manual, el primer paso consiste en unir la parte A del juguete con la parte B mediante un tornillo de 2 pulgadas. Cuando termina con esta tarea verifica el paso 1. Inspirado en su éxito, prosigue con el paso 2 y luego con el paso 3. Apenas acaba usted de completar el paso 3 cuando su vecino, que está cortando el césped, grita pidiendo ayuda al cortarse con una herramienta eléctrica.

Usted verifica rápidamente las instrucciones del paso 3, de modo que sepa por donde se quedó en la construcción de su juguete; luego deja sus herramientas y corre en auxilio del vecino (quiere rematarlo, pero le da pena...). Después de todo, las necesidades inmediatas de alguien son más importantes que su éxito momentáneo con su juguete. Ahora se encuentra implicado en una tarea muy diferente a montar un juguete, y es seguir las instrucciones de un botiquín de primeros auxilios para tapar la herida de su vecino mediante vendas y antisépticos.

Una vez que la herida ha sido atendida exitosamente, usted vuelve a su trabajo previo. A medida que recoge sus herramientas, mira las instrucciones y observa que debe empezar por el paso 4. Luego prosigue con su proyecto de montaje hasta que lo termina por completo.

Pues bien, en terminología de sistemas operativos, la historia anterior se puede describir así:

Usted empeño el papel de la CPU o procesador. Había dos programas o tareas, una el ensamblado del juguete y la otra atender la herida de su vecino. Cuando estaba ensamblando el juguete (tarea A), cada paso seguido pertenece a un proceso. La llamada por ayuda fue una interrupción y cuando dejó el juguete para atender a su amigo herido, lo dejó por un proceso con prioridad superior. Cuando usted fue interrumpido, realizó un cambio de contexto cuando marcó el paso 3 como la última instrucción completada y dejó sus herramientas. Atender la herida del vecino se volvió la tarea B. Mientras usted ejecutaba las instrucciones de primeros auxilios, cada uno de los pasos que ejecutó pertenecian a un proceso. Y, por supuesto, cuando cada tarea se completaba, el proceso estaba acabado..

Como se ha mostrado, un solo procesador puede compartirse entre varios procesos, pero sólo si el sistema operativo cuenta con una política de planificación para determinar cuándo detener un proceso con una actividad y proceder con el otro. En el ejemplo anterior, el algoritmo de planificación estaba basado en prioridades: usted trabajaba en el proceso perteneciente al trabajo A hasta que se presentó un proceso con prioridad superior. Aunque en este caso se trataba de un buen algoritmo, un algoritmo de planificación basado en prioridades no siempre es lo mejor, como se verá en más profundidad en el capítulo de planificación.

2 Bloque de control de procesos o BCP

Supongamos que el procesador comienza a ejecutar un código de proceso o conjunto de instrucciones. En cualquier instante puntual del tiempo, mientras el proceso está en ejecución, éste se puede caracterizar por una serie de elementos cuyo conjunto se llama **contexto de ejecución.**

El contexto de ejecución es por tanto un conjunto de datos por el cual el sistema operativo es capaz de supervisar y controlar un proceso:

- **Identificación.** Un identificador único asociado al proceso para distinguirlo del resto de procesos. Puede también almacenar el identificador del proceso padre creador de este proceso y el identificador del usuario del proceso, dependiendo del sistema.
- **Estado**. Si el proceso está actualmente corriendo, está en el estado Ejecución. Hay más tipos de estado, dependiendo del sistema operativo, como por ejemplo estado Listo, Bloqueado, Suspendido, Terminado o Zombie (se estudiarán más adelante).
- **Información de planificación:** Nivel de prioridad relativo al resto de procesos, y evento o eventos por los que espera el proceso cuando está en estado Bloqueado. Esta información se usa por parte del planificador para decidir qué trabajo ejecutar a continuación.
- **Descripción de los segmentos de memoria asignados al proceso:** Espacio de direcciones o límites de memoria asignado al proceso en el espacio de usuario en RAM.
- **Punteros a memoria**. Incluye los punteros al código de programa y los datos asociados a dicho proceso, además de cualquier bloque de memoria compartido con otros procesos, e incluso si el proceso utiliza memoria virtual.
- **Datos de contexto**. Estos son datos que están presenten en los registros del procesador cuando el proceso está corriendo, banderas de estado, señales, datos temporales en registros,

contador de programa, etc., es decir, todo lo necesario para poder continuar la ejecución del proceso cuando el planificador del sistema operativo lo decida.

- **Información de estado de E/S y recursos asignados:** Incluye las peticiones de E/S pendientes, los dispositivos de E/S asignados a dicho proceso (por ejemplo, un disco duro), una lista de los ficheros usados, etc.
- **Comunicación entre procesos.** Se guarda cualquier dato que tenga que ver con la comunicación entre dos procesos independientes.
- **Información de auditoría**. Puede incluir la cantidad de tiempo de procesador o ciclos utilizados por el proceso, cuánto le queda por ejecutar.

El contexto de ejecución comentado anteriormente se almacena de manera diferente dependiendo del sistema operativo, pero de forma general se hace en una **estructura de datos** que se llama **Bloque de Control de Proceso** o **BC**P (*process control block* - PCB en Inglés) (Figura 3.1).

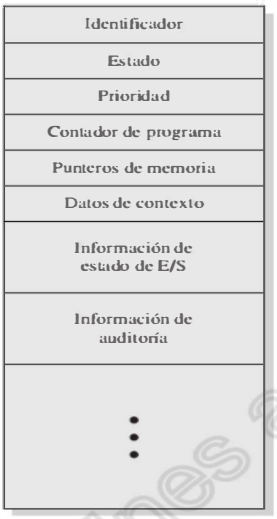


Figura 3.1. Bloque de control de programa (BCP) simplificado.

El punto más significativo en relación al BCP, es que contiene suficiente información de forma que es posible interrumpir un proceso cuando está corriendo y posteriormente restaurar su estado de ejecución como si no hubiera tenido interrupción alguna. El BCP es la herramienta clave que permite al sistema operativo dar soporte a múltiples procesos y proporcionar multiprogramación. Un BCP se crea cuando se acepta un proceso en el sistema por parte de planificador, y se actualiza a medida que la ejecución del proceso avanza desde el inicio hasta el final. En algunos sistemas basados en GNU/Linux, cada proceso tiene asociado un BCP que está almacenado en una **lista simplemente enlazada**, llamada **tabla de procesos**.

Se llama **imagen del proceso** al conjunto del código, datos, pila, montículo y BCP de un proceso. Para ejecutar un proceso es necesario que su imagen esté cargada en memoria principal.

3 Dispatcher

Ampliando con la sección anterior, para un computador monoprocesador (y mononúcleo), como máximo un único proceso puede estar ejecutándose en un instante de tiempo, y dicho proceso estará en el estado Ejecución. La conmutación entre unos procesos y otros es lo que da lugar a la multiprogramación, y el programa que se encarga de conmutar según la política de planificación del sistema, se llama **activador** o ***dispatcher,*** el cual reside en el núcleo del sistema.

El *dispatcher* pasa a ejecución cuando se producen alarmas de temporización (**rodaja o cuanto de tiempo**) o interrupciones de otro tipo y cuando se producen operaciones de E/S. Es el planificador el que dice al *dispatcher* qué proceso es el que debe introducir en la CPU (no confundir el planificador con el *dispatcher*). Gracias al **BCP**, el *dispatcher* puede introducir un proceso nuevo o hacer que se continué con otro por la instrucción donde se quedó (salvado y restauración de contexto).

Recuerde que una interrupción, ya sea software o hardware, puede suceder en cualquier momento y, por tanto, en cualquier punto de la ejecución de un programa de usuario. Su aparición es imprevisible.

3.1 Ejemplo de ejecución del dispatcher

La Figura 3.2 muestra el despliegue en memoria de tres procesos. Para simplificar la exposición, se asume que dichos procesos no usan memoria virtual; por tanto, los tres procesos residen en memoria principal y representan a tres programas. De manera adicional, existe un pequeño programa activador que forma parte del núcleo del sistema operativo y que intercambia procesos en el procesador.

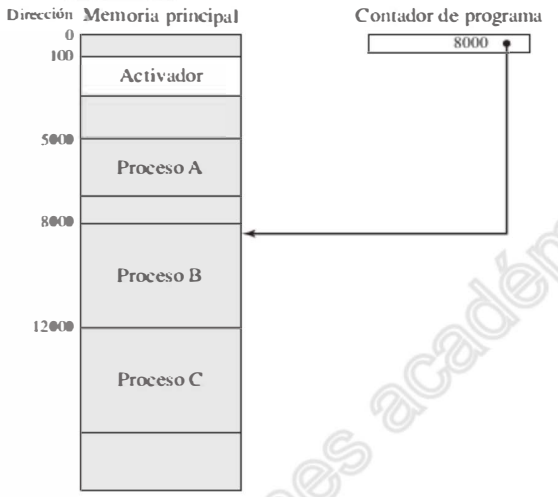


Figura 3.2. Instantánea de un ejemplo de ejecución (Figura 3.4) en el ciclo de instrucción 13.

La Figura 3.3 muestra las trazas de cada uno de los tres procesos en los primeros instantes de ejecución. Se muestran las 12 primeras instrucciones ejecutadas por los procesos A y C. El proceso B ejecuta 4 instrucciones y se asume que la cuarta instrucción invoca una operación de E/S, a la cual el proceso debe esperar.

(a) Traza del Proceso A	(b) Traza del Proceso B	(c) Traza del Proceso C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000 = Dirección de comienzo del programa del Proceso A.
8000 = Dirección de comienzo del programa del Proceso B.
12000 = Dirección de comienzo del programa del Proceso C.

Figura 3.3. Traza de los procesos de la Figura 3.2.

Ahora vea estas trazas desde el punto de vista del procesador. La Figura 3.4 muestra las trazas entrelazadas resultante de los 52 primeros ciclos de ejecución (por conveniencia los ciclos de instrucciones han sido numerados).

En este ejemplo, se asume que el sistema operativo sólo deja que un proceso continúe durante 6 ciclos de instrucción (rodaja de tiempo), después de los cuales se interrumpe (alarma temporizador). Esto previene que un solo proceso monopolice el uso del tiempo del procesador y provoque inanición en los demás. Además se supone que el planificador decide comenzar a ejecutar el proceso A. Por simplicidad se omite la ejecución de la rutina ISR por alarma de reloj y la ejecución del planificador.

1	5000		27	12004	
2	5001		28	12005	
3	5002				Temporización
4	5003		29	100	
5	5004		30	101	
6	5005		31	102	
		Temporización	32	103	
7	100		33	104	
8	101		34	105	
9	102		35	5006	
10	103		36	5007	
11	104		37	5008	
12	105		38	5009	
13	8000		39	5010	
14	8001		40	5011	
15	8002				Temporización
16	8003		41	100	
		Petición de E/S	42	101	
17	100		43	102	
18	101		44	103	
19	102		45	104	
20	103		46	105	
21	104		47	12006	
22	105		48	12007	
23	12000		49	12008	
24	12001		50	12009	
25	12002		51	12010	
26	12003		52	12011	
					Temporización

100 = Dirección de comienzo del programa activador.
Las zonas sombreadas indican la ejecución del proceso de activación;
la primera y la tercera columna cuentan ciclos de instrucciones;
la segunda y la cuarta columna las direcciones de las instrucciones que se ejecutan

Figura 3.4. Traza combinada de los procesos de la Figura 3.2.

Como muestra la Figura 3.4, las primeras 6 instrucciones del proceso A se ejecutan seguidas de una alarma de temporización y de la ejecución de cierto código del activador, que ejecuta 6 instrucciones antes de devolver el control al proceso B[1].

Después de que se ejecuten 4 instrucciones, el proceso B solicita una acción de E/S, para la cual debe esperar. Por tanto, el procesador deja de ejecutar el proceso B y pasa a ejecutar el proceso C, por medio del activador.

Después de otra alarma de temporización, el procesador vuelve al proceso A. Cuando este proceso llega a su temporización, el proceso B aún estará esperando que se complete su operación de E/S, por lo que el activador pasa de nuevo al proceso C.

4 Control de procedimientos mediante pila

Una técnica habitual para controlar la ejecución de llamadas a procedimiento o subrutinas y los retornos de las mismas en un proceso es utilizar una pila.

Una pila es un conjunto ordenado de elementos, tal que en cada momento solamente se puede acceder a uno de ellos, el más recientemente añadido. El número de elementos de la pila, o longitud de la pila, es variable. Por esta razón, se conoce también a una **pila** como una **lista donde el último que entra es el primero que sale** (*Last-In First-Out*, LIFO). Cada elemento ordenado contiene información sobre las rutinas que se invocan en un programa.

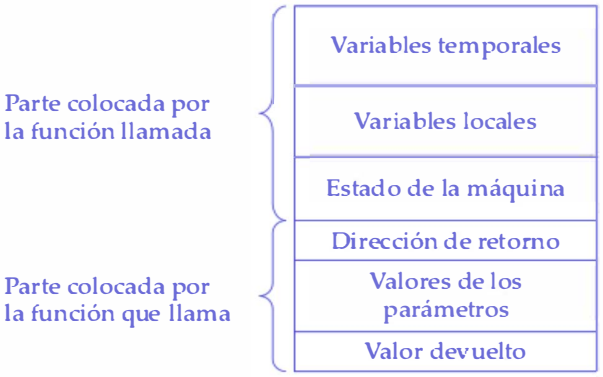
La implementación de la pila de un proceso requiere que haya un conjunto contiguo de posiciones de memoria dedicado a almacenar sus elementos. La mayoría de las veces el bloque de posiciones está parcialmente lleno con elementos y el resto está disponible para el crecimiento de la pila.

Por cada llamada que se realice desde el proceso en ejecución a una función o subrutina, en la pila (*stack*) se almacenará lo siguiente:

- La dirección de retorno, que es la dirección de memoria de la siguiente instrucción de código donde retornar después de ejecutarse la función o subrutina.
- Los datos pasados como parámetros a la función o subrutina.
- El valor de retorno de la función. No confundir con la dirección de retorno, es el valor devuelto por la función o subrutina (entero, flotante, etc).
- Las variables locales utilizadas por la función o subrutina llamada.
- Las variables temporales utilizadas por la función o subrutina llamada.
- El estado de la máquina (registros SP y FP del procesador), de los cuales se habla después.

1 El reducido número de instrucciones ejecutadas por los procesos y por el activador es irreal; se ha utilizado para simplificar el ejemplo y clarificar las explicaciones.

En la siguiente imagen se muestra lo que se llama **registro de activación**, que contiene todo lo anteriormente expuesto. Para cada función o subrutina llamada **se apila un registro de activación**.

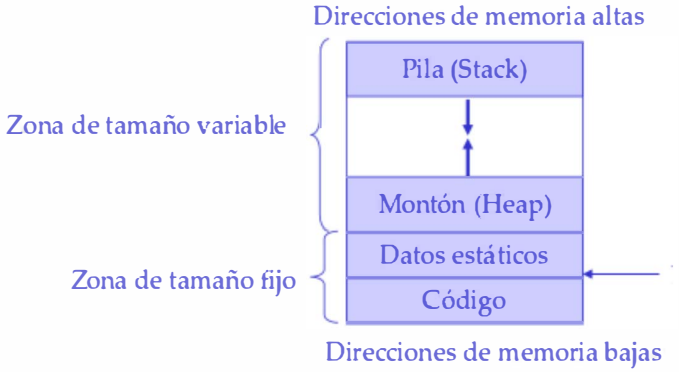


Ejemplo de variable temporal:

```
main()
{
  …
  a = funcion (b, x+y);
  …
  printf("%d\n", a + 20);
}
```

```
int funcion (int a, int b)
{
  int local = a+b;
  …
  return local;
}
```


Una vez que el sistema operativo ha reservado las zonas de memoria en el espacio de usuario y ha cargado el código de un programa, este está listo para ser ejecutado. Dicho esto, un proceso se compone de varias zonas de memoria principal diferenciadas (no tienen porque tener ese orden):



- **La pila de llamadas (*stack*).** Zona de memoria variable para las llamadas a subrutinas, que almacenará los registros de activación de llamadas a funciones o subrutinas.

  El procesador maneja dos registros especiales referentes a la pila (*stack*), que sirven para que una subrutina pueda acceder a los parámetros y variables que necesita, de forma que se tenga bien localizado su registro de activación. Por tanto, estos dos registros delimitan el espacio de memoria del último registro de activación apilado:

  - Puntero de pila o *Stack Pointer* (SP): Dirección de la cima de la pila.
  - Puntero de base o *Frame Pointer* (FP): Dirección base del registro de activación de la función que invoca a la subrutina.

- **El área de datos dinámicos o montón (*heap*).** Una zona para la gestión dinámica de la memoria que se solicita durante la ejecución, por tanto es una zona de memoria variable. Por ejemplo en C con la orden "*malloc()*", "*calloc()*", "*free()*", "*realloc()*", o en los lenguajes orientados a objetos con el operador "*new()*".
- **El área de datos estáticos:** Se usa para almacenar las variables globales, las constantes (enteros, reales, cadenas, …) y variables estáticas (*static* en C), reservando el espacio justo ya que se conoce en tiempo de compilación.
- **El área del código.** Se usa para almacenar el código fuente en instrucciones máquina, por consiguiente se reserva solamente el espacio justo.

Véase a continuación el procedimiento de llamada y retorno de una función según lo comentado hasta ahora.

Procedimiento de llamada:

- Se reserva espacio para el valor devuelto.
- Se almacena el valor de los parámetros que se pasan a la función llamada.
- Se almacena el valor de la dirección de retorno donde comenzar a ejecutar cuando finalice la función.
- Se almacena el estado de la máquina (incluye SP y FP).
- Se almacenan las variables locales y temporales.
- Comienza la ejecución del código de la función llamada.

Procedimiento de retorno es el siguiente:

- Se asigna el valor devuelto.
- Se restaura el contenido del estado de la máquina. Esto modifica el valor de FP y SP a los valores del proceso llamante, y provoca la liberación de la memoria ocupada la función llamada.
- Se restaura el valor de la dirección de retorno de la función que llama.

- Se devuelve el control a la función que llama.
- Se almacena el valor devuelto en la variable local o temporal de la función que llama.

5 Creación y terminación de procesos

Hay cuatro eventos principales que provocan la creación de procesos:

1) **El arranque del sistema:** Demonios y servicios como correo electrónico, servidor de páginas Web, gestor de impresión, gestor de bluetooth, gestor del entorno gráfico, gestor de planificación, gestor del sistema de ficheros, gestor de módulos de E/S, gestor de red, gestor de sonido, etc. En Gnu/Linux podemos utilizar el comando `service --status-all` para visualizar los servicios que se están ejecutando en el sistema.
2) La ejecución, desde un proceso, de una **llamada al sistema para creación de procesos**. Por ejemplo *fork()* mediante llamadas Posix.
3) **Una petición de usuario para crear un proceso.** En sistemas interactivos un doble clic para abrir un programa, o simplemente invocando a un ejecutable desde terminal.
4) **El inicio de un trabajo por lotes** (colección o batería de trabajos en cola). Se aplica sólo a los sistemas de procesamiento por lotes que se encuentran en las *mainframes* grandes. Aquí los usuarios pueden enviar trabajos de procesamiento por lotes al sistema, posiblemente en forma remota. Cuando el sistema operativo decide que tiene los recursos para ejecutar un trabajo de la cola, crea un proceso y ejecuta el siguiente trabajo de la cola de entrada.

Una vez que se crea un proceso, empieza a ejecutarse y realiza el trabajo al que está destinado. Sin embargo, nada dura para siempre, ni siquiera los procesos. Tarde o temprano el nuevo proceso terminará, por lo general debido a una de las siguientes condiciones (**voluntarias** o **involuntarias**):

- **Salida normal (voluntaria).** La mayoría de los procesos terminan debido a que han concluido su trabajo. Por ejemplo, cuando un compilador compila un programa añade implícitamente una llamada al sistema para indicar al sistema operativo su terminación normal. Esta llamada es *exit()* en sistemas POSIX. En sistemas con entorno gráfico, los procesadores de palabras, navegadores de Internet y programas similares siempre tienen un icono o elemento de menú en el que el usuario puede hacer clic para indicar al proceso que elimine todos los archivos temporales que tenga abiertos y después termine. Esa acción en realidad lleva a invocar a *exit()*.
- **Salida por error (voluntaria).** La segunda razón de terminación es que el proceso descubra un error. Por ejemplo, si un usuario escribe el comando "*cc foo.c*" para compilar el programa "foo.c" y no existe dicho archivo, el compilador simplemente termina. Note que este error es predecible y tenido en cuenta por el proceso, ya que es el programador el que en su código contempló la opción de terminar el programa en el caso de que no se encontrase el fichero.
- **Error fatal (involuntaria)**. La tercera razón de terminación es un error fatal producido por el proceso, a menudo debido a un error en el programa. Algunos ejemplos incluyen el ejecutar una instrucción ilegal, hacer referencia a una parte de memoria no existente o la división entre cero. En algunos sistemas, un proceso puede indicar al sistema operativo que desea manejar ciertos errores por sí mismo, en cuyo caso el proceso recibe una señal que

puede tratar en vez de terminar el proceso. Esto forma parte del tratamiento y captura de excepciones de los programas. Java tiene un tratamiento de excepciones muy potente.

- **Eliminado por otro proceso (involuntaria).** La cuarta razón por la que un proceso podría terminar es que algún otro proceso ejecute una llamada al sistema que indique que lo elimine. En sistemas POSX esta llamada es *kill()*.

6 *Un modelo de procesos de dos estados*

La responsabilidad principal del sistema operativo es controlar la ejecución de los procesos y asignar recursos a los mismos. El primer paso en el diseño de un sistema operativo para el control de procesos es construir un modelo de comportamiento para los procesos. Se puede construir el modelo más simple posible observando que, en un instante dado, un proceso está siendo ejecutando por el procesador o no. En este modelo, un proceso puede estar en dos estados: Ejecutando o No Ejecutando, como se muestra en la Figura 3.5(a).

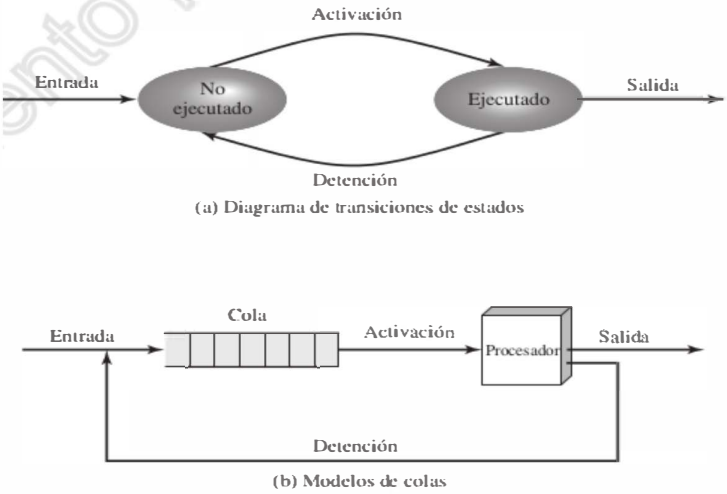


Figura 3.5. Modelo de proceso de dos estados.

Cuando el sistema operativo crea un nuevo proceso, se crea el BCP para el nuevo proceso e inserta dicho proceso en el sistema en estado No Ejecutando. El proceso existe, es conocido por el sistema operativo, y está esperando su oportunidad de ejecutar. De cuando en cuando, el proceso actualmente en ejecución se interrumpirá y una parte del sistema operativo, el planificador, seleccionará otro proceso a ejecutar y se lo hará saber al *dispatcher*. El proceso saliente pasará del estado Ejecutando a No Ejecutando y pasará a Ejecutando un nuevo proceso.

En este modelo simple los procesos que no están ejecutando deben estar en una especie de cola FIFO, esperando su turno de ejecución. Existe una sola cola cuyas entradas son punteros al BCP de un proceso en particular (Figura 3.5(b)). Un proceso que se interrumpe por rodaja de tiempo se transfiere a la cola de procesos en espera. Alternativamente, si el proceso ha finalizado o ha sido abortado, se descarta y sale del sistema. En cualquier caso, el *dispatcher* recoge un proceso de la cola para ejecutar.

7 Un modelo de procesos de cinco estados

Si todos los procesos estuviesen siempre preparados para ejecutar, la gestión de colas proporcionada en la Figura 3.5(b) sería efectiva. La cola es una lista de tipo FIFO y el procesador opera siguiendo una estrategia cíclica *round-robin* o turno rotatorio sobre todos los procesos disponibles (cada proceso de la cola tiene cierta cantidad de tiempo, por turnos, para ejecutar y regresar de nuevo a la cola).

Sin embargo, este modelo es inadecuado: algunos procesos que están en el estado de No Ejecutando están listos para ejecutar, mientras que otros podrían estar bloqueados, esperando a que se complete una operación de E/S. Por tanto, utilizando una única cola, el activador no puede seleccionar únicamente los procesos que lleven más tiempo en la cola, en su lugar debería recorrer la lista buscando los procesos que no estén bloqueados y que lleven en la cola más tiempo.

Una forma más natural para manejar esta situación es dividir el estado de No Ejecutando en dos estados, **Listo** y **Bloqueado**. Esto se muestra la Figura 3.6.

Figura 3.6. Modelo de proceso de cinco estados.

Para gestionarlo correctamente, se han añadido dos estados adicionales que resultarán muy útiles. Estos cinco estados en el nuevo diagrama son los siguientes:

- **Ejecutando**. El proceso está actualmente en ejecución. Para una mejor comprensión, asumimos que el computador tiene un único procesador con un solo núcleo, de forma que sólo un proceso puede estar en este estado en un instante determinado.
- **Listo**. Un proceso que se prepara para ejecutar cuando tenga oportunidad y lo decida el planificador.
- **Bloqueado**. Un proceso que no puede ejecutar hasta que se cumpla un evento determinado, por ejemplo que se complete una operación E/S.
- **Nuevo**. Un proceso que se acaba de crear y que aún no ha sido admitido en el grupo de procesos ejecutables por el sistema operativo. Típicamente, se trata de un nuevo proceso que no ha sido cargado en memoria principal, aunque su BCP si ha sido creado.
- **Saliente**. Un proceso que ha sido liberado del grupo de procesos ejecutables por el sistema

operativo, debido a que ha sido detenido o que ha sido abortado por alguna razón (ya sea voluntaria o involuntaria).

Los estados Nuevo y Saliente son útiles para construir la gestión de procesos:

- El estado Nuevo se corresponde con un proceso que acaba de ser definido. Cuando se solicita un nuevo trabajo a un sistema de proceso por lotes, el sistema operativo realiza todas las tareas internas que correspondan: se asocia un identificador a dicho proceso y se construyen todas aquellas tablas que se necesiten para gestionar al proceso. En este punto, el proceso se encuentra el estado Nuevo. Esto significa que el sistema operativo ha realizado todas las tareas necesarias para crear el proceso pero el proceso en sí, aún no se ha puesto en ejecución, no está cargado en RAM.

  Por ejemplo, un sistema operativo puede limitar el número de procesos que puede haber por razones de rendimiento (CPU) o limitaciones de memoria principal (RAM). Mientras un proceso está en el estado Nuevo, la información relativa al proceso que se necesite por parte del sistema operativo se mantiene en tablas de control de memoria principal. Sin embargo, el proceso en sí mismo no se encuentra en memoria principal, es decir, la zona de código, datos, pila y montículo no se encuentra en memoria principal, sino que permanece en almacenamiento secundario, normalmente en disco duro.

- De forma similar, para que un proceso salga del sistema, primero ha de terminar cuando alcance su punto de finalización natural, cuando es abortado debido a un error no recuperable, o cuando otro proceso con autoridad apropiada causa que el proceso se aborte. La terminación mueve el proceso al estado Saliente. En este punto, el proceso no es elegible de nuevo para su ejecución.

  Las tablas y otra información asociada con el proceso saliente se encuentran temporalmente preservadas por el sistema operativo, el cual proporciona tiempo para que programas auxiliares o de soporte extraigan la información necesaria. Por ejemplo, un programa de auditoría puede requerir registrar el tiempo de proceso y otros recursos utilizados por este proceso saliente con objeto de realizar una contabilidad de los recursos del sistema; un programa de utilidad puede requerir extraer información sobre el histórico de los procesos por temas relativos con el rendimiento o análisis de la utilización. Una vez que estos programas han extraído la información necesaria, el sistema operativo no necesita mantener ningún dato relativo al proceso y el proceso se borra del sistema, junto con sus estructuras utilizadas.

7.1 Transición entre estados

La Figura 3.6 indica que tipos de eventos llevan a cada transición de estado (en un modelo de 5 estados) para cada proceso. Las posibles transiciones son las siguientes:

- **Null → Nuevo**. Se crea un nuevo proceso para ejecutar un programa. Este evento ocurre por cualquiera de las relaciones indicadas en secciones anteriores.
- **Nuevo → Listo**. El sistema operativo mueve a un proceso del estado nuevo al estado listo cuando éste se encuentre preparado para ejecutar un nuevo proceso. La mayoría de sistemas fijan un límite basado en el número de procesos existentes o la cantidad de memoria virtual

que se podrá utilizar por parte de los procesos existentes. Este límite asegura que no haya demasiados procesos activos y que se degrade el rendimiento del sistema por falta de memoria RAM, memoria virtual o capacidad de computo de la CPU.

- **Listo → Ejecutando**. Cuando llega el momento de seleccionar un nuevo proceso para ejecutar, el sistema operativo selecciona uno de los procesos que se encuentre en el estado Listo. Esta es una tarea que lleva acabo el planificador.
- **Ejecutando → Saliente**. El proceso actual en ejecución se finaliza por parte del sistema operativo tanto si el proceso indica que ha completado su ejecución como si éste se aborta.
- **Ejecutando → Listo**. La razón más habitual para esta transición es que el proceso en ejecución haya alcanzado el máximo tiempo posible de ejecución de forma ininterrumpida (interrupción por *alarma de reloj*). Prácticamente todos los sistemas operativos multiprogramados imponen este tipo de restricción de tiempo.

  Existen otras posibles causas alternativas para esta transición que no están incluidas en todos los sistemas operativos, siendo de particular importancia el caso en el cual el sistema operativo asigna diferentes *niveles de prioridad a diferentes procesos*. Supóngase, por ejemplo, que el proceso A está ejecutando a un determinado nivel de prioridad, y el proceso B a un nivel de prioridad mayor, y que se encuentra bloqueado. Si el sistema operativo se da cuenta de que se produce un evento por el cual el proceso B está esperando, moverá el proceso B al estado de Listo. Esto puede interrumpir al proceso A y poner en ejecución al proceso B. Decimos, en este caso, que el sistema operativo ha expulsado al proceso A.
- **Ejecutando → Bloqueado**. Un proceso se pone en el estado Bloqueado si solicita algo por lo cual debe esperar o no está disponible, por ejemplo:
  - Un proceso ha solicitado un servicio que el sistema operativo no puede realizar en ese momento.
  - Un proceso ha solicitado una operación de E/S.
  - Cuando un proceso se comunica con otro, un proceso puede bloquearse mientras está esperando a que otro proceso le proporcione datos o esperando un mensaje de ese otro proceso.
- **Bloqueado → Listo**. Un proceso en estado Bloqueado se mueve al estado Listo cuando sucede el evento por el cual estaba esperando.
- **Listo → Saliente.** Por claridad, esta transición no se muestra en el diagrama de estados. En algunos sistemas, un padre puede terminar la ejecución de un proceso hijo en cualquier momento. Incluso se puede producir si un proceso es terminado mediante una señal por otro proceso.
- **Bloqueado → Saliente.** Se aplican los comentarios indicados en el caso anterior.

Si regresamos a nuestro ejemplo de secciones anteriores, en la Figura 3.7 se muestra la transición entre cada uno de los estados de proceso.

1	5000	
2	5001	
3	5002	
4	5003	
5	5004	
6	5005	
		Temporización
7	100	
8	101	
9	102	
10	103	
11	104	
12	105	
13	8000	
14	8001	
15	8002	
16	8003	
		Petición de E/S
17	100	
18	101	
19	102	
20	103	
21	104	
22	105	
23	12000	
24	12001	
25	12002	
26	12003	
27	12004	
28	12005	
		Temporización
29	100	
30	101	
31	102	
32	103	
33	104	
34	105	
35	5006	
36	5007	
37	5008	
38	5009	
39	5010	
40	5011	
		Temporización
41	100	
42	101	
43	102	
44	103	
45	104	
46	105	
47	12006	
48	12007	
49	12008	
50	12009	
51	12010	
52	12011	
		Temporización

100 = Dirección de comienzo del programa activador.
Las zonas sombreadas indican la ejecución del proceso de activación;
la primera y la tercera columna cuentan ciclos de instrucciones;
la segunda y la cuarta columna las direcciones de las instrucciones que se ejecutan

Figura 3.4. Traza combinada de los procesos de la Figura 3.2.

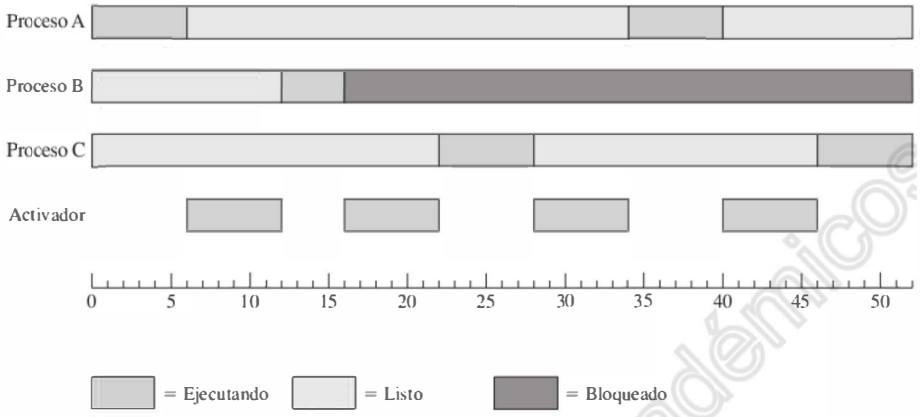
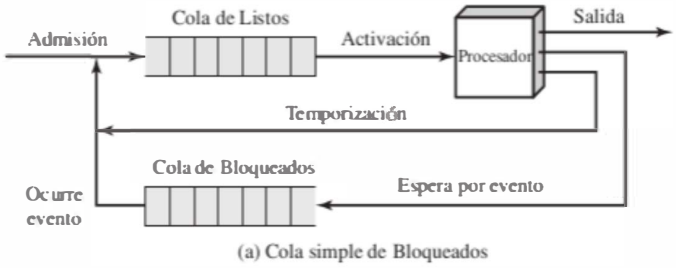


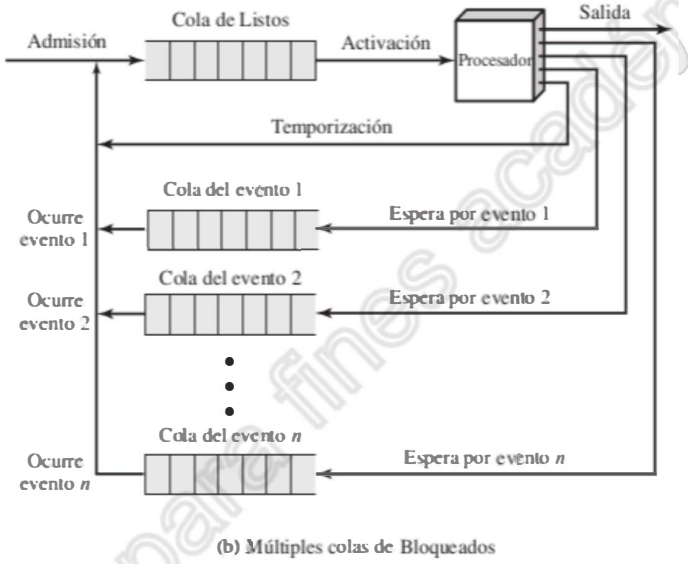
Figura 3.7. Estado de los procesos de la traza de la Figura 3.4.

7.2 Un posible esquema de colas

La figura 3.8 (a) sugiere la forma de aplicar un esquema de dos colas, la **colas de Listos** y la **cola de Bloqueados**.



(a) Cola simple de Bloqueados



(b) Múltiples colas de Bloqueados

Figura 3.8. Modelo de colas de la Figura 3.6.

Cada proceso admitido por el sistema, se coloca en la cola de Listos. Cuando llega el momento de que el sistema operativo seleccione otro proceso a ejecutar, selecciona uno de la cola de Listos. En ausencia de un esquema de prioridad, esta cola puede ser una lista de tipo FIFO (*first-in-first-out*). Cuando el proceso en ejecución termina de utilizar el procesador: o bien finaliza, o bien se coloca en la cola de Listos o de Bloqueados, dependiendo de las circunstancias.

Por último, cuando sucede un evento, cualquier proceso en la cola de Bloqueados que únicamente esté esperando a dicho evento, se mueve a la cola de Listos. Esta última transición significa que, cuando sucede un evento, el sistema operativo debe recorrer la cola entera de Bloqueados, buscando aquellos procesos que estén esperando por dicho evento. En los sistemas operativos con muchos procesos, esto puede significar cientos o incluso miles de procesos en esta lista, por lo que sería mucho más eficiente tener una cola por cada evento. De esta forma, cuando

sucede un evento, la lista entera de procesos de la cola correspondiente se movería al estado de Listo, Figura 3. 8(b).

Un refinamiento final sería: si la activación de procesos está dictada por un esquema de prioridades, sería conveniente tener varias colas de procesos listos, una por cada nivel de prioridad. El sistema operativo podría determinar cual es el proceso listo de mayor prioridad simplemente seleccionando éstas en orden.

Los tres principales estados descritos (Listo, Ejecutando, Bloqueado) proporcionan una forma sistemática de modelizar el comportamiento de los procesos y diseñar la implementación del sistema operativo. Se han construido algunos sistemas operativos utilizando únicamente estos tres estados.

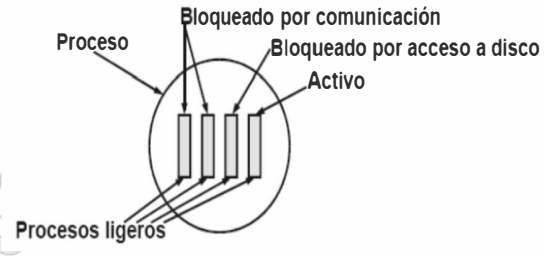
8 Hilos (hebras)

El concepto de proceso que se ha presentado hasta ahora supone que es un programa en ejecución con una sola hebra o hilo de control. Ahora, la mayoría de los sistemas operativos modernos proporcionan características que permiten que un proceso tenga múltiples hilos de control.

Un **hilo**, también llamado **proceso ligero o hebra** (se usará indistintamente a partir de ahora), es una unidad básica de utilización de la CPU. Si un **proceso** tradicional, también llamado **proceso pesado**, tiene múltiples hilos de control, puede realizar más de una tarea a la vez, al mismo tiempo.

Dicho esto, dentro de un proceso puede haber uno o más hilos, cada uno con:

- Un estado de ejecución por hilo (Ejecutando, Listo, etc.).



- Un contexto o BCP de hilo, al igual que con los procesos. Una forma de ver a un hilo es como un contador de programa independiente dentro de un proceso.
- Una pila de ejecución.
- Un espacio de almacenamiento para variables locales.
- Acceso a la memoria y recursos de su proceso, compartido con todos los hilos de su mismo proceso.

Entonces, en un entorno multihilo, sigue habiendo un único BCP y un espacio de direcciones de usuario asociado al proceso, pero ahora hay varias pilas separadas para cada hilo, así como **un bloque de control para cada hilo** que contiene los valores de los registros, la prioridad, y otra información relativa al estado del hilo. De esta forma, todos los hilos de un proceso comparten el

estado y los recursos de ese proceso, residen en el mismo espacio de direcciones y tienen acceso a los mismos datos. Cuando un hilo cambia determinados datos en memoria, otros hilos ven los resultados cuando acceden a estos datos (no en las variables locales). Si un hilo abre un archivo con permisos de lectura, los demás hilos del mismo proceso pueden también leer ese archivo. La Figura 4.1 ilustra la diferencia entre un proceso tradicional monohilo y un proceso multihilo:

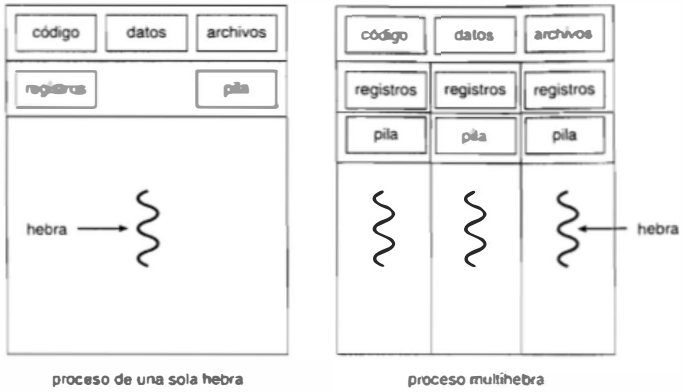
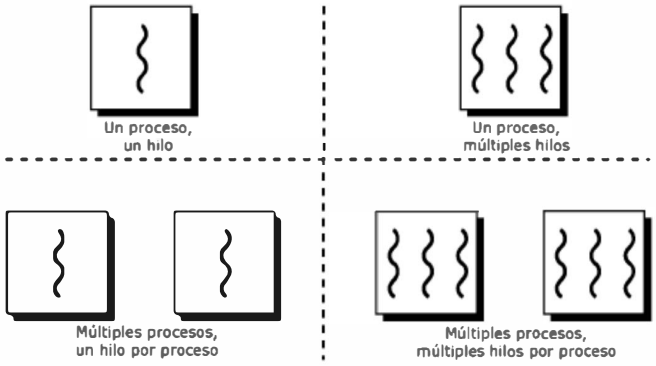


Figura 4.1 Procesos monohebra y multihebra.

En la siguiente Figura, las dos configuraciones que se muestran en la parte izquierda son estrategias monohilo y las de la derecha multihilo:



8.1 Estados de los hilos

En un sistema operativo que soporte hilos, la planificación y la activación se realizan a nivel de hilo, por tanto, los principales estados de los hilos son: Ejecutando, Listo y Bloqueado. Generalmente, no tiene sentido aplicar estados de suspensión a un hilo, ya que dichos estados son conceptos de nivel de proceso. Existen, sin embargo, diversas acciones que afectan a todos los hilos de un proceso y que el sistema operativo debe gestionar a nivel de proceso. Suspender un proceso implica expulsar el espacio de direcciones de un proceso de memoria principal para dejar hueco a otro espacio de direcciones de otro proceso. Ya que todos los hilos de un proceso comparten el mismo espacio de direcciones, todos los hilos se suspenden al mismo tiempo. De forma similar, la

finalización de un proceso finaliza todos los hilos de ese proceso.

Hay cuatro operaciones básicas relacionadas con los hilos que están asociadas con un cambio de estado del hilo:

- **Creación**. Cuando se crea un nuevo proceso, también se crea un hilo de dicho proceso. Posteriormente, un hilo del proceso puede crear otro hilo dentro del mismo proceso, proporcionando un puntero a las instrucciones y los argumentos para el nuevo hilo. Al nuevo hilo se le proporciona su propio registro de contexto y espacio de pila y se coloca en la cola de Listos.
- **Bloqueo**. Cuando un hilo necesita esperar por un evento se bloquea, almacenando los registros de usuario, contador de programa y punteros de pila. El procesador puede pasar a ejecutar otro hilo en estado Listo, dentro del mismo proceso o en otro diferente.
- **Desbloqueo**. Cuando sucede el evento por el que el hilo está bloqueado, el hilo se pasa a la cola de Listos.
- **Finalización**. Cuando se completa un hilo, se liberan su registro de contexto y pilas.

8.2 Motivación y ventajas

Muchos paquetes de software que se ejecutan en los PC modernos de escritorio son multihebra. Normalmente, una aplicación se implementa como un proceso propio con varias hebras de control. Actualmente muchos *kernel* de sistemas operativos son multihebra; hay varias hebras operando en el kernel y cada hebra realiza una tarea específica, tal como gestionar dispositivos o tratar interrupciones. Por ejemplo, Solaris crea un conjunto de hebras en el kernel específicamente para el tratamiento de interrupciones; Gnu/Linux utiliza una hebra del kernel para gestionar la cantidad de memoria libre en el sistema.

Como ejemplo de aplicación multihebra, un servidor web acepta solicitudes de los clientes que piden páginas web, imágenes, sonido, etc. Un servidor web sometido a una gran carga puede tener varios (quizá, miles) de clientes accediendo de forma concurrente a él. Si el servidor web funcionara como un proceso tradicional de una sola hebra, sólo podría dar servicio a un cliente cada vez y la cantidad de tiempo que un cliente podría tener que esperar para que su solicitud fuera servida podría ser enorme. Una solución es que el servidor funcione como un solo proceso de aceptación de solicitudes. Cuando el servidor recibe una solicitud, crea otro proceso para dar servicio a dicha solicitud. Según este método, lo que se hace es dividir en múltiples hebras el proceso servidor web. El servidor crea una hebra específica para escuchar las solicitudes de cliente y cuando llega una solicitud, en lugar de crear otro proceso, el servidor crea otra hebra para dar servicio a la solicitud.

Los hilos también se pueden utilizar para **trabajos en primer plano y en segundo plano.** Por ejemplo, un procesador de textos puede tener una hebra para mostrar gráficos, otra hebra para responder a las pulsaciones de teclado del usuario y una tercera hebra para el corrector ortográfico y gramatical que se ejecuta en segundo plano.

También se usan en procesamiento asíncrono, por ejemplo, se puede diseñar un procesador de textos con protección contra un fallo de corriente que escriba el buffer de su memoria RAM a

disco una vez por minuto.

Dicho esto, las **ventajas de la programación multihebra** pueden dividirse en cuatro categorías principales:

1. **Capacidad de respuesta**. El uso de múltiples hebras en una aplicación interactiva permite que un programa continúe ejecutándose incluso aunque parte de él esté bloqueado o realizando una operación muy larga, lo que incrementa la capacidad de respuesta al usuario. Por ejemplo, un explorador web multihebra permite la interacción del usuario a través de una hebra mientras que en otra hebra se está cargando una imagen.
2. **Compartición de recursos.** Por omisión, las hebras comparten la memoria y los recursos del proceso al que pertenecen, no siendo necesario la duplicidad.
3. **Economía y tiempo de ejecución**. La asignación de memoria y recursos para la creación de procesos es costosa. En general, se consume mucho más tiempo en crear y gestionar los procesos que las hebras. Por ejemplo, en Solaris, crear un proceso es treinta veces lento que crear una hebra, y el cambio de contexto es aproximadamente cinco veces más lento.
    - **Crear o terminar una hebra** es mucho más rápido que crear o terminar un proceso, ya que las hebras comparten determinados recursos del padre que no hay que crear de nuevo. Cuando se termina un proceso se debe eliminar el BCP del mismo y todas las estructuras de datos y espacio de memoria de usuario ocupado, mientras que si se termina un hilo se elimina solo su contexto y pila.
    - El **cambio de contexto entre hebras** es realizado mucho más rápidamente que el cambio de contexto entre procesos, mejorando el rendimiento del sistema. Al cambiar de un proceso a otro, el sistema operativo genera lo que se conoce como *overhead*, que es tiempo "desperdiciado" por el procesador para realizar un cambio de contexto por medio del *dispatcher*. En los hilos, como pertenecen a un mismo proceso, al realizar un cambio de hilo el tiempo perdido es casi despreciable, ya que hay estructuras de datos que no son necesarias de salvar/restaurar, concretamente aquellas que comparten todas las hebras de un mismo proceso.
    - Si se necesitase **comunicación entre hebras** también sería mucho más rápido que intercomunicar procesos, ya que los datos están inmediatamente habilitados y disponibles entre hebras (los hilos dentro de un mismo proceso comparten memoria y archivos). En la mayoría de los sistemas, en la comunicación entre procesos, debe intervenir el núcleo para ofrecer protección de los recursos y realizar la comunicación . En cambio, entre hilos pueden comunicarse entre sí sin la invocación al núcleo. Por lo tanto, si hay una aplicación que debe implementarse como un conjunto de unidades de ejecución relacionadas, es más eficiente hacerlo con una colección de hilos que con una colección de procesos separados.
4. **Utilización sobre arquitecturas multiprocesador**. Las ventajas de usar configuraciones multihebra pueden verse incrementadas significativamente en una arquitectura multiprocesador, donde las hebras pueden ejecutarse en paralelo en los diferentes procesadores. Un proceso monohebra sólo se puede ejecutar en una CPU, independientemente de cuántas haya disponibles. Los mecanismos multihebra en una

máquina con varias CPU incrementan el grado de concurrencia.

De esta forma, si se desea implementar una aplicación o función como un conjunto de unidades de ejecución relacionadas, es mucho más eficiente hacerlo con un conjunto de hilos que con un conjunto de procesos independientes.

8.3 Modelos multihilo

Hasta ahora, nuestra exposición se ha ocupado de las hebras en sentido genérico. Sin embargo, desde el punto de vista práctico, el soporte para hebras puede proporcionarse en el nivel de usuario (para las hebras de usuario) o por parte del kernel (para las hebras del kernel). El soporte para las hebras de usuario se proporciona por encima del kernel y las hebras se gestionan sin soporte del mismo, mientras que el sistema operativo soporta y gestiona directamente las hebras del kernel.

Casi todos los sistemas operativos actuales, incluyendo Gnu/Linux, Windows y Mac OS, soportan las hebras a nivel de kernel (son las que se usan en las prácticas de la asignatura).

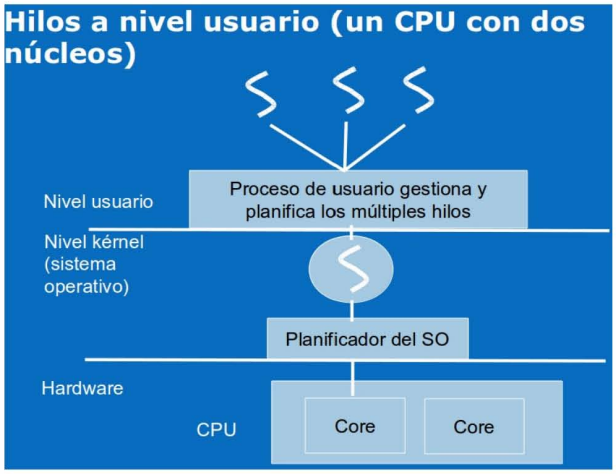
8.3.1 Modelos muchos a uno

El modelo muchos-a-uno asigna múltiples hebras del nivel de usuario a una hebra del kernel. La gestión de hebras se hace mediante la biblioteca de hebras en el espacio de usuario, por lo que resulta eficiente, pero el proceso completo se bloquea si una hebra realiza una llamada bloqueante al sistema. También, dado que sólo una hebra puede acceder al kernel cada vez, no podrán ejecutarse varias hebras en paralelo sobre múltiples procesadores. La biblioteca de hilos contiene código para la creación y destrucción de hilos, para paso de mensajes y datos entre los hilos, para planificar la ejecución de los hilos, y para guardar y restaurar el contexto de los hilos.

Toda esto tiene lugar dentro de un solo proceso, el núcleo no es consciente de esta actividad, los hilos son invisibles para el. El núcleo continúa planificando el proceso como una unidad y asigna al proceso un único estado (Listo, Ejecutando, Bloqueado, etc.).

Principalmente se utilizaban en las aplicaciones que se ejecutaban en sistemas operativos que no podían planificar hilos. El sistema de hebras *Green*, una biblioteca de hebras disponible en Solaris, usa este modelo, así como *GNU Portable Threads* (antiguos hilos basados en POSIX). Este tipo de hilos gestionados por bibliotecas tienen una ventaja, y es que son muy portables, ya que pueden pueden ejecutar en cualquier sistema operativo, acepte o no hilos. No se necesita ningún cambio en el nuevo núcleo para darles soporte. La biblioteca de los hilos es un conjunto de utilidades a nivel de aplicación que pueden usar todas las aplicaciones del sistema.

Como desventaja principal, si tenemos en cuenta que en un sistema operativo típico muchas llamadas al sistema son bloqueantes, cuando un proceso realiza una operación de E/S bloqueadora, como puede ser una lectura del teclado, éste pasa a estado bloqueado mientras se espera esa entrada y el sistema operativo pasa a ejecutar otro proceso. Si esto lo aplicamos a hilos, cuando un hilo a nivel de biblioteca realiza una llamada al sistema bloqueante, no sólo se bloquea ese hilo, sino que se bloquean todos los hilos del proceso. Por tanto, no hay paralelismo real, no se saca ventaja al concepto de multiprocesamiento.

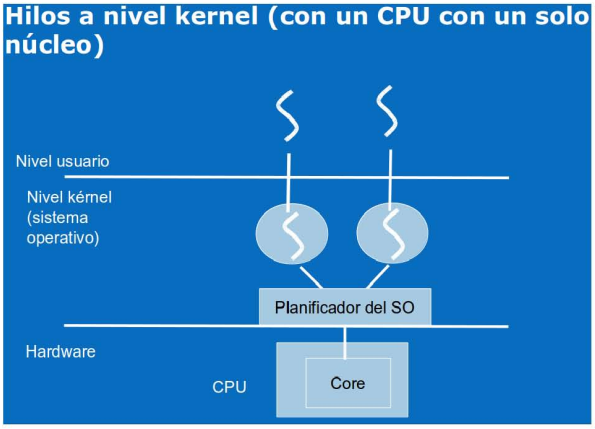


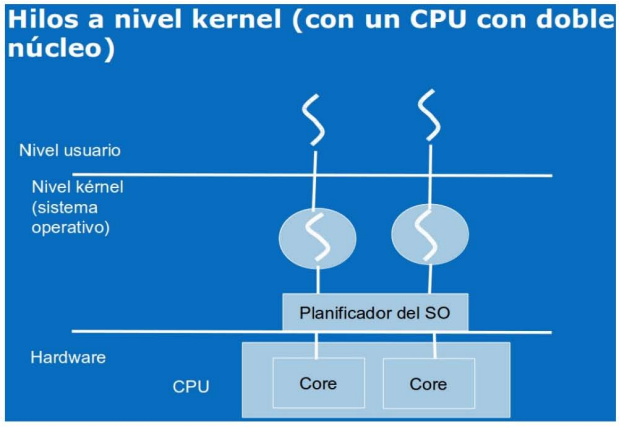
8.3.2 Modelos uno a uno

El modelo uno-a-uno asigna cada hebra de usuario a una hebra del kernel. Proporciona una mayor concurrencia que el modelo muchos-a-uno, permitiendo que se ejecute otra hebra mientras una hebra hace una llamada bloqueante al sistema; también permite que se ejecuten múltiples hebras en paralelo sobre varios procesadores.

No hay código de gestión de hilos en la aplicación a nivel de biblioteca (planificación), solamente una interfaz de programación de aplicación (API) para acceder a las utilidades de hilos del núcleo (creación, terminación, comunicación entre hilos). El núcleo mantiene información de contexto del proceso como una entidad y de los hilos individuales del proceso. La planificación realizada por el núcleo se realiza a nivel de hilo.

El único inconveniente de este modelo es que crear una hebra, como cualquier otro proceso, requiere una llamada al kernel, produciéndose un cambio a modo núcleo. Lo mismo para la terminación y otras gestiones que el usuario pueda realizar dese la API. También sabemos que cuando el sistema operativo decide que la rodaja o cuanto de tiempo de un proceso ha terminado o tiene que bloquearlo por una interrupción, hay que buscar otro proceso de la lista de Listos, y esto supone guardar el contexto actual y cargar otro nuevo contexto de proceso a partir del *dispatcher*, que también se ejecuta en modo núcleo. Todo ello supone una carga de trabajo administrativa adicional que repercutir en el rendimiento de una aplicación, por lo que la mayoría de las implementaciones de este modelo restringen el número de hebras soportadas por el sistema. Gnu/Linux, junto con la familia de sistemas operativos Windows implementan el modelo uno-a-uno.





Dicho esto, existen dos formas principales de implementar una biblioteca de hebras:

1. El primer método consiste en proporcionar una biblioteca enteramente en el espacio de usuario, sin ningún soporte del kernel. Todas las estructuras de datos y el código de la biblioteca se encuentran en el espacio de usuario. Esto significa que invocar a una función de la biblioteca es como realizar una llamada a una función local en el espacio de usuario y no una llamada al sistema.
2. El segundo método consiste en implementar una biblioteca en el nivel del kernel, soportada directamente por el sistema operativo. En este caso, el código y las estructuras de datos de la biblioteca se encuentran en el espacio del kernel. Invocar una función en la API de la biblioteca normalmente da lugar a que se produzca una llamada al sistema dirigida al kernel.

   En las prácticas de la asignatura se usaran este último tipo de bibliotecas de hebras, concretamente la biblioteca *pthead* de POSIX.