Unidad 2:
Gestión de Procesos

Tema 4, Procesos:

4.1 El concepto de proceso.
4.2 Planificación de procesos.
4.3 Procesos cooperativos.
4.4 Hilos (threads).

4.1 El concepto de proceso.

- Un proceso es **cualquier programa en ejecución**.
- Un proceso **necesita** ciertos **recursos** para realizar satisfactoriamente su tarea:
  - Tiempo de CPU.
  - Memoria.
  - Archivos.
  - Dispositivos de E/S.
- Los recursos se asignan a un proceso:
  - Cuando se crea.
  - Durante su ejecución.

4.1 El concepto de proceso.

- Un **sistema** consiste en una **colección de procesos** que podrían ejecutarse **concurrentemente**.

- Las obligaciones del SO como gestor de procesos son:

  - **Creación y eliminación** de procesos.
  - **Planificación de procesos** (procurando la ejecución de múltiples procesos maximizando la utilización del procesador).
  - Establecimiento de mecanismos para la **sincronización y comunicación** de procesos.
  - Manejo de **bloqueos mutuos**.

4.1 El concepto de proceso.

- Un proceso es:
  - **Sección de texto** (código del programa).
  - Actividad actual, representada por:
    - Valor del **contador de programa.**
    - Contenido de **registros** del procesador.
  - Además, también incluye:
    - **Pila** (stack), que contiene **datos temporales** (parámetros de subrutinas, direcciones de retorno y variables locales).
    - **Sección de datos**, que contiene **variables globales y memoria dinámica**.

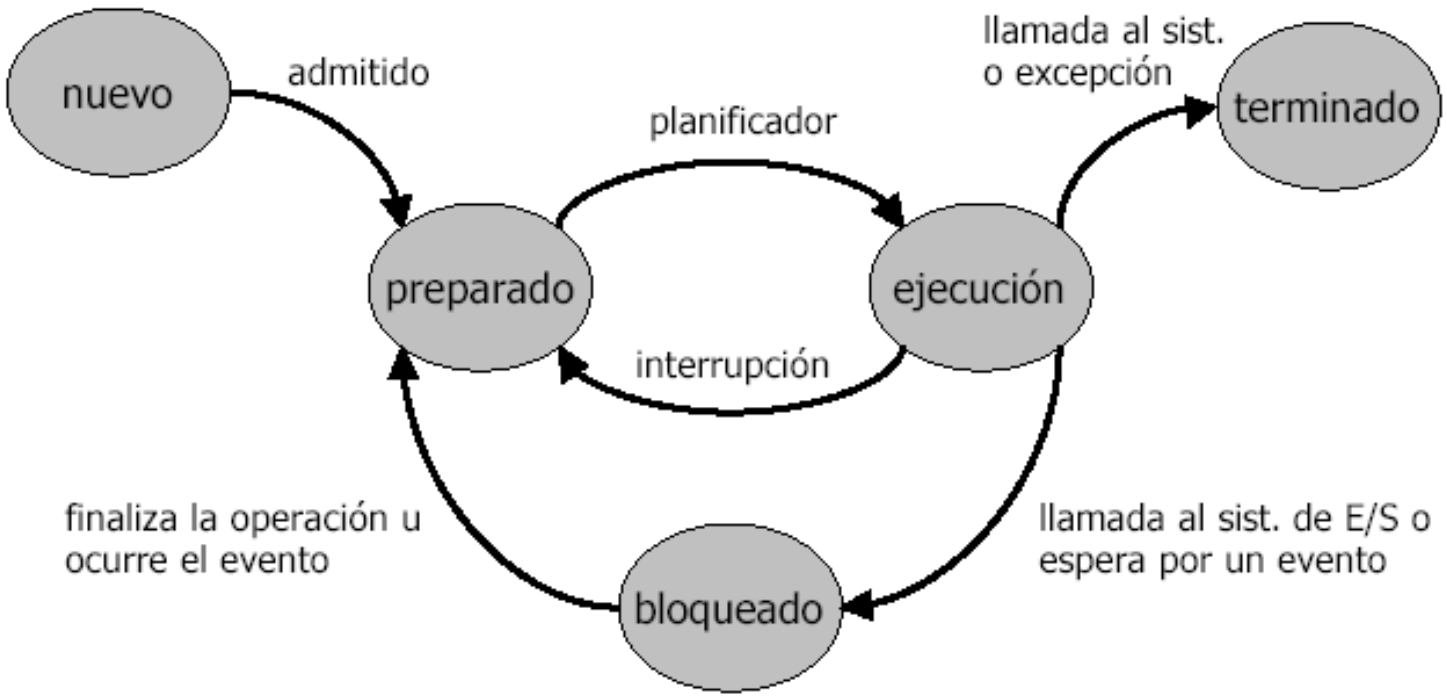
4.1 El concepto de proceso: Estados de un proceso.

- A medida que un **proceso se ejecuta cambia de estado**. Cada proceso puede estar en uno de los estados:
  - **Nuevo** (new): el proceso se está creando.
  - **En ejecución** (running): el proceso está en la CPU ejecutando instrucciones.
  - **Bloqueado** (waiting, en espera): proceso esperando a que ocurra un suceso (ej. terminación de E/S o recepción de una señal).
  - **Preparado** (ready, listo): esperando que se le asigne a un procesador.
  - **Terminado** (terminated): finalizó su ejecución, por tanto no ejecuta más instrucciones y el SO le retirará los recursos que consume.

- Nota: **Sólo un proceso puede estar ejecutándose en cualquier procesador en un instante dado**, pero muchos procesos pueden estar listos y esperando.

4.1 El concepto de proceso: Estados de un proceso.

- Diagrama de estados de un proceso:



4.1 El concepto de proceso: Estados de un proceso.

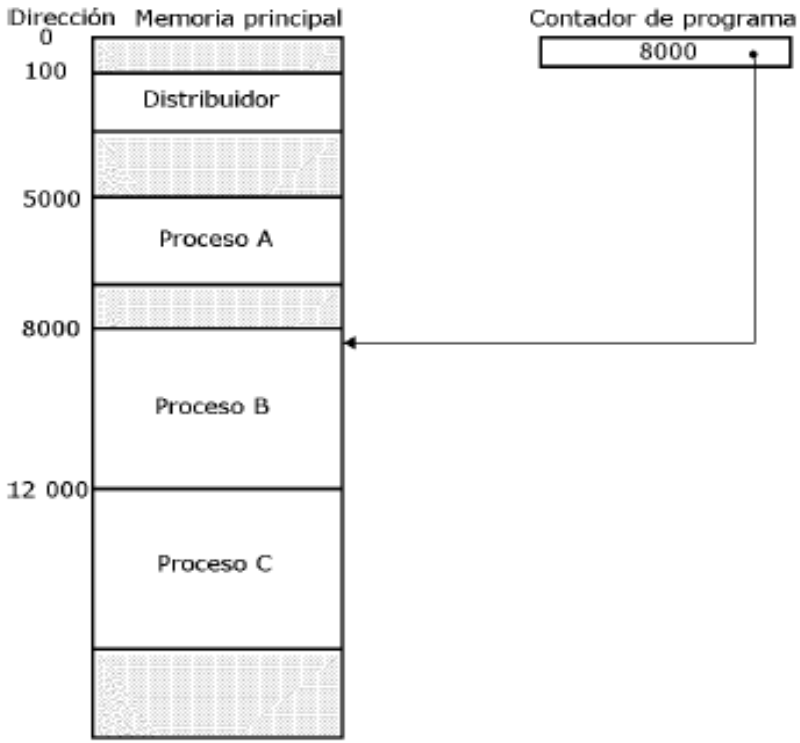
- Para que un programa se ejecute, el SO debe crear un proceso para él. En un **sistema con multiprogramación el procesador ejecuta código de distintos programas que pertenecen a distintos procesos**.

- Aunque dos procesos estén asociados al mismo programa, se consideran **dos secuencias de ejecución separadas**, cada una de las cuales se considera un proceso.

- Llamamos **traza** de un proceso al **listado** de la secuencia **de instrucciones** que se ejecutan para el mismo.

4.1 El concepto de proceso: Estados de un proceso.

- Ejemplo: disposición en memoria de tres procesos.



4.1 El concepto de proceso: Estados de un proceso.

- Ejemplo: traza de los tres procesos.

(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process 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

4.1 El concepto de proceso: Estados de un proceso.

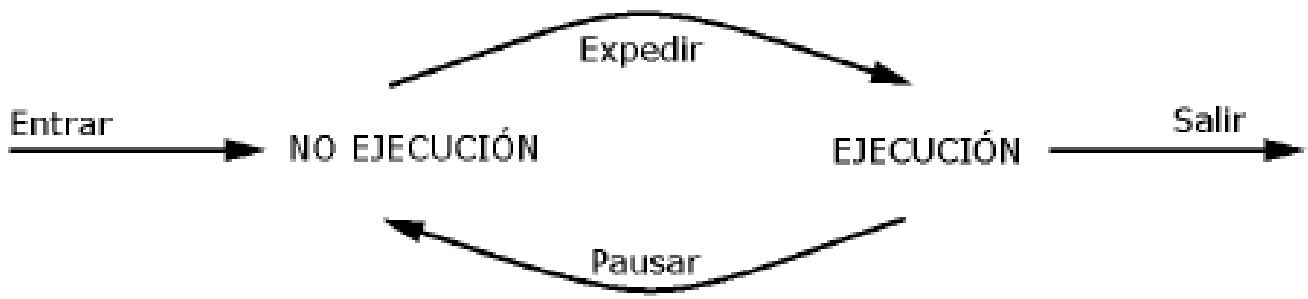
- Ejemplo: traza combinada de los tres procesos para los primeros 52 ciclos de instrucciones del sistema:

1	5000	27	12004
2	5001	28	12005
3	5002	----------	Time out
4	5003	29	100
5	5004	30	101
6	5005	31	102
----------	Time out	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	----------	Time out
16	8003	41	100
----------	I/O request	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
		----------	Time out

100 = Dirección de comienzo del programa distribuidor
Las áreas sombreadas indican ejecución del proceso distribuidor; la primera y tercera columna cuentan los ciclos de instrucción; la segunda y cuarta columna muestran la dirección de la instrucción a ejecutar.

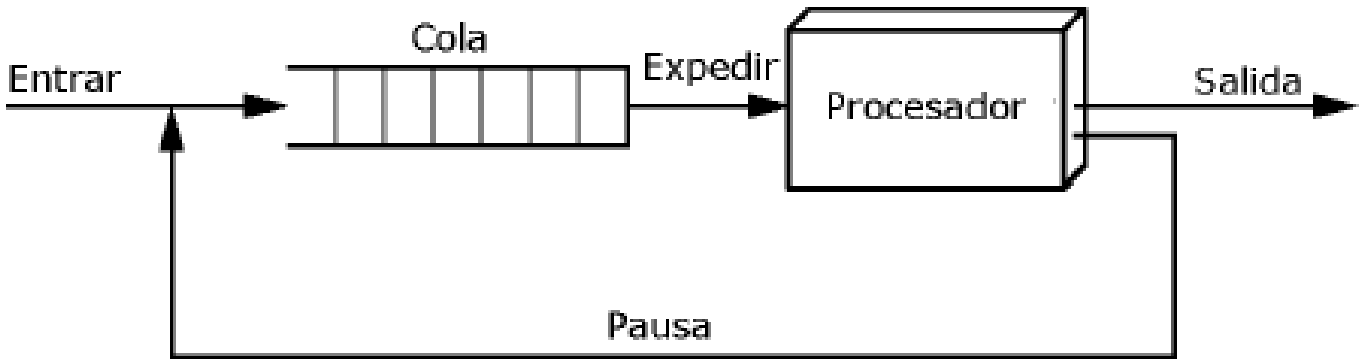
4.1 El concepto de proceso: Estados de un proceso.

- Modelo de proceso con dos estados:
  - El modelo más sencillo es el que considera que **en un cierto instante el proceso está ejecutándose en el procesador o no** => sólo dos estados posibles:
    - **Ejecución.**
    - **No ejecución.**



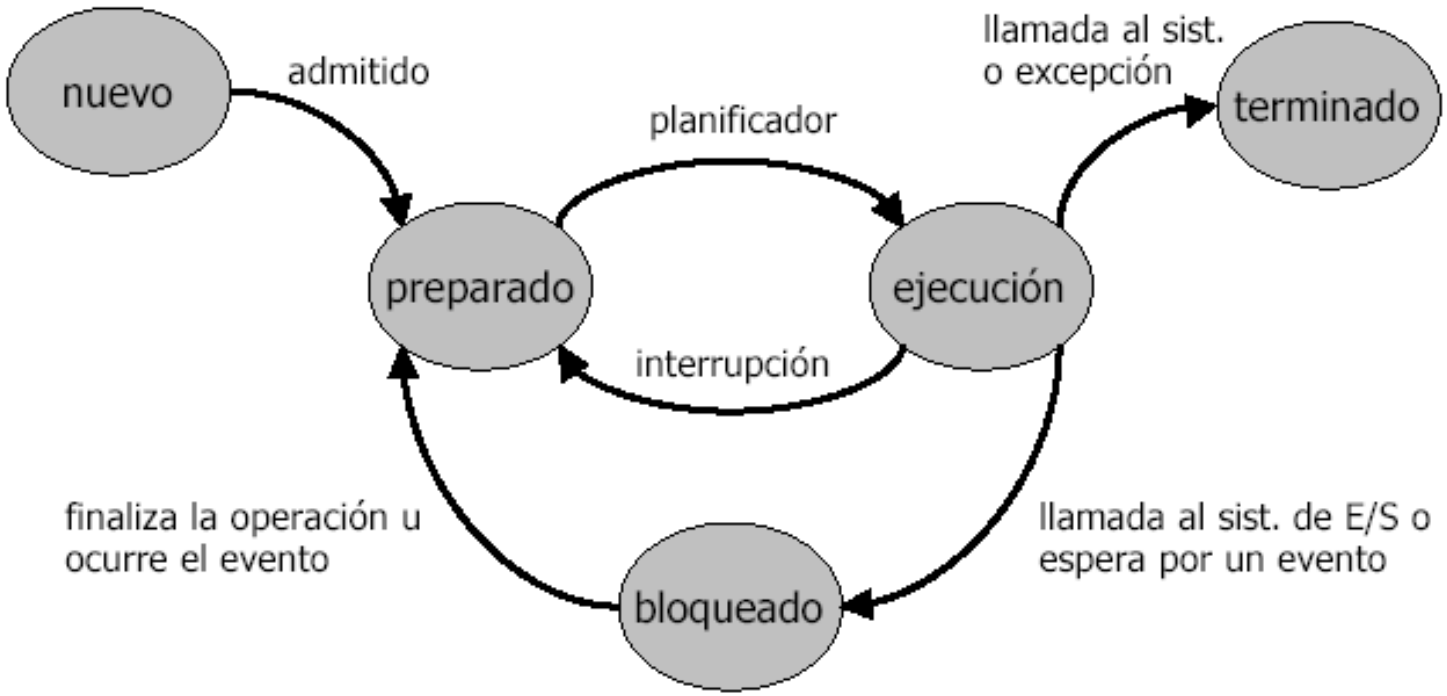
4.1 El concepto de proceso: Estados de un proceso.

- Modelo de proceso con dos estados:
  - Los procesos que no estén ejecutándose se guardan en una cola de procesos, donde esperan su turno de ejecución en el procesador.
  - Cada entrada de la cola es un puntero a un proceso en particular. Cuando un proceso se interrumpe, se le pasa a la cola de procesos en espera. Si un proceso termina o se abandona, se le saca del sistema.



4.1 El concepto de proceso: Estados de un proceso.

- Modelo de proceso de cinco estados:



4.1 El concepto de proceso: Estados de un proceso.

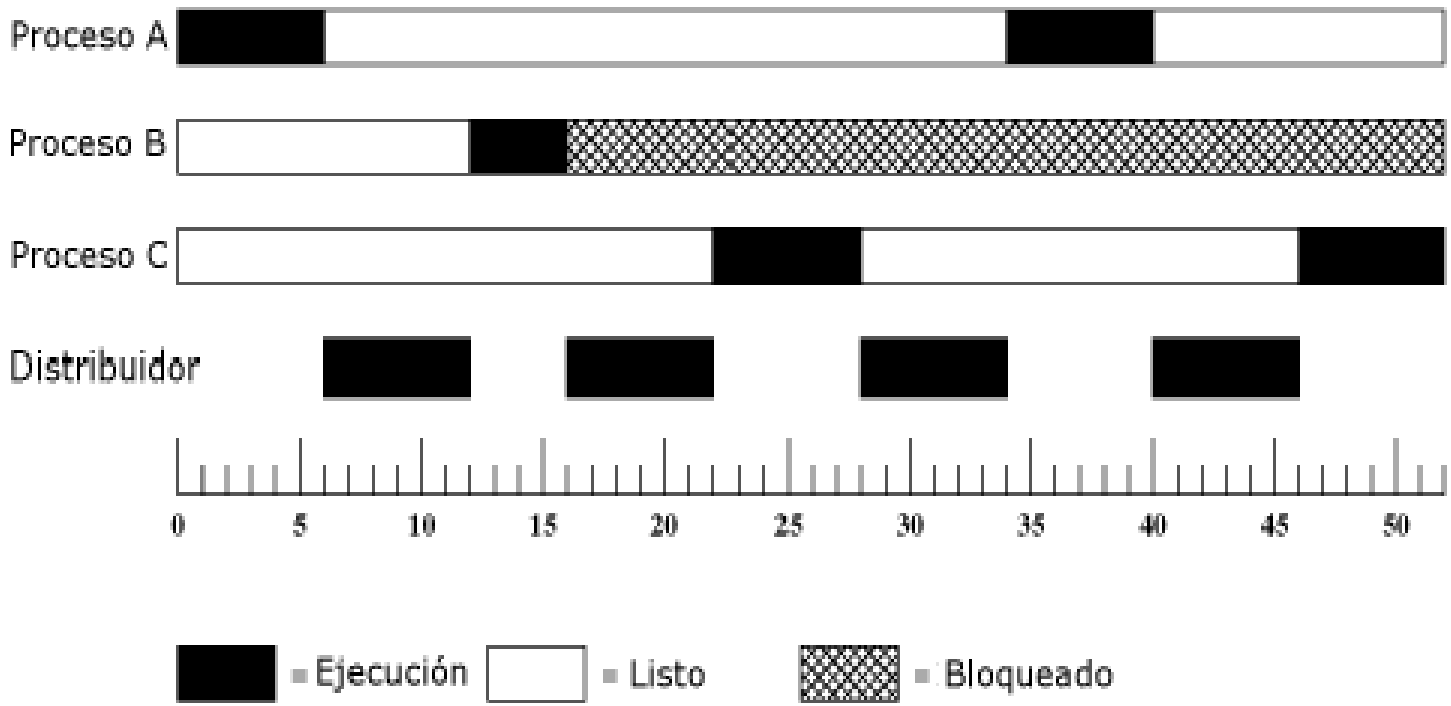
- Modelo de proceso de cinco estados:
  - Los sucesos que pueden dar lugar a una transición de estados en este modelo son los siguientes:
    - **Ninguno a nuevo:** se crea un nuevo proceso para ejecutar un programa
    - **Nuevo a preparado:** el sistema está preparado para aceptar un proceso más porque dispone de recursos para ello.
    - **Preparado a ejecución:** el sistema elige uno de los procesos en estado preparado para llevarlo a ejecución.
    - **Ejecución a terminado:** el proceso que se está ejecutando es finalizado por el SO si indica que terminó, se abandona o se cancela.
    - **Ejecución a preparado:** el proceso ha agotado su tiempo de ejecución, cede voluntariamente su tiempo de ejecución o se interrumpe para atender a otro de mayor prioridad.

4.1 El concepto de proceso: Estados de un proceso.

- Modelo de proceso de cinco estados:
  - Los sucesos que pueden dar lugar a una transición de estados en este modelo son los siguientes (continuación):
    - **Ejecución a bloqueado:** el proceso solicita algo por lo que debe esperar.
    - **Bloqueado a preparado:** se produce el suceso por el que el proceso estaba esperando.
    - **Preparado a terminado** (no aparece en la figura): un padre puede terminar con un proceso hijo en cualquier momento, o bien, si el padre termina todos sus hijos se pueden terminar.
    - **Bloqueado a terminado:** el mismo criterio que el anterior.

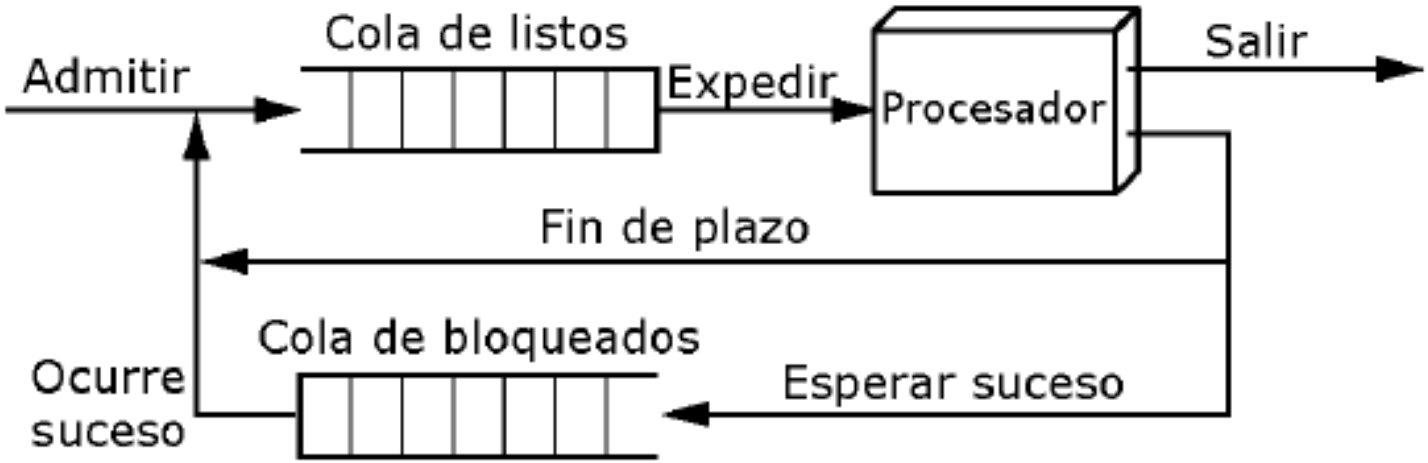
4.1 El concepto de proceso: Estados de un proceso.

- Ejemplo: estados asociados a la traza de los tres procesos.



4.1 El concepto de proceso: Estados de un proceso.

- Sería necesario disponer de dos colas: una de **listos** y otra de **bloqueados**. Los procesos **nuevos** que se van admitiendo pasan a la **cola de listos**, el sistema elige de esta cola alguno para pasarlo a ejecución. Cuando ocurre un **suceso**, todos los procesos que esperan por él pasan de la **cola de bloqueados a la cola de listos**.



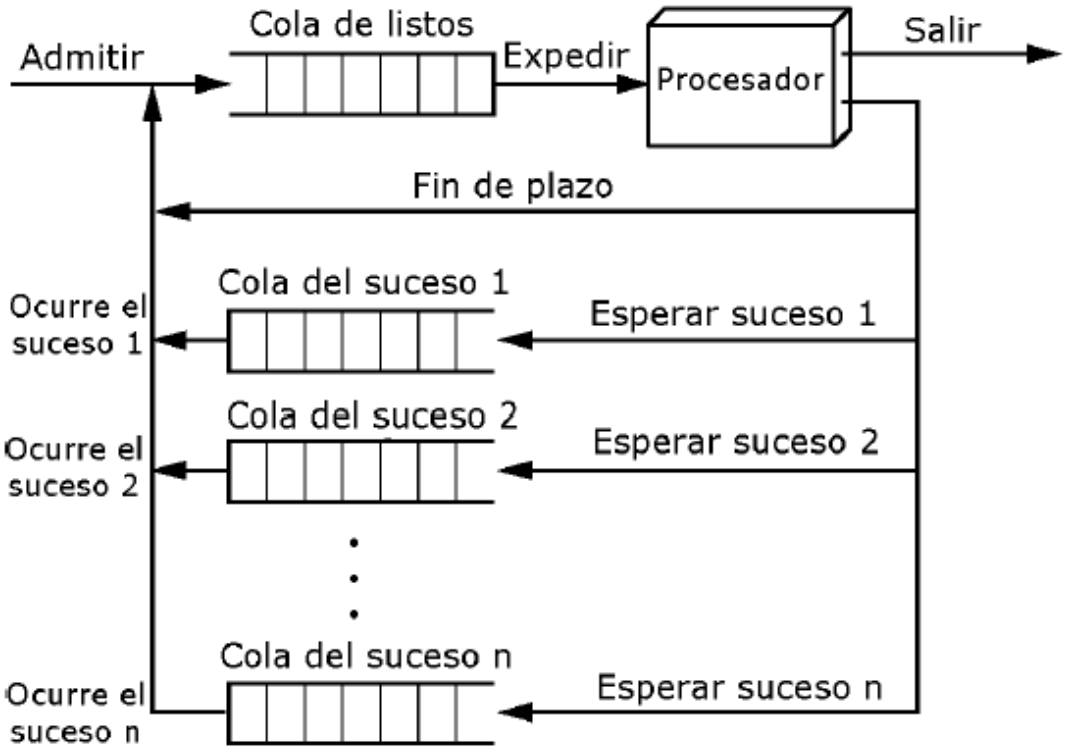
4.1 El concepto de proceso: Estados de un proceso.

- El modelo con una sola cola de **bloqueados** tiene la limitación de que cuando ocurre un evento del sistema debe recorrer la cola para buscar a aquellos procesos que esperan dicho suceso. Esto sería muy costoso en un SO grande con cientos o miles de procesos en dicha cola.

- Sería más eficiente tener varias colas, una asociada a cada suceso. De esta manera, cuando ocurra ese suceso todos los procesos de la cola asociada se pasarían a la cola de **listos**.

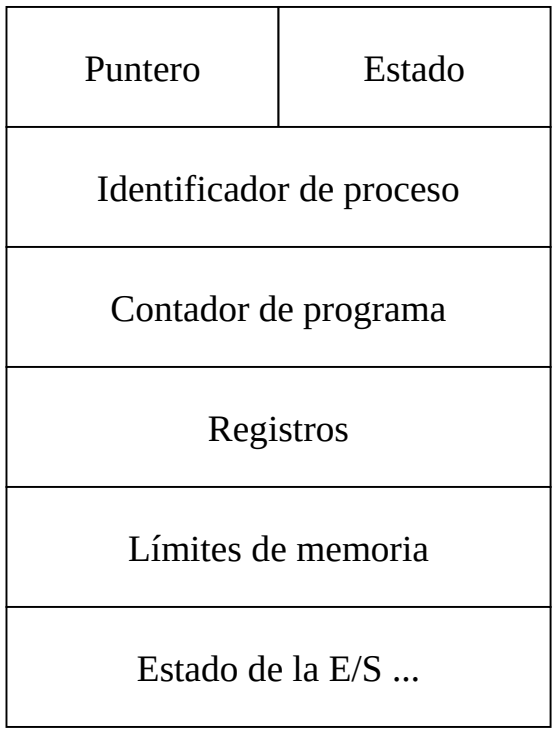
4.1 El concepto de proceso: Estados de un proceso.

- Modelo con varias colas de bloqueados para el proceso de cinco estados:



4.1 El concepto de proceso: Bloque de control de proceso (PCB).

Cada **proceso se representa** en el SO con un **bloque de control de proceso** (también llamado bloque de control de tarea).



4.1 El concepto de proceso: Bloque de control de proceso (PCB).

- Los elementos de información asociados son:
  - **Estado** actual del proceso.
  - **Contador de programa:** indica la dirección de la siguiente instrucción que se ejecutará de ese proceso.
  - **Registros de CPU:** acumuladores, registros índice, punteros de pila y registros generales.
  - **Información de planificación de CPU:** prioridad del proceso, punteros a colas de planificación, etc.
  - **Información de gestión de memoria:** valor de los registros de base y límite, tabla de páginas o tabla de segmentos.
  - **Información de contabilidad:** tiempo de CPU, tiempo consumido, números de procesos, etc.
  - **Información de estado de E/S:** dispositivos de E/S asignados a este proceso, lista de archivos abiertos, etc.

4.2 Planificación de procesos.

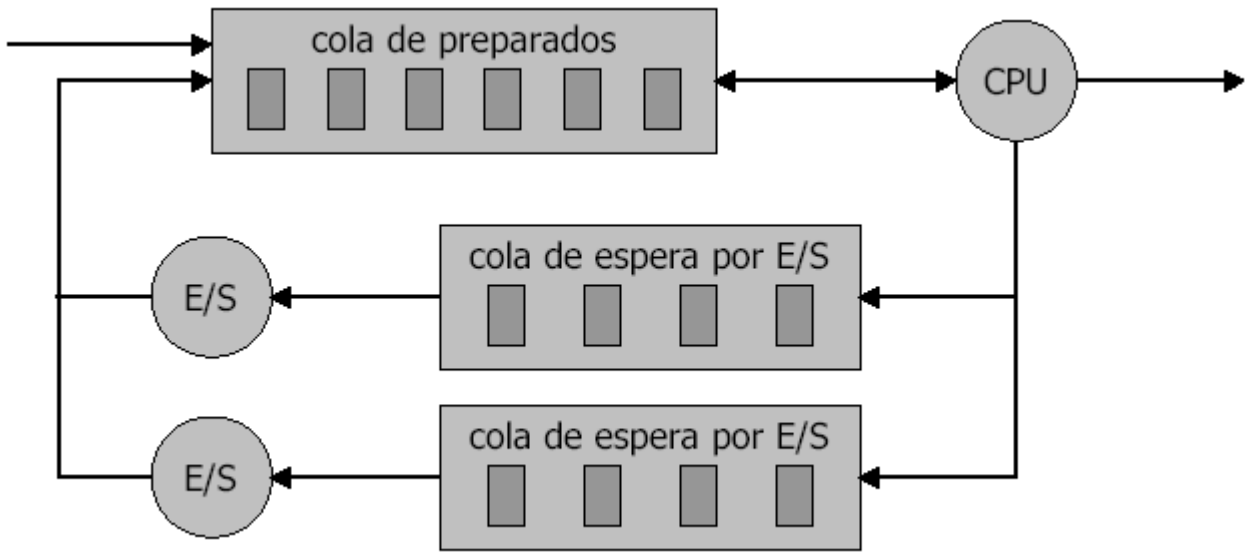
- Las políticas de planificación de procesos surgen como necesidad dada por los objetivos perseguidos por el SO: multiprogramación y tiempo compartido.
  - Objetivo de la **multiprogramación**:
    - Tener **algún proceso en ejecución en todo momento** (maximizar el aprovechamiento de la CPU).
  - Objetivo del **tiempo compartido**:
    - Conmutar la CPU entre procesos con tal frecuencia que **los usuarios puedan interactuar con cada programa durante su ejecución**.
- Nota: En el caso de un sistema con un solo procesador nunca habrá más de un proceso en ejecución, si hay más de un proceso, los otros tendrán que esperar hasta que la CPU esté libre y pueda replanificarse.

4.2 Planificación de procesos: Colas de planificación.

- Colas de planificación:
  - **Cola de procesos:** incluye **todos los procesos** que van ingresando en el sistema.
  - **Cola de procesos listos:** son los **procesos** que están **en memoria principal esperando para ejecutarse**. La cabecera de esta cola contiene punteros al primer y último bloque del PCB de la lista. Cada PCB tiene un puntero que apunta al siguiente proceso de la cola de procesos listos.
  - **Cola de dispositivos:** lista de **procesos que esperan un dispositivo de E/S** en particular.

4.2 Planificación de procesos: Colas de planificación.

- Colas de planificación:
  - El SO organiza los PCB en colas de espera por el procesador o por los dispositivos de E/S (colas de planificación: colas de procesos, colas de dispositivos).



4.2 Planificación de procesos: Colas de planificación.

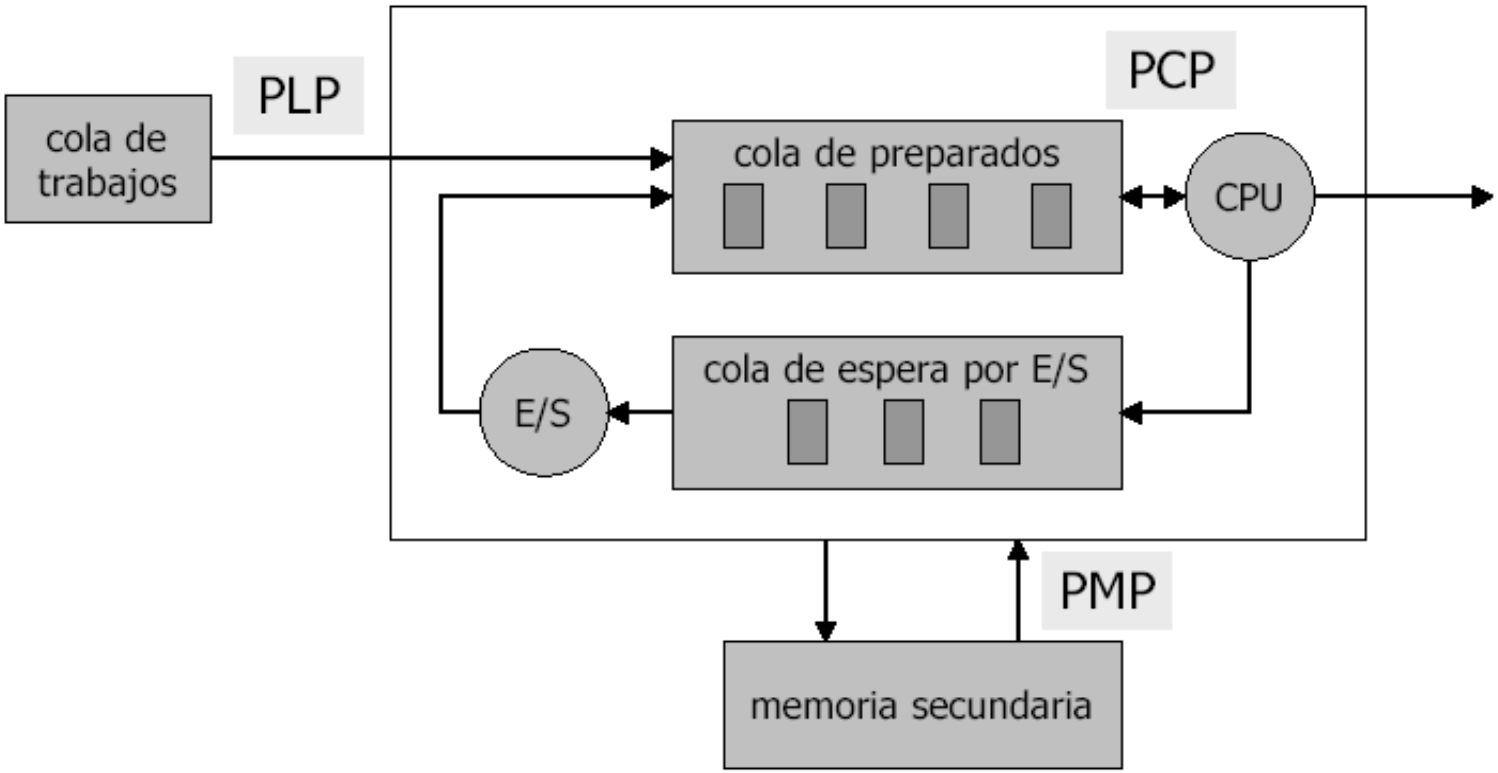
- Un **proceso** se coloca **inicialmente en la cola de listos**, a la espera de que se le ejecute. **Una vez asignada la CPU al proceso** y cuando éste se está ejecutando puede ocurrir que:
  - El proceso pueda **emitir una solicitud de E/S**, y entonces colocarse en una cola de E/S.
  - El proceso puede **crear un nuevo proceso y esperar a que termine**.
  - El proceso puede ser **desalojado por la fuerza de la CPU**, como resultado de una interrupción, y **ser colocado otra vez en la cola de procesos listos**.
- Nota: en los dos primeros casos el proceso pasará del estado de **espera** al estado de **listo**, y se colocará de nuevo en la **cola de procesos listos**.

4.2 Planificación de procesos: Niveles de planificación.

- Niveles de planificación: planificadores.
  - En los sistemas por lotes, existe un **planificador de largo plazo (PLP)** o de alto nivel, que suministra procesos a la cola de preparados.
  - El **planificador de corto plazo** o de bajo nivel es el que **asigna y desasigna la CPU**.
  - El PLP trata de conseguir una mezcla adecuada de trabajos intensivos en CPU y en E/S.
  - **Planificador de medio plazo:** envía al disco procesos de poco interés, para abrir memoria principal para nuevos procesos => Intercambio (swapping).

4.2 Planificación de procesos: Niveles de planificación.

- Niveles de planificación: planificadores.

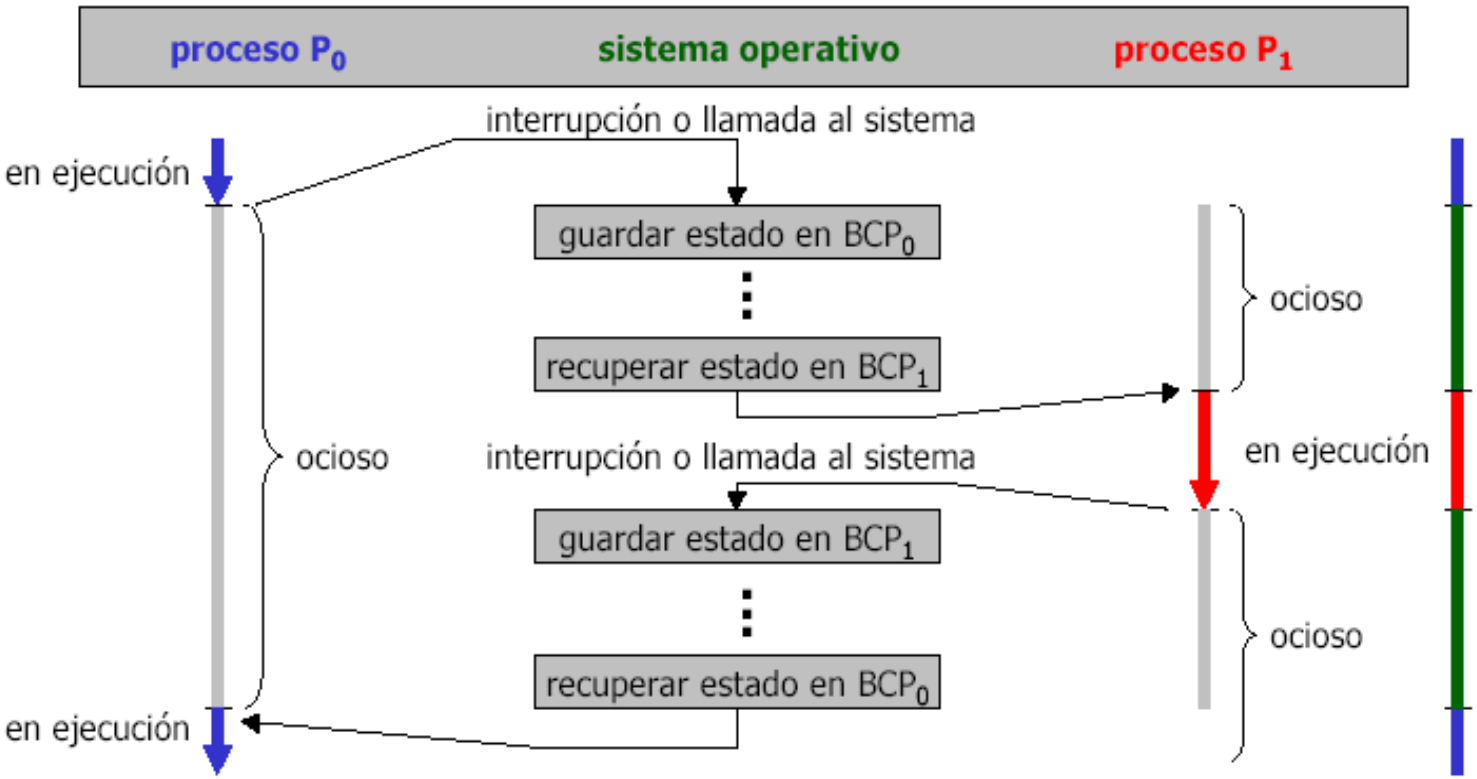


4.2 Planificación de procesos: Cambio de contexto.

- Cambio de contexto (context switch):
  - Consiste en **desalojar a un proceso de la CPU y reanudar otro**.
  - Se **guarda el estado del proceso saliente** en su PCB y se **recuperan los registros del proceso que entra**.
  - El **cambio de contexto** (tiempo de conmutación) es tiempo perdido => debe de ser **lo más rápido posible**. El tiempo de conmutación varía entre 1 y 1000 μs.
  - El **hardware** en ocasiones **facilita el cambio de contexto**, haciendo que un cambio simplemente implique **cambiar el puntero** al conjunto de registros actual.

4.2 Planificación de procesos: Cambio de contexto.

- Cambio de contexto:



4.2 Planificación de procesos: Creación de procesos.

- Los procesos se crean:
  - Mediante una **llamada al sistema** de **“crear proceso”,** durante el curso de su ejecución.
  - El **proceso creador** se denomina proceso **padre**, y el **nuevo proceso, hijo.**
  - Variantes en las relaciones padre/hijo:
    - **Compartición de recursos**: ¿todos, algunos, ninguno?.
    - **Espacio de memoria**.
    - **Sincronización del padre**: ¿espera a que el hijo termine?.
    - **Terminación**.

4.2 Planificación de procesos: Creación de procesos.

- Cuando un proceso crea un proceso nuevo, hay dos posibilidades en términos de ejecución:
  - Padre e hijo **se ejecutan concurrentemente**.
  - **Padre espera por la finalización del hijo**.

- En cuanto al espacio de direcciones del nuevo proceso:
  - El **hijo** es un **duplicado del padre**.
  - Se carga un **programa en el proceso hijo**.

4.2 Planificación de procesos: Creación de procesos.

- En UNIX existen dos funciones básicas para crear procesos:
  - **Función fork():**
    - Cuando se la llama **crea un proceso hijo** que es una **copia casi exacta del proceso padre (duplicado del padre)**. Ambos procesos continúan ejecutándose desde el punto en el que se hizo la llamada a **fork().**
    - En UNIX los procesos se identifican mediante un **"identificador de proceso" (PID)** que es un entero único. Ambos procesos continúan su ejecución con la instrucción que sigue al fork() con una diferencia:
      - **El código que el hijo recibe del fork es cero.**
      - **El que recibe del padre es el propio pid.**

4.2 Planificación de procesos: Creación de procesos.

- **Funciones exec:**
  - Tras crear un nuevo proceso, después de llamar a fork, Linux llama a una función de la familia exec. Éstas funciones **reemplazan el programa ejecutándose en el proceso por otro programa**. Cuando un programa llama a una función exec, su ejecución cesa de inmediato y comienza a ejecutar el nuevo programa desde el principio, suponiendo que no ocurriera ningún error durante la llamada.
  - Generalmente uno de los dos procesos (padre o hijo) utiliza la **llamada al sistema execve después del fork** para **reemplazar su espacio de memoria con un programa nuevo**.
  - Nota: Si el padre no tiene nada que hacer mientras el hijo se ejecuta, puede emitir una llamada **wait (esperar) para sacarse a sí mismo de la cola de procesos listos hasta que el hijo termine.**

4.2 Planificación de procesos: Identificación de procesos.

- Todo proceso en Linux lleva asociado un **identificador** (nº de 16 bits que **se asigna secuencialmente en Linux** por cada nuevo proceso que se crea), que resulta **único** para cada proceso y se conoce como **PID**. Además, salvo el proceso raiz (init), todo proceso lleva asociado un proceso padre, que también tiene un identificador, en este caso **PPID**, lo que generará toda una estructura en árbol.

- **Ejemplo:** El siguiente programa escribe el identificador de un proceso (PID) y el identificador de su proceso padre (PPID).

```
#include <stdio.h>
#include <unistd.h>
    int main (){
      printf("El identificador de este proceso es PID = %d\n",(int)getpid ());
      printf("El identificador del proceso padre es PPID = %d\n",(int)getppid ());
}
```

4.2 Planificación de procesos: Identificación de procesos.

- Un programa puede obtener el identificador del proceso en el que se está ejecutando por medio de la función de llamada al sistema **getpid(),** mientras que el identificador del proceso padre desde el que se ejecutase puede obtener por medio de la función de llamada al sistema **getppid().**

- Al finalizar el fork() tanto el proceso padre como el hijo continúan su ejecución a partir de la siguiente instrucción. Si un padre quiere esperar a que su hijo termine deberá utilizar la llamada al sistema **wait (); wait() detiene la ejecución del proceso (lo pasa al estado bloqueado)** hasta que un hijo de éste termine. wait() regresa de inmediato si el proceso no tiene hijos. Cuando wait() regrese por terminación de un hijo, el valor devuelto es positivo e igual al pid de dicho proceso. De lo contrario devuelve –1 y pone un valor en errno.

4.2 Planificación de procesos: Terminación de procesos.

- **Procesos Zombie:**
  - Proceso que **ha finalizado su ejecución** pero **aún no ha sido eliminado**.
  - Supongamos que un programa crea un proceso hijo y luego llama a la función **wait():**
    - Si el proceso **hijo no ha finalizado** en ese punto, el proceso **padre se bloqueará** en la llamada hasta que el proceso hijo finalice.
    - Si el proceso **hijo finaliza antes de** que el proceso padre llame al **wait ()** el proceso **hijo se convierte en un proceso zombie**.
  - Cuando el padre llame al wait(), captura el estado del proceso hijo **(terminación),** el proceso hijo es borrado y la llamada wait() nos devuelve al programa inmediatamente.

4.2 Planificación de procesos: Terminación de procesos.

- Un proceso finaliza:
  - Cuando tras ejecutar su última instrucción **le pide al SO que lo elimine** utilizando una llamada al sistema **“exit”.**
  - Cuando el padre emite una **llamada al sistema para abortarlo**.
  - Un padre podría terminar la ejecución de uno de sus hijos por **diversas razones**, como:
    - El **hijo se excedió** en la utilización de alguno de los **recursos** que se le asignaron.
    - La **tarea** que se asignó al hijo ya **no es necesaria**.
    - El padre va a salir, y el SO no permite que un hijo continúe si su padre termina => **terminación en cascada**.

4.2 Planificación de procesos: Terminación de procesos.

- Matar un proceso:
  - En ocasiones nos sucederá que **queramos finalizar un proceso que esta corriendo**, las razones para hacer esto pueden ser múltiples, que el proceso esté consumiendo demasiado tiempo del procesador, que esté bloqueado, que no genere información o que genere demasiada, ...
  - Para matar un proceso utilizaremos la orden **kill**, que resulta muy útil en el caso de procesos que no tienen asociada ninguna terminal de control. **Para matar un proceso necesitaremos su PID** de tal manera que escribiremos:

kill PID(s)

4.3 Procesos cooperativos:

- Procesos concurrentes:
  - Los procesos pueden tener distintas relaciones de comunicación entre sí:
    - **Independientes:** no puede afectar, ni ser afectado por los demás procesos que se ejecutan en el sistema, **compiten por el uso de recursos escasos.**
    - **Cooperativos:** puede afectar o ser afectado por los demás procesos que se ejecutan en el sistema, **colaboran entre sí** buscando un objetivo común.
  - Obviamente, **cualquier proceso que comparte datos con otro proceso es cooperativo**.

4.3 Procesos cooperativos:

- Razones para crear un entorno que permita la cooperación entre procesos:
  - **Compartir información:** acceso concurrente a elementos de información comunes.
  - **Aceleración de los cálculos:** para ejecutar una tarea con mayor rapidez, la dividimos en subtareas, cada una de las cuales se ejecuta en paralelo con las otras.
  - **Modularidad:** posibilidad de dividir las funciones del sistema en procesos individuales.
  - **Comodidad:** para evitar que a un usuario individual se le acumule gran número de tareas.

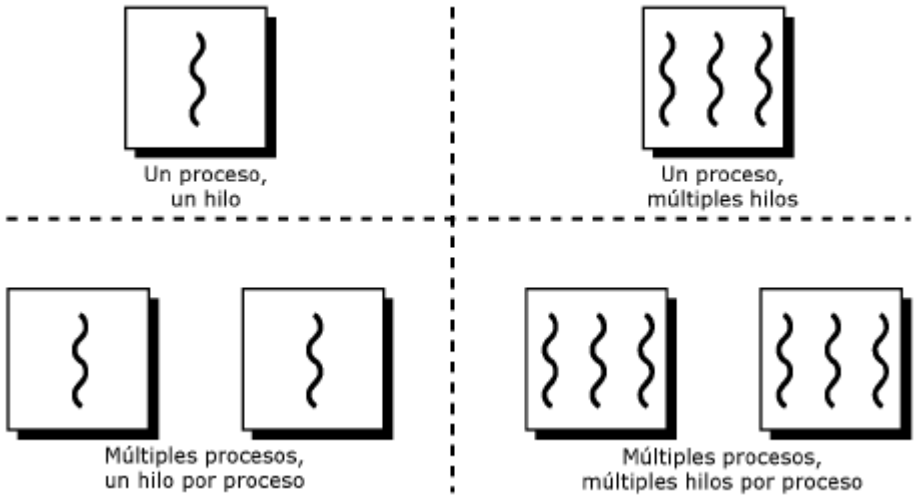
4.3 Procesos cooperativos:

- Ejemplo de proceso cooperativo:

  Problema del productor-consumidor.
  - Un proceso **productor produce información** que es **consumida por un proceso consumidor**, para que se ejecuten concurrentemente, es preciso contar con un **buffer** de elementos que el productor puede llenar y el consumidor puede vaciar.
  - **Productor y consumidor** deben de estar **sincronizados** para que el consumidor no trate de consumir un elemento que aún no se ha producido (el consumidor espera al productor).
  - **Restricciones de espera** para productor y consumidor con buffer limitado:
    - Productor: **espera si buffer lleno**.
    - Consumidor: **espera si buffer vacío**.

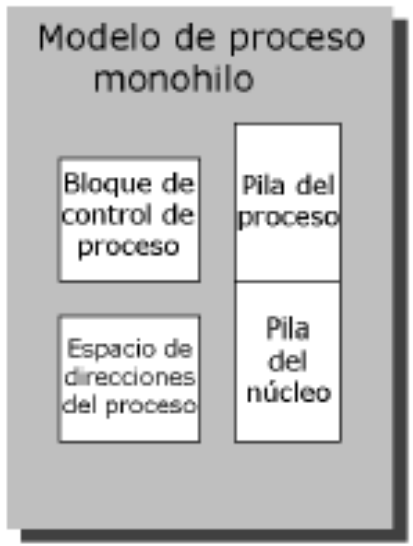
4.4 Hilos (threads).

- Procesos e hilos (threads):
  - Los hilos son un concepto relativamente nuevo de los SO. En este contexto, un **proceso** recibe el nombre de **proceso pesado**, mientras que un **hilo** recibe el nombre de **proceso ligero**. El término hilo se refiere sintáctica y semánticamente a **hilos de ejecución**.
  - El término **multihilo** hace referencia a la capacidad de un SO para mantener **varios hilos de ejecución dentro del mismo proceso**.



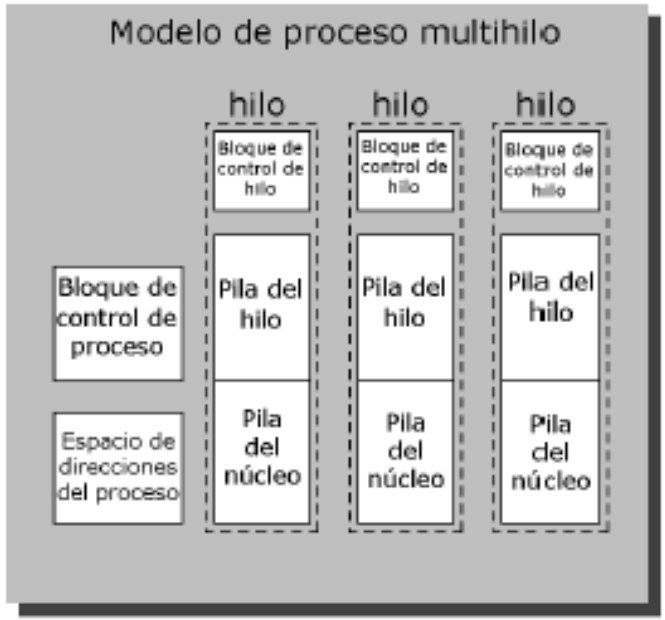
4.4 Hilos (threads).

- En un SO con procesos **monohilo (un solo hilo de ejecución por proceso),** en el que no existe el concepto de hilo, la representación de un proceso **incluye su PCB**, un **espacio de direcciones del proceso**, una **pila de proceso** y una **pila núcleo**.



4.4 Hilos (threads).

- En un SO con procesos **multihilo**, sólo hay un PCB y un espacio de direcciones asociados al proceso, sin embargo, ahora hay **pilas separadas para cada hilo** y **bloques de control para cada hilo**.



4.4 Hilos (threads).

- Estructura de los hilos:
  - Un **hilo** (proceso ligero) es una **unidad básica de utilización de la CPU**, y consiste en un **contador de programa, un juego de registros y un espacio de pila**.
  - **Los hilos** dentro de una misma aplicación **comparten**:
    - La **sección de código**.
    - La **sección de datos**.
    - Los **recursos del SO** (archivos abiertos y señales).
  - Un proceso tradicional o pesado es igual a una tarea con un solo hilo.

4.4 Hilos (threads):Recursos compartidos y no compartidos.

- Los hilos permiten la **ejecución concurrente** de varias secuencias de instrucciones asociadas a diferentes funciones dentro de un mismo proceso, **compartiendo** un mismo **espacio de direcciones** y las mismas **estructuras de datos** del núcleo.

- **Recursos compartidos entre los hilos:**
  - **Código** (instrucciones).
  - **Variables globales**.
  - **Ficheros y dispositivos abiertos**.

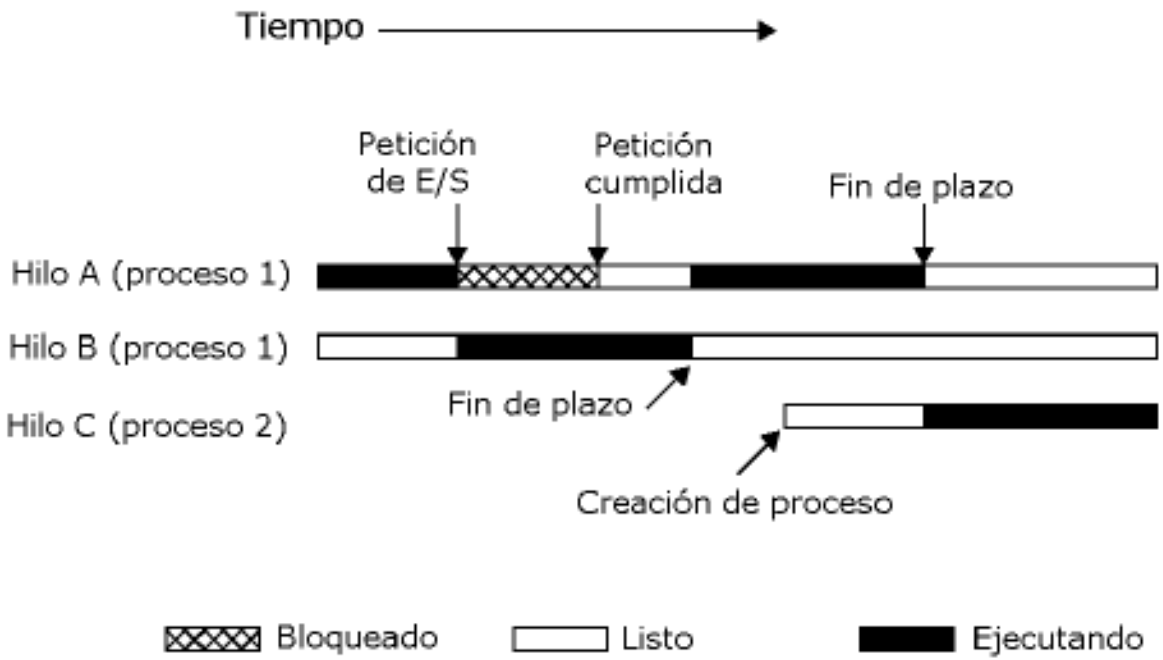
- **Recursos no compartidos entre los hilos:**
  - **Contador del programa** (cada hilo puede ejecutar una sección distinta de código).
  - **Registros de CPU**.
  - **Pila para las variables locales** de los procedimientos a las que se invoca después de crear un hilo.
  - **Estado:** distintos hilos pueden estar en ejecución, listos o bloqueados esperando un evento.

4.4 Hilos (threads): Estados de un hilo.

- Los principales estados de un hilo son: **ejecución, preparado y bloqueado** y hay cuatro operaciones básicas relacionadas con el **cambio de estado** de los hilos:
  - **Creación:** En general, cuando se crea un nuevo proceso se crea también un hilo para ese proceso. Posteriormente, ese hilo puede crear nuevos hilos dándoles un puntero de instrucción y algunos argumentos. Ese hilo se colocará en la cola de preparados.
  - **Bloqueo:** Cuando un hilo debe esperar por un suceso, se le bloquea guardando sus registros. Así el procesador pasará a ejecutar otro hilo preparado.
  - **Desbloqueo:** Cuando se produce el suceso por el que un hilo se bloqueó pasa a la cola de listos.
  - **Terminación:** Cuando un hilo finaliza, se liberan su contexto y sus pilas.
- Nota: Un punto importante es la posibilidad de que **el bloqueo de un hilo lleve al bloqueo de todo el proceso**. Es decir, que el bloqueo de un hilo lleve al bloqueo de todos los hilos que lo componen, aún cuando el proceso está preparado.

4.4 Hilos (threads): Estados de un hilo.

- En un sistema monoprocesador, la multiprogramación permite intercalar la ejecución de múltiples hilos dentro del mismo proceso:



4.4 Hilos (threads): Procesos e hilos.

- Los hilos operan, en muchos sentidos, igual que los procesos:
  - **Pueden estar en uno o varios estados**: listo, bloqueado, en ejecución o terminado.
  - También **comparten la CPU**.
  - **Sólo hay un hilo activo** (en ejecución) en un instante dado.
  - Un hilo dentro de un proceso se **ejecuta secuencialmente**.
  - Cada **hilo** tiene **su propia pila y contador de programa**.
  - **Pueden crear sus propios hilos hijos**.

- Diferencia con los procesos:
  - Los hilos **no son independientes entre sí**. Como todos los hilos pueden acceder a todas las direcciones de la tarea, **un hilo puede leer la pila de cualquier otro hilo o escribir sobre ella**. Aunque pueda parecer lo contrario **la protección no es necesaria** ya que el diseño de una tarea con múltiples hilos tiene que ser un usuario único.

4.4 Hilos (threads): Procesos e hilos.

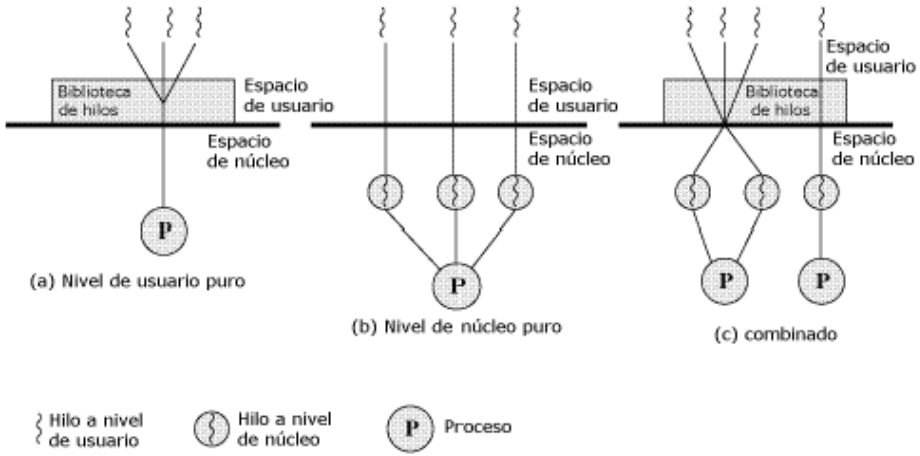
- Ventajas de los hilos sobre los procesos:
  - Se tarda mucho **menos tiempo en crear un nuevo hilo** en un proceso existente que en crear un nuevo proceso.
  - Se tarda mucho **menos tiempo en terminar un hilo** que un proceso.
  - Se tarda mucho **menos tiempo en conmutar entre hilos** de un mismo proceso que entre procesos.
  - Los hilos hacen **más rápida la comunicación entre procesos**, ya que al compartir memoria y recursos, se pueden comunicar entre sí sin invocar el núcleo del SO.

4.4 Hilos (threads).

- Ejemplos de uso de los hilos:
  - **Trabajo interactivo y en segundo plano:** En un programa de hoja de cálculo, un hilo podría estar leyendo la entrada del usuario y otro podría estar ejecutando las órdenes y actualizando la información.
  - **Procesamiento asíncrono:** Se podría implementar, con el fin de protegerse de cortes de energía, un hilo que se encargará de salvaguardar el buffer de un procesador de textos una vez por minuto.
  - **Estructuración modular de los programas:** Los programas que realizan una variedad de actividades se pueden diseñar e implementar mediante hilos.

4.4 Hilos (threads).

- Hilos a nivel de núcleo y a nivel de usuario:
  - Existen dos categorías para la implementación de hilos: **hilos a nivel de usuario (ULT) e hilos a nivel de núcleo (KLT).**
  - **Hilos a nivel de usuario:**
    - La **gestión de hilos la realiza una aplicación** y el núcleo no es consciente de la existencia de hilos. Se programa una aplicación como multihilo mediante una biblioteca de funciones para gestionar ULT (crear, destruir, comunicar, planificar, etc.)



4.4 Hilos (threads).

- Hilos a nivel de núcleo (KLT):
  - Todo el trabajo de **gestión de hilos lo realiza el núcleo**. En el área de la aplicación no hay código para la gestión de hilos, únicamente una Interfaz de Programas de Aplicación (API) para las funciones de gestión de hilos en el núcleo.
  - Este tipo de soluciones se han implementado en Windows 2000 y Linux. **Las aplicaciones se programan como multihilo y todos los hilos de esa aplicación pertenecen al mismo proceso**. El núcleo del SO mantiene la información de contexto del proceso como un todo y la de cada hilo dentro del proceso.
  - La principal desventaja de la solución KLT, respecto de la ULT, es que el **paso de control** de un hilo a otro, dentro del mismo proceso, **requiere cambios de modo.**

4.4 Hilos (threads).

- Ventajas del usar ULT en vez de KLT:
  - 1. El **intercambio** de hilos **no requiere los privilegios del modo núcleo**. Así, el proceso no debe cambiar al modo núcleo y volver al modo usuario para gestionar los hilos. Esto evita una sobrecarga de procesamiento.
  - 2. Se puede **manejar una planificación específica para los hilos**. Para algunas aplicaciones puede ser mejor una planificación basada en prioridades mientras que para otras puede ser mejor un turno rotatorio.
  - 3. **Los ULT se pueden ejecutar en cualquier SO**. Para dar soporte a los ULT no es necesario realizar cambios en el núcleo del SO ya que se utilizan herramientas de gestión de hilos que proporciona una biblioteca para todas las aplicaciones.

4.4 Hilos (threads).

- Desventajas del uso de ULT en vez de KLT:
  - 1. En un SO **la mayoría de las llamadas a sistema son "bloqueantes".** Así, cuando un ULT ejecuta una llamada al sistema no solo se bloquea ese hilo, sino todos los hilos del proceso.
  - 2. Una aplicación multihilo basada en ULT **no puede sacar partido de los sistemas multiprocesadores**. El núcleo del SO asigna un proceso a un solo procesador y sólo se puede ejecutar en él un hilo del proceso a la vez.
- Para evitar estas desventajas:
  - 1. Podemos **modularizar la aplicación por procesos** en vez de hilos. Pero esto elimina la principal ventaja de los hilos: cada cambio de proceso es más costoso que un cambio de hilo.
  - 2. Usar una **técnica de recubrimiento** (convierte una llamada bloqueante en una no bloqueante), para evitar el bloqueo de los hilos.