CAPITULO 12

ADMINISTRACIÓN DEL PROCESADOR

12.1. – THREADS – Introducción [10][11][13]

En la mayoría de los sistemas operativos tradicionales cada proceso tiene un espacio de direcciones y un hilo de control. De hecho, ésa es casi la definición de un proceso. Consideremos, por ejemplo, un servidor de archivos que debe bloquearse en forma ocasional en espera de acceso al disco. Si el servidor tiene varios hilos de control podría ejecutarse un segundo hilo mientras el primero duerme. Esto no es posible si se crean dos procesos servidores independientes ya que deben compartir un buffer en común, lo que implicaría que deben estar en el mismo espacio de direcciones.

Un Thread (hilo), a veces llamado un lightweight process, es una unidad básica de uso de CPU, y cada hilo posee un Program Counter, un Register Set y un Stack. En muchos sentidos los hilos son como pequeños miniprocesos. Cada thread se ejecuta en forma estrictamente secuencial compartiendo la CPU de la misma forma que lo hacen los procesos, solo en un multiprocesador se pueden realizan en paralelo.

Los hilos pueden crear hilos hijos y se pueden bloquear en espera de llamadas al sistema, al igual que los procesos regulares. Mientras un hilo está bloqueado se puede ejecutar otro hilo del mismo proceso.

Puesto que cada hilo tiene acceso a cada dirección virtual (comparten un mismo espacio de direccionamiento), un hilo puede leer, escribir o limpiar la pila de otro hilo. No existe protección entre los hilos debido a que es imposible y no es necesario, ya que no son hostiles entre sí, es más a veces cooperan. Aparte del espacio de direcciones, comparten el mismo conjunto de archivos abiertos, procesos hijos, cronómetro, señales, etc.

ELEMENTOS POR HILOS	ELEMENTOS POR PROCESOS
Contador del programa	Espacio de dirección
Pila	Variables globales
Conjunto de Registros	Archivos Abiertos
Hilos hijos	Procesos hijos
Estado	Cronómetros
	Señales
	Semáforos
	Información contable

12.2. - USO DE LOS HILOS [10][11]

Los hilos se inventaron para permitir la combinación del paralelismo con la ejecución secuencial y el bloqueo de las llamadas al sistema. Existen 3 formas de organizar un proceso de muchos hilos en un server.

Estructura Servidor Trabajador
Estructura En Equipo
Estructura Entubamiento

12.2.1. - Estructura Servidor Trabajador [10][11]

Existe un hilo en el servidor que lee las solicitudes de trabajo en un buzón del sistema, examina éstas y elige a un hilo trabajador inactivo y le envía la solicitud, la cual se realiza con frecuencia al colocarle un puntero al mensaje en una palabra especial asociada a cada hilo. El servidor despierta entonces al trabajador dormido (un signal al semáforo asociado).

El hilo verifica entonces si puede satisfacer la solicitud desde el bloque cache compartido, sino puede inicia la operación correspondiente (por ejemplo podría lanzar una lectura al disco) y se duerme nuevamente a la espera de la conclusión de ésta. Entonces, se llama al planificador para iniciar otro hilo, ya sea hilo servidor o trabajador.

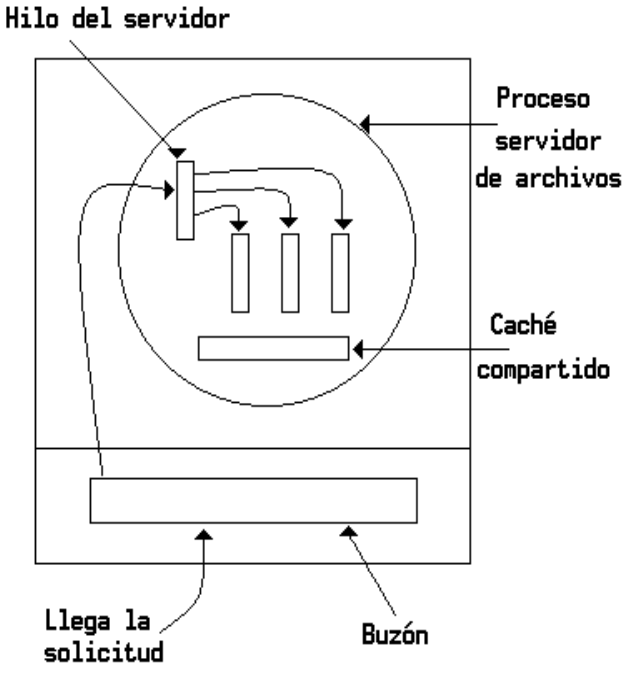


Fig. 12.1. - Hilo trabajador.

Otra posibilidad para esta estructura es que opere como un hilo único. Este esquema tendría el problema de que el hilo del servidor lanza un pedido y debe esperar hasta que éste se complete para lanzar el próximo (secuencial).

Una tercera posibilidad es ejecutar al servidor como una gran máquina de estado finito. Esta trata de no bloquear al único hilo mediante un registro del estado de la solicitud actual en una tabla, luego obtiene de ella el siguiente mensaje, que puede ser una solicitud de nuevo trabajo, o una respuesta de una operación anterior. En el caso del nuevo trabajo, éste comienza, en el otro caso se busca la información relevante en la tabla y se procesa la respuesta. Este modelo no puede ser usado en RPC puesto que las llamadas al sistema no son bloqueantes, es decir no se permite enviar un mensaje y bloquearse en espera de una respuesta (como veremos más adelante).

En este esquema hay que guardar en forma explícita el estado del cómputo y restaurarlo en la tabla para cada mensaje enviado y recibido. Este método mejora el desempeño a través del paralelismo pero utiliza llamadas no bloqueantes y por lo tanto es difícil de programar.

Modelo	**Características**
Hilos	Paralelismo; llamadas al sistema bloqueantes
Servidor de un solo hilo; Sin paralelismo	llamadas al sistema bloqueantes
Máquina de estado finito; Paralelismo	llamadas al sistema no bloqueantes

12.2.2. - Estructura en Equipo [10][11]

Todos los hilos son iguales y cada uno obtiene y procesa sus propias solicitudes. Cuando un hilo no puede manejar un trabajo por ser hilos especializados se puede utilizar una cola de trabajo. Esto implica que cada hilo verifique primero la cola de trabajo antes de mirar el buzón del sistema.

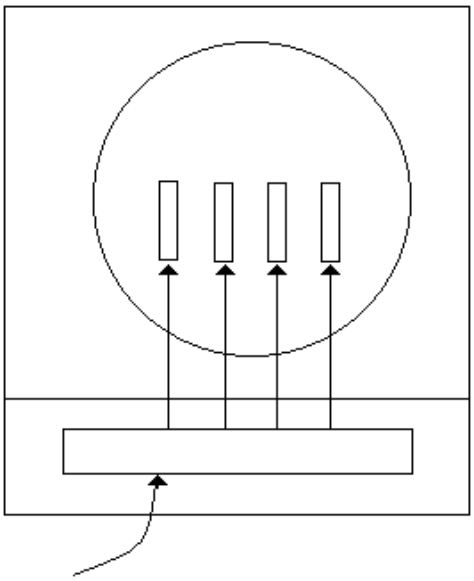


Fig.12.2. - Estructura en equipo.

12.2.3. - Estructura de Entubamiento (pipeline) [10][11]

El primer hilo genera ciertos datos y los transfiere al siguiente para su procesamiento. Los datos pasan de hilo en hilo y en cada etapa se lleva a cabo cierto procesamiento. Esta puede ser una buena opción para productor/consumidor, no así para los servidores de archivos.

12.2.4. - Otros usos de los hilos [10][11]

En un cliente los threads son usados para copiar un archivo a varios servidores, por ejemplo si un cliente quiere copiar un mismo archivo a varios servidores puede dedicar un hilo para que se comunique con cada uno de ellos.

Otro uso de los hilos es el manejar las señales, como las interrupciones del teclado. Por ejemplo se dedica un hilo a esta tarea que permanece dormido y cuando se produce una interrupción el hilo se despierta y la procesa.

Existen aplicaciones que se prestan para ser programadas con el modelo de hilos, por ejemplo el problema de los productores-consumidores. Nótese que como comparten un buffer en común no funcionaría el hecho de tenerlos en procesos ajenos.

Por último nótese la utilidad de los hilos en sistemas multiprocesadores en dónde pueden realmente ejecutarse en forma paralela.

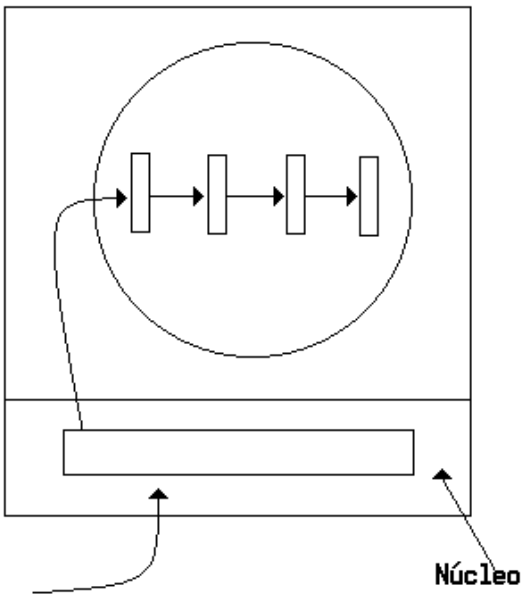


Fig. 12.3.- Estructura pipeline.

12.3. - ASPECTOS DEL DISEÑO DE PAQUETES DE THREADS [10][11]

Un conjunto de primitivas relacionadas con los hilos disponibles para los usuarios se llama un *paquete de hilos*. Se pueden tener hilos estáticos o dinámicos.

En un diseño estático se elige el número de hilos al escribir el programa, o durante su compilación. Cada uno de ellos tiene una pila fija.

En el dinámico se permite la creación y destrucción de los hilos en tiempo de ejecución. La llamada para la creación de un hilo establece el programa principal del hilo (como si fuera un apuntador a un procedimiento) y un tamaño de pila, también otros parámetros, como ser prioridad de planificación.

La llamada devuelve un identificador de hilo para ser usado en posteriores llamadas relacionadas con ese hilo. Entonces un proceso se inicia con un único hilo, pero puede crear el numero necesario de ellos.

La terminación de los hilos puede ser de dos maneras, al terminar su trabajo o bien desde el exterior. En el ejemplo del servidor de archivos los hilos se crean una vez iniciado el programa y nunca se eliminan.

Ya que los hilos comparten una memoria común el acceso a datos comunes se programa mediante regiones críticas que se implementan mediante el uso de semáforos, monitores u otras construcciones similares. Se utiliza el semáforo de exclusión *mutex* sobre el cual se definen las operaciones de cierre (LOCK) y liberación (UNLOCK). Existe además una operación TRYLOCK que para el cierre funciona igual que el Lock pero si el semáforo está cerrado en lugar de bloquear el hilo regresa un código de estado que indica falla.

Se encuentra disponible también la *variable de condición*, similar en todo a la que se utiliza en monitores (como veremos más adelante).

El código de un hilo consta al igual que un proceso de varios procedimientos, pudiendo tener variables locales, globales y variables del procedimiento. De éstas las globales producen problemas, ya que el valor de una variable global puesto por un hilo que se duerme puede ser modificado por otro, creando así una incoherencia cuando se despierta. Las soluciones que se presentan son:

a) Prohibir las variables globales: esto presenta conflictos con el software ya existente, como por ejemplo Unix.

b) Asignarle a cada hilo sus propias variables globales particulares. Esto introduce un nuevo nivel de visibilidad, ya que las variables son visibles a todos los procedimientos de un hilo además de las variables visibles a un procedimiento específico y las visibles a todo el programa. Esta alternativa tiene el inconveniente de no ser posible de implementar en la mayoría de los lenguajes de programación. Una forma de hacerlo es asignar un bloque de memoria a las variables globales y transferirlas a cada procedimiento como un parámetro adicional.

c) Nuevos procedimientos en bibliotecas para crear, leer y escribir estas variables. La creación de una variable global implicaría la asignación de un apuntador en un espacio de almacenamiento dedicado a ese hilo de forma que solo él tiene acceso a la variable global definida.

12.3.1 - Llamadas. [10][11]

Presentamos un resumen de las llamadas que pueden existir para el manejo de los hilos o threads:

Llamadas de manejo de Threads
- Create, Exit, Join, Detach

Llamadas de Sincronización (Manejo de regiones criticas)
- Mutex_init, Mutex_destroy, Mutex_lock, Mutex_trylock, Mutex_unlock

Llamadas de Condición Variables (usados para el bloqueo de recursos)
- Cond_init, Cond_destroy, Cond_wait, Cond_signal, Cond_broadcast

Llamadas de Scheduling (administran las prioridades de los hilos)
- Setscheduler, Getscheduler, Setprio, Getprio

Llamadas de Eliminación
- Cancel, Setcancel

12.4. - IMPLEMENTACIÓN DE UN PAQUETE DE HILOS [10][11]

Aquí trataremos la planificación de los hilos mediante distintos algoritmos, ya que el usuario puede especificar el algoritmo de planificación y establecer las prioridades en su caso. Existen dos formas de implementar un paquete de hilos:

1) Colocar todo el paquete de hilos en el espacio del usuario (el núcleo no conoce su existencia),
2) Colocar todo el paquete de hilos en el espacio del núcleo (el núcleo sabe de su existencia)

12.4.1. - Paquete de hilos en el espacio del usuario [10][11]

La ventaja es que este modelo puede ser implantado en un sistema operativo que no tenga soporte para hilos.

Los hilos se ejecutan arriba de un SISTEMA DE TIEMPO DE EJECUCIÓN (Intérprete), que se encuentra en el espacio del usuario en contacto con el núcleo. Por ello, cuando un hilo ejecuta una llamada al sistema, se duerme, ejecuta una operación en un semáforo o mutex, o bien cualquier operación que pueda provocar su suspensión, se llama a un procedimiento del sistema de tiempo de ejecución el cual verifica si debe suspender al hilo.

En caso afirmativo almacena los registros del hilo (propios) en una tabla, busca un hilo no bloqueado para ejecutarlo, vuelve a cargar los registros guardados del nuevo hilo, y apenas intercambia el apuntador a la pila y el apuntador del programa, el hilo comienza a ejecutar.

El intercambio de hilos es de una magnitud menor en tiempo que una interrupción siendo esta característica un fuerte argumento a favor de esta estructura.

Este modelo tiene una gran escalabilidad y además permite a cada proceso tener su propio algoritmo de planificación. El sistema de tiempo de ejecución mantiene la ejecución de los hilos de su propio proceso hasta que el núcleo le retira el CPU.

12.4.2. - Paquete de hilos en el núcleo [10][11]

Para cada proceso, el núcleo tiene una tabla con una entrada por cada hilo, con los registros, estado, prioridades, y demás información relativa al hilo (ídem a la información en el caso anterior), solo que ahora están en el espacio del núcleo y no dentro del sistema de tiempo de ejecución del espacio del usuario.

Todas las llamadas que pueden bloquear a un hilo se implantan como llamadas al sistema, esto representa un mayor costo que el esquema anterior por el cambio de contexto. Al bloquearse un hilo, el núcleo opta entre ejecutar otro hilo listo, del mismo proceso, o un hilo de otro proceso.

12.4.3. - PROBLEMAS [10][11]

a) Implementación de las llamadas al sistema con bloqueo.

Un hilo hace algo que provoque un bloqueo, entonces en el esquema de hilos a nivel del usuario no se puede permitir que el thread realice en realidad la llamada al sistema ya que con esto detendría todos los demás hilos y uno de los objetivos es permitir que usen llamadas bloqueantes pero evitando que este bloqueo afecte a los otros hilos.

En cambio en el esquema de hilos en el núcleo directamente se llama al núcleo el cual bloquea al hilo y luego llama a otro.

Hay una forma de solucionar el problema en el primer esquema y es verificar de antemano que una llamada va a provocar un bloqueo, en caso positivo se ejecuta otro hilo. Esto lleva a escribir parte de las rutinas de la biblioteca de llamadas al sistema, si bien es ineficiente no existen muchas alternativas. Se denomina jacket.

Algo similar ocurre con las fallas de página. Si un hilo produce una falla de página el núcleo que desconoce que dentro del proceso del usuario hay varios hilos bloquea todo el proceso.

b) Administración de los hilos (scheduling).

En el esquema de hilos dentro del proceso del usuario cuando un hilo comienza su ejecución ninguno de los demás hilos de ese proceso puede ejecutarse a menos que el primer hilo entregue voluntariamente el CPU. Dentro de un único proceso no existen interrupciones de reloj, por lo tanto no se puede planificar con quantum.

En el esquema de hilos en el núcleo las interrupciones se presentan en forma periódica obligando a la ejecución del planificador.

c) Constantes llamadas al sistema.

En el esquema de hilos a nivel usuario se necesitaría la verificación constante de la seguridad de las llamadas al sistema, es decir es mucho más simple el bloqueo en los hilos a nivel núcleo que a nivel usuario puesto que en el núcleo solo bloquea al hilo y conmuta al próximo hilo mientras que a nivel usuario necesita antes de bloquearse llamar al sistema de tiempo de ejecución para conmutar y luego bloquearse.

d) Escalabilidad

El esquema de hilos a nivel usuario tiene mejor escalabilidad ya que en el esquema de hilos a nivel núcleo éstos requieren algún espacio para sus tablas y su pila en el núcleo lo que se torna inconveniente si existen demasiados hilos.

e) Problema general de los hilos

Muchos de los procedimientos de biblioteca no son reentrantes. El manejo de las variables globales propias es dificultoso. Hay procedimientos (como la asignación de memoria) que usan tablas cruciales sin utilizar regiones críticas, pues en un ambiente con un único hilo eso no es necesario. Para poder solucionar estos problemas de manera adecuada habría que reescribir toda la biblioteca. Otra forma es proveer un jacket a cada hilo de manera que cierre un mutex global al iniciar un procedimiento. De esta forma la biblioteca se convierte en un enorme monitor.

El uso de las señales (traps - interrupciones) también producen dificultades, por ejemplo dos hilos que deseen capturar la señal de una misma tecla pero con propósitos distintos. Ya es difícil manejar las señales en ambientes con un solo hilo y en ambientes multithreads las cosas no mejoran. Las señales son un concepto típico por proceso y no por hilo, en especial, si el núcleo no está consciente de la existencia de los hilos.

12.5 – INTRODUCCION ADMINISTRACIÓN DEL PROCESADOR [9][44][S.O.]

La administración del procesador es, prácticamente, el tema central de la multiprogramación. Esta administración involucra las distintas maneras a través de las cuales el Sistema Operativo comparte el recurso procesador entre distintos procesos que están compitiendo por su uso. Esto implica directamente la multiprogramación y conlleva simultáneamente la sincronización de los mismos.

La idea de administrar el procesador eficientemente está enfocada en dos aspectos: el primero es la cantidad de procesos por unidad de tiempo que se pueden ejecutar en un sistema; y el segundo, el que importa más al usuario, es el tiempo de respuesta de esos procesos.

- Cantidad de Procesos por Unidad de Tiempo (throughput)
- Tiempo de Respuesta (turnaround time)

La idea de repartir el recurso procesador entre distintos procesos se debe a que tenemos la posibilidad de utilizar el tiempo de procesador abandonado por un proceso para que lo pueda usar otro. O sea aprovechar los tiempos muertos de un determinado proceso para que se puedan ejecutar otros.

Estos tiempos muertos se producen porque existen otras actividades que están desarrollándose sobre cierto proceso. Esas otras actividades generalmente son de E/S, y esto es posible porque existe algo que está ayudando a realizar esa E/S, es decir, existen canales o procesadores de E/S que ayudan a descargar del procesador central esa actividad.

12.6. - Turnaround [9][44]

Normalmente nos encontramos con un proceso A que tiene una cantidad de tiempo de procesador, y otra cantidad de tiempo muerto desde el punto de vista del procesador, porque está realizando una operación de E/S, procesa nuevamente, E/S, y procesa nuevamente.

La idea es tratar de, en esos momentos en donde la actividad está descargada en un procesador especializado en E/S, usar ese tiempo para que otro proceso ejecute.

Supóngase que existe otro proceso homogéneo a éste, es decir, las mismas ráfagas de procesador, y los mismos tiempos de E/S (3 ráfagas de procesador y 2 operaciones de E/S), y ambos procesos se ejecutaran en monoprogramación , el primero tardaría 5 unidades, y el segundo tardará otras 5 unidades. Es decir que recién después de 10 unidades de tiempo se tendría la finalización de ambos programas, con lo cual se obtiene un tiempo de respuesta promedio de 7.5 unidades (Ver Fig. 12.4 - turnaround (10+5) / 2= 7.5 -).

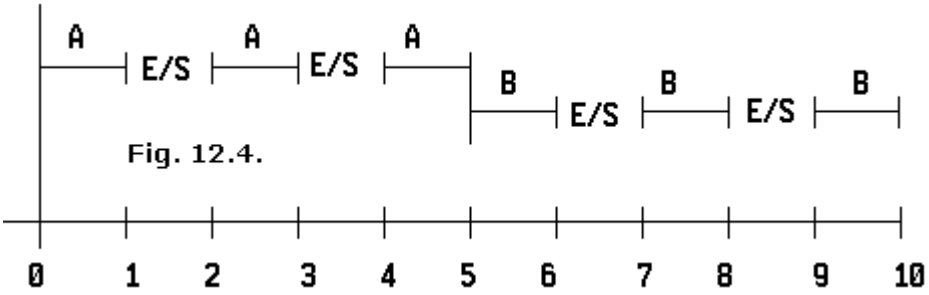


Fig. 12.4.

Si se sumergiera esta situación en un ambiente de multiprogramación, suponiendo que los tiempos de E/S de uno de los programas coinciden con las ráfagas de procesador del otro, se podría obtener un esquema de distribución como se ve en la Fig. 12.5.

De esta manera el tiempo de respuesta promedio es de 5.5 (Turnaround) (6+5)/2= 5.5.

La ventaja es que los procesos terminan antes del valor esperado en promedio en monoprogramación. Un elemento de medida que es el tiempo medio de terminación de programa permite verificar las bondades de los distintos algoritmos que se verán a continuación.

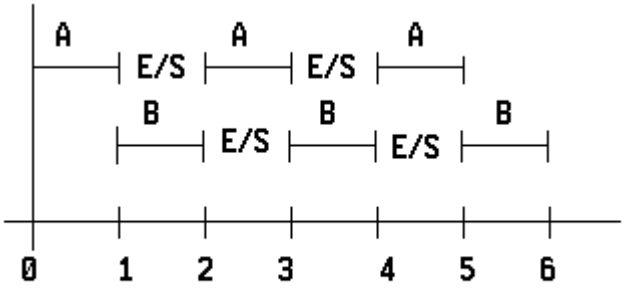
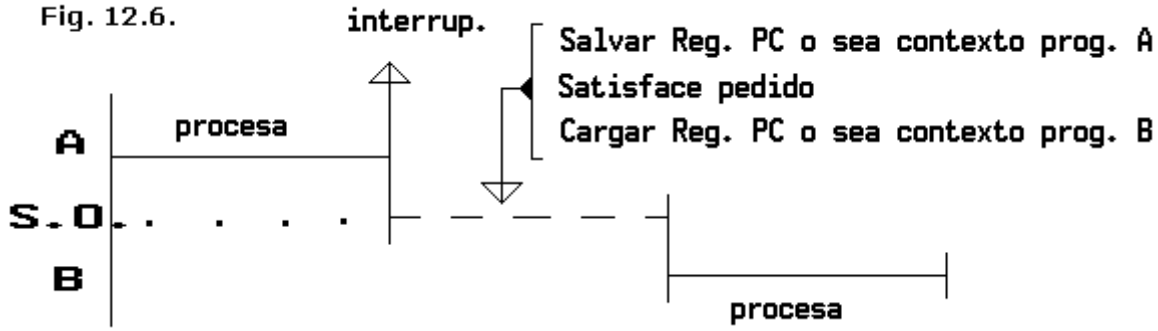


Fig. 12.5.

El tiempo medio de respuesta, turnaround, se obtiene como la suma de los tiempos de terminación de los procesos dividida la cantidad de procesos. Este turnaround, en realidad, está indicando el índice de felicidad de un usuario, da un valor esperado que debe interpretarse de la siguiente manera : A las x unidades de tiempo se puede tener la esperanza de que existen procesos que ya han finalizado, por lo tanto cuánto menor es x menos se debe esperar, en promedio, para que el sistema haya dado una respuesta.

Lógicamente que suponer que las ráfagas de procesador de uno de los procesos coinciden exactamente con los tiempos de E/S del otro proceso es prácticamente una ilusión, más aún considerando que existen otros

tiempos asociados a la multiprogramación que complican aún más la interacción entre los procesos. Consideremos por ejemplo el grafo de la Fig. 12.6.



O sea, en el primer momento se está ejecutando el proceso A, luego se produce alguna interrupción provocada por el mismo programa o recibida del exterior, se debe entonces salvar los registros y la palabra de control del proceso A (o sea su contexto), suponiendo que fue una interrupción por una llamada al supervisor para realizar una E/S, se satisfará ese pedido, y después se cargarán los registros y la palabra de control para el proceso B (o sea su contexto).

Es decir, que los tiempos, de acuerdo a lo visto anteriormente, no van a ser tan exactos, sino que se tendrá una cantidad de tiempo para realizar las tareas antes enumeradas, y recién después de esto se podrá ejecutar el siguiente proceso.

A este tiempo dedicado a la atención de interrupciones, salvaguarda y carga de contextos, en suma, el tiempo dedicado por el sistema operativo a ejecutar sus propias rutinas para proveer una adecuada administración entre los diferentes procesos se lo suele denominar **el overhead** (sobrecarga) del Sistema Operativo.

12.7 - Tablas y Diagrama de Transición de Estados

[9][44]

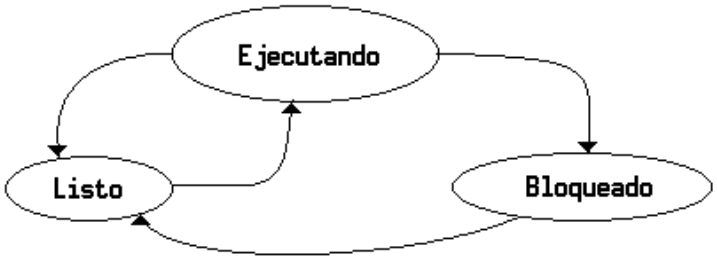


Fig. 12.7 - Diagrama de Transición de Estados.

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

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

- Ejecutando,
- Listo para la ejecución, o
- Bloqueados por alguna razón.

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 a ese proceso le está haciendo falta es el recurso 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.

El correspondiente Diagrama de Transición sería pues el de la Fig. 12.7.

Para manejar esa cola de listos se requiere de una tabla, y esa tabla, básicamente, lo que debe tener es una identificación de los procesos (Fig. 12.8).

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

Esta tabla contiene los Bloques de Control de Procesos. En este caso se agrupan los BCP en una Tabla de Bloques de Control de Procesos (TBCP).

Fig. 12.8.

Puntero al primero de los Listos →

Id. Proceso	Estado
A	Ejec.
B	Listo
C	Bloq.
D	Listo

12.8. - Bloque de Control de Proceso (BCP)

[9][44]

El Bloque de Control de Procesos contiene el contexto de un proceso y todos los datos necesarios para hacer posible la ejecución de ese proceso y satisfacer sus necesidades.

Cada entrada de la tabla tiene un apuntador al bloque anterior y uno al posterior, una identificación del proceso, la palabra de control, los registros, si se está trabajando en un sistema de administración de memoria paginada un apuntador a su tabla de distribución de páginas, dispositivos que esté usando, archivos que esté usando, tiempos que hacen a la vida del proceso, el estado, y apuntadores al anterior y posterior en el mismo estado.

Un esquema de una entrada de un Bloque de Control de Procesos puede verse en la Fig. 12.9.

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.

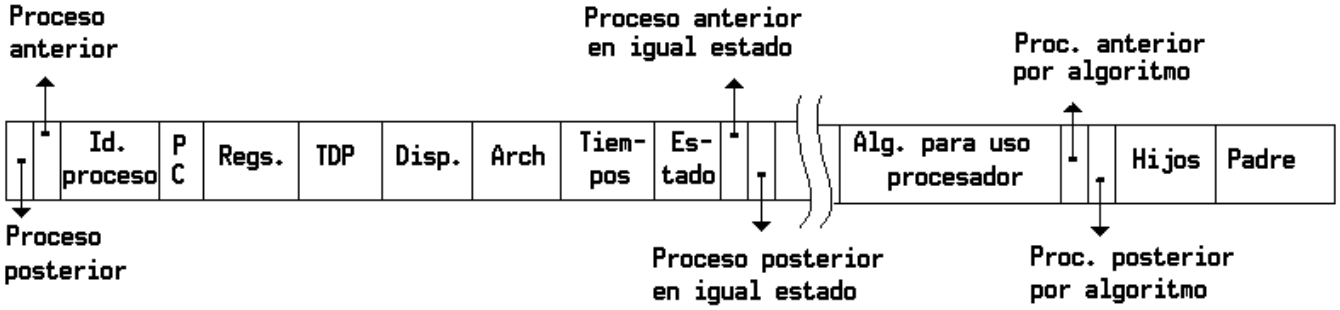


Fig. 12.9. - Contenido del Bloque de Control de Proceso (BCP).

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 nil.
- Accounting: información que servirá para contabilizar los gastos que produce este proceso (números contables, cantidad de procesos de E/S, etc.)

12.9 - Programa y Proceso [9][44]

Un programa es una entidad pasiva, mientras que un proceso o tarea (se usan indistintamente ambos términos) es una entidad activa.

O sea que un programa es un conjunto de instrucciones, un proceso es un conjunto de instrucciones más su contexto (BCP) y en ejecución.

Qué significa realmente un proceso o una tarea dentro de un programa?. Dados dos conjuntos de instrucciones que están dentro de un programa, si esos conjuntos son completamente independientes, de manera tal que su ejecución independiente no afecta el resultado del programa, luego cada conjunto es un proceso. Por ejemplo, si se tiene el siguiente conjunto de instrucciones :

A = A + C
Z = A + 1

y por otro lado :

D = E + F
Y = D + 1

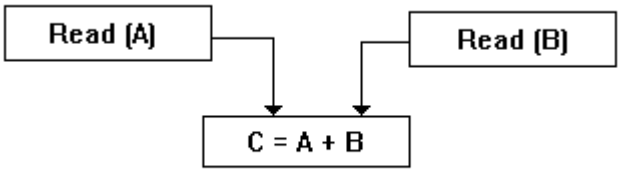


Fig. 12.10.

Estos dos conjuntos de instrucciones son completamente independientes, y que se ejecuten en forma independiente no afecta el resultado final del programa ya que están manejando datos distintos.

Para verlo en un caso bastante claro en el cual se podría dividir un programa en un par de tareas, supóngase que existe el siguiente conjunto de instrucciones:

READ A
READ B
C = A + B

Cada instrucción READ, que desencadena una serie de operaciones, es completamente independiente de la otra que desencadena otro conjunto de instrucciones completamente independientes del otro. En cambio, la operación C = A + B es dependiente de las otras dos, desde el momento que está usando resultados que le dan las dos anteriores. O sea que este fragmento de programa podría ser descompuesto en un grafo de la manera que se ve en la Fig. 12.10.

Pudiendo descomponer de esta manera a un programa, es realmente posible que las dos operaciones, más tarde conjunto de instrucciones, pueden ser ejecutadas en forma simultánea, desde el punto de vista de la lógica del programa, es decir no afecta hasta aquí que cualquiera de las dos se ejecute antes, a partir de la asignación en sí.

Existe toda una serie de condiciones para establecer cuándo es posible hacer esto que se verán más adelante cuando se estudien los procesos concurrentes, pero hay que tener bien en claro que es posible dividir un proceso en subconjuntos de procesos que llamamos secuenciales, y esos procesos son secuenciales si están compuestos por instrucciones que dependen unas de otras.

Si el ejemplo anterior se diera en un ambiente de multiprocesamiento sería posible efectuar las dos operaciones READ en forma simultánea, algo que en cursos anteriores se veía cómo una llamada al supervisor **sin devolución** del recurso procesador al sistema operativo, porque se volvía al programa para ejecutar otra operación que era completamente independiente de la operación anterior. Esto va a requerir algún elemento de sincronización, ya que, volviendo al ejemplo anterior, no se podrá realizar la asignación hasta que no se hayan completado los READ. Esto también es tema para el futuro.

Dado un programa es posible generar, por situaciones especiales, que se divida en distintas tareas que son independientes. Y estas también compiten por el recurso procesador, luego se tendrá que administrar el procesador también para distintas tareas que componen un programa y que van a utilizar ese recurso. Con lo cual también es posible que desde un bloque de control de un programa exista un apuntador X al bloque de control de procesos correspondientes a ese programa, y que dentro de ese bloque de control de procesos, cada entrada corresponda a un proceso o tarea independiente que lo conforman.

Luego, se puede hablar de programas o de procesos en forma indistinta ya que dado un programa se puede descomponer en tareas disjuntas. En definitiva se están manejando bloques de control de procesos que van a requerir del uso del procesador.

Un programa completo puede dividirse en procesos. Esto es típico de lenguajes que permiten multitareas, como PL-1, y Pascal concurrente, ya que, o bien lo indica el programador o porque el compilador es lo suficientemente inteligente como para darse cuenta de segmentar el programa en distintos procesos, y esos procesos pueden después competir por el recurso como si fueran procesos independientes.

Utilizar este esquema dentro de una computadora que tiene un solo procesador es hacer multiprogramación . En los sistemas que tienen más de un procesador, y con la posibilidad de ejecutar cada READ en cada uno de ellos, se está en presencia de un sistema de multiprocesamiento. Más aún, si se inserta este esquema en computadoras distintas con algún mecanismo de comunicación, considerando inclusive que pueden estar ubicadas en sitios remotos, se puede decir que se está frente a un sistema de computación distribuido.

Dividir a los programas en tareas requiere luego, obviamente, de un sistema de comunicación que permita que una vez que estén divididos en tareas, se los pueda insertar en cualquier tipo de arquitectura y aprovechar entonces sus facilidades específicas.

12.10 - Fin de un Proceso (total o temporal) [9][44]

Mientras se está ejecutando un proceso este puede abandonar su estado por diversas razones. Lo más deseable sería que el proceso terminase en forma satisfactoria.

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 causales de finalización se refieren al fin Total 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 (Fig. 12.11) visualizando además la existencia de colas para los recursos compartidos.

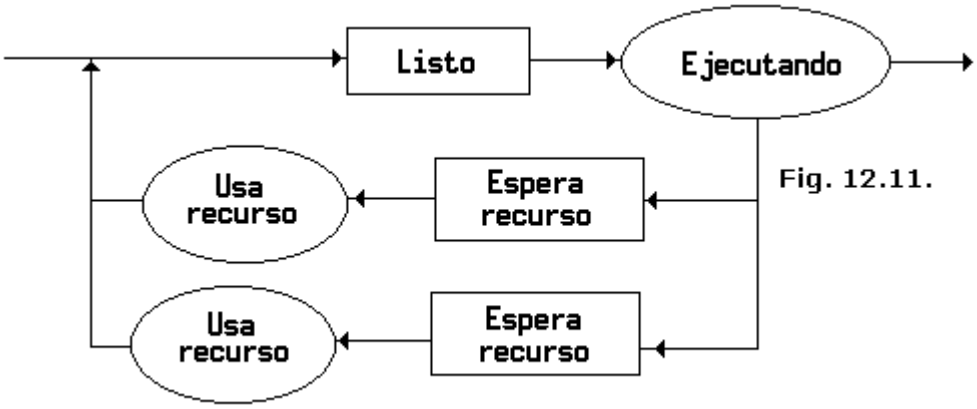


Fig. 12.11.

12.11 - Rutinas de Administración del Procesador [9][44]

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.

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

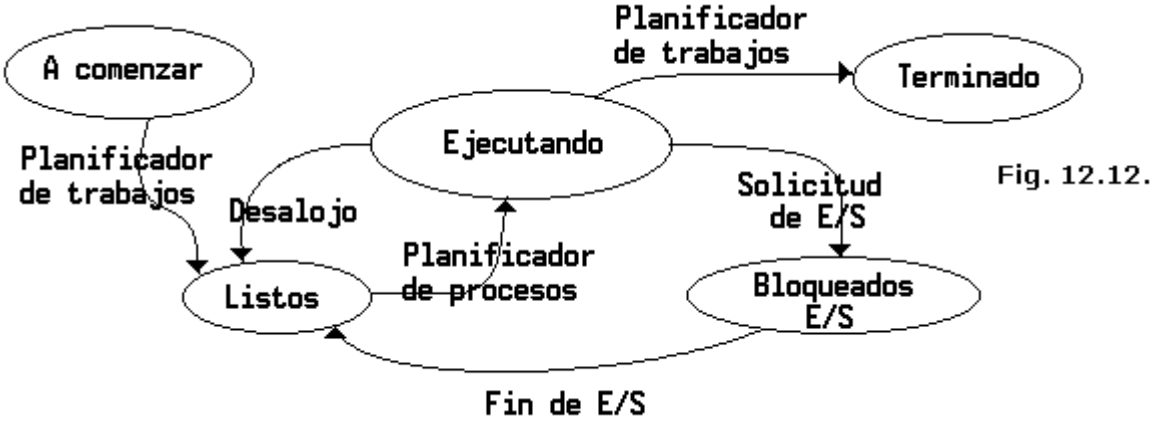


Fig. 12.12.

Sus funciones principales son :

- 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".

La administración de la cola de listos que es en donde se centrará nuestro estudio, es también llamada muchas veces planificador de bajo nivel, y se lo suele denominar **Planificador del Procesador o Planificador de Procesos**.

El Planificador de procesos 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, entre los cuales el primero que se visualiza es el **Planificador del Procesador**, que es el que aplica la política de selección, luego sigue un **Controlador de Tráfico**, que es el que realiza el manejo de las tablas, y además (dependiendo de la bibliografía que se consulte) un **Dispatcher**, que sería el que pone en estado de ejecución al programa, o sea 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 que 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.

Es muy importante tener bien en claro cuáles son las funciones :

- Manejo de las tablas,
- Selección del proceso de la cola de listos, y
- Poner en ejecución al proceso.

12.12 - Políticas de asignación [9][44]

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 termina
- 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.

12.12.1 - FIFO o FCFS [9][44]

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.

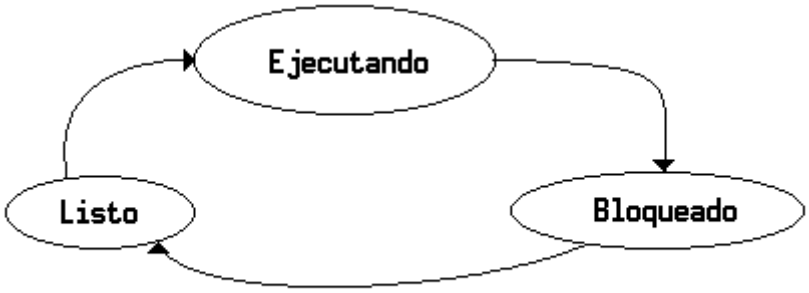


Fig. 12.13. - Administración del procesador FIFO.

Su esquema es el que se puede ver en la Fig. 12.13.

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.

12.12.2 - Más Corto Primero (JSF) Sin Desalojo. [9][44]

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 (Nota: 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.

Hay distintas formas de calcular la calificación, se pueden llegar a tener promedios de calificaciones anteriores, por ejemplo :

$$T_{n+1} = \S\ T_n + ... + (1-\S)\ \S\ T_{n-1} + ... + (1-\S)^j\ T_{n-j} + ...$$

en donde a § se lo designa con anterioridad. Si vale 1, significa que se tiene en cuenta la ráfaga anterior, si vale 0, la anterior no es tenida en cuenta, luego con 0 << § << 1 se pueden dar distintos pasos a los momentos históricos del proceso.

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.

12.12.3 - Más Corto Primero Con Desalojo.

[9][44]

Una variante del método del más corto primero es que exista desalojo. El diagrama de transición de estados tendría el aspecto que se puede ver en la Fig. 12.14.

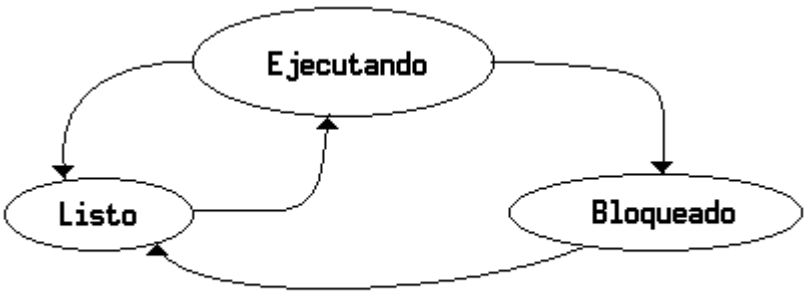


Fig. 12.14 - Más corto primero con desalojo.

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 faltarán 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.

12.12.4 - Administración por Prioridades [9][44]

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.

12.12.5 - Round-Robin [9][44]

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. Se puede visualizar esto como una calesita de la forma de la Fig. 12.15.

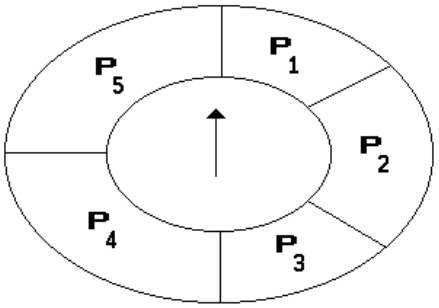


Fig. 12.15. - Calesita circular del método Round-Robin.

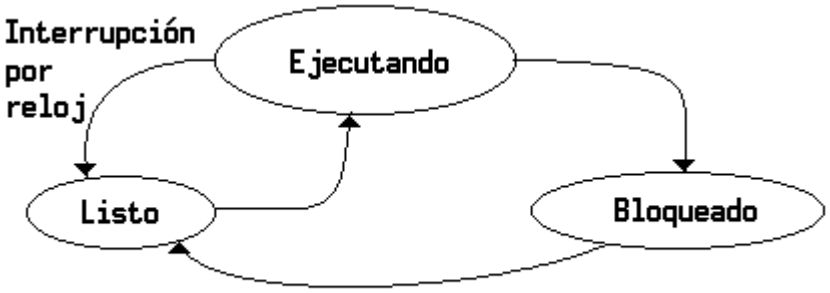


Fig. 12.16. - Administración del procesador Round-Robin.

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.

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.

12.12.6 - Multicolas [9][44]

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. Veamos el ejemplo de la Fig. 12.17.

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.

Además se podría considerar que cada uno 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.

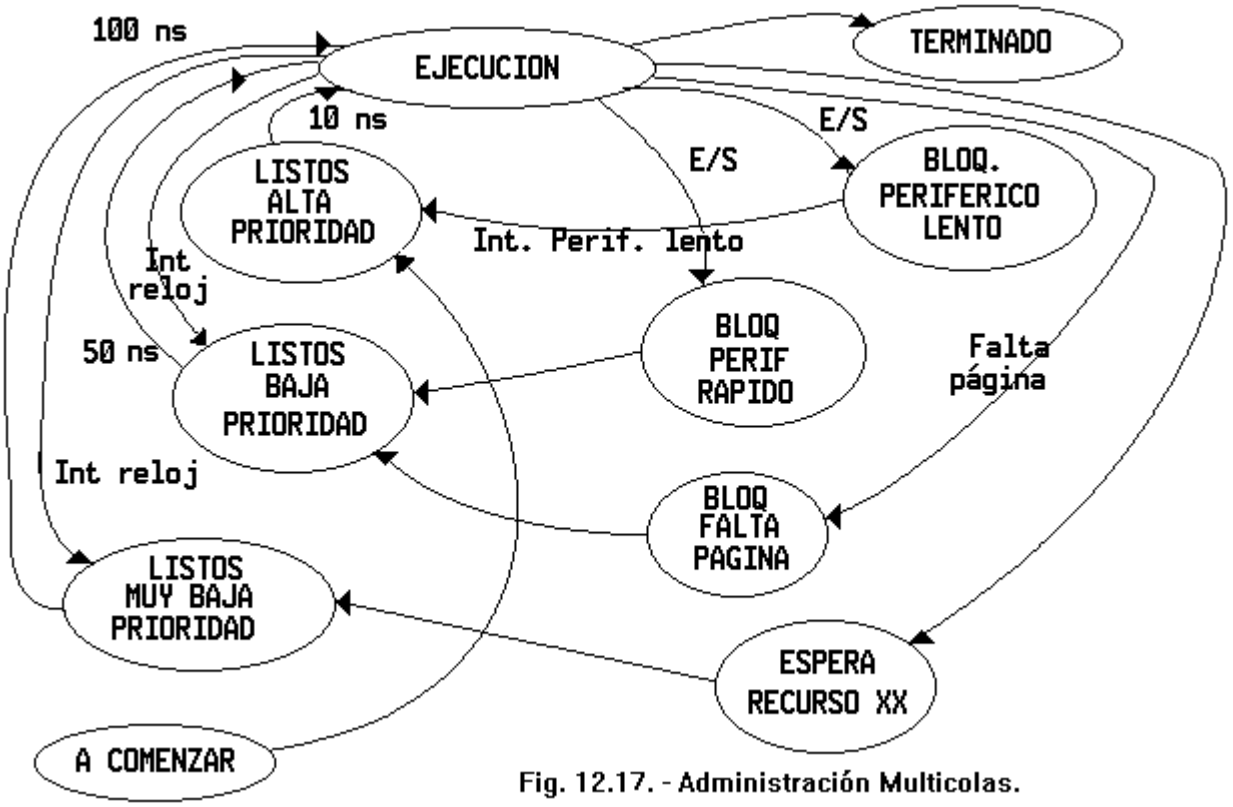


Fig. 12.17. - Administración Multicolas.

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 listos 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.

12.13 - Mediciones de performance [9][44][S.O.]

Una forma analítica de comparar los distintos algoritmos es haciendo cuentas con los tiempos que se conocen de los procesos. Supongamos un ejemplo en el cual hay 5 trabajos (5 procesos) cuyas ráfagas a considerar son las siguientes :

PROCESO	RAFAGA
A	5
B	30
C	3
D	10
E	12

y propongamos evaluar cuáles son los tiempos de espera. O sea cuánto espera cada uno de esos procesos. Los guarismos son :

FIFO (un.promedio)	25.2
El más corto primero (sin desalojo)	11.8
Round-Robin (quantum = 10)	20.2

Estas cifras se calculan de la siguiente forma :

En el caso del FIFO, la primera ráfaga del trabajo A dura 5, en el caso del B dura 30, en el caso del C dura 3, etc.

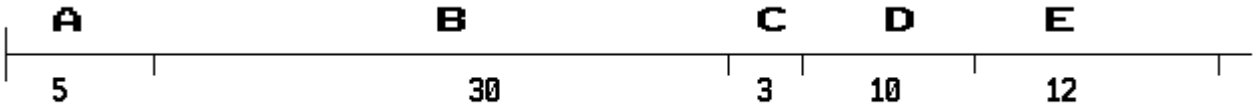


Fig. 12.18.

El primero no ha esperado nada para obtener su ráfaga, el segundo ha esperado 5, el tercero 35, etc. El cálculo final para hallar el tiempo de espera promedio sería entonces (0 + 5 + 35 + 38 + 48) / 5 = 25.2

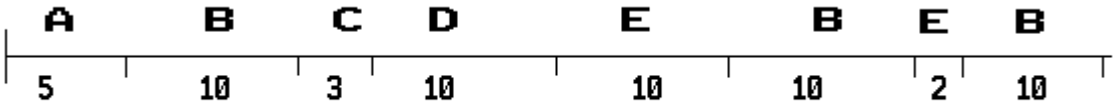


Fig. 12.19.

Para Round-Robin se tendrán pedacitos. Se tiene un quantum de 10, en el A se tiene 5, para el B se tiene 30, pero en 10 le llega la interrupción por reloj (le quedan 20 por ejecutar), para el C se tiene 3, etc.

Los tiempos de espera serán : para el A cero, para el B hay que sumar (5 + 23 + 2), para el C será 15, para el D será 18 y para el E habrá que sumar (28 + 10). El tiempo promedio de espera será igual a 20.20 (101 / 5)

El más corto primero es el mejor de todos, pero no se deben olvidar los problemas de implementación. Recuérdese que si se mezclan procesos interactivos con procesos de tipo batch de mucho tiempo de procesador estos últimos no ejecutarán nunca.

Para medir un sistema en general, lo que conviene hacer es, conociendo la carga del mismo, armar un conjunto de programas representativos de la instalación, correrlos todos juntos y obtener resultados. A partir de esos resultados, se deben tomar decisiones. Esto se conoce como **Benchmark** (banco de pruebas). Teniendo un subconjunto representativo de la carga natural de un sistema, sabiendo que el mismo tiene un 10% de programas COBOL, 50% de programas FORTRAN, y 40% de una actividad determinada, hay que hacer un banco de pruebas con esa representación, cargar los distintos sistemas, tomar las medidas necesarias y compararlos.

Esto no está midiendo sólo la actividad del procesador, sino también cómo funciona el sistema frente a la carga que se le presenta.

12.14 - Ejemplo: Caso Real [26][S.O.]

Veamos un ejemplo real de una administración multicola. Estas colas se manejan por medio del algoritmo Round-Robin. Este sistema intenta premiar las actividades interactivas. Es un sistema interactivo y deja siempre con la más alta prioridad a todo aquel que tenga una interacción con su terminal. Se manejan básicamente tres colas : Q1, Q2, Q3., las cuales comparten el recurso procesador de la siguiente manera :

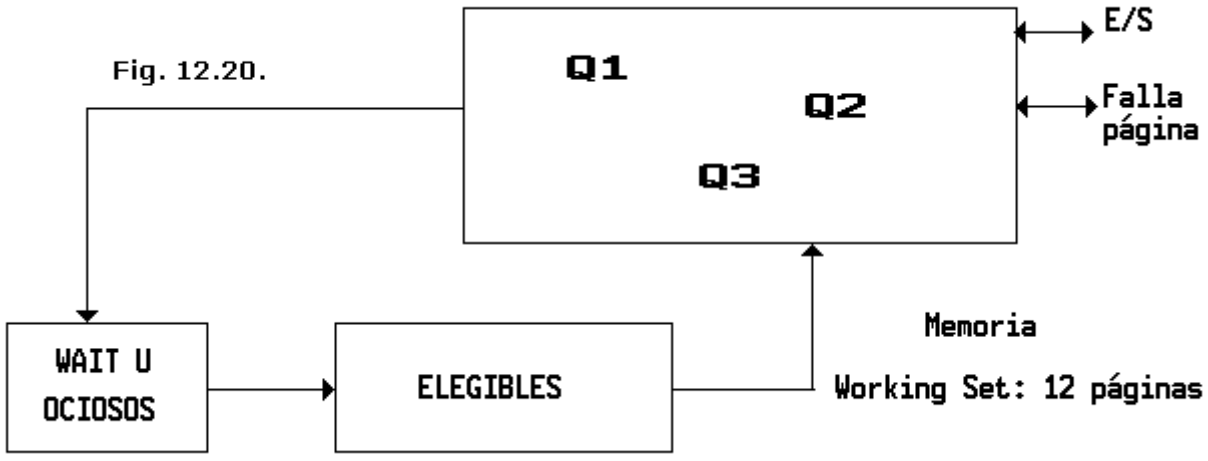


Fig. 12.20.

- dentro de Q1 la administración es Round-Robin y solamente se pasa a la Q2 cuando Q1 está vacía o cuando todos los procesos que están dentro de la Q1 están bloqueados. Lo mismo ocurre con la Q2 y la Q3.

En el Round-Robin no importa si un proceso está bloqueado. La representación en forma de calesita es la de la Fig. 12.21.

El proceso 1 es el que toma el recurso procesador, ejecuta lo que necesita y después se pasa al proceso 2. Si éste está en estado de bloqueado, automáticamente, se pasará al proceso 3, o sea se sigue estando dentro

de la lista. En el caso de Multicolas ocurre lo mismo, dentro de la Q1 va a haber procesos que van a estar en estado de bloqueado, por razones de E/S. La calificación de cada una de estas colas sería que los procesos que están en Q1 son altamente interactivos (tienen mucha actividad con la terminal) y los procesos que están en Q2 tienen un poco menos. Los de Q3 tienen poca actividad con la terminal (por ejemplo una compilación estaría en Q3).

Existe la posibilidad de salir de estas colas y entrar a la cola de Elegibles. Allí se encuentran procesos a los que les falta el recurso procesador, han excedido en mucho el uso del recurso procesador, o están ociosos. Antes de poder volver a pelearse con el resto de los procesos por el uso del procesador, tienen que pasar una especie de "prueba de fuego" que está relacionada con la memoria.

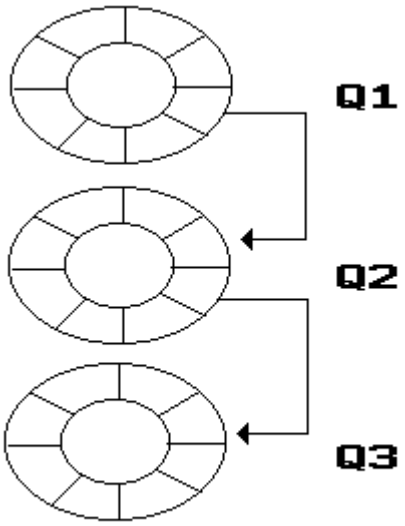


Fig. 12.21.

Existe una medida de memoria llamada **Working Set**, que para este sistema en particular está midiendo el número de páginas esperadas en memoria cuando éste proceso toma el recurso procesador para que trabaje. Sería la cantidad de páginas promedio que necesita para trabajar, cada vez que este proceso toma el control del procesador.

Para poder volver a competir para ocupar el procesador, cuando un proceso está calificado con un Working Set de **n**, no lo puede volver a hacer, mientras la cantidad de bloques de memoria real no sea mayor o igual que esta cantidad (n). Si un proceso está en elegibles porque usaba mucho procesador y está calificado con un working set de 12, no va a poder ingresar a ninguna de las colas a menos que la cantidad de páginas no sea menor o igual que la cantidad de bloques libres que hay en memoria.

Aquí se lo empieza a penalizar, no solo por haber usado mucho el recurso procesador, sino por haber estado utilizando demasiada cantidad de páginas.

El paso de una cola a otra es la siguiente :

	Quantum	Sale de cola Qi	Entra a cola Qi
Q1 Interactivo	8	Int. por reloj	Int. Terminal
Q2 No-Interactivo	64	Int .por reloj (*6)	sale de Q1
Q3 Pesados	512	Int.por reloj(*8)	sale de Q2

La cola 1 es interactiva (por definición). El quantum que se le da es de 8 unidades de tiempo y sale de la cola 1 cuando excedió ese quantum, o sea por interrupción de reloj. Pero lo importante es que se vuelve a entrar a la cola 1 por interrupción de terminal. No importa dónde se encuentre, va a volver siempre a la cola uno cada vez que exista una operación de E/S desde la terminal. O sea se están priorizando las operaciones por terminal.

La cola 2 es menos interactiva que Q1; el quantum es de 64 unidades de tiempo (generalmente se colocan como primer quantum un número cualquiera y luego múltiplos de éste porque dependen de cada modelo de máquina), y sale por interrupción de reloj, pero por 6, o sea seis veces ha sido interrumpido mientras estaba en la cola 2 por reloj. En Q1 cuando llega una sola interrupción por el quantum que teníamos, pasa a Q2. Pero en Q2 se le da bastante más tiempo y además soporta hasta 6 veces estas interrupciones, o sea permanece más tiempo en esa cola menos interactiva.

La Q3 tiene los procesos más pesados. Su quantum es de 512 unidades, interrupción por reloj, pero 8 veces, y entra porque sale de Q2. Entonces en el momento que excedió las 8 interrupciones de ráfagas de 512 cada una, es cuando tenemos la salida de cola y se pasa a la cola de Elegibles.

Si Q1 y Q2 están muy llenas, o sea están con muchos procesos, posiblemente los demás procesos que han caído a la cola 3 no entren "más" al recurso procesador. Es el problema del efecto pila, han quedado abajo y no toman nunca más el recurso procesador.

12.15 - Efecto Residual [S.O.]

El problema de los procesos que están en el resto de las colas, es que hay que encontrar un mecanismo para que vuelvan a tener derecho a tomar el recurso procesador, para evitar el efecto residual de los procesos que no lo tomarían nunca.

Para evitar ese defecto lo que se hace es en el momento en que el proceso abandona el recurso procesador porque fue interrumpido o por que el mismo decide abandonarlo, se realiza una cuenta de cuándo debería ser la próxima vez que debería tomar el recurso procesador. Cada cierta cantidad de tiempo se controla si existen procesos que se encuentran con una hora de próximo proceso muy atrasada. El cálculo, podría ser considerando el ejemplo anterior, si estuviera nada más que en Q1 :

$$\text{Hora próxima procesador} = \text{Hora actual} + (n-1)_{Q1} * \text{Quantum}_{Q1}$$

siendo $(n-1)_{Q1}$ los (n-1) procesos de la cola 1.

Si estuviera en la cola 2:

$$\text{Hora próxima procesador} = \text{Hora actual} + (n)_{Q1} * \text{Quantum}_{Q1} + (n-1)_{Q2} * \text{Quantum}_{Q2}$$

Si el proceso uno abandona el recurso procesador en un determinado instante, se toma la hora actual, se mira qué cantidad de procesos hay en esa cola y se lo multiplica por el quantum, o sea se lo multiplica por cuanto debería esperar una vuelta completa.

Puede suceder que existan procesos en Q2 y Q3 cuya hora de próximo uso de procesador ya haya pasado hace un rato, esos son procesos que quedaron en espera. Entonces para evitar el problema pila, el efecto residual, lo que se hace es examinar las colas y, cuando se detectan procesos que se encuentran en esta situación, se les asigna el procesador por más que haya procesos a los cuales les correspondan su uso en colas de más alta prioridad.

De la misma manera que se examinan las colas 1, 2 y 3, para detectar el problema de los procesos que quedan en forma residual, se hace exactamente lo mismo con los procesos que están en la cola de elegibles. Aquí el impedimento es su working set, y se usa una pequeña trampa que es descontar a ese working set del proceso un determinado porcentaje (puede ser 10%). Cada vez que se lo examine y se determine que hace rato que está ahí, la próxima vez se le descontará un 10%, hasta que realmente esa cifra pueda llegar prácticamente a cero y pueda tomar el recurso procesador.

Cuando se pasa un proceso a la cola de elegibles no se calcula más la hora próxima de procesador, pues de allí se extraen con el cálculo del Working Set.

Para que este sistema funcione bien, toda la actividad del SO. (incluida la paginación, las administraciones de otras colas, las operaciones de E/S, etc.) no debe exceder su permanencia en el procesador más del 5% de su tiempo total. Si se excede comienzan los problemas.

12.16 - SEMAFOROS [9][44]

Evidentemente, si existe más de un proceso ejecutando en una arquitectura SIMD o MIMD que de distintas maneras pueden llegar a acceder al mismo lugar de memoria se genera un problema de administración de la concurrencia. Entendiendo como concurrencia el intento de acceso simultáneo a una misma variable, un mismo dato, una misma dirección de memoria.

Para evitar el problema de que se cambie la información de una manera no deseada, lo que se debe hacer es establecer un sistema de protección para posibilitar que esta concurrencia se dé, pero en forma ordenada. O sea que accedan en orden, de a uno por vez.

La forma de realizar esto, es la de colocar algún tipo de señal, y que esa señal sea respetada, de tal manera que si dice que un proceso no puede acceder a un determinado lugar de memoria, no lo haga.

Pero además esa señal debe comportarse de tal manera que una vez que un proceso ha hecho uso de la zona restringida permita el acceso de un nuevo proceso a la misma. O sea que se busca protección y sincronización entre procesos.

Lo mejor que se puede hacer es colocar algún tipo de bandera, de flag; algo que se denomina, normalmente, semáforo.

Definición : Semáforo es un conjunto de bits que va a indicar un determinado estado que indica si se puede o no acceder a un determinado dato, o a una determinada posición de memoria, o si se puede realizar algún tipo de acción.

Habría dos acciones para manejarse con un semáforo. Una sería cerrar el semáforo porque se va a acceder a ese lugar, y una vez que se accede, liberarlo, o sea ponerlo en verde para que pueda acceder algún otro proceso.

La forma de hacerlo la llamaremos cierre y escribiremos

CIERRE (X)

para indicar que se cierra el semáforo X. Obviamente, si lo que se quiere hacer es controlar que no se está intentando acceder a un lugar que no corresponde, en ese momento lo que habrá que hacer es mirar ese valor de X, y lo notaremos :

EXAMINAR(X)

Establecemos la siguiente convención : si X es igual a 0, es de libre acceso; y si X es distinto de 0, significa que está ocupado.

Entonces se examina X y si X es igual a 0, se toma y se realiza la acción; si X es distinto de 0, se debe esperar. La liberación, APERTURA(X), sería, sencillamente, asignarle a X una valor igual a 0.

Sea el siguiente programa pequeño de la rutina CIERRE(X):

```
1:  If X not = 0   Then  LOOP(X)
                    2:  X <<--- 1
                    3:  Tomar el recurso
```

Examinar X será preguntar por el valor de X, o sea preguntar si X es igual a 0. Si resulta afirmativamente, lo que se hace es asignarle a X un valor cualquiera distinto de 0, o sea, ocuparlo y luego vendrá la acción de tomar el recurso. Y si X no vale 0, la acción a tomar será esperar, es decir examinar nuevamente el valor de X hasta que cambie.

Esto tiene un par de problemas. El primero de ellos es que, si una vez que, ante la pregunta, se sale por X = 0, y en ese momento llega una interrupción y se quedó con el estado de X igual a 0, cuando vuelva a ejecutarse

ya no se pregunta nuevamente, se pone en 1 y se lo toma. Pero no se sabe lo que sucedió en el ínterin. Podría haber venido otra rutina equivalente, preguntar si X estaba en 0 (a lo que respondería que sí ya que no se llegó a actualizar previamente) le colocaría un 1 y tomaría el recurso, produciéndose luego la doble asignación. Luego, como sistema de protección no es eficiente.

El otro caso es que si se queda constantemente preguntando por el semáforo significa que se está ejecutando una instrucción, o sea se está utilizando procesador inútilmente ya que no se está haciendo nada productivo. Y de hecho, si estuviera en un sistema de monoprocesamiento hasta que no existiera una interrupción, se quedaría eternamente preguntando por el valor del semáforo.

Atacando el primer problema, que es el asunto de la interrupción, se sabe que las instrucciones no son interrumpibles por la mitad. Entonces lo que habría que lograr es que estas dos acciones, o sea la pregunta y la asignación del semáforo, se tradujesen en una sola instrucción. O sea generar una nueva instrucción que, simultáneamente haga el cambio del valor del semáforo, y testee el valor que tenía.

Para esto existe una instrucción que se llama **TEST-AND-SET (TS**).

Nótese aquí que lo que se hace es cambiar el hardware, que haga la observación de cuánto es el valor del semáforo, y además asigne el valor de ocupación.

Esta instrucción TEST-AND-SET se podría traducir de la siguiente manera :

```
CRB1   5,X          (Cargar registro base número 5 con valor X)
COMPR 5,0           (Comparar el contenido del registro 5 con valor 0)
```

El funcionamiento sería cargar en el registro 5 el valor que tiene el semáforo X y ponerle en ese lugar un 1.

Con esta instrucción, ya no importa lo que suceda después, pues si luego llegara una interrupción, una de las primeras cosas que se harían sería salvar los registros, y por ende el registro 5 en particular, y además, ya también se cerró el semáforo sin todavía saber cuál era el valor del mismo.

Luego, lo que se hará será comparar el registro 5 contra el valor 0. Si el semáforo valía 0, en el momento que se ejecutó la primera instrucción, ya vale 1; y se entera cuál era el valor anterior del semáforo en el momento que testee el registro 5.

Si el semáforo valía 1, o sea que ya estaba cerrado, se le volvió a escribir un 1, o sea que no cambió su estado. Y tampoco importa qué es lo que va a suceder después porque, en realidad, se va a testear ese 1 en el registro 5. O sea que de esta manera se esta asegurando que las interrupciones no afectarán el resultado de ese fragmento de código.

El otro problema era que se quedaba ciclando, preguntando por el semáforo, y se consumía mucho tiempo ejecutando esa instrucción que, hasta que no venga una interrupción no se podía abandonar (si estamos en monoprocesamiento).

Para eso, lo que se aplica es lo siguiente : dada esa situación, poder pasar a un estado de espera; donde ese estado de espera implica que se encuentran esperando la apertura de ese semáforo. Se puede usar una instrucción WAIT(X), que lo que haría sería que poner en una cola asociada al semáforo X a la rutina que se quedó esperando la apertura de ese semáforo. Cola que tiene una estructura muy sencilla.

```
X ------------>> Procesos en espera de apertura de X
                 .....
                 .....
```

En el momento en que se hace la apertura de ese semáforo, habría que examinar esta cola para saber si realmente hay procesos que estén esperando la apertura de ese semáforo. Para eso existe otra primitiva que se denomina SIGNAL(X), que toma al primer proceso que se encuentra a la espera de ese semáforo y lo manda a la cola de listos. O sea que hace la función de despertar procesos.

Entonces, las rutinas de cierre y de apertura quedarán de la siguiente forma :

```
                CIERRE(X)                       APERTURA(X)
EXAM:  T.S.(X)                                  X = 0
       If X = Ocupado  then WAIT(X)             SIGNAL(X)
       Go to EXAM
       Tomo recurso
```

Pero, porqué no inhibir todas las interrupciones, mientras se realiza esta tarea ? El utilizar el mecanismo de inhibir todas las interrupciones en lugar de utilizar la instrucción Test-and-Set tiene sus desventajas. Supongamos que se están sincronizando procesos, básicamente esos procesos van a tener una zona crítica, que es donde no se desea que ocurran problemas de concurrencia, y seguramente, después, otra zona que deja de ser crítica nuevamente. La extensión de esto pueden ser muchísimas cosas, por ejemplo, como retirar información de buffers , etc. Luego, si se inhiben todo tipo de interrupción, se ejecuta sin problemas, pero se estarían perdiendo eventos que están sucediendo.

Existe un caso en el cual tiene bastante sentido que el procesador se quede ciclando sobre un semáforo, y no que se produzca un Wait. Este es el caso en el cual se tiene, por ejemplo, dos procesadores que quieren acceder a la cola de listos para seleccionar el próximo proceso a ejecutar. En un momento dado está uno de ellos seleccionando por tanto se está ejecutando el planificador de procesos, y en ese mismo instante, también en el otro procesador se quiere ejecutar el planificador de procesos.

Uno de los procesadores justamente tiene que ir a la cola de listos para seleccionar el proceso que tiene que ejecutar. Pero el otro está haciendo la selección y tiene el recurso (cola de listos) tomado. Si no puede elegir, no tiene sentido que se espere nada, porque lo que quiere hacer es justamente tomar un proceso para continuar la ejecución. Este sería un caso en el cual uno de los procesadores se quedará loopeando en la pregunta esta de ver cuándo cambia el valor del semáforo. Esos semáforos reciben el nombre de SPIN.

12.16.1 - Semáforos Contadores [9][44]

Dijkstra ideó lo que se denomina semáforos contadores, con el siguiente concepto: los semáforos pueden servir también para contar la cantidad de procesos que hay en espera de un recurso. Además con la idea de ir después contando algunas otras cosas.

Para ello, inventó dos operadores, para trabajar sobre los semáforos. Uno de ellos es el operador P, cuya función es la siguiente :

x = x - 1

si x $<$ 0, se espera.

Esto considerando un valor inicial de x = 1. El otro operador es el operador V, que correspondería a la apertura del semáforo, que hace :

x = x + 1

si x $\leq$ 0, despierta.

Por supuesto que la espera lo podemos transformar en un WAIT (X),y el despierta en un SIGNAL(X).

P(X)	V(X)	Valor inicial x = 1
x = x - 1	x = x + 1	
If x $<$ 0 WAIT(X)	If x $\leq$ 0 SIGNAL(x)	

Entonces, si se tuviera un buffer, sobre el cual alguien va a poner información, que llamaremos emisor, y algún otro que llamaremos receptor, retira información, se puede controlar, si el lugar en el cual se desea colocar información está libre y además, se puede controlar si existe la información que se desea. O sea se controla y se sincroniza la actividad de ambos procesos (Caso del Productor-Consumidor).

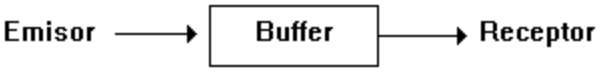


Fig. 12.22 - Caso del Productor-Consumidor

La idea es la siguiente : no se quiere que el emisor escriba antes que el receptor haya retirado la información, y tampoco que el receptor retire información que el emisor todavía no escribió.

La rutina que emite, podría tener un semáforo que controlara el espacio. Una vez que colocó la información, la cierra, es decir no se puede colocar más información ahí, y abrir un semáforo que habilitaría la extracción de información. Con lo cual podría realizar lo siguiente:

Mensaje = 0	EMISOR	RECEPTOR
Espacio = 1	P(Espacio)	P(Mensaje)
	.	.
	Coloca inf.	Retira
	en el Buffer	.
	.	.
	V(Mensaje)	V(Espacio)

El receptor, tendría que preguntar si ya existe el mensaje. En el caso en que existiera, retirarlo y volver a habilitar el espacio.

Si se piensa un ejemplo en el que se deposite información en un vector (Vec(i)) de dimensión 3 se puede tener un par de variables, i=0, j=0, que van diciendo a cada uno de los módulos, el emisor y el receptor, poner la información que es remitida. La idea es no tapar información que no ha sido retirada, no retirar información que no existe y llegar a la situación de acceder en forma exclusiva a esta zona de memoria.

La rutina de emisión es la siguiente :

E	Mensaje = 0
P(Espacio)	Espacio = 3
VEC(i) = M	
i = (i+1) módulo 3	
V(Mensaje)	

en VEC(i) se está poniendo el mensaje del emisor, después se incrementa i con i+1 pero con módulo 3 para no superar la cantidad total de elementos que contiene el vector. Y que después se habilita la posibilidad de retirar un mensaje.

El receptor hace la operación contraria :

R
P(Mensaje)
X = VEC(j)
j = (j+1) módulo 3

V(Espacio)

toma de alguna manera la información desde VEC(j), luego lo incrementa en uno (todo esto también módulo 3) y avisa de un espacio nuevo para seguir colocando información.

Y esto funciona muy bien mientras no exista la posibilidad de que ocurra una interrupción y nuevamente antes de poder actualizar el subíndice, con el cual se está recorriendo el vector, una nueva tarea quisiera utilizar este procedimiento de emisión.

Luego lo que se hace para solucionar esto es colocarle un semáforo exclusivo (Exclu), cuyo valor inicial es 1, y después se lo habilita. Con el emisor se hace lo mismo.

E
P(Exclu)
P(Espacio)
VEC(i) = M
i = (i+1) módulo 3
V(Mensaje)
V(Exclu)

R
P(Exclu)
P(Mensaje)
X = VEC(j)
j = (j+1) módulo 3
V(Espacio)
V(Exclu)

Hasta aquí todo va bien, pero si los valores fuesen :

Espacio = 0 Mensaje = 3 Exclu = 1

significa que hay información en todo el buffer y no se puede seguir colocando porque sino se taparía la que existe, el receptor no está retirando información, y por ende se produce un problema.

Si E hace un P(Exclu), o sea Exclu que en ese momento permite la entrada (vale 1), pero luego de realizar P(Exclu) , éste queda con valor 0 y cuando hace P(Espacio), se da cuenta de que no puede entrar (está en -1). Si R hace P(Exclu) estando E interrumpido a la altura que corresponde entre P(Espacio) y VEC(i) = M y el receptor intentando retirar información, hace que con su P(Exclu), Exclu tome valor (-1) y se interrumpe ahí, nos encontramos ante la situación de que el emisor (E) está esperando que alguien retire información mientras el receptor está esperando tener la exclusividad de acceso para poder retirar la información

Lo que hemos armado es un **abrazo mortal**, y allí quedó trabado el sistema de sincronización.

El verdadero problema de esto es que en realidad se está exigiendo una condición, la de exclusividad, previa a la de conocer si realmente corresponde entrar a la zona exclusiva, o sea se está pidiendo exclusividad de algo cuando aún no sé sabe si puede utilizarse.

Con el pequeño cambio de preguntar primero si corresponde entrar y después preguntar si se tiene el recurso en forma exclusiva, se ha arreglado el problema, o sea lo que se hace es :

E
P(Espacio)
P(Exclu)
VEC(i) = M
i = (i+1) módulo 3
V(Exclu)
V(Mensaje)

y para el receptor se hace lo mismo, primero se pregunta si puede entrar, y después pregunta si lo tiene exclusivo, y cuando se va hace exactamente lo mismo

R
P(Mensaje)
P(Exclu)
X = VEC(j)
j = (j+1) módulo 3
V(Exclu)
V(Espacio)

Y ahora no existe el problema anterior porque lo primero que se hace es preguntar si realmente corresponde entrar, y una vez que se puede, se pide con exclusividad el uso del recurso. Entonces al implementar el orden correcto, alcanza con un solo semáforo.

El Exclu sirve para dejar una zona de exclusividad y para escribir o retirar algo. La razón de entrar en exclusividad es porque si un proceso se interrumpe en el medio y a continuación ingresa otro que realiza una actividad del mismo estilo podría destruir información que tenía el primer proceso.

Este tipo de problema es la razón por la cual los semáforos en algunos textos son bastante criticados, y se proponen algunos otros tipos de solución.

Una solución son los monitores que es como tener una especie de programas de servicios que se encarguen de realizar la tarea de exclusividad y sincronización.

12.16.2 - Productor-Consumidor (Implementado con Stack) [S.O.]

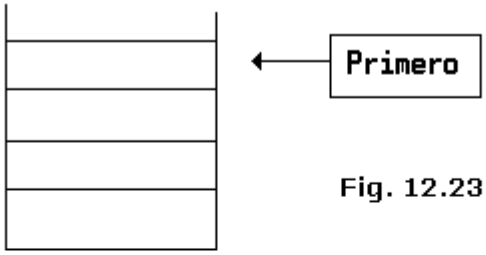
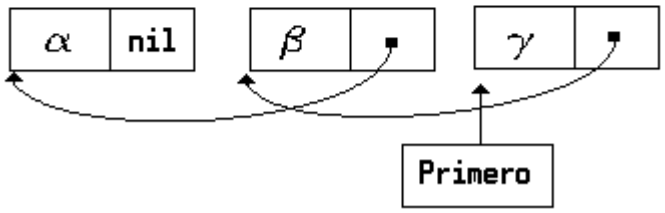


Fig. 12.23.

Veamos otro ejemplo de Productor-Consumidor, pero implementado con stack, donde se puede apreciar mucho mejor la necesidad del semáforo de exclusión.

Supóngase que se desea una cantidad máxima de mensajes (que aquí llamaremos MAX) y que se cuenta para ello con una estructura encadenada (ver Fig. 12.23) y con la siguiente información:



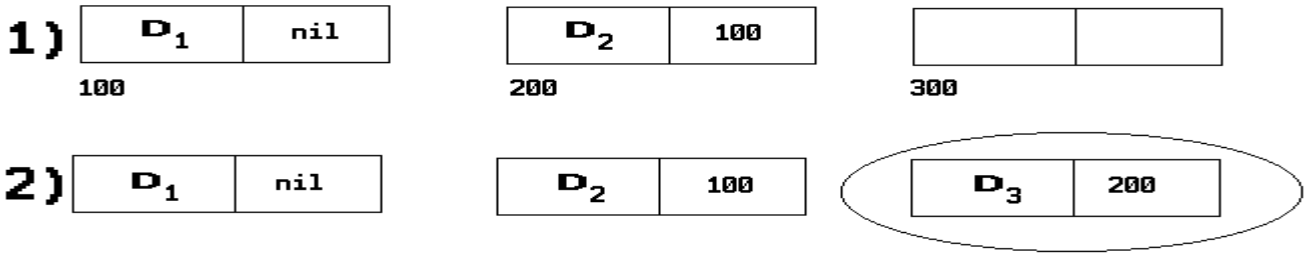
Primero (apuntador) = Nil (apunta al último mensaje generado o primero a consumir),
P(apuntador) = dirección del mensaje que se está produciendo,
C (apuntador) = dirección del mensaje que se está consumiendo,
E (semáforo) = MAX = 3 (cuenta el máximo de mensajes producibles)
S (semáforo) = 0 (cuenta el número de mensajes disponibles para consumo)
X (semáforo de exclusión) = 1

Las rutinas serán :

	A (Productor)		B (Consumidor)
	P(E)		P(S)
	P(X)		P(X)
[1]	P = genera apuntador	[2]	C = Primero
	P.Mensaje = Dato		Primero = C Próximo
	P. Próx = Primero		Dato = C.Mensaje
[3]	Primero = P		Libera C
	V(X)		V(X)
	V(S)		V(E)

Ambas rutinas cuentan con el semáforo de exclusión X.

Supóngase momentáneamente que no se tiene el semáforo X, y que los valores de las variables luego de emitidos dos mensajes son los indicados en la Fig. 12.24 situación 1).



	Valores iniciales	1) A	B	A	B	A	Valores finales
Primero	nil	100	200		300	100	100
E	3	2	1	0		1	1
S	0	1	2		1	2	2
P	nil	100	200	300			
C	nil				200		

Fig. 12.24.

2)

La rutina A comienza a producir el tercer mensaje y llega a ejecutar hasta la instrucción [1] luego de lo cual es interrumpida. Poco tiempo después comienza a ejecutarse la rutina B la cual llega a ejecutar hasta la instrucción [2] inclusive antes de ser interrumpida también.

Retoma la rutina A que ejecuta hasta la instrucción [3] inclusive, continúa B y luego se completa A.

Nótese que se produjo el mensaje de la dirección 300 y se consumió el mensaje almacenado en la dirección 200, sin embargo el valor del apuntador al próximo mensaje a consumir (Primero) indica que se debe consumir el mensaje de la dirección 100. Aún cuando los contadores de mensajes indican que existían 2 mensajes para consumir se ha perdido el mensaje que se almacenó en la dirección 300.

Luego se perdieron datos a causa de la falta del semáforo de exclusión.

La implementación de semáforos es posible, y sería muy bueno que su ejecución fuera atómica, es decir que un P y un V sean directamente una instrucción. Tener implementado desde hardware una instrucción "operador P" y otra "operador V", y sencillamente cuando se les indicase sobre cuál semáforo trabajar sabrían sobre qué zona de memoria actuar.

La implementación más usual es que esto se transforme en una llamada al S.O., con lo cual se hace una interrupción, indicando que se quiere usar la rutina que se llama P y la rutina que se llama V.

Obviamente, una forma de implementarlo sería en el momento que se lo va a usar inhibir todo tipo de interrupción, eso generalmente trae problemas porque : primero se debe estar completamente seguro de que no se podrá producir un loop en lo que se hará, y segundo no siempre se pueden inhibir todo tipo de interrupción; si, por ejemplo, se tiene un sistema de control de procesos no se pueden inhibir todas las interrupciones, porque posiblemente alguna de ellas sean muy importantes.

12.16.3 - Ejemplo de administración de Procesador con semáforos [S.O.]

Veremos un caso en el cual se visualizará cómo se va desarrollando el tema de ir pidiendo recursos que estén protegidos por lo menos por un semáforo, el esquema de sistema va a ser el siguiente : existe una zona donde se reciben las interrupciones

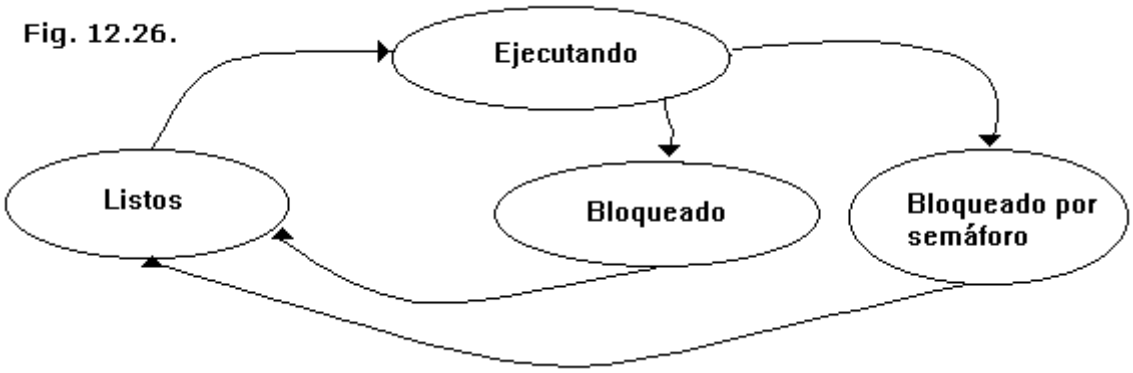
Por ejemplo el primero es el nivel de fin de E/S, otro será el nivel de incidente de programa, el siguiente el de llamada al supervisor y luego el nivel de interrupción por reloj, si es que existe, o cualquier otro tipo de interrupción que se tenga.

El núcleo de este sistema será el siguiente, todos los programas van a poder estar en los siguientes estados :

- un estado de ejecución,
- un estado de listos,
- un estado de bloqueados y
- un estado más que será bloqueado en espera de semáforo.

Fig. 12.25.

Fig. 12.26.



En este sistema supóngase un recurso único, y ese recurso está controlado por una rutina de asignación (rutina S). Esa rutina de asignación lo que hace es ejecutar un operador P sobre un determinado semáforo que llamamos X:

Rutina de asignación :
S(asignación)
P(X)
Asigna

donde tenemos que :

P(X)
X = X - 1
If X < 0 Wait (X)

Y la rutina de liberación será :

Desasignar
V(X)

donde el operador V(X) será :

V(X)
X = X + 1
If X $\leq$ 0 Signal(X)

Veamos qué sucede con este sistema en el cual en un momento determinado se tienen los siguientes estados : en ejecución, una cola de listos, bloqueados y bloqueados por semáforo, y supóngase que hay un proceso P1 que está ejecutando y en la cola de listos hay otros dos P2 y P3. La administración de la cola de listos es por lo menos FIFO, y en caso de querer una administración del procesador que también fuera FIFO, entonces se elimina la interrupción por reloj (ver Fig. 12.27).

El proceso P1 está ejecutando, y en un momento dado hace una llamada al supervisor para solicitar el recurso S, mientras tanto los procesos P2 y P3 están en la cola de listos. Al realizarse la llamada al supervisor, hay una interrupción, un intercambio de la palabra de control, la PC del programa P1 va a ir a la izquierda de la zona que tenemos asignada como llamadas al supervisor y se tomará la palabra de control que está a la derecha, que es la que corresponde a la rutina que atiende las llamadas al supervisor. En conclusión lo que se tiene aquí es la rutina de atención al supervisor, mientras que tenemos a P2 y a P3 en la cola de listos.

P1 tiene suspendida su ejecución, o sea su palabra de control está guardada. Esta rutina de atención del supervisor se tiene que dar cuenta de dos cosas:

1) Le han pedido el recurso
2) P1 que pidió el recurso S tiene que pasar a bloqueado, o va a estar esperando ese recurso.

Entonces la rutina hará dos cosas:

1) Va a preparar información sobre P1 que va a pasar a bloqueado.
2) Tiene que preparar información que genera en este momento una rutina pidiendo el recurso S para el programa 1 (Rut(S,P1), que es una nueva tarea.

El encargado en un sistema de administración de procesador de colocar los nuevos estados para un programa que se ha generado y otro que está siendo suspendido es el controlador de tráfico (CT). Entonces se va a llamar con estos datos al controlador de tráfico, todavía P2 y P3 están en la cola de listos.

El controlador de tráfico generará las dos cosas que le han pedido que haga

- poner en estado de bloqueado al P1 y
- colocar dentro de la cola de listos a la rutina que pide el recurso S para el P1.

Luego de hacer esto, el controlador de tráfico terminó su tarea, entonces se tiene que seleccionar cuál es el próximo proceso que tiene que tomar el recurso procesador. Como conclusión el planificador de procesos es el que entra a funcionar y siguen los mismos procesos en la cola de listos.

Supóngase que la cola de listos es FIFO y no da ningún tipo de prioridad a la rutina S, el PP lo que hace es seleccionar algún proceso de la cola de listos por medio de algún algoritmo, entonces selecciona a P2, y en la cola quedarán P3 y la Rut(S,P1), y en bloqueado sigue quedando P1.

Supóngase que en el ínterin P2 hace cosas. No importa qué es lo que hace. Pero en algún momento aparece el planificador de procesos que elige a P3 para que se ejecute, entonces ahora queda la Rut(S,P1) en la cola de listos y en bloqueado quedan P1 y P2.

Ahora supóngase que P3 pide exactamente lo mismo que P1, o sea que P3 hace la llamada al Supervisor, pidiendo el recurso S, nuevamente se tendrá una rutina que atiende la llamada (At.Sup.), cuando sucede esto ahora en la zona de interrupciones no se va a tener más la palabra de control de P1, sino que se tendrá la de P3.

Ahora es P3 quien tiene que pasar al estado de bloqueado, el sistema tiene que armar una rutina para el recurso S pero ahora para el proceso P3, y nuevamente se llama al controlador de tráfico. Mientras tanto en la cola de listos y en bloqueados siguen los mismos estados.

Una vez que se llama al controlador de tráfico este va a poner en la cola de listos Rut(S.P3) y pondrá en el estado de bloqueado a P3.

Después va a llamar otra vez al planificador de procesos y éste va a elegir de acuerdo al orden que tenían las cosas a la rutina que pide S para P1 (Rut(S,P1)), en la cola de listos tenemos la Rut(S,P3) y tenemos bloqueados a P1,P2 y a P3.

Esta rutina de alguna manera invoca, o tiene dentro de sí el pedido de asignación del recurso S, durante esa situación se va a hacer P(X), luego X = X - 1, si X<<0 se pasa a un Wait, digamos que inicialmente el valor del semáforo es 1, por lo tanto X va a valer 0 y se asigna el recurso al proceso 1.

Asignárselo al P1 significa que ahora el proceso lo puede usar. Supóngase que lo que está pidiendo es una porción de memoria, si está listo para usarla, hay que hacer ciertas cosas como sacarlo del estado de bloqueado en que se encuentra y pasarlo a la cola de listos para que la pueda usar. Entonces se tiene que llamar al controlador de tráfico para el proceso P1, informándole que lo saque del estado de bloqueado. En conclusión, se tendrá a la rutina que pide S para P3, luego, se tiene en este momento el P1 que acaba de salir del estado de bloqueado y pasa a la cola de listos y P2 y P3 continúan bloqueados.

Cuando termine el controlador de tráfico de ordenar todas sus listas, llama otra vez al Planificador de procesos, y el planificador de procesos seleccionará la rutina que pide el recurso S para P3, en la cola de listos estará P1 y en bloqueados P2 y P3.

Entonces se invocará nuevamente la rutina de asignación, va a quedar en (-1) el valor del semáforo y pasará a un WAIT, y como ya se ha dicho pasar a un WAIT es pasar al estado de bloqueado por semáforo, enton-

ces la primitiva WAIT coloca en el estado de bloqueado del semáforo X a esa rutina del controlador de tráfico, siempre es el controlador de tráfico el que toca las tablas, y la situación queda con que P1 está listo, P2 y P3 también bloqueados y se agrega en bloqueado por semáforo a Rut(S,P3).

En realidad a pesar de que el recurso no estaba siendo usado, no se lo otorgó porque había sido asignado a otro proceso.

Luego seguirá el planificador de procesos como corresponde y finalmente P1 hará uso de su recurso. El resto es trivial.

La rutina S al asignar, le da el puntero. Si fuera un buffer de memoria entonces le dice cuál es la zona de memoria que le está asignando, le da la dirección del lugar donde puede escribir, por ejemplo. En el caso de operaciones de E/S que asigna el canal significa que está poniendo al proceso en la posición necesaria para que lo use.

EJECUCION	**LISTOS**	**BLOQUEADOS**	**BLOQ. POR SEMAFORO**
P1	P2 P3		
P1 (Sup S)	P2 P3		
Atenc. sup.	P2 P3		
(P1 Bloq)			
Rut (S,P1)			
Llamar CT			
CT	P2 P3		
CT	P2 P3 Rut(S,P1)	P1	
PP	P2 P3 Rut(S,P1)	P1	
P2	P3 Rut(S,P1)	P1	
.....	.....	.....	
PP		P1 P2	
P3	Rut(S,P1)	P1 P2	
P3 (Sup S)	Rut(S,P1)	P1 P2	
At. Sup.	Rut(S,P1)	P1 P2	
(P3 Bloq)	Rut(S,P1)	P1 P2	
Rut(S,P3)	Rut(S,P1)	P1 P2	
Llamar CT	Rut(S,P1)	P1 P2	
CT	Rut(S,P1) Rut(S,P3)	P1 P2 P3	
PP			
Rut(S,P1)	Rut(S,P3)	P1 P2 P3	
CT(P1)	Rut(S,P3) P1	P2 P3	
PP			
Rut(S,P3)	P1	P2 P3	
CT	P1	P2 P3	Rut(S,P3)
PP			
P1		P2 P3	Rut(S,P3)

Fig. 12.27.