Unidad 4: PROCESOS

Introducción. Estados. Planificación, Bloque de control de procesos. Servicios. Algoritmos de asignación de la CPU en multiprogramación

PROCESOS

Introducción

La **multiprogramación** (multitarea) es esencialmente la **multiplexación** de los recursos del sistema (procesador, memoria central y dispositivos de E/S) entre una serie de **programas activos**.

Los **procesos** son un **mecanismo esencial** para **definir y gestionar** la **ejecución concurrente** de los programas bajo el control del SO.

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

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

En esencia, un **proceso** es una **instancia de un programa en ejecución**.

- Es la unidad más pequeña de de trabajo (**job step**) individualmente planificable por un SO.

Una definición más completa de proceso puede ser:

Un **proceso** es la **porción de programa en ejecución** más una **estructura de datos** llamada **Vector de Estado** o **Bloque de Control de Proceso** o **Process Control Block (PBC)** que posee los valores actuales del **espacio de nombres** del programa y **otros datos** usados por el SO.

Resumiendo:

Un proceso es una porción de programa en ejecución más su contexto.

PROCESO = porción de programa en ejecución + PCB

El **PCB** es utilizado por el SO para poder conmutar el uso de la CPU entre los distintos programas en ejecución.

Los procesos son creados y destruidos por el sistema operativo, así como también este se debe hacer cargo de la comunicación entre procesos, pero lo hace a petición de otros procesos. El mecanismo por el cual un proceso crea otro proceso se denomina bifurcación (*fork*). Los nuevos procesos pueden ser independientes y no compartir el espacio de memoria con el proceso que los ha creado o ser creados en el mismo espacio de memoria.

De todo lo expuesto surgen términos que deben ser definidos:

- **Contexto:** conjunto de entidades que definen los estados del proceso.
- **Espacio de nombres de un programa**: Conjunto de nombres sobre el cual el programa puede actuar directamente (relación directa). Por Ej., un Archivo debe ser trasladado a Memoria para que el Programa pueda actuar sobre él.
- **Espacio de nombres de un proceso**: Conjunto de objetos que pueden ser usados por el proceso.
- **Espacio de nombres del procesador**: Conjunto de objetos que pueden ser usados por todos los procesos.

- **Espacio de memoria**: Conjunto de direcciones de memoria usadas para implementar el espacio de nombres del procesador.
- **Poder de un proceso (Modo Master- Slave)**: Conjunto de información que define los recursos accesibles por dicho proceso, así como su modo de acceso. Puede evolucionar dinámicamente.

Los espacios de nombres del proceso y del procesador no coinciden, en general el primero es un subconjunto del segundo.

CICLO DE VIDA DE UN PROCESO

El Proceso nace mediante un **System Call** emitido por otro proceso, y durante su ciclo de vida adopta los siguientes estados:

- **Nuevo o New (create):** El proceso está siendo creado.
- **Listo (ready):** Proceso espera que se le asigne el procesador.
- **Ejecutando (running):** Instrucción se está ejecutando.
- **Esperando (waiting):** Proceso está esperando que suceda algún evento.
- **Terminado (completed):** Proceso ha finalizado.

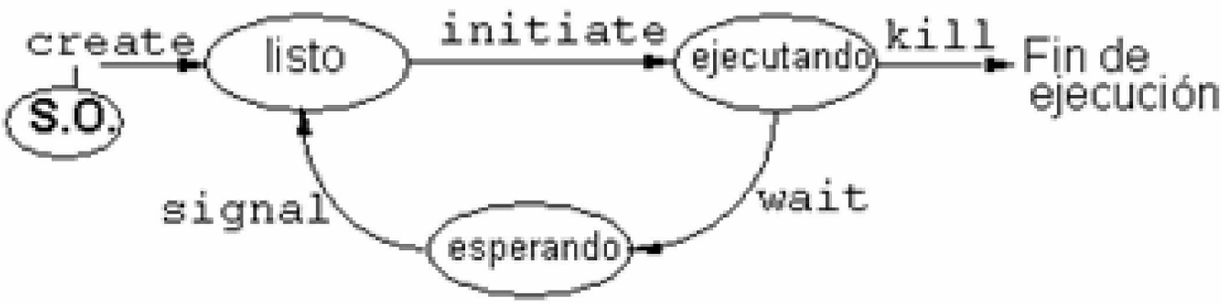
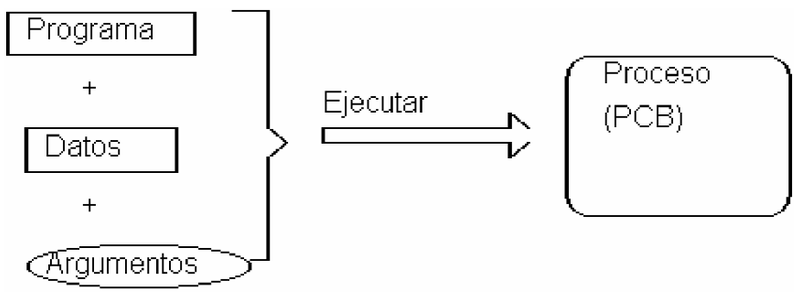


Diagrama de Estados de un Proceso

Crear un Proceso:

Significa:

1. **Darle un nombre** (referencias unívocas - no ambiguas-)
2. **En el PCB inicial** se deben **especificar los objetos y datos que va a usar** el Proceso



Dos tipos de creación:

1. Jerárquica

Cada proceso que se crea es hijo del proceso creador y hereda el entorno de ejecución del su padre. Un proceso durante su ejecución puede crear varios procesos hijos a través de llamadas al sistema para creación de procesos.

Proceso creador es el proceso padre y el proceso nuevo es el proceso hijo, el cual podrá crear otros procesos creando un **árbol** (**tree) de procesos.**

Los subprocesos pueden obtener recursos del SO o restringirse a recursos del proceso padre, el cual divide recursos entre sus hijos o los comparte entre varios hijos. Al restringir un proceso hijo a un subconjunto de recursos del padre se evita que este sature al sistema creando demasiados procesos hijos.

Al crearse un proceso este obtiene recursos físicos y lógicos, datos iniciales del proceso padre.

Hay dos alternativas:

- - El Padre continúa ejecutándose concurrentemente con sus hijos.
- - El Padre espera a que todos sus hijos hayan terminado.

2. No Jerárquica

Cada proceso creado por otro proceso se ejecuta independientemente de su creador en un entorno diferente. (no muy común).

Fin de un proceso

Un proceso terminar por diversas razones. Las causales posibles de abandono son:

- FIN NORMAL (Proceso completo)
- ERROR (Fin anormal)
- NECESITA RECURSOS (E/S, etc.) (Pasa a Bloqueado)
- DESALOJO (Por algún proceso de mayor prioridad) (Pasa a Listos)

Las dos primeras causas se refieren al fin total (muerte) del proceso, en tanto que las dos últimas indican solamente un fin temporal del mismo.

Desalojo significa que, por alguno de los algoritmos de administración de procesador, se considera que el tiempo de uso del procesador por parte de ese proceso ha sido demasiado alto.

Se puede graficar esta situación visualizando además la existencia de colas para los recursos compartidos.

El padre puede provocar la terminación de hijos por diferentes causas:

- el hijo se excedió en el uso de algún recurso asignado;
- ya no requiere tarea del hijo.

Para esto el padre necesita conocer la identidad de sus hijos (cuenta con mecanismo para inspeccionar estado de sus hijos).

Cuando un proceso muere ocurre los siguientes pasos:

- Desaparece el PCB
- Recursos comunes liberados
- Recursos locales destruidos

Cuando un proceso termina también deben terminar sus hijos (normal o anormalmente). A esto se lo denomina **terminación en cascada.**

TIPOS DE RECURSOS DE UN PROCESO

Definimos los siguientes tipos de variables para los procesos:

- **Variables locales**: Son aquellas que sólo se acceden desde el Proceso y sólo éste las conoce.
- **Variables Globales o Públicas**: son aquellas que también se acceden desde otros Procesos (Compartidas).

Se dice que un **recurso es local** de un proceso cuando **sólo es utilizado por dicho proceso**.

Un recurso es **común o repartible con *n* puntos de accesos** cuando lo comparten ***n*** **procesos al mismo tiempo y no es local a ninguno de ellos.**

Los recursos que se comparten en un punto de acceso se dice que son **críticos**.

TIPOS DE PROCESOS

Procesos Independientes o Disjuntos

Son aquellos que sólo tienen variables locales (operan sobre variables disjuntas) o comparten variables globales sin modificarlas, es decir la intersección de su PCB es vacía.

Procesos Paralelos o Concurrentes

Son aquellos que **pueden usar simultáneamente un recurso**. Si el recurso es modificado entonces es **critico** y se usa la **mutua exclusión** para sincronizar su uso.

También se denominan **procesos concurrentes** cuando **sus ejecuciones se superponen en el tiempo (**si la primera operación de uno de ellos es comenzada antes que se complete la última operación del otro).

Razones para la ejecución concurrente:

- Información compartida
- Acelerar los cálculos
- Modularidad
- Comodidad

Para esto se requiere cooperación entre procesos, necesitando un mecanismo para la sincronización y comunicación.

Procesos Interactuantes

Son procesos concurrentes que comparten variables locales.

Dos procesos que están relacionados (son concurrentes) si la intersección de sus vectores de estados (PCB) o es vacía: uno de ellos puede hacer que un recurso quede accesible al otro, o privarlo de este recurso, es decir, uno de los procesos puede hacer cambiar el estado del otro.

Clasificación de los procesos según el código de su programa

Reutilizables

Son los que pueden ser ejecutados con diferentes juegos de datos.

Reentrantes

Sólo tienen código puro y no tienen asociados datos.

ESTADOS DE LOS PROCESOS

Para poder manejar convenientemente una administración de procesador es necesario contar con un cierto juego de datos. Ese juego de datos se encuentra en una tabla en la cual se refleja en qué estado se encuentra el proceso, por ejemplo, si está ejecutando o no.

Los procesos, básicamente, se van a encontrar en tres estados:

§ Listo para la ejecución (Ready)
§ Ejecutando (Running)
§ Bloqueados por alguna razón (Waiting).

Los estados de los procesos son internos del SO y transparente al usuario. Para el usuario el proceso está en ejecución independientemente del estado en que se encuentra internamente en el sistema.

Estados activos

Son los estados que van desde el momento en que se le ha asignado el recurso hasta que lo devuelve

- **Listo o Preparado (Ready)**
  Disponen de todos los recursos para su ejecución y aguardan su turno.
- **Ejecución (Running)**
  El proceso está en uso (control) del procesador.

Estados inactivos

Son los que esperan por largo tiempo los recursos.

- **Bloqueado (Waiting)**
  Son los que aguardan momentáneamente por un recurso.
- **Suspendido (Suspended)**
  Son los que esperan un evento.

COLAS DE PROCESOS

Los **procesos se almacenan en colas**, cada una de ellas representa un **estado (status**) particular de los procesos.

1) Job queue (cola de trabajos)

Procesos que residen en almacenamiento secundario (disco) y que esperan su ejecución.

2) Ready queue (cola de procesos listos)

Procesos que residen en la memoria principal y que están “listos” esperando su ejecución.

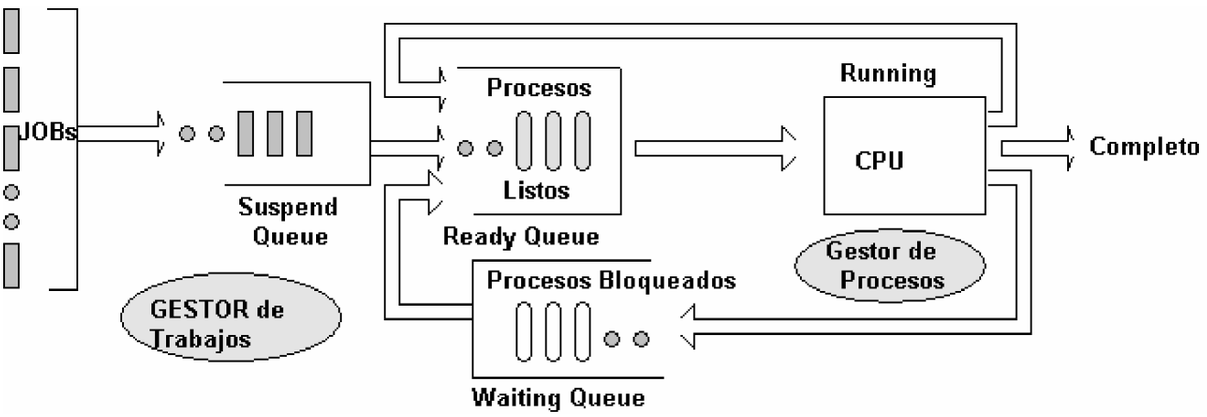
Es una lista ligada. Un encabezado de la cola de procesos listos contendrá apuntadores al primer y último PCB de la lista, cada PCB tiene campo apuntador que indica el siguiente proceso de la cola de procesos listos.

3) Waiting queue (cola de procesos bloqueados)

Procesos que residen en la memoria principal y que están esperando que acontezca algún suceso (ej. E/S).

4) Suspended queue (cola de procesos suspendidos)

Procesos que residen en la memoria secundaria (disco) y que están esperando que acontezca algún suceso (ej. E/S). Son los procesos residentes en memoria virtual (swap).



BLOQUE DE CONTROL DE PROCESOS (PCB)

El Bloque de Control de Procesos (**PCB, Vector de Estado, Tabla de Estado o Descriptor**) contiene el contexto de un proceso y todos los datos necesarios para hacer posible la ejecución de ese proceso y satisfacer sus necesidades.

En forma más detallada cada uno de sus campos contiene:

- **Apuntador a proceso anterior:** dirección del BCP anterior (anterior en tiempo pues fue creado antes). El primer BCP tendrá una identificación que lo señale como tal y deberá ser conocida su ubicación por el Planificador de Procesos.
- **Apuntador a proceso posterior:** dirección del BCP posterior (posterior en tiempo, pues fue creado después). El último BCP tendrá un nil (no se descartan encadenamientos circulares, pero por ahora se los presenta como lineales).
- **Identificación de Proceso:** identificación única para este proceso que lo hace inconfundible con otro.
- **Palabra de control:** espacio reservado o apuntador en donde se guarda la PC cuando el proceso no se encuentra en ejecución.
- **Registros:** ídem anterior, pero para los registros de uso general del proceso.
- **TDP:** apuntador al lugar en donde se encuentra la **Tabla de Distribución de Páginas** correspondiente a este proceso. En el supuesto de tratarse de otro tipo de administración de memoria en esta ubicación se encontraría la información necesaria para conocer en qué lugar de memoria está ubicado el proceso.
- **Dispositivos:** apuntador a todos los dispositivos a los que tiene acceso el proceso al momento. Esta información puede ser estática si es necesario que el proceso declare antes de comenzar su ejecución los dispositivos a usar, o completamente dinámica si existe la capacidad de obtener y liberar dispositivos a medida que se ejecuta el proceso.
- **Archivos:** ídem Dispositivos pero para los archivos del proceso.
- **Tiempos:** Tiempo de CPU utilizado hasta el momento. Tiempo máximo de CPU permitido a este proceso. Tiempo que le resta de CPU a este proceso. Otros tiempos.
- **Estado:** Ejecutando. Listo. Bloqueado. Wait (En espera). Ocioso.
- **Apuntador al BCP del proceso anterior en el mismo estado:** dirección del BCP correspondiente al proceso anterior en ese mismo estado.
- **Apuntador al BCP del proceso posterior en el mismo estado:** ídem anterior pero al proceso posterior.
- **Información para el algoritmo de adjudicación del procesador:** aquí se tendrá la información necesaria de acuerdo al algoritmo en uso.
- **Apuntador al BCP del proceso anterior en función del algoritmo:** dependerá del algoritmo.
- **Apuntador al BCP del proceso posterior en función del algoritmo:** dependerá del algoritmo.
- **Apuntador al BCP del Proceso Padre:** dirección del BCP del proceso que generó el actual proceso.

- **Apuntador a los BCP Hijos:** apuntador a la lista que contiene las direcciones de los BCP hijos (generados por) de este proceso. Si no tiene contendrá null.
- **Accounting:** información que servirá para contabilizar los gastos que produce este proceso (números contables, cantidad de procesos de E/S, etc)

TABLAS Y DIAGRAMA DE TRANSICIÓN DE ESTADOS

En base a estos estados se construye lo que se denomina **Diagrama de Transición de Estados**.

Estar en la **Cola de Listos** significa que el único recurso que le está haciendo falta a ese proceso es el procesador. O sea, una vez seleccionado de esta cola pasa al estado de **Ejecución**.

Se tiene una transición al estado de **Bloqueados** cada vez que el proceso pida algún recurso. **Una vez** que ese requerimiento ha sido **satisfecho**, el proceso **pasará** al estado de **Listo** porque ya no necesita otra cosa más que el recurso procesador.

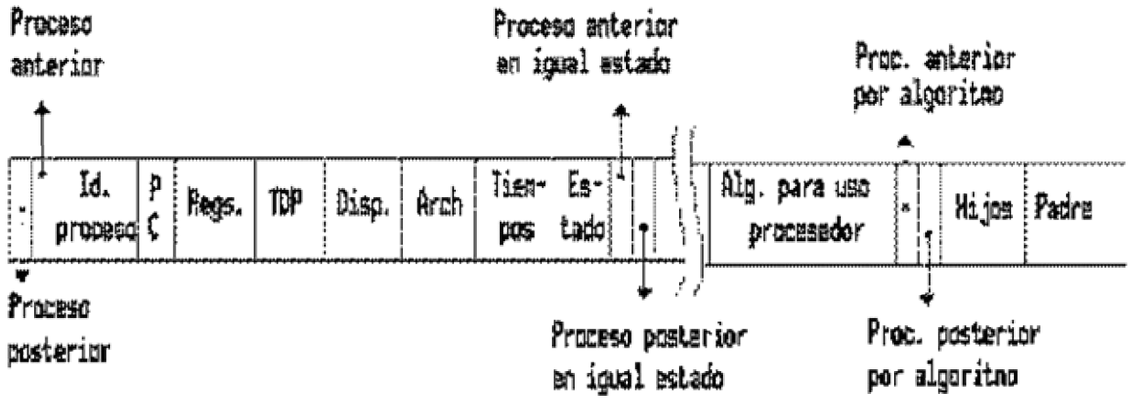
Para manejar esa **Cola de Listos** se requiere de una tabla, denominada **Tabla de Bloques de Control de Procesos (TBCP)**, que **contiene**, básicamente, una identificación de los procesos es decir los **Bloques de Control de Procesos** (**PCB**) .

Como los listos pueden ser muchos, hace falta un apuntador al primero de esa cola de listos, y posiblemente un enganche entre los siguientes en el mismo estado.

Cada entrada de la **TBCP** tiene un apuntador al bloque anterior y uno al posterior y un conjunto de datos inherentes al proceso referido (identificación del proceso, la palabra de control, los registros, dispositivos que esté usando, archivos que esté usando, tiempos que hacen a la vida del proceso, el estado, etc.).

Si bien es cierto que es más fácil pensar a la **TBCP** como una matriz, este tipo de implementación es **muy rígida** y rápidamente podría desembocar en que el espacio reservado para BCP's se termine.

Pensando en una implementación más dinámica se puede implementar a la **TBCP** como un encadenamiento de **BCP**s.



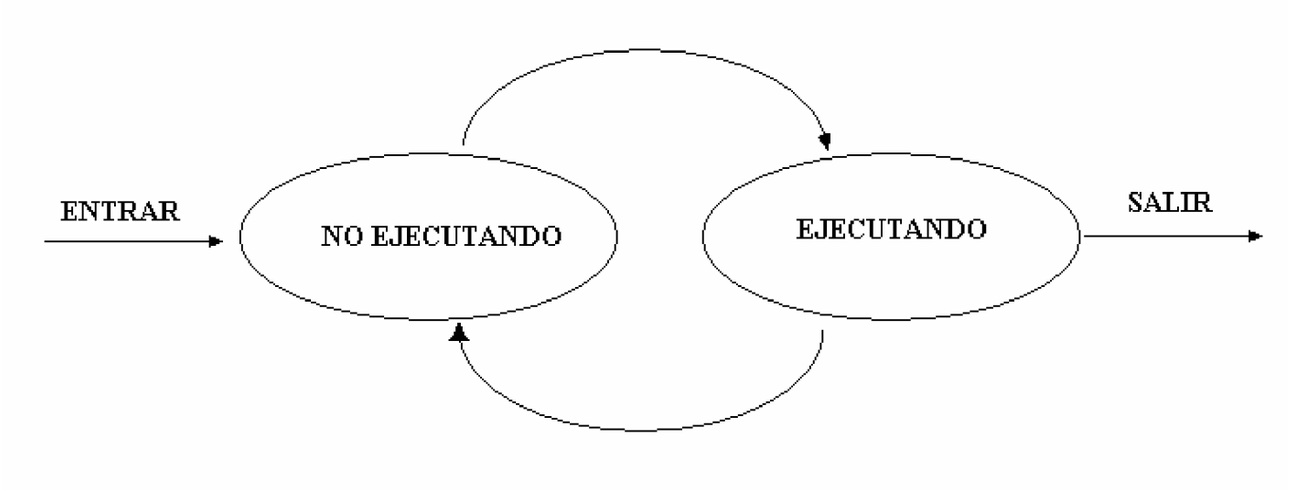
DIAGRAMAS DE TRANSICIÓN

Los **diagramas de transición** o **diagramas de estados** son gráficos que representan los estados de los procesos.

Modelo de dos estados

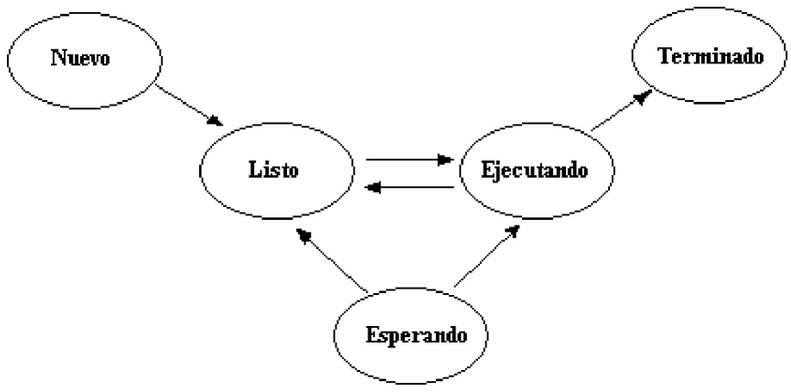
El modelo de estados más simple es el de dos estados. En este modelo, un proceso puede estar ejecutándose o no. Cuando se crea un nuevo proceso, se pone en estado de *No ejecución*. En algún momento el proceso que se está ejecutando pasará al estado *No ejecución* y otro proceso se elegirá de la lista de procesos listos para

ejecutar para ponerlo en estado *Ejecución*. De esta explicación se desprende que es necesario que el sistema operativo pueda seguirle la pista a los procesos, conociendo su estado y el lugar que ocupa en memoria. Además los procesos que no se están ejecutando deben guardarse en algún tipo de cola mientras esperan su turno para ejecutar.



Modelo de cinco estados

El modelo anterior de dos estados funcionaría bien con una cola FIFO y planificación por turno rotatorio para los procesos que no están en ejecución, si los procesos estuvieran siempre listos para ejecutar. En la realidad, los procesos utilizan datos para operar con ellos, y puede suceder que no se encuentren listos, o que se deba esperar algún suceso antes de continuar, como una operación de Entrada/Salida. Es por esto que se necesita un estado donde los procesos permanezcan esperando la realización de la operación de Entrada Salida por parte del Sistema Operativo hasta que puedan proseguir. Se divide entonces al estado *No ejecución* en dos estados: *Listo* y *Espera*. Se agregan además un estado *Nuevo* y otro *Terminado*.



Los cinco estados de este diagrama son los siguientes:

- **Ejecución:** el proceso está actualmente en ejecución.
- **Listo:** el proceso está listo para ser ejecutado, sólo está esperando que el planificador de corto plazo así lo disponga.
- **Espera:** el proceso no puede ejecutar hasta que no se produzca cierto suceso, como la finalización de una operación de Entrada/Salida solicitada por una llamada al sistema operativo.
- **Nuevo:** El proceso recién fue creado y todavía no fue admitido por el sistema operativo. En general los procesos que se encuentran en este estado todavía no fueron cargados en la memoria principal.
- **Terminado:** El proceso fue expulsado del grupo de procesos ejecutables, ya sea porque terminó o por algún fallo, como un error de protección, aritmético, etc.

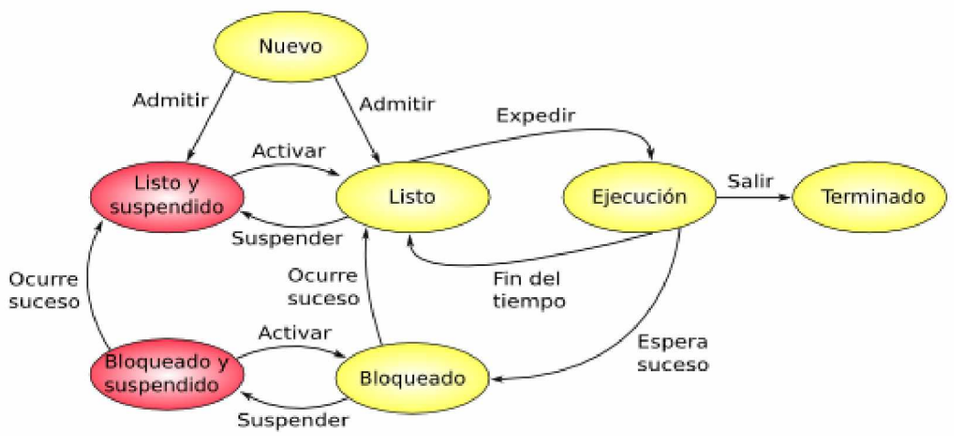
Los nuevos estados *Nuevo* y *Terminado* son útiles para la gestión de procesos. En este modelo los estados *Espera* y *Listo* tienen ambos colas de espera. Cuando un nuevo proceso es admitido por el sistema operativo, se sitúa en la cola de listos. A falta de un esquema de prioridades ésta puede ser una cola FIFO. Cuando se da un suceso se pasan a la cola de listos los procesos que esperaban por ese suceso.

Si existe un esquema con diferentes niveles de prioridad de procesos es conveniente mantener varias colas de procesos listos, una para cada nivel de prioridad, lo que ayuda a determinar cuál es el proceso que más conviene ejecutar a continuación.

Asimismo, existen varias colas en estado de espera, como mínimo una por cada periférico.

Modelo de cinco estados más suspendidos

Una de las razones para implementar el estado *Espera* era poder hacer que los procesos se puedan mantener esperando algún suceso, por ejemplo una Entrada/Salida. Sin embargo, al ser mucho más lentas estas operaciones, puede suceder que en nuestro modelo de cinco estados todos los procesos en memoria estén esperando en el estado *Espera* y que no haya más memoria disponible para nuevos procesos. Podría conseguirse más memoria, aunque es probable que esto sólo permita procesos más grandes y no necesariamente nuevos procesos. Además hay un costo asociado a la memoria y de cualquier forma es probable que se llegaría al mismo estado con el tiempo. Otra solución es el **intercambio**. El intercambio se lleva a cabo moviendo una parte de un proceso o un proceso completo desde la memoria principal al disco, quedando en el estado *Suspendido*. Después del intercambio, se puede aceptar un nuevo proceso o traer a memoria un proceso suspendido anteriormente. El problema que se presenta ahora es que puede ser que si se decide traer a memoria un proceso que está en el estado *Suspendido*, el mismo todavía se encuentre en espera. Sólo convendría traerlo cuando ya está listo para ejecutar, esto implica que ya aconteció el suceso que estaba esperando. Para tener esta diferenciación entre procesos suspendidos, ya sean listos como en espera, se utilizan cuatro estados: *Listo*, *Espera*, *Espera y suspendido* y *Listo y suspendido*.



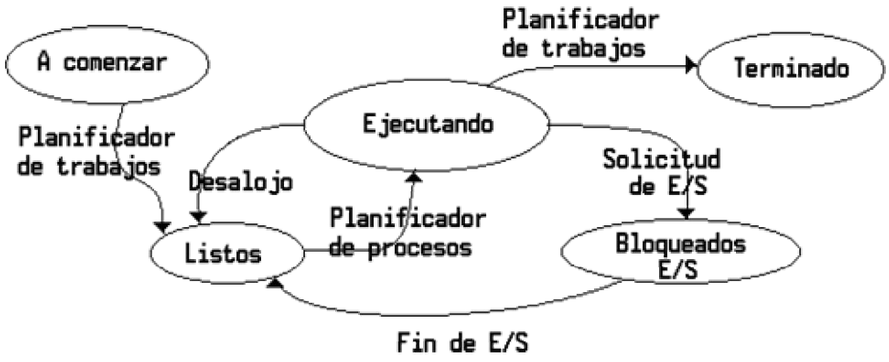
RUTINAS DE ADMINISTRACIÓN DEL PROCESADOR

Generalmente, y dependiendo de la literatura que se consulte, se encuentra que la administración del procesador incluye el pasaje de Retenido a Listos (la entrada del exterior a listos) como una de las partes de la administración del procesador. Lógicamente, mientras no se seleccione un proceso, va a ser imposible que éste compita por el recurso procesador.

PLANIFICADOR DE TRABAJOS

Se denomina **Planificador de Trabajos** al conjunto de rutinas que realizan esta función de ingresar un proceso al sistema desde el exterior y se lo llama muy a menudo **administrador de alto nivel**. Es además ca-

paz de comunicarse con el resto de los administradores para ir pidiendo los recursos que el trabajo necesitará para iniciar su ejecución.



Funciones Principales del Planificador de Trabajos:

- Seleccionar trabajos a ingresar al Sistema.
- Asignar recursos (solicitándolos a los administradores correspondientes)
- Liberar recursos (ídem anterior)

Si posee datos suficientes, el **Planificador de Trabajos**, puede planificar la carga de un sistema. Esta última capacidad carece de sentido si se está trabajando en un sistema que pierde su característica de "batch".

PLANIFICADOR DEL PROCESADOR O PLANIFICADOR DE PROCESOS

El **Planificador de Procesos**, también llamado **planificador de bajo nivel**, es el que tiene que **inspeccionar la Cola de Listos y seleccionar**, de acuerdo a algún criterio, **cual de los procesos que se encuentran allí hará uso del procesador.**

Para administrar eficientemente esta situación se cuenta con un conjunto de módulos:

PLANIFICADOR DEL PROCESADOR

Es el que aplica la política de selección.

CONTROLADOR DE TRÁFICO

Es el que realiza el manejo de las tablas.

DISPATCHER

Es el que pone en estado de ejecución al programa, o sea que carga en el procesador su contexto.

Dado un proceso que pide un servicio sobre un recurso (por ejemplo una E/S) **cambiar del estado de ejecución al estado de bloqueado** es una actividad que le corresponde al **Controlador de tráfico**, que es el maneja las tablas. El **seleccionar el próximo proceso a ejecutar** le corresponde al que aplica la política de selección, que es el **Planificador de Procesos**. Y el que realmente realiza la operación final de **cargar los registros, la palabra de control y los relojes** que sean necesarios (dependiendo de la política de administración del procesador o sea el contexto), es función del **Dispatcher**.

En alguna bibliografía se adjudica la función del Dispatcher al Controlador de Tráfico.

POLÍTICAS DE ASIGNACIÓN

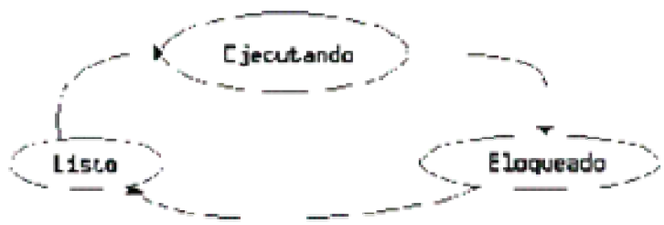
Los criterios para seleccionar un algoritmo de asignación del procesador deben responder lo mejor posible a las siguientes pautas:

- Utilización del procesador: 100 %
- Troughput: Cantidad de trabajos por unidad de tiempo
- Tiempo de Ejecución: Tiempo desde que ingresa un proceso hasta que ter- mina
- Tiempo de Espera: Permanencia en Listos
- Tiempo de Respuesta: Tiempo que tarda en obtenerse un resultado

Luego, el mejor algoritmo será el que maximice las dos primeras pautas y minimice las 3 siguientes.

FIFO o FCFS

Entre las políticas que puede aplicar el planificador de procesos para la selección de procesos que deben pasar del estado de listos al estado de ejecución existe obviamente **la más trivial**, como siempre, que es la **FIFO (first-in first-out)** o **FCFS (first come first served)**. Que significa que el primero que está en la cola es el primero que va a usar el recurso procesador.



De un estado listo, pasa a un estado de ejecución, y de ese estado de ejecución el proceso abandona el recurso procesador solo por decisión propia pasando al estado de bloqueado, y después, una vez satisfecha su necesidad pasa otra vez al estado de listo. O sea que no es desalojado del uso del recurso procesador ya que una vez que lo toma lo sigue usando.

En el momento en que se produce una interrupción por fin de E/S, se atenderá ese fin de E/S el que momentáneamente hará que el proceso abandone el uso del procesador, pero después de finalizada la atención de tal interrupción, el proceso original retomará el uso de la CPU.

Más Corto Primero (JSF) Sin Desalojo.

Una de las políticas que siempre **da el mejor resultado** para aquello que se quiere **ordenar en función del tiempo** es la del **más corto primero (Job Short First).**

Esto implica ordenar los distintos procesos de acuerdo al tiempo que van a necesitar del recurso procesador. O sea, la cola se ordena en función de las ráfagas que se espera que van a emplear de procesador lo distintos procesos.

Ráfaga o **Quantum** es el tiempo continuo de uso del procesador por parte de un proceso que va desde que éste toma el procesador hasta que lo abandona por algún evento.

Este algoritmo es perfecto, ya que si se hace cualquier medición, por ejemplo del turnaround, da siempre mejor que cualquier otro algoritmo de administración.

La **dificultad** radica en que **es necesario conocer el futuro**. **En la práctica es casi imposible** saber con anterioridad cuánto tiempo de procesador va a necesitar un proceso. Lo que se puede hacer es calcular lo que se presume que va a utilizar.

La forma de implementarlo sería, conociendo esa medida, calificar al proceso, es decir si se tiene:

Proceso	Tiempo
P1	5
P2	3
P3	4

La cola de listos ordenada quedaría como P2, P3 y P1.

No se tiene en cuenta el tiempo total de ejecución, sino el tiempo de la ráfaga. O sea cada uno de los segmentos de uso del procesador.

Puede haber varios criterios para conocer la ráfaga. Uno de ellos sería que alguien declare un determinado valor.

Otro criterio sería, medir la cantidad de tiempo en la cual el proceso 1 ejecuta una ráfaga, supóngase que son 5 unidades de tiempo, y calificarlo luego como 5. En realidad, como no se puede conocer el futuro, lo que se hace es llevar algún tipo de estadística sobre lo que pasó en pasos anteriores. Cuando el proceso ingresa por primera vez es indistinto que se lo coloque primero o último, aunque convendrá ponerlo primero para que comience a conocerse su historia.

Una vez que el proceso sale de bloqueado dependerá de cuánto tiempo utilizó el procesador para que tenga otra calificación. En la política Más corto primero no importa el orden en el que entraron a la cola de listos, sino la calificación que se les ha dado.

En este algoritmo no tenemos, todavía, desalojo. Es decir que, como en FIFO o FCFS, el proceso abandona voluntariamente el uso del procesador.

Más Corto Primero Con Desalojo

Una variante del método del más corto primero es que exista desalojo.

Este desalojo se da a través de dos hechos fundamentales (estableciendo cada uno una política de asignación distinta):

- Porque llegó un programa calificado como **más corto** que el que está en ejecución
- Porque llegó un programa cuyo tiempo es menor que el **tiempo remanente**.

En el primero de los casos si se tiene en ejecución un programa calificado con 3, y en la cola de listos hay otros dos calificados con 4 y con 5, y llega en este momento, a la cola de listos, uno calificado como 2, se produce una interrupción del que está calificado como 3, se lo coloca en la cola de listos en el orden que le corresponda, y pasa a ejecutarse el programa calificado como 2.

En el caso de tiempo remanente, lo que se va a comparar es cuánto hace que está ejecutando el que está calificado como 3. Es decir, dada la misma situación del ejemplo anterior, si llega uno calificado como 2, pero resulta que al que está ejecutando sólo le falta una unidad de tiempo de ejecución, no se lo interrumpe, y el nuevo programa pasa a la cola de listos en el orden que le corresponda.

Si al programa que está ejecutándose le faltaran más de 2 unidades de tiempo de ejecución, se interrumpe, se lo pasa a la cola de listos, y pasa a ejecutarse el nuevo programa.

Administración por Prioridades

Otro tipo de administración del procesador consiste en **dar prioridades a los procesos**. De alguna manera, el algoritmo del más corto primero con desalojo constituye uno de estos.

La idea es que algunos procesos tengan mayor prioridad que otros para el uso del procesador por razones de criterio.

Estos criterios pueden ser de índole administrativa o por el manejo de otros recursos del sistema. Ejemplos concretos serían:

§ Por prioridad administrativa por ejemplo el "Proceso de Sueldos"

§ Por recursos: Administración de Memoria Particionada; por ejemplo los procesos se ejecutan en particiones de direcciones más bajas tienen mayor (o menor) prioridad que los procesos que se ejecutan en las particiones de direcciones más altas

En estos casos es necesario implementar un mecanismo que evite un bloqueo indefinido para aquellos procesos que tienen las más bajas prioridades.

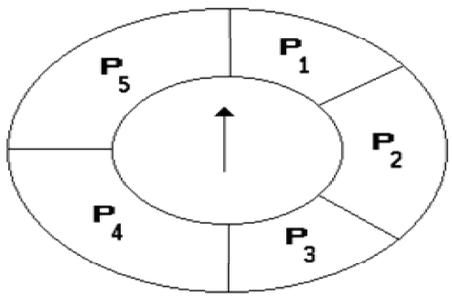
Una solución puede ser que a medida que transcurre el tiempo la prioridad de los procesos relegados se incrementa paulatinamente.

Round-Robin

Otra forma de administrar el procesador es el **Round-Robin**. El origen del término Round-Robin proviene de que antiguamente en los barcos de la marina cuando los oficiales deseaban presentar una queja al capitán redactaban un documento el pie del cual estampaban sus firmas en forma circular, de forma tal que era imposible identificar cual de ellos había sido el promotor de tal queja.

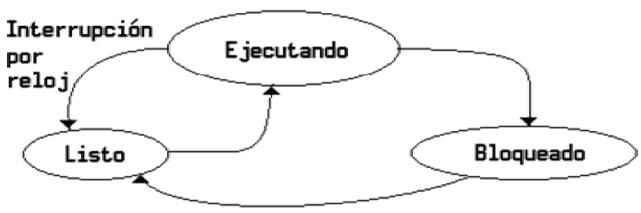
Esta administración consiste en dar a cada proceso la misma cantidad o cuota de uso del procesador.

La asignación se comporta como una manecilla que recorre el segmento circular (que representa la ráfaga asignada) y que al pasar al próximo proceso genera una interrupción por reloj. Si el próximo proceso no se encuentra en estado de listo, se pasa al siguiente y así sucesivamente.



Calesita circular del método Round-Robin.

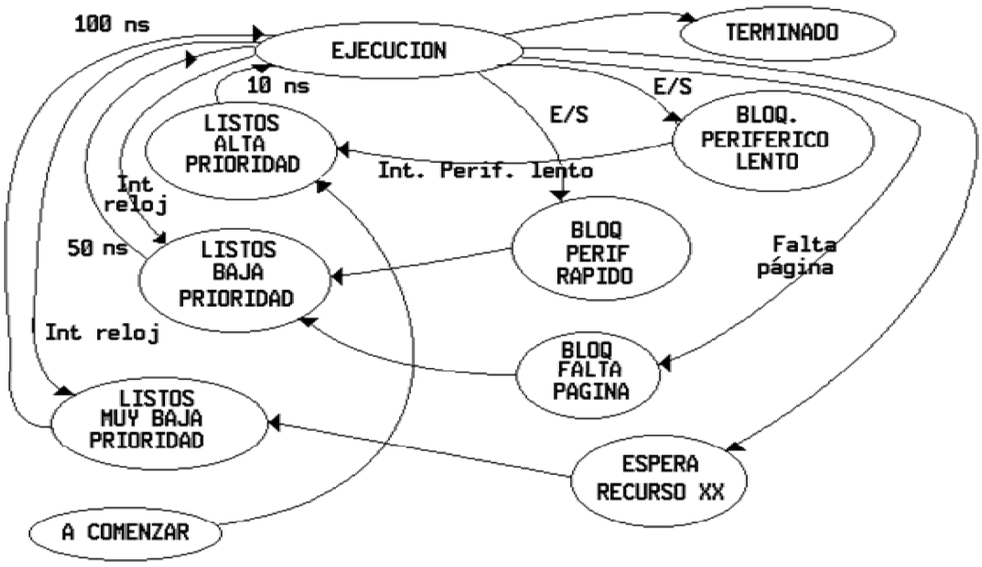
Es decir, a todos los procesos se les da un quantum, que es una medida del tiempo que podrán usar el procesador antes de ser interrumpidos por reloj. Obviamente puede ocurrir que cuando le toque al siguiente proceso, éste se encuentre bloqueado por operaciones de E/S, en tal caso se pasa al siguiente proceso, y el anterior tendrá que esperar que le toque el procesador nuevamente la próxima vuelta.



Administración del procesador Round- Robin.

Multicolas

Otra forma de prioridad es asignar distintas colas y distintos quantums dependiendo del tipo de bloqueo al que llega el proceso (lectora/impresora antes que cinta) o por las características del proceso.



Administración Multicolas.

A este esquema se le podría agregar que el programa en ejecución cuando es interrumpido vaya a alguna de las colas perdiendo el uso del procesador. De esta forma se está beneficiando a los de alta E/S, en caso contrario se guardaría el valor del reloj de intervalos.

Se podría considerar que cada una de las colas es administrada por medio de una política diferente.

Las variantes del Round-Robin y Multicolas pueden ser:

- Todos los procesos tienen el mismo Quantum
- Si un proceso usó poco procesador se lo puede colocar por la mitad de la cola
- Si un proceso acaba de ingresar se le otorga más tiempo
- Si un proceso ejecutó muchas veces hasta el límite más alto de quantum sólo se le permitirá ejecutar cuando no haya otro proceso
- Dar preferencia a los procesos que realizan mucha E/S
- Dar preferencia a los procesos interactivos
- Dar preferencia a los procesos más cortos
- Pasar procesos de una cola a otra según el comportamiento que demuestren
- Se puede seleccionar de las colas de acuerdo a su historial

De todas maneras en los sistemas **Multicolas** no debe olvidarse que existe la posibilidad de procesos que queden relegados, luego es necesario implementar alguna variante para solucionar esto. Más adelante se discute una de ellas.

Este tipo de administración logra lo que se denomina un "**Balance General del Sistema**" debido a que compensa la utilización de los recursos respecto de la ejecución de los procesos otorgando mayor prioridad de ejecución (cola de Listos de mayor prioridad) a los procesos que utilizan los periféricos más lentos y viceversa.

Nótese en el grafo que los procesos que utilizan un periférico lento luego de finalizar su E/S retornan a la cola de Listos de mayor prioridad. Esto permite que dicho proceso luego de estar durante bastante tiempo fuera del circuito de listo-ejecutando (su E/S demora más ya que el periférico es más lento) pueda tener pronto una chance de retomar el uso del recurso Procesador. Si se lo hubiera colocado en una cola de lis- tos de menor prioridad debería esperar que las colas anteriores se vaciaran antes de poder acceder al procesador.

Las colas de menor prioridad que contienen a los procesos que realizan E/S sobre periféricos veloces se encuentran cargadas continuamente ya que el proceso demora muy poco tiempo en realizar su E/S (ya que el periférico es rápido) y vuelve rápidamente a la cola de Listos, en tanto que las colas de mayor prioridad usualmente se vacían con bastante frecuencia ya que sus procesos demoran un tiempo considerable en volver del estado de Bloqueado.

Finalmente este mecanismo logra asimismo que los periféricos lentos se utilicen más frecuentemente logrando su máximo aprovechamiento.

HILO DE EJECUCIÓN (Threads)

Un **hilo de ejecución**, en sistemas operativos, es una característica que permite a una aplicación realizar varias tareas concurrentemente. Los distintos hilos de ejecución comparten una serie de recursos tales como el espacio de memoria, los archivos abiertos, situación de autenticación, etc. Esta técnica permite simplificar el diseño de una aplicación que debe llevar a cabo distintas funciones simultáneamente.

Los hilos de ejecución que comparten los mismos recursos, sumados a estos recursos, son en conjunto conocidos como un proceso. El hecho de que los hilos de ejecución de un mismo proceso compartan los recursos hace que cualquiera de estos hilos pueda modificar éstos. Cuando un hilo modifica un dato en la memoria, los otros hilos acceden e ese dato modificado inmediatamente.

Lo que es propio de cada hilo es el contador de programa, la pila de ejecución y el estado de la CPU (incluyendo el valor de los registros).

El proceso sigue en ejecución mientras al menos uno de sus hilos de ejecución siga activo. Cuando el proceso es terminado, todos sus hilos de ejecución también lo son. Asimismo en el momento en el que todos los hilos de ejecución finalizan, el proceso no existe más y todos sus recursos son liberados.

Algunos lenguajes de programación tienen características de diseño expresamente creadas para permitir a los programadores lidiar con hilos de ejecución (como Java). Otros (la mayoría) desconocen la existencia de hilos de ejecución y éstos deben ser creados mediante llamadas de biblioteca especiales que dependen del sistema operativo en el que estos lenguajes están siendo utilizados (como es el caso del C y del C++).

Un ejemplo de la utilización de hilos es tener un hilo atento a la interfaz gráfica (iconos, botones, ventanas), mientras otro hilo hace una larga operación internamente. De esta manera el programa responde de manera más ágil a la interacción con el usuario. También pueden ser utilizados por una aplicación servidora para dar servicio a múltiples clientes.

Diferencias entre hilos y procesos

Los hilos se distinguen de los tradicionales procesos en que los procesos son generalmente independientes, llevan bastante información de estados, e interactúan sólo a través de mecanismos de comunicación dados por el sistema. Por otra parte, muchos hilos generalmente comparten otros recursos directamente. En muchos de los sistemas operativos que proveen facilidades para los hilos, es más rápido cambiar de un hilo a otro dentro del mismo proceso, que cambiar de un proceso a otro. Este fenómeno se debe a que los hilos comparten datos y espacios de direcciones, mientras que los procesos al ser independientes no lo hacen. Al cambiar de un proceso a otro el sistema operativo (mediante el dispatcher) genera lo que se conoce como overhead, que es tiempo desperdiciado por el procesador para realizar un cambio de modo (mode switch), en este caso pasar del estado de Running al estado de Waiting o Bloqueado y colocar el nuevo proceso en Running. En los hilos como pertenecen a un mismo proceso al realizar un cambio de hilo este overhead es casi despreciable.

Sistemas operativos como Windows NT, OS/2 y Linux (2.5 o superiores) han dicho tener hilos 'baratos', y procesos 'costosos' mientras que en otros sistemas no hay una gran diferencia.

Funcionalidad de los hilos

Al igual que los procesos, los hilos poseen un estado de ejecución y pueden sincronizarse entre ellos para evitar problemas de compartimiento de recursos. Generalmente, cada hilo tiene una tarea especifica y determinada, como forma de aumentar la eficiencia del uso del procesador.

Estados de un hilo

Los principales estados de los hilos son: Ejecución, Listo y Bloqueado. No tiene sentido asociar estados de suspensión de hilos ya que es un concepto de proceso. En todo caso, si un proceso está expulsado de la memoria principal (RAM), todos sus hilos deberán estarlo ya que todos comparten el espacio de direcciones del proceso.

Cambio de estados

***Creación**:* Cuando se crea un proceso se crea un hilo para ese proceso. Luego, este hilo puede crear otros hilos dentro del mismo proceso. El hilo tendrá su propio contexto y su propio espacio de pila, y pasara a la cola de listas.

***Bloqueo**:* Cuando un hilo necesita esperar por un suceso, se bloquea (salvando sus registros). Ahora el procesador podrá pasar a ejecutar otro hilo que esté en la cola de Listos mientras el anterior permanece bloqueado.

***Desbloqueo**:* Cuando el suceso por el que el hilo se bloqueó se produce, el mismo pasa a la cola de Listos.

***Terminación**:* Cuando un hilo finaliza se liberan tanto su contexto como sus pilas.

Ventajas de los hilos contra procesos

Si bien los hilos son generados a partir de la creación de un proceso, podemos decir que un proceso es un hilo de ejecución, conocido como Monohilo. Pero las ventajas de los hilos se dan cuando hablamos de Multihilos, que es cuando un proceso tiene múltiples hilos de ejecución los cuales realizan actividades distintas, que pueden o no ser cooperativas entre sí. Los beneficios de los hilos se derivan de las implicaciones de rendimiento.

Se tarda mucho menos tiempo en crear un hilo nuevo en un proceso existente que en crear un proceso. Algunas investigaciones llevan al resultado que esto es así en un factor de 10.

Se tarda mucho menos en terminar un hilo que un proceso, ya que cuando se elimina un proceso se debe eliminar el BCP del mismo, mientras que un hilo se elimina su contexto y pila.

Se tarda mucho menos tiempo en cambiar entre dos hilos de un mismo proceso

Los hilos aumentan la eficiencia de la comunicación entre programas en ejecución. 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 misma. 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.

Sincronización de hilos

Todos los hilos comparten el mismo espacio de direcciones y otros recursos como pueden ser archivos abiertos. Cualquier modificación de un recurso desde un hilo afecta al entorno del resto de los hilos del mismo proceso. Por lo tanto, es necesario sincronizar la actividad de los distintos hilos para que no interfieran unos con otros o corrompan estructuras de datos.

Una ventaja de la programación multihilo es que los programas operan con mayor velocidad en sistemas de computadores con múltiples CPUs (sistemas multiprocesador o a través de grupo de máquinas) ya que los hilos del programa se prestan verdaderamente para la ejecución concurrente. En tal caso el programador necesita ser cuidadoso para evitar condiciones de carrera (problema que sucede cuando diferentes hilos o procesos alteran datos que otros también están usando), y otros comportamientos no intuitivos. Los hilos generalmente requieren reunirse para procesar los datos en el orden correcto. Es posible que los hilos requieran de operaciones atómicas para impedir que los datos comunes sean cambiados o leídos mientras estén siendo modificados, para lo que usualmente se utilizan los semáforos. El descuido de esto puede generar interbloqueo.

Formas de multihilos

Los sistemas operativos generalmente implementan hilos de dos maneras:

***Multihilo apropiativo**:* permite al sistema operativo determinar cuándo debe haber un cambio de contexto. La desventaja de esto es que el sistema puede hacer un cambio de contexto en un momento inadecuado, causando un fenómeno conocido como inversión de prioridades y otros problemas.

***Multihilo cooperativo**:* depende del mismo hilo abandonar el control cuando llega a un punto de detención, lo cual puede traer problemas cuando el hilo espera la disponibilidad de un recurso.

El soporte de *hardware* para multihilo desde hace poco se encuentra disponible. Esta característica fue introducida por Intel en el Pentium 4, bajo el nombre de HyperThreading.

Usos más comunes

Trabajo interactivo y en segundo plano

Por ejemplo, en un programa de hoja de cálculo un hilo puede estar visualizando los menús y leer la entrada del usuario mientras que otro hilo ejecuta las órdenes y actualiza la hoja de calculo. Esta medida suele aumentar la velocidad que se percibe en la aplicación, permitiendo que el programa pida la orden siguiente antes de terminar la anterior.

Procesamiento asíncrono

Los elementos asíncronos de un programa se pueden implementar como hilos. Un ejemplo es como los softwares de procesamiento de texto guardan archivos temporales cuando se está trabajando en dicho programa. Se crea un hilo que tiene como función guardar una copia de respaldo mientras se continúa con la operación de escritura por el usuario sin interferir en la misma.

Aceleración de la ejecución

Se pueden ejecutar, por ejemplo, un lote mientras otro hilo lee el lote siguiente de un dispositivo.

Estructuración modular de los programas

Puede ser un mecanismo eficiente para un programa que ejecuta una gran variedad de actividades, teniendo las mismas bien separadas mediante a hilos que realizan cada una de ellas.

Implementaciones

Hay dos grandes categorías en la implementación de hilos:

Hilos a nivel de usuario

Hilos a nivel de Kernel

También conocidos como **ULT** (*User Level Thread*) y **KLT** (*Kernel Level Thread*)

Hilos a nivel de usuario (ULT)]

En una aplicación ULT pura, todo el trabajo de gestión de hilos lo realiza la aplicación y el núcleo o kernel no es consciente de la existencia de hilos. Es posible programar una aplicación como multihilo mediante una biblioteca de hilos. La misma contiene el código para crear y destruir hilos, intercambiar mensajes y datos entre hilos, para planificar la ejecución de hilos y para salvar y restaurar el contexto de los hilos.

Todas las operaciones descritas se llevan a cabo en el espacio de usuario de un mismo proceso. El kernel continua planificando el proceso como una unidad y asignándole un único estado (Listo, bloqueado, etc.).

Ventajas de los ULT]

El intercambio de los hilos no necesita los privilegios del modo kernel, por que todas las estructuras de datos están en el espacio de direcciones de usuario de un mismo proceso. Por lo tanto, el proceso no debe cambiar a modo kernel para gestionar hilos. Se evita la sobrecarga de cambio de modo y con esto el sobrecoste o overhead.

Se puede realizar una planificación específica. Dependiendo de que aplicación sea, se puede decidir por una u otra planificación según sus ventajas.

Los ULT pueden ejecutar en cualquier sistema operativo. La biblioteca de hilos es un conjunto compartido.

Desventajas de los ULT]

En la mayoría de los sistemas operativos las llamadas al sistema (System calls) son bloqueantes. Cuando un hilo realiza una llamada al sistema, se bloquea el mismo y también el resto de los hilos del proceso.

En una estrategia ULT pura, una aplicación multihilo no puede aprovechar las ventajas de los multiprocesadores. El núcleo asigna un solo proceso a un solo procesador, ya que como el núcleo no interviene, ve al conjunto de hilos como un solo proceso.

Una solución al bloqueo mediante a llamadas al sistema es usando la técnica de jacketing, que es convertir una llamada bloqueante en no bloqueante.

Hilos a nivel de núcleo (KLT)]

En una aplicación KLT pura, todo el trabajo de gestión de hilos lo realiza el kernel. En el área de la aplicación no hay código de gestión de hilos, únicamente un API (interfaz de programas de aplicación) para la gestión de hilos en el núcleo. Windows 2000, Linux y OS/2 utilizan este método. Linux utiliza un método muy particular en que no hace diferencia entre procesos e hilos, para linux si varios proceso creados con la llamada al sistema "clone" comparten el mismo espacio de direcciones virtuales el sistema operativo los trata como hilos y lógicamente son manejados por el kernel.

Ventajas de los KLT]

El kernel puede planificar simultáneamente múltiples hilos del mismo proceso en múltiples procesadores.

Si se bloquea un hilo, puede planificar otro del mismo proceso.

Las propias funciones del kernel pueden ser multihilo

Desventajas de los KLT

El paso de control de un hilo a otro precisa de un cambio de modo.

Combinaciones ULT y KLT

Algunos sistemas operativos ofrecen la combinación de ULT y KLT, como Solaris.

La creación de hilos, así como la mayor parte de la planificación y sincronización de los hilos de una aplicación se realiza por completo en el espacio de usuario. Los múltiples ULT de una sola aplicación se asocian con varios KLT. El programador puede ajustar el número de KLT para cada aplicación y máquina para obtener el mejor resultado global.

SEMÁFORO

Un **semáforo** es una variable especial protegida (o tipo abstracto de datos) que constituye el método clásico para restringir o permitir el acceso a recursos compartidos (por ejemplo, un recurso de almacenamiento del sistema o variables del código fuente) en un entorno de multiprocesamiento (en el que se ejecutarán varios procesos concurrentemente). Fueron inventados por Edsger Dijkstra y se usaron por primera vez en el sistema operativo THEOS.

Operaciones

Los semáforos sólo pueden ser manipulados usando las siguientes operaciones (*este es el código con espera activa*):

```
Inicia(Semáforo s, Entero v)
{
     s = v;
}
```

En el que se iniciará la variable semáforo s a un valor entero v.

```
P(Semáforo s)
{
    while (s >= 0); /* espera hasta que s < 0 */
    s = s-1;
}
```

La cual mantendrá en espera activa al regido por el semáforo si este tiene un valor inferior o igual al nulo.

```
V(Semáforo s)
{
    s = s+1;
}
```

Estas instrucciones pueden modificarse para evitar la espera activa, haciendo que la operación P duerma al mismo proceso que la ejecuta si no puede decrementar el valor, mientras que la operación V despierta a un proceso que no es quien la ejecuta. En un pseudolenguaje más entendible, la operación P suele denominarse "wait" o "espera" y la operación V "signal" o "señal".

El por qué de los nombres de estas funciones, V y P, tienen su origen en el idioma holandés. "Verhogen" significa incrementar y "Proberen" probar, aunque Dijkstra usó la palabra inventada *prolaag*. que es una combinación de *probeer te verlagen* (intentar decrementar). El valor del semáforo es el número de unidades del recurso que están disponibles (si sólo hay un recurso, se utiliza un "semáforo binario" con los valores 0 y 1).

Si hay n recursos, se inicializará el semáforo al número n. Así, cada proceso, al ir solicitando un recurso, verificará que el valor del semáforo sea mayor de 0; si es así es que existen recursos libres, seguidamente acaparará el recurso y decrementará el valor del semáforo.

Cuando el semáforo alcance el valor 0, significará que todos los recursos están siendo utilizados, y los procesos que quieran solicitar un recurso deberán esperar a que el semáforo sea positivo, esto es: alguno de los procesos que están usando los recursos habrá terminado con él e incrementará el semáforo con un signal o V(s).

Inicia se utiliza para inicializar el semáforo antes de que se hagan peticiones sobre él, y toma por argumento a un entero. La operación *P* cuando no hay un recurso disponible, detiene la ejecución quedando en espera activa (o durmiendo) hasta que el valor del semáforo sea positivo, en cuyo caso lo reclama inmediatamente decrementándolo. *V* es la operación inversa: hace disponible un recurso después de que el proceso ha terminado de usarlo. Las operaciones *P* y *V* han de ser indivisibles (o atómicas), lo que quiere decir que cada una de las operaciones no debe ser interrumpida en medio de su ejecución.

La operación *V* es denominada a veces *subir* el semáforo (*up*) y la operación *P* se conoce también como *bajar* el semáforo (*down*), y también son llamadas *signal* y *wait* o *soltar* y *tomar*.

Para evitar la espera activa, un semáforo puede tener asociada una cola de procesos (normalmente una cola FIFO). Si un proceso efectúa una operación *P* en un semáforo que tiene valor cero, el proceso es detenido y añadido a la cola del semáforo. Cuando otro proceso incrementa el semáforo mediante la operación *V* y hay procesos en la cola asociada, se extrae uno de ellos (el primero que entró en una cola FIFO) y se reanuda su ejecución.

Usos

Los semáforos se emplean para permitir el acceso a diferentes partes de programas (llamados **secciones críticas**) donde se manipulan variables o recursos que deben ser accedidos de forma especial. Según el valor con que son inicializados se permiten a más o menos procesos utilizar el recurso de forma simultánea.

Un tipo simple de semáforo es el **binario**, que puede tomar solamente los valores 0 y 1. Se inicializan en 1 y son usados cuando sólo un proceso puede acceder a un recurso a la vez. Son esencialmente lo mismo que los mutex. Cuando el recurso está disponible, un proceso accede y decrementa el valor del semáforo con la operación P. El valor queda entonces en 0, lo que hace que si otro proceso intenta decrementarlo tenga que esperar. Cuando el proceso que decrementó el semáforo realiza una operación *V*, algún proceso que estaba esperando puede despertar y seguir ejecutando.

Para hacer que dos procesos se ejecuten en una secuencia predeterminada puede usarse un semáforo inicializado en 0. El proceso que debe ejecutar primero en la secuencia realiza la operación *V* sobre el semáforo antes del código que debe ser ejecutado después del otro proceso. Éste ejecuta la operación *P*. Si el segundo proceso en la secuencia es programado para ejecutar antes que el otro, al hacer *P* dormirá hasta que el primer proceso de la secuencia pase por su operación *V*. Este modo de uso se denomina señalación (*signaling*), y se usa para que un proceso o hilo de ejecución le haga saber a otro que algo ha sucedido.

Ejemplo de uso

Los semáforos pueden ser usados para diferentes propósitos, entre ellos:

Implementar cierres de exclusión mutua o locks

Barreras

Permitir a un máximo de N threads acceder a un recurso, inicializando el semáforo en N

Notificación. Inicializando el semáforo en 0 puede usarse para comunicación entre threads sobre la disponibilidad de un recurso

En el siguiente ejemplo se crean y ejecutan n procesos que intentarán entrar en su sección crítica cada vez que puedan, y lo lograrán siempre de a uno por vez, gracias al uso del semáforo s inicializado en 1. El mismo tiene la misma función que un lock.

```
const int n    /* número de procesos */
variable semaforo s; /* declaración de la variable semáforo de valor entero*/
Inicia (s,1)   /* Inicializa un semáforo con nombre s con valor 1 */
void P (int i)
{
   while (cierto)
   {
      P(s)    /* En semáforos binarios, lo correcto es poner un P(s) antes de entrar en
              la sección crítica, para restringir el uso de esta región del código*/

      /* SECCIÓN CRÍTICA */
      V(s)    /* Tras la sección crítica, volvemos a poner el semáforo a 1 para que otro
              proceso pueda usarla */

      /* RESTO DEL CÓDIGO */
   }
}
  void main()
{
   Comenzar-procesos(P(1), P(2), ... ,P(n));
}
```