# Administración de Procesos (versión beta)

La actividad más importante del núcleo del sistema operativo es implementar los procesos. Cada proceso es un procesador virtual en donde se ejecuta una aplicación o una herramienta del sistema operativo. El núcleo debe encargarse entonces de administrar los recursos del *hardware* del computador para que sean asignados convenientemente a los procesos.

En este capítulo examinaremos cómo el núcleo asigna el o los procesadores reales a los procesos. En una primera etapa supondremos que el hardware sólo ofrece un procesador real. Más tarde generalizaremos a varios procesadores.

## Scheduling de Procesos

En la mayoría de los sistemas computacionales existe un solo procesador real. Por lo tanto el núcleo debe asignar el procesador por turnos a los numerosos procesos que pueden estar activos. Hay distintas estrategias para asignar estos turnos, dependiendo del objetivo que se persiga.

Por ejemplo en un sistema de multiprogramación se busca maximizar el tiempo de uso del procesador, mientras que en un sistema de tiempo compartido se busca atender en forma expedita a muchos usuarios que trabajan interactivamente.

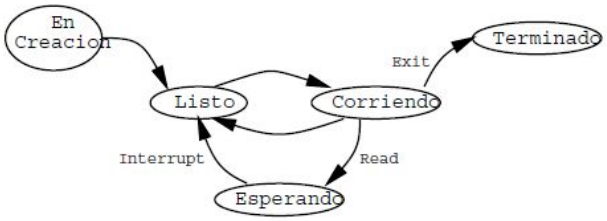
La asignación *estratégica* del procesador a los procesos es lo que se denomina *scheduling de procesos*. Es estratégica porque se intenta lograr algún objetivo particular como alguno de los que se mencionó más arriba y para ello se usan estrategias que pueden funcionar muy bien en determinados sistemas, pero muy mal en otros. La componente del núcleo que se encarga de esta labor se denomina *scheduler* del procesador.

Además de scheduling de procesos, también se realiza scheduling de los accesos a disco y para ello existe un scheduler asociado a cada disco. Este scheduler ordena estratégicamente los mútiples accesos a disco de varios procesos con el fin de minimizar el tiempo de acceso.

Para abreviar la notación, cuando se hable de ``el scheduler´´ se subentenderá que se trata del scheduler del procesador.

### Estados de un proceso

Mientras un proceso se ejecuta puede pasar por distintos estados. Estos estados se aprecian en la siguiente figura:




- **En creación**: el núcleo está obteniendo los recursos que necesita el proceso para poder correr, como por ejemplo memoria o disco.
- **Corriendo** (RUN): El proceso está en poseción del procesador, el que ejecuta sus instrucciones.
- **Esperando** (WAIT): El proceso espera que se lea un sector del disco, que llegue un mensaje de otro proceso, que transcurra un intervalo de tiempo, que termine otro proceso, etc.
- **Listo** (READY): El proceso está activo pero no está en poseción del procesador.
- **Terminado**: El proceso terminó su ejecución, pero sigue existiendo para que otros procesos puedan determinar que terminó.

Un proceso pasa de un estado a otro constantemente y varias veces por segundo. Por ejemplo cuando un proceso está **corriendo** el scheduler puede quitarle el procesador para entregárselo a otro proceso. En este caso el primer proceso queda **listo** para ejecutarse. Es decir en cualquier momento, el scheduler puede entregarle nuevamente el procesador y quedar **corriendo**. En este estado el proceso puede leer del terminal y por lo tando quedar en **espera** de que el usuario ingrese algún texto.

El número exacto de estados depende del sistema. Por ejemplo en nSystem un proceso puede pasar por los siguientes estados:

- READY: Elegible por el scheduler o corriendo (también incluye el estado RUN).
- ZOMBIE: La tarea llamó `nExitTask` y espera `nWaitTask`. Este es equivalente al estado terminado.
- WAIT_TASK: La tarea espera el final de otra tarea.
- WAIT_REPLY: La tarea hizo `nSend` y espera `nReply`.
- WAIT_SEND: La tarea hizo `nReceive` y espera `nSend`.
- WAIT_READ: La tarea está bloqueada en un `read`.
- WAIT_WRITE: La tarea está bloqueada en un `write`.
- otros.

## El descriptor de proceso

El descriptor de proceso es la estructura de datos que utiliza el núcleo para mantener toda la información asociada a un proceso. Ella contiene:

- El estado del proceso: En creación, listo, corriendo, etc.
- Registros del procesador: Aquí se guardan los valores del contador de programa, del puntero a la pila y de los registros del procesador real cuando el proceso abandonó el estado **corriendo**.
- Información para el scheduler: Por ejemplo la prioridad del proceso, punteros de enlace cuando el descriptor se encuentra en alguna cola, etc.
- Asignación de recursos: Memoria asignada, archivos abiertos, espacio de *swapping*, etc.
- Contabilización de uso de recurso: Tiempo de procesador consumido y otros.

El núcleo posee algún mecanismo para obtener el descriptor de proceso a partir del *identificador del proceso*, que es el que conocen los procesos.

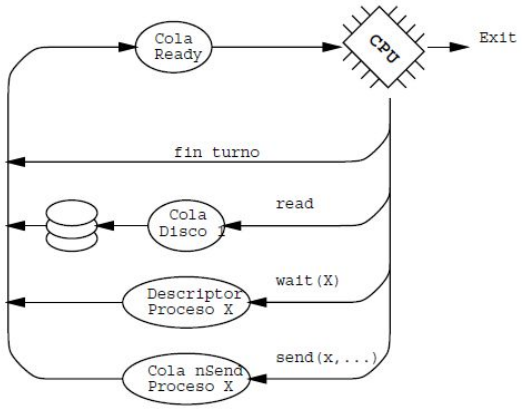
La siguiente tabla muestra el descriptor que usa nSystem. En nSystem el identificador de tarea es un puntero a este descriptor. Sin embargo, por razones de encapsulación este puntero se declara del tipo `void *` en `nSystem.h`.

```
typedef struct Task { /* Descriptor de una tarea */
  int status;       /* Estado de la tarea (READY, ZOMBIE ...) */
  char *taskname;   /* Util para hacer debugging */

  SP sp;            /* El stack pointer cuando esta suspendida */
  SP stack;         /* El stack */

  struct Task *next_task;   /* Se usa cuando esta en una cola */
  void *queue;              /* Debugging */

  /* Para el nExitTask y nWaitTask */
  int  rc;                  /* codigo de retorno de la tarea  */
  struct Task *wait_task;   /* La tarea que espera un nExitTask */
```

```
  /* Para nSend, nReceive y nReply */
  struct Queue *send_queue; /* cola de emisores en espera de esta tarea */
  union { void *msg; int rc; } send; /* sirve para intercambio de info */
  int wake_time;            /* Tiempo maximo de espera de un nReceive */
}
  *nTask;
```

## Colas de Scheduling

Durante la ejecución, un proceso pasa por numerosas colas a la espera de algún evento. Esto se puede observar en la siguiente figura:




Mientras un proceso espera la obtención del procesador, este proceso permanece en una cola del scheduler del procesador. Esta cola puede estar organizada en una lista enlazada simple (cada descriptor de proceso tiene un puntero al próximo descriptor en la cola).

Además de la cola del scheduler del procesador existen la colas de scheduling de disco. El proceso permanece en estas colas cuando realiza E/S.

Cuando un proceso envía un mensaje síncrono a otro proceso que no está preparado para recibirlo, el primer proceso queda en una cola de recepción de mensajes en el descriptor del proceso receptor.

Por último el proceso puede quedar en una cola del reloj regresivo, a la espera de que transcurra un instante de tiempo.

## Cambio de contexto

Cambio de contexto (*context switch*, *task switch* o *process switch*) es la acción que efectúa el scheduler cuando transfiere el procesador de un proceso a otro. Para realizar el cambio de contexto, el scheduler debe realizar diversas labores que detallamos a continuación:

- Resguardar/restaurar Registros.

  Cuando un proceso está **corriendo**, se utiliza el contador de programa, el puntero a la pila y los registros del procesador real. Al hacer el cambio de contexto, el nuevo proceso que toma el control del procesador usará esos mismos registros. Por ello es necesario resguardar los registros del proceso saliente en su descriptor de proceso. De igual forma, el descriptor del proceso entrante mantiene los valores que contenían los registros en el momento en que perdió el procesador. Estos valores deben ser restaurados en los registros reales para que el proceso funcione correctamente.
- Cambiar espacio de direcciones virtuales.

  El espacio de direcciones del proceso saliente no es el mismo del espacio de direcciones del

proceso entrante. En el capítulo sobre administración de memoria veremos como se logra el cambio de espacio de direcciones virtuales.

- Contabilización de uso de procesador.

  Usualmente, el núcleo se encarga de contabilizar el tiempo de procesador que utiliza cada proceso. Una variable en el descriptor de proceso indica el tiempo acumulado. En cada cambio de contexto el núcleo consulta un cronómetro que ofrece el hardware de cada computador. El scheduler suma el tiempo transcurrido desde el último cambio contexto al tiempo acumulado del proceso saliente.

Estas labores del scheduler consumen algo de tiempo de procesador y por lo tanto son sobrecosto puro. Cada cambio de contexto puede significar unos pocos microsegundos en los procesadores de hoy en día, pero podía llegar a ser de hasta un milisegundo en los años 80 y por eso se consideraba crítico evitar los cambios de contexto que no fuesen imprescindibles.

Normalmente la componente más cara en tiempo de procesador es la del cambio de espacio de direcciones virtuales. Ahí radica el origen del nombre de procesos livianos o pesados. Como los procesos livianos comparten el mismo espacio de direcciones no es necesario cambiarlo durante un cambio de contexto. Por lo tanto pueden ser implementados más eficientemente. En este caso el peso de un proceso corresponde al tiempo de procesador que se consume al realizar un cambio de contexto. A mayor tiempo, mayor peso.

## Interrupciones vs. cambio de contexto

Es importante hacer notar que cuando se produce una interrupción, el hecho de invocar la rutina de atención de la interrupción *no es un cambio de contexto*. Esto se debe a que el código de la rutina de atención es parte del núcleo y por lo tanto no pertenece a ningún proceso en particular. Para que haya un cambio de contexto *es necesario* que se pase de la ejecución del código de un proceso a la ejecución del código de otro proceso.

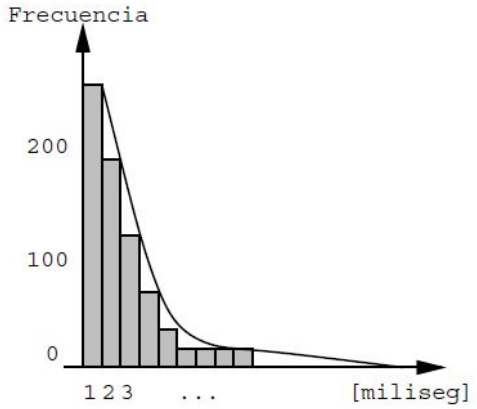
Sin embargo muchos cambios de contexto ocurren durante una interrupción, por ejemplo cuando interrumpe el reloj regresivo. En otros casos, la interrupción no se traduce en un cambio de contexto. Por ejemplo si un disco interrumpe, esto puede significar que el proceso que esperaba esta interrupción se coloca en la cola del scheduler pero no se le transfiere de inmediato el procesador, se continúa ejecutando el proceso interrumpido.

## Ráfagas de CPU

Una ráfaga de CPU es una secuencia de instrucciones que puede ejecutar un proceso sin pasar a un estado de espera. Es decir el proceso *no espera* un acceso al disco, o la recepción de un mensaje, o el término de un proceso, etc. La ejecución de un proceso consta de innumerables ráfagas de CPU.

## Tiempo de duración efectiva de una ráfaga de CPU

Es el tiempo de CPU que consume una ráfaga durante la ejecución. En este tiempo no se contabiliza el tiempo en que el proceso estuvo esperando obtener la CPU. Empíricamente se ha determinado que la mayoría de las ráfagas toman pocas instrucciones. Esto se observa gráficamente en esta figura:




El primer tramo indica el número de ráfagas con duración efectiva entre 0 y 1 milisegundo, el segundo tramo indica las ráfagas de 1 a 2 milisegundos, etc.

El scheduler puede aprovecharse de este hecho empírico para disminuir el sobrecosto de los cambios de contexto. En efecto, al final de una ráfaga el cambio de contexto es inevitable porque el proceso en curso pasa a un modo de espera. Al contrario, los cambios de contexto en medio de una ráfaga son evitables y por lo tanto son sobrecosto puro. Un buen scheduler tratará de evitar los cambios de contexto en medio de una ráfaga corta y solo introducirá cambios de contexto en ráfagas prolongadas, que son las menos.

### Tiempo de despacho de una ráfaga

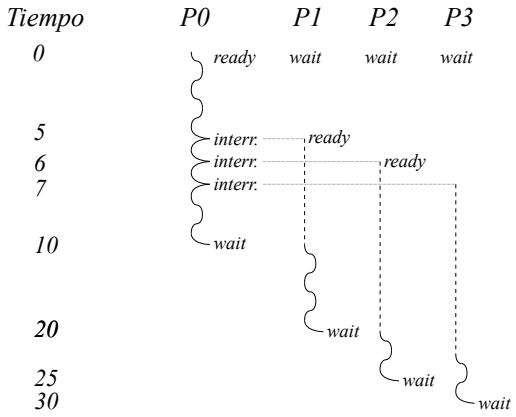
El tiempo de despacho de una ráfaga R es el tiempo real que transcurre desde que el proceso pasó a estado listo para ejecutar la ráfaga R, hasta que esa ráfaga termina de ejecutarse, es decir cuando el proceso pasa nuevamente a estado listo para ejecutar una nueva ráfaga. En este tiempo también se contabiliza el tiempo en que el proceso estuvo esperando obtener el procesador.

## Estrategias de scheduling de procesos

Como dijimos anteriormente el scheduling de procesos intenta lograr algún objetivo particular como maximizar la utilización del procesador o atender en forma expedita a usuarios interactivos. Para lograr estos objetivos se usan diversas estrategias que tienen sus ventajas y desventajas. A continuación se explican las estrategias más enseñadas en los libros de sistemas operativos, aunque en la práctica se usan estrategias bastante más complejas.

### First Come First Served o FCFS

Ésta es una estrategia para procesos non-preemptive. En ella se atiende las ráfagas en forma ininterrumpida en el mismo orden en que llegan a la cola de procesos, es decir en orden FIFO. La siguiente figura muestra esta estrategia en acción:




En T=0, el proceso P0 está en estado RUN, mientras que P1, P2 y P3 están en estado de espera. En T=5, producto de una interrupción, el proceso P1 pasa al estado READY, pero como la estrategia es non preemptive, el proceso P0 continúa con el procesador. Lo mismo ocurre en T=6 y T=7 en donde P2 y P3 pasan a estado READY. En T=10, P0 pasa a un estado de espera. Como se otorga el procesador por orden de llegada, P1 gana primero el procesador de manera continua, ejecutando toda su ráfaga sin cambios de contexto, hasta que pasa a un estado de espera en T=20. En ese instante P2 gana el procesador y continua hasta T=25, instante en que comienza a ejecutarse la ráfaga de P3.

Esta estrategia tiene la ventaja de ser muy simple de implementar. Sin embargo tiene varias desventajas:

- ✗ Pésimo comportamiento en sistemas de tiempo compartido. Por lo tanto sólo se usa en sistemas batch.
- ✗ El tiempo de despacho promedio puede ser malo por culpa del orden de llegada.

  Por ejemplo el tiempo de despacho de P1 fue de 20-5=15, el de P2, 25-6=19 y el de P3, 30-7=23. El tiempo promedio de despacho de estas 3 ráfagas es de (15+19+23)/3= 19.

  Si el scheduler hubiese elegido primero P3, luego P2 y después P1, las ráfagas hubiesen terminado en 15, 20 y 30 respectivamente dando tiempos de despacho de 15-7=8, 20-6=14 y 30-5=25. El tiempo de despacho promedio se reduciría a ~ 15.6. Se puede demostrar matemáticamente que el tiempo de despacho de las ráfagas se minimiza atendiendo primero las ráfagas de menor duración.
- ✗ Los procesos intensivos en tiempo de CPU tienden a procesarse antes que los intensivos en E/S. Por lo tanto, al comienzo el cuello de botella es la CPU mientras que al final son los canales de E/S y la CPU se utiliza poco. Es decir esta estrategia no maximiza el tiempo de utilización de la CPU.

Observe que aún cuando esta estrategia es non-preemptive esto no impide que en medio de una ráfaga de CPU pueda ocurrir una interrupción de E/S que implique que un proceso en espera pase al estado listo para correr.

## Shortest Job First o SJF

Como vimos en el ejemplo de más arriba, en principio si se atiende primero las ráfagas más breves se minimiza el tiempo promedio de despacho. Por ejemplo si en la cola de procesos hay 4 ráfagas con duraciones efectivas de 5, 2, 20 y 1, estas ráfagas deberían ser atendidas en el orden 1, 2, 5 y 20. Se demuestra que este promedio es inferior a cualquier otra permutación en el orden de atención de las ráfagas.

Desde luego el problema es que el scheduler no puede predecir el tiempo que tomará una ráfaga de CPU. Sin embargo es posible calcular un predictor del tiempo de duración de una ráfaga a partir de la

duración de las ráfagas anteriores. En efecto, se ha comprobado empíricamente que la duración de las ráfagas tiende a repetirse. El predictor que se utiliza usualmente es:

$$\tau_{n+1}^{P} = \alpha\, t_n^P + (1-\alpha)\,\tau_n^P$$

En donde:

| | |
|---|---|
| $\tau_{n+1}^{P}$ | predictor de la próxima ráfaga |
| $t_n^P$ | duración efectiva de la ráfaga recién ejecutada |
| $\tau_n^P$ | predictor de la ráfaga recién ejecutada |
| $\alpha$ | ponderador para la última ráfaga |

Típicamente $\alpha = 0.5$ . Con esta fórmula se pondera especialmente la duración de las últimas ráfagas, ya que al expandir esta fórmula se obtiene:

$$\tau_{n+1}^{P} = \alpha\, t_n^P + (1-\alpha)\,\alpha\, t_{n-1}^P + (1-\alpha)^2\,\alpha\, t_{n-2}^P + (1-\alpha)^3\,\alpha\, t_{n-3}^P + \cdots$$

De este modo cuando hay varios procesos en la cola de procesos listos para correr, se escoge aquel que tenga el menor predictor.

SJF puede ser non-preemptive y preemptive. En el caso non-preemptive una ráfaga se ejecuta completamente, aún cuando dure mucho más que su predictor. Desde luego, la variante non-preemptive se usa sólo en sistemas batch.

En el caso preemptive se establece una cota al tiempo de asignación del procesador. La cota podría ser el segundo predictor más bajo entre los procesos listos para correr. También hay que considerar el caso en que mientras se ejecuta una ráfaga, llega una nueva ráfaga. Se podría considerar que el scheduler ceda el procesador a la ráfaga que llega si su predictor es menor al máximo entre el predictor de la ráfaga en ejecución y su duración efectiva hasta ese momento. Pero enfín, estos son solo ejemplos de las muchas variantes que se pueden adoptar.

## Colas con prioridad

En esta estrategia se asocia a cada proceso una prioridad. El scheduler debe velar porque en todo momento el proceso que está corriendo es aquel que tiene la mayor prioridad entre los procesos que están listos para correr. La prioridad puede ser fija o puede ser calculada. Por ejemplo SJF es un caso particular de scheduling con prioridad en donde la prioridad p del proceso Q se calcula como:

$$p = -\tau_{n+1}^{Q}$$

El problema de esta estrategia es que puede provocar hambruna (starvation), ya que lo procesos con baja prioridad podrían nunca llegar a ejecutarse porque siempre hay un proceso listo con mayor prioridad.

## Aging

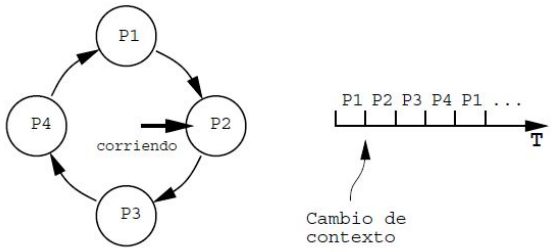
Aging es una variante de la estrategia de prioridades que resuelve el problema de la hambruna. Esta estrategia consiste en aumentar cada cierto tiempo la prioridad de todos los procesos que se encuentran listos para correr. Por ejemplo, cada un segundo se puede aumentar en un punto la prioridad de los procesos que se encuentran en la cola *ready*. Cuando el scheduler realiza un cambio de contexto el proceso saliente recupera su prioridad original (sin las bonificaciones).

Desde este modo, los procesos que han permanecido mucho tiempo en espera del procesador, tarde o temprano ganarán la prioridad suficiente para recibirlo.

## Round-Robin

Es una estrategia muy usada en sistemas de tiempo compartido. En Round-Robin los procesos *listos*

*para correr* se organizan en una lista circular. El scheduler se pasea por la cola dando *tajadas* pequen as de tiempo de CPU ( *time slicing*): de 10 a 100 milisegundos. Esta estrategia se aprecia en la figura:




Esta estrategia minimiza el *tiempo de respuesta* en los sistemas interactivos. El tiempo de respuesta es el tiempo que transcurre entre que un usuario ingresa un comando interactivo hasta el momento en que el comando despliega su primer resultado. No se considera el tiempo de despliegue del resultado completo. Esto se debe a que para un usuario interactivo es mucho más molesto que un comando no dé ninguna señal de avance que un comando que es lento para desplegar sus resultados. Por esta razón el tiempo de respuesta sólo considera el tiempo que demora un comando en dar su primer señal de avance.

## Implementación

Esta estrategia se implementa usando una cola FIFO con todos los procesos que están listos para correr. Al final de cada tajada el scheduler coloca el proceso que se ejecutaba al final de la cola y extrae el que se encuentra en primer lugar. El proceso extraído recibe una nueva tajada de CPU. Para implementar la tajada, el scheduler programa una interrupción del reloj regresivo de modo que se invoque una rutina de atención (que pertenece al scheduler) dentro del tiempo que dura la tajada.

Si un proceso termina su ráfaga sin agotar su tajada (porque pasa a un modo de espera), el scheduler extrae el próximo proceso de la cola y le da una nueva tajada completa. El proceso que terminó su ráfaga *no se coloca* al final de la cola del scheduler. Este proceso queda en alguna estructura de datos esperando algún evento.

## Tamaño de tajada

Una primera decisión que hay que tomar es qué tamaño de tajada asignar. Si la tajada es muy grande la estrategia degenera en FCFS con todos sus problemas. Al contrario si la tajada es muy pequeña el sobrecosto en cambios de contexto es demasiado alto.

Además si la tajada es muy pequeña aumenta el tiempo promedio de despacho. En efecto, si se tienen dos ráfagas de duración **t**. El tiempo medio de despacho de ambas ráfagas en FCFS será:

$$\overline{T} = \frac{t + 2t}{2} = \frac{3}{2}t$$

En cambio en RR con tajada mucho más pequeña que **t**, ambas ráfagas serán despachadas prácticamente al mismo tiempo en **2t**, por lo que el tiempo medio de despacho será:

$$\overline{T} \rightarrow 2t$$

Una regla empírica que da buenos resultados en Round-Robin es que se fija el tamaño de la tajada de modo que el 80% de las ráfagas deben durar menos que la tajada y por lo tanto se ejecutan sin cambios de contexto de por medio.

## Llegada de procesos

El segundo problema que hay que resolver es donde colocar un proceso P que pasa de un estado de espera al estado listo para correr, suponiendo que el proceso que estaba en ejecución en ese momento es Q. Las alternativas son:

(a) El proceso P se agrega al final de la cola de procesos para el round robin.

(b) El proceso P se agrega al comienzo de la cola.

(c) El proceso P le roba el procesador a Q. El proceso Q queda en primer lugar en la cola de procesos para Round Robin.

La alternativa (a) perjudica a los procesos intensivos en E/S porque pasan mucho más frecuentemente a estado de espera que los intensivos en CPU. Cada vez que regresan al estado listo para ejecutarse se van al final de la cola. Por otra parte se puede demostrar que en las alternativas (b) y (c) los procesos intensivos en CPU pueden sufrir hambruna. Pero existe una cuarta alternativa:

(d) El proceso P le roba el procesador a Q o queda al comienzo de la cola. Pero su tajada de tiempo se reduce al tiempo que le restaba en el momento de pasar a estado de espera.

# Jerarquías de Scheduling

Normalmente los sistemas operativos implementan una jerarquía de scheduling de procesos. Esto se debe a que, primero, la CPU no es el único recurso que hay que administrar, la memoria también es escasa y hay que administrarla; y segundo, no todos los procesos tienen la misma urgencia, por lo que es conveniente que los procesos no urgentes se pospongan hasta que el computador se encuentre más desocupado.

Es así como el scheduling se podría organizar en 3 niveles:

- Scheduling de corto plazo.

  En este nivel se administra sólo la CPU usando las estrategias vistas en la sección anterior. Esta administración la realiza el núcleo.

- Scheduling de mediano plazo.

  En este nivel se administra la memoria. Cuando la memoria disponible no es suficiente se llevan procesos a disco. Esta transferencia a disco toma mucho más tiempo que un traspaso de la CPU (segundos vs. microsegundos). Por lo tanto las transferencias a disco deben ser mucho menos frecuentes que los cambios de contexto. Por esta razón se habla de mediano plazo. Este nivel de scheduling también se realiza en el núcleo.

- Scheduling de largo plazo.

  Este nivel está orientado al procesamiento batch. Los procesos batch o jobs no son urgentes, por lo que su lanzamiento puede ser postergado si el computador está sobrecargado. No todos los sistemas operativos implementan este nivel de scheduling y si lo hacen, lo realizan fuera del núcleo a través de procesos ``demonios´´.

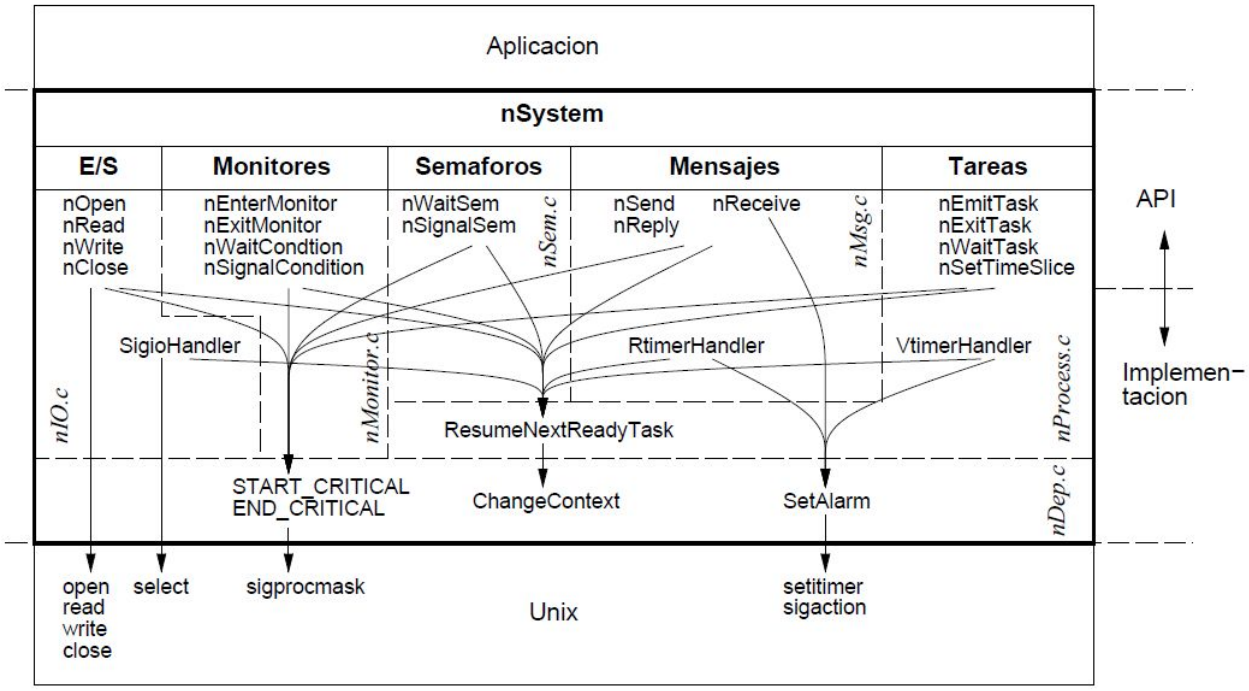
  En este nivel hay una cola de jobs esperando ser lanzados. El scheduler lanza jobs sólo en la medida que el procesador tiene capacidad disponible. Una vez que un job fue lanzado, éste se administra sólo en los niveles de corto y mediano plazo. Es decir que el scheduler de largo plazo no tiene ninguna incidencia sobre los jobs ya lanzados.

# Administración de procesos en nSystem

El nSystem es un sistema de procesos livianos o nano tareas que corren en un solo proceso pesado de Unix. Las tareas de nSystem se implementan multiplexando el tiempo de CPU del proceso Unix en tajadas. Esto se puede hacer gracias a que un proceso Unix tiene asociado su propio reloj regresivo, un mecanismo de interrupciones (señales), E/S no bloqueante y todas las características del hardware que se necesitan para implementar multiprogramación en un solo procesador, con la salvedad de que este hardware es emulado en software por Unix.

La siguiente figura muestra la organización de nSystem en módulos. nSystem es una biblioteca de

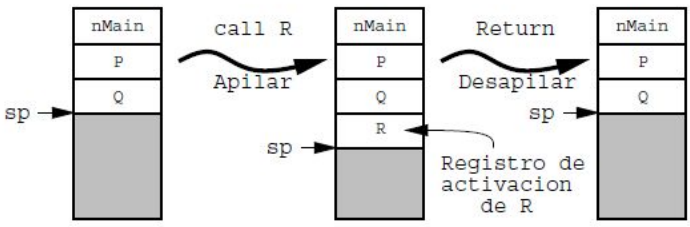
procedimientos que se sitúa entre la aplicación (el código del programador) y la API de Unix:




Los procedimientos superiores corresponden a la API de nSystem y son por lo tanto invocados por la aplicación. Los procedimientos de más abajo son procedimientos internos que se requieren en la implementación de la API. El nombre que aparece verticalmente es el archivo en donde se implementa un módulo en particular. Las flechas indican que el procedimiento desde donde se origina la flecha necesita invocar el procedimiento de destino de la flecha.

## La pila

Cada nano tarea tiene su propia pila. Los registros de activación de los procedimientos que se invocan en una tarea se apilan y desapilan en esta estructura de datos (ver figura). Un registro del procesador, denominado `sp`, apunta hacia el tope de la pila.




Llamadas y retornos de procedimientos usando una pila

La pila es de tamaño fijo, por lo que una recursión demasiado profunda puede desbordarla. El nSystem chequea el posible desborde de la pila en cada cambio de contexto. De ocurrir un desborde el nSystem termina con un mensaje de error. Sin embargo si no hay cambios de contexto no puede detectar el desborde. En este caso, el nSystem podría terminar en un *segmentation fault* o *bus error*, sin mayor explicación de la causa de la caída.

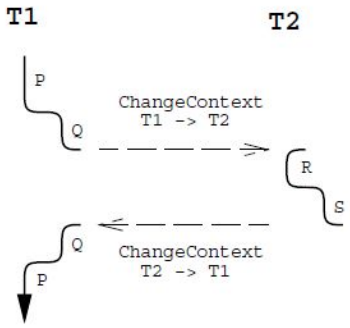
## El cambio de contexto

En nSystem nunca hay más de una nano tarea corriendo realmente. El registro `sp` del procesador

apunta al tope de la pila de la tarea que está corriendo.

El cambio de contexto en nSystem lo realiza el scheduler llamando al procedimiento:

```
void ChangeContext(nTask out_task, nTask in_task);
```

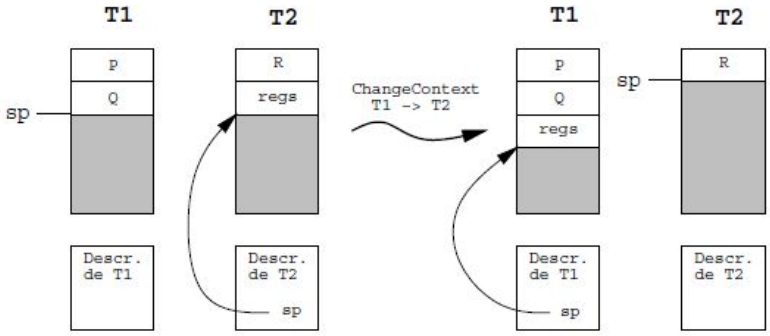



Cambio de pila durante un cambio de contexto.

Este procedimiento realiza las siguientes actividades:

1. Resguarda los registros del procesador en el tope de la pila de la tarea saliente ( `out_task`).
2. Guarda `sp` en el descriptor de esta tarea saliente (`out_task->sp`).
3. Rescata el puntero al tope de la pila de la tarea entrante a partir de su descriptor de proceso (`in_task->sp`).
4. Restaura los registros de la tarea entrante que se encuentran en el tope de la pila.
5. Retoma la ejecución a partir del llamado a `ChangeContext` de la tarea entrante.

Es decir que el llamado a `ChangeContext` de la tarea saliente no retorna hasta que se haga un nuevo cambio hacia aquella tarea:




Cambio de contexto en nSystem.

Observe que las labores que realiza `ChangeContext` no pueden llevarse a cabo en C y por lo tanto parte de este procedimiento está escrito en *assembler* y es completamente dependiente de la arquitectura del procesador. Sin embargo, este cambio de contexto *no requiere una llamada al sistema*, el cambio de contexto se realiza sólo con instrucciones del modo usuario en el proceso Unix en donde corre el nSystem.

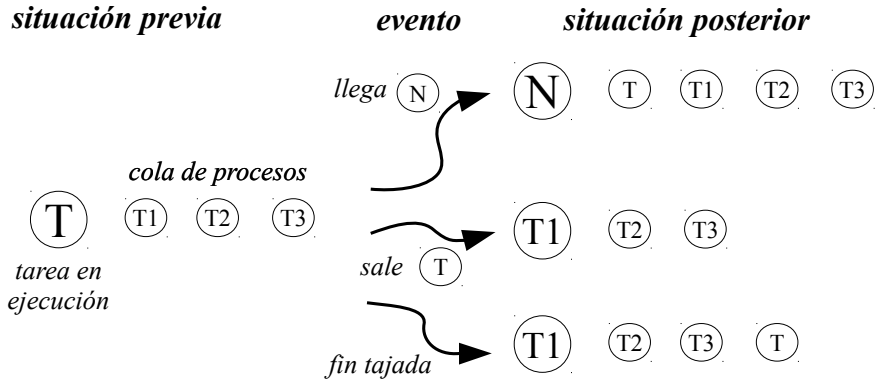
## Scheduling en nSystem

La estrategia de scheduling de nSystem podría denominarse como *preemptive last come first served* o PLCFS. Es necesario recalcar que esta estrategia no se implementa en ningún sistema operativo conocido. Fue escogida para nSystem por su extrema simplicidad, lo que permite escribir su código en una sola pizarra durante una clase. Si bien no se implementa en los sistemas operativos, se inspira en el scheduling que realiza por hardware el microprocesador de los años 80-90 *transputer*. La característica

de este microprocesador era ofrecer threads a nivel del set de instrucciones.

En PLCFS las ráfagas son atendidas en orden LIFO (last in first out), es decir que las tareas que llegan al scheduler toman el control de procesador de inmediato y la tarea que se ejecutaba en ese instante se coloca en primer lugar en la cola del scheduler. Para evitar que una tarea acapare indefinidamente el procesador, cada **t** milisegundos el scheduler le quita el procesador a la tarea en ejecución, cualquiera sea esta tarea, y la coloca al final de una cola.

Esta estrategia se implementa usando una cola dual, es decir en donde se pueden agregar tareas ya sea al final como al comienzo. La siguiente figura muestra una situación previa en donde el procesador ejecuta la tarea T y la cola de procesos mantiene las tareas T1, T2 y T3. La configuración cambia según el evento de scheduling que ocurra. En primer caso es cuando llega una tarea N. Esto puede ser el resultado de una interrupción o una acción de la tarea T. En ese caso la tarea N roba el procesador y la tarea T se agrega al comienzo de la cola. El segundo caso ocurre cuando la tarea T pasa a un estado de espera. Entonces la tarea T1 se queda con el procesador y todas las tareas de la cola avanzan una posición. La llegada de la señal del timer corresponde al último caso. En ese caso la tarea T se coloca al final de la cola, T1 se queda con el procesador y todas las demás tareas avanzan en una posición.




Scheduling en nSystem

El fin de tajada se implementa programando una señal que se invoca cada **t** milisegundos de tiempo virtual del proceso Unix. Para esta señal se usa el tiempo virtual del proceso Unix, es decir el tiempo de uso del procesador real por parte del proceso Unix sin contar el tiempo en que el procesador estaba en poseción de otro proceso Unix. En efecto, cambiar de tarea cada **t** milisegundos de tiempo real no tendría sentido, pues durante ese intervalo podría haber sucedido que el proceso Unix en donde corre nSystem no hubiese tenido el procesador real y por lo tanto la tarea en ejecución no pudo avanzar tampoco.

## Scheduling simplificado

La siguiente es una versión simplificada del código del scheduler. En ella no se consideran ciertas condiciones de borde que obscurecen innecesariamente el código. Estas condiciones son el caso en que no hay tareas listas para correr y el caso en que el scheduling es non-preemptive.

El scheduler mantiene las siguientes variables:

| | |
|---|---|
| `nTask current_task;` | La tarea en ejecución |
| `Queue ready_queue;` | La cola dual |
| `int current_slice;` | El intervalo entre señales |

El scheduler retoma una tarea con el siguiente procedimiento:

```
ResumeNextReadyTask() {
```

```
  nTask this_task= current_task;
  nTask next_task= GetTask(ready_queue);
  ChangeContext(this_task, next_task);
  current_task= this_task; /* (1) */
}
```

Observe que si **T** era la tarea que corría cuando se invocó `ResumeNextReadyTask`, el llamado a `ChangeContext` no retornará hasta que se retome **T**. Es decir cuando se invoque nuevamente `ResumeNextReadyTask` y **T** aparezca en primer lugar en la cola `ready_queue`. Mientras tanto, el nSystem ejecutó otras tareas y realizó eventualmente otros cambios de contexto. Por esta razón en (1) se asigna **T** a la tarea actual y no `next_task`.

En cada señal periódica se invoca el siguiente procedimiento:

```
void VtimerHandler() {
  /* Programa la siguiente interrupcion */
  SetAlarm(VIRTUALTIMER, current_slice, VtimerHandler);
  /* Coloca la tarea actual al final de la cola */
  PutTask(ready_queue, current_task);
  /* Retoma la primera tarea en la cola */
  ResumeNextReadyTask();
}
```

Para implementar `SetAlarm` se usaron las llamadas al sistema `setitimer` y `sigaction`.

Note que la tarea que recibe el procesador estaba bloqueada en el llamado a `ChangeContext` de `ResumeNextReadyTask` y por lo tanto es ella la que actualizará la variable `current_task` con su propio identificador. En nSystem todos los cambios de contexto se realizan a través de `ResumeNextReadyTask`. La única excepción es el cambio de contexto que se realiza al crear una tarea.

# Implementación de semáforos en nSystem (incorrecta)

Un semáforo se representa mediante una estructura de datos que tiene la siguiente información:

| | |
|---|---|
| `int count;` | El número de tickets |
| `Queue queue;` | Los procesos que esperan un ticket |

El procedimiento `nWaitSem` debe bloquear la tarea que lo invoca cuando `count` es 0:

```
void nWaitSem(nSem sem) {
  if (sem->count>0)
    sem->count--;
  else {
    current_task->status= WAIT_SEM;
    PutTask(sem->queue, current_task);
    /* Retoma otra tarea */
    ResumeNextReadyTask();
  }
}
```

El procedimiento `nSignalSem` debe desbloquear una tarea cuando `count` es 0 y hay tareas en espera. La tarea desbloqueada toma de inmediato el control del procesador (ya que el scheduling es LCFS).

```
void nSignalSem(nSem sem) {
  if (EmptyQueue(sem->queue))
    sem->count++;
```

```
   else {
      nTask wait_task= GetTask(sem->queue);
      wait_task->status= READY;
      PushTask(ready_queue, current_task);
      PushTask(ready_queue, wait_task);
      /* wait_task es la primera en la cola! */
      ResumeNextReadyTask();
   }
}
```

Esta implementación es incorrecta porque el proceso Unix podría recibir una señal de timer virtual en un mal momento, gatillando un datarace. Por ejemplo, si la señal ocurre justo en el momento de incrementar o decrementar `sem->count`, se pueden perder tickets o inventar tickets.

# Implementación de secciones críticas en nSystem

El código de los procedimientos nWaitSem y nSignalSem corresponde a una sección crítica y por lo tanto se debe garantizar la exclusión mutua de las tareas mientras se ejecutan. Como hemos supuesto que hay un solo procesador (1 solo core), la única manera de que 2 tareas lleguen a entremezclar la ejecución de estos procedimientos es por medio de una señal del timer. Por lo tanto la implementación más simple de la sección crítica consiste en inhibir las señales para evitar cambios de contexto dentro de la sección crítica. Esto se logra con los llamadas `START_CRITICAL` y `END_CRITICAL`. En consecuencia la versión correcta para los semáforos es:

```
int nWaitSem(...) {
  START_CRITICAL();
    ... como antes ...
  END_CRITICAL();
}
```

```
int nSignalSem(...) {
  START_CRITICAL();
    ... como antes ...
  END_CRITICAL();
}
```

El procedimiento `START_CRITICAL` inhibe las señales de Unix usando la llamada al sistema `sigprocmask`. La misma llamada se usa en `END_CRITICAL` para reactivar las señales.

No es necesario inhibir las señales en `VtimerHandler` o `SigioHandler` porque estos procedimientos son para atender señales de Unix, y en este caso se configuran para que Unix los invoque desde ya con las señales inhibidas. De igual forma cuando estos procedimientos retornan, se reactivan automáticamente las señales.

Es importante destacar que cuando en `nSignalSem` se inhiben las señales de Unix y se retoma otra tarea llamando a `ResumeNextReadyTask`, es esta otra tarea la que reactivará las señales en el `END_CRITICAL` del final de `nWaitSem`.

El `END_CRITICAL` que se encuentra al final de `nSignalSem` sólo se invocará cuando se retome nuevamente la tarea que invocó este procedimiento. Y entonces se reactivarán las señales que fueron inhibidas por otra tarea, aquella que perdió el procesador.

## Entrada/Salida no bloqueante

En Unix se puede colocar un descriptor de archivo (`fd`) en un modo no bloqueante y hacer que se gatille la señal `SIGIO` cuando haya algo para leer en ese descriptor. Esto significa que `read(fd,...)` no se bloquea cuando no hay nada que leer en `fd`, sino que retorna **-1**. Esto se configura en Unix con la llamada al sistema `fcntl`.

Esto es importante porque de no programar los descriptores en modo bloqueante, cuando una tarea intente leer un descriptor se bloquearán todas las tareas hasta que haya algo para leer en ese descriptor. Esto es inadmisible en un sistema multiprogramado.

Una primera implementación de `nRead` es:

```
int nRead(int fd, char *buf, int nbyte) {
  int rc;
  START_CRITICAL();
  /* Intentamos leer */
  rc= read(fd, buf, nbyte);
  while (rc<0 && errno==EAGAIN) {
    /* No hay nada disponible */
    AddWaitingTask(fd, current_task);
    current_task->status= WAIT_READ;
    /* Pasamos a la proxima tarea ready */
    ResumeNextReadyTask();
    /* Ahora si que deberia funcionar */
    rc= read(fd, buf, nbyte);
  }
  END_CRITICAL();
  return rc;
}
```

El valor `EAGAIN` en `errno` indica que no hay nada para leer y que `read` debería ser reinvocado más tarde. Unix informa al proceso de nSystem que hay algo para leer en algún descriptor invocando la señal `SIGIO`. El procedimiento que atiende esta señal es:

```
SigioHandler() {
  PushTask(current_task, ready_queue);
  ... preparar argumentos para select ...
  select(...);
  ... por cada tarea T que puede avanzar:
      PushTask(T, ready_queue);
  ...
  ResumeNextReadyTask();
}
```

El llamado al sistema `select` determina qué descriptores de archivo tienen datos para leer o en qué descriptores se puede escribir sin llenar los buffers internos de Unix que obligarían a bloquear el proceso Unix. En el código que sigue al `select` se busca si hay tareas bloqueadas a la espera de alguno de los descriptores cuya lectura/escritura no causará un bloqueo temporal de nSystem. Estas tareas quedarán antes en la cola que la tarea en ejecución cuando se invocó `SigioHandler`, porque el scheduling es preemptive LCFS.

Observe que el código de `nRead` es una sección crítica y por lo tanto se usan los procedimientos `START_CRITICAL` y `END_CRITICAL`. para garantizar la exclusión mutua, como se hizo para `nWaitSem` y `nSignalSem`. En cambio, dado que `SigioHandler` es una rutina de atención de señales, la exclusión mutua está garantizada gracias a que cuando ocurre la señal, este procedimiento se invoca con las señales inhibidas, como ocurre con `VtimerHandler`.

# Implementación de procesos en Unix para monoprocesadores: núcleo clásico

Las implementaciones clásicas de Unix administran los procesos en un esquema similar al de nSystem. La estrategia de scheduling de procesos es un poco más elaborada pues implementa procesos con prioridades y al mismo tiempo intenta dar un buen tiempo de respuesta a los usuarios interactivos, algo que no es fácil de lograr debido a las restricciones que impone la memoria del procesador.

Más interesante que la estrategia de scheduling es estudiar, primero, a partir de qué momento los

procesos comienzan a compartir parte de su espacio de direcciones, y segundo, cómo se logra la exclusión mutua en secciones críticas.

Cuando un proceso ejecuta una aplicación el procesador se encuentra en modo usuario, y por lo tanto algunas instrucciones están inhibidas. Si se ejecutan se produce una interrupción que atrapa el núcleo.

Los segmentos del espacio de direcciones que son visibles desde una aplicación son el de código, datos y pila. Cuando ocurre una interrupción el procesador pasa al modo sistema e invoca una rutina de atención de esa interrupción. Esta rutina es provista por el núcleo y su dirección se rescata del vector de interrupciones, el que no es visible desde un proceso en modo usuario.

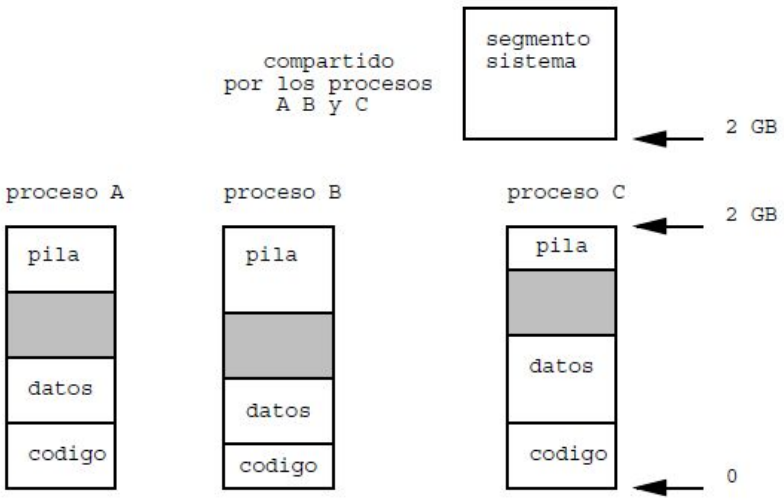
## Llamadas al sistema

Las llamadas al sistema siempre las ejecuta el núcleo en el modo sistema. La única forma de pasar desde el modo usuario de una aplicación al modo sistema del núcleo es por medio de una interrupción. Por lo tanto las llamadas al sistema se implementan precisamente a través de una interrupción.

Cuando se hace una llamada al sistema como `read`, `write`, etc., se invoca un procedimiento de biblioteca cuya única labor es colocar los argumentos en registros del procesador y luego ejecutar una instrucción de máquina que gatilla explícitamente la interrupción (por ejemplo la instrucción INT en x86). En algunos procesadores no existe un instrucción legal para gatillar la interrupción, y entonces se coloca un instrucción ilegal, lo que gatilla una interrupción debido a la ejecución de una instrucción ilegal. El núcleo reconoce estas interrupciones como una llamada al sistema.

## El segmento sistema

Al pasar a modo sistema, comienza a ejecutarse código del núcleo. El espacio de direcciones que se ve desde el núcleo contiene los mismos segmentos que se ven desde una aplicación: código, datos y pila. Pero además se agrega un nuevo segmento: el segmento sistema. Este segmento se aprecia en esta figura:




El espacio de direcciones en modo sistema.

El segmento sistema es *compartido* por todos los procesos. Es decir no existe un segmento sistema por proceso, sino que todos los procesos ven los mismos datos en este segmento. Si un proceso realiza una modificación en el segmento sistema, los demás procesos verán la modificación. Sin embargo, solo lo verán cuando trabajen en modo sistema, porque en modo usuario el segmento sistema es invisible para los procesos.

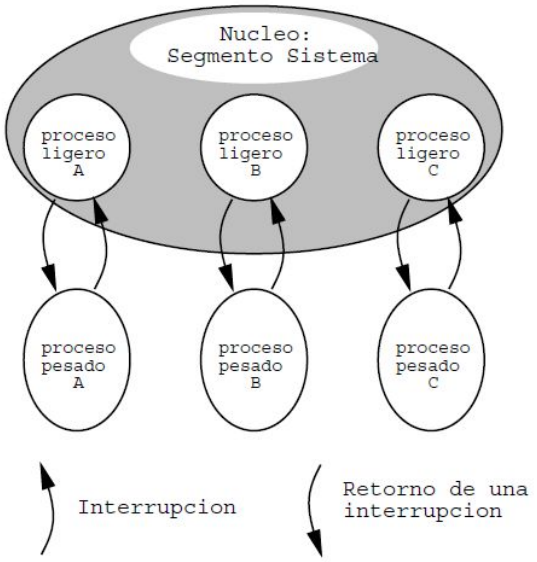
El segmento sistema contiene el código del núcleo, el vector de interrupciones, los descriptores de

proceso, la cola de procesos *ready*, las colas de E/S de cada dispositivo, los *buffers* del sistema de archivos, las puertas de acceso a los dispositivos, la memoria de video, etc. Enfín, todos los datos que los procesos necesitan compartir.

Por simplicidad, en máquinas con hasta 1 GB de memoria es usual que las implementaciones de Unix ubiquen los segmentos de la aplicación por debajo de los 2 gigabytes (la mitad del espacio direccionable) y el segmento sistema por arriba de los 2 gigabytes. Además en el segmento sistema se coloca toda la memoria real del computador. De esta forma el núcleo no solo puede ver la memoria del proceso interrumpido, sino que también pueden ver la memoria de los demás procesos. Sin embargo, este esquema no se puede usar con máquinas con más de 1 GB de memoria.

## Procesos ligeros en el núcleo

Cada proceso pesado tiene asociado en el núcleo un proceso ligero encargado de atender sus interrupciones (ver la siguiente figura). Este proceso ligero tiene su pila en el segmento sistema y comparte sus datos y código con el resto de los procesos ligeros que corren en el núcleo.




Procesos ligeros en el núcleo

Es en este proceso ligero en donde corre la rutina de atención de cualquier interrupción que se produzca mientras corre una aplicación en el modo usuario, sea esta interrupción una llamada al sistema, un evento de E/S relacionado con otro proceso, una división por cero, etc.

Conceptualmente, una llamada al sistema es un mensaje síncrono que envía el proceso pesado al proceso ligero en el núcleo. De igual forma el retorno de la rutina de atención corresponde a la respuesta a ese mensaje que permite que el proceso pesado pueda continuar.

Los procesos del núcleo se ejecutan con las interrupciones inhibidas y por lo tanto no pueden haber cambios de contexto no deseados. De esta forma se logra la exclusión en las secciones críticas del núcleo. Los cambios de contexto sólo ocurren explícitamente.

Para implementar este esquema, el hardware de los microprocesadores modernos ofrece dos punteros a la pila, uno para el modo usuario y otro para el modo sistema y su intercambio se realiza automáticamente cuando ocurre una interrupción. Desde luego, en el modo usuario, el puntero a la pila del modo sistema no es visible, mientras que en el modo sistema sí se puede modificar el puntero a la pila del modo usuario.

## Relación con nSystem

Cuando un proceso Unix pasa al modo sistema, le entrega el control a su respectivo proceso ligero en el

núcleo. Este último corre con las interrupciones inhibidas, comparte el segmento sistema y tiene acceso a toda la máquina. Por lo tanto conceptualmente es similar a una nano tarea en nSystem que inhibe las interrupciones con `START_CRITICAL()`, comparte el espacio de direcciones con otras nano tareas y tiene acceso a todo el procesador virtual.

Es decir, *un proceso pesado en Unix que llama al sistema o es interrumpido se convierte en un proceso ligero ininterrumpible en el núcleo*. Luego, los algoritmos y estrategias que se vieron para nSystem pueden ser usados para implementar mensajes, semáforos, E/S, etc. en un monoprocesador bajo Unix.

Este esquema es el que se uso en las primeras versiones de Unix como por ejemplo Unix System V 3.x, Unix BSD y Linux 1.x, y por eso se les denomina *núcleos clásicos*. Su principal ventaja es que es muy simple de implementar, porque dentro del núcleo no hay que preocuparse de los dataraces. No es necesario garantizar la exclusión mutua a nivel de cada estructura de datos del núcelo, ya que todo el núcleo es una gran y única sección crítica. Sin embargo tiene la desventaja de que el sistema permanece largos períodos con las interrupciones inhibidas, lo que puede ser inaceptable para sistemas de tiempo real. Este tipo de sistemas requieren respuestas rápidas a eventos en un tiempo acotado, usualmente en el rango de unos pocos micro-segundos. Este requisito no se puede cumplir si las interrupciones están inhibidas dentro del núcleo.

# Unix en multiprocesadores

Un multiprocesador es un computador que posee varios procesadores que comparten la misma memoria física. En esta sección se debe entender que procesador y *core* son sinónimos, pero se preferirá el uso del término procesador por sobre el término *core* introducido por Intel. Por la misma razón un multiprocesador es lo mismo que en la jerga de Intel se llama un procesador multicore.

Un multiprocesador puede correr varios procesos Unix en paralelo. Sin embargo, la visión conceptual de un proceso Unix sigue siendo la misma: *un solo procesador* en donde corre un solo thread. Es decir que los múltiples procesos Unix que corren en un multiprocesador no comparten sus espacios de direcciones.

## Scheduling

En este esquema hay *un reloj regresivo por procesador*. De modo que el tiempo de cada procesador se puede multiplexar en tajadas de tiempo que se otorgan por turnos a los distintos procesos que estén listos para correr. Cada vez que ocurre una interrupción del reloj regresivo, la rutina de atención debe colocar el proceso en ejecución en la cola global de procesos *ready*, extraer un nuevo proceso y ejecutarlo.

El problema se presenta cuando la interrupción se produce simultáneamente en dos procesadores. Entonces los dos procesadores modificarán paralelamente la cola global de procesos, con el peligro de dejar la cola en un estado inconsistente. Entonces la pregunta es ¿cómo se evitan los dataraces cuando hay 2 procesadores ejecutando código del núcleo en paralelo?

## Primera solución: un solo procesador en el núcleo

Cuando todos los procesadores ejecutan instrucciones en el modo usuario, no hay secciones críticas puesto que los procesos que corren en cada procesador no comparten sus espacios de direcciones. De igual forma si sólo uno de los procesadores se encuentra en el modo sistema, no hay peligro de modificación concurrente de una estructura de datos compartida.

El peligro de las secciones críticas se presenta cuando hay 2 o más procesadores reales en el modo sistema al mismo tiempo, es decir en el núcleo compartiendo el mismo segmento sistema.

Por lo tanto, la forma más simple de implementar Unix en un multiprocesador es impedir que dos procesadores puedan correr simultáneamente en el modo sistema.

El problema es cómo impedir que dos procesadores ejecuten en paralelo código del núcleo. La solución estándar en un monoprocesador que consiste en inhibir las interrupciones no funciona en un multiprocesador. En efecto, sólo se pueden inhibir las interrupciones externas a la CPU, es decir las del reloj regresivo y las de E/S. Las interrupciones causadas por el software, como por ejemplo una llamada al sistema, no se pueden inhibir. Un proceso en el modo usuario todavía puede realizar llamadas al sistema aunque las interrupciones estén inhibidas. Por lo demás un procesador solo puede inhibir sus propias interrupciones externas, no las de los otros procesadores.

Es decir que si dos procesos que se ejecutan en procesadores reales llaman al sistema al mismo tiempo, ambos entrarán al núcleo y habrá dos procesadores corriendo en el modo sistema. Esto originará problemas de sección crítica.

## Exclusión a través de candados

Para impedir que dos procesadores pasen al modo sistema simultáneamente por medio de una llamada al sistema, se coloca un candado (*lock*) a la entrada del núcleo. Cuando un procesador entra al núcleo cierra el candado. Si otro procesador intenta entrar al núcleo tendrá que esperar hasta que el primer proceso abra el candado al salir del núcleo.

El candado es una estructura de datos que posee dos operaciones:

- `spinLock(&lock)`: Si el candado está cerrado espera hasta que esté abierto. Cuando el candado esta abierto, cierra el candado y retorna de inmediato.
- `spinUnlock(&lock)`: Abre el candado y retorna de inmediato.

El candado se cierra al principio de la rutina de atención de las interrupciones por llamadas al sistema. Es decir cuando el procesador ya se encuentra en el modo sistema, pero todavía no ha realizado ninguna acción peligrosa. Si el procesador encuentra cerrado el candado, se queda bloqueado hasta que el candado sea abierto por el procesador que se encuentra en el modo sistema. Mientras tanto el procesador bloqueado *no puede ejecutar otros procesos*, pues dado que la cola *ready* es una estructura de datos compartida, no puede tener acceso a ella para extraer y retomar un proceso.

El candado se abre en la rutina de atención justo antes de retornar a la aplicación.

Observe que no es razonable implementar el candado antes de que la aplicación pase al modo sistema, puesto que nadie puede obligar a una aplicación a verificar que el candado está abierto antes de hacer una llamada al sistema.

## Implementación de candados

La implementación específica de un Lock que se verá a continuación se denomina *spin-lock* porque cuando el lock está tomado, se recurre a un ciclo de *busy-waiting* para esperar a que se libere. Por esta razón las operaciones para tomar o liberar el candado se llaman `spinLock` y `spinUnlock`.

La siguiente implementación es *errónea* debido a que pueden ocurrir dataraces. El candado se representa mediante un entero que puede estar `OPEN` o `CLOSED`.

```
void spinLock(int *plock) {
  // Incorrecto:
  while (*plock==CLOSED)
    ;
  *plock=CLOSED;
}
```

```
void spinUnlock(int *plock) {
  *plock= OPEN;
}
```

El error está en que dos procesadores pueden cargar simultáneamente el valor de `*plock` y determinar que el candado esta abierto y pasar ambos al modo sistema.

Para implementar un candado, la mayoría de los microprocesadores ofrecen instrucciones de máquina atómicas (indivisibles) que permiten implementar un spin-lock eficientemente. En Sparc esta

instrucción es:

`swap` *dir. en memoria*, *reg*

Esta instrucción intercambia atómicamente el contenido de la dirección especificada con el contenido del registro. Esta instrucción es equivalente al procedimiento:

```
int Swap(int *ptr, int value) {
  int ret= *ptr;
  *ptr= value;
  return ret;
}
```

Sólo que por el hecho de ejecutarse atómicamente, dos procesadores que tratan de ejecutar en paralelo esta instrucción, serán *secuencializados*.

La siguiente es una implementación correcta pero ineficiente de un spin-lock:

```
void spinLock(int *plock) {
  while (Swap(plock, CLOSED)==CLOSED)
    ; /* busy-waiting */
}
```

```
void spinUnlock(int *plock) {
  *plock= OPEN;
}
```

Observe que si un procesador encuentra el candado cerrado, la operación `swap` se realiza de todas formas, pero el resultado es que el candado permanece cerrado. En inglés, un candado implementado de esta forma se llama *spin-lock*.

Esta implementación es ineficiente porque puede producir mucho tráfico en el bus de datos que es compartido por todos los procesadores para comunicarse entre sí y para llegar a la memoria. No hay problema si se pide el spin-lock y nadie lo posee. Tampoco hay problema si un solo procesador pide el spin-lock que está tomado por otro procesador. En este caso la función spinLock realizará la operación swap hasta que el spin-lock se libere, pero estas operaciones ocurrirán en la memoria cache del procesador solicitante, sin interferir con la ejecución de los demás procesadores.

La situación desfavorable se produce cuando un procesador posee el spin-lock y al menos otros 2 procesadores pretenten obtener el mismo spin-lock. El problema se presenta porque las constantes operaciones de swap de ambos procesadores ya no pueden realizarse a nivel se sus respectivos caches porque swap lee y escribe el spin-lock. Para asegurar la consistencia de las memorias caches se requiere el uso del bus compartido en cada operación swap. Cada swap puede tomar hasta unos 50 nanosegundos si los procesadores están en chips distintos (o unos 10 nanosegundos si están en el mismo chip). Compárese esta cifra a solo 1 nanosegundo cuando la operación swap se realiza a nivel del cache del procesador. Aunque el problema no es tanto el tiempo que toma en ejecutarse swap. El problema es que se usa el 100% del bus compartido. Si cualquier otro procesador, incluyendo el procesador que posee el spin-lock, requiere usar el bus, tendrá que competir con las operaciones swap de los procesadores que pretenden obtener el spin-lock. La conclusión es que esta situación degrada considerablemente el desempeño de todos los procesadores.

La siguiente es una implementación correcta y eficiente de un spin-lock:

```
void spinLock(volatile int *plock) {
  do {
    while (*plock==CLOSED)
      ; /* busy-waiting */
  } while (Swap(plock, CLOSED)!=OPEN)
}
```

```
void spinUnlock(int *plock) {
  *plock= OPEN;
}
```

La clave en esta solución es que los procesadores que esperan obtener el spin-lock solo leen, no escriben. Esto permite que los procesadores puedan mantener una copia del spin-lock es sus propias memorias caches. Solo la primera lectura requiere usar el bus compartido, el resto de las lecturas no.

Los demás procesadores no verán ningún impacto en su desempeño. Cuando se invoque spinUnlock, sí se requerirá usar el bus compartido para invalidar las copias del spin-lock en los procesadores que lo pretenden, pero esto ocurre unas pocas veces, no en un ciclo.

Observación: Este es uno de los pocos casos en donde se debe usar un ciclo de busy-waiting. Retomar otro proceso no es una opción porque para hacerlo se necesitaría entrar al núcleo. No queda otra que hacer esperar al procesador que necesita obtener un spin-lock.

### Ventajas y desventajas

Se ha presentado una solución simple para implementar Unix en un multiprocesador. La solución consiste en impedir que dos procesadores puedan ejecutar simúltanemente código del núcleo, dado que comparten el mismo segmento sistema.

Este mecanismo se usa en SunOS 4.X ya que es muy simple de implementar sin realizar modificaciones mayores a un núcleo de Unix diseñado para monoprocesadores, como es el caso de SunOS 4.X. Este esquema funciona muy bien cuando la carga del multiprocesador está constituida por procesos que requieren mínima atención del sistema.

La desventaja es que dependiendo del uso que se dé al computador, el núcleo podría transformarse en un cuello de botella. Esto ocurre cuando los procesadores pasan la mayor parte del tiempo esperando poder entrar al núcleo. Por ejemplo esto se puede dar en un multiprocesador que se utiliza como servidor de disco en una red porque la carga está constituida sobretodo por muchos procesos ``demonios´´ que atienden los pedidos provenientes de la red (servidores NFS o *network file system*). Estos procesos demonios corren en SunOS sólo en modo sistema. Por lo tanto sólo uno de estos procesos puede estar ejecutándose en un procesador en un instante dado. El resto de los procesadores está eventualmente bloqueado a la espera del candado para poder entrar al modo sistema.

Si un procesador logra extraer un proceso para hacerlo correr, basta que este proceso haga una llamada al sistema para que el procesador vuelva a bloquearse en el candado por culpa de que otro procesador esta corriendo un proceso NFS en el núcleo.

## Núcleos modernos o multi-threaded

Para evitar definitivamente que el núcleo de un sistema operativo para multiprocesadores se transforme en un cuello de botella se debe permitir que varios procesadores puedan ejecutar procesos ligeros en el núcleo. Además estos procesos ligeros deben ser interrumpibles la mayor parte del tiempo. Para lograrlo se necesita implementar herramientas de sincronización de procesos ligeros en el núcleo. La herramienta usada dependerá de cuanto tiempo dura la operación de la estructura. Cuando se trata de garantizar la exclusión mutua de una estructura de datos para una operación que dura del orden de unos pocos microsegundos se usa un spin-lock. Si la estructura estuviese siendo ocupada por otro procesador, no tiene sentido retomar otro proceso mientras tanto, porque hacer el cambio de contexto tomará varios microsegundos. Además de pedir el spin-lock de la estructura, es eficiente inhibir también las interrupciones, para así evitar que una interrupción del timer provoque un cambio de contexto y el spin-lock permanezca tomado más tiempo de lo que corresponde.

En cambio si la operación es compleja o necesitamos esperar un evento que puede tomar mucho más tiempo que el tiempo que dura un cambio de contexto, entonces sí conviene retomar otro proceso. Por lo tanto, los procesos ligeros del núcleo deben sincronizarse a través de semáforos, mensajes o monitores, los que se implementan a partir de los spin-locks, como veremos a continuación. Al implementar la espera por largos períodos de tiempo haremos que se retome otro proceso.

En esta sección implementaremos mutex y condiciones de bajo nivel para luego implementar un scheduler para un núcleo moderno, seguido de semáforos y monitores de alto nivel. En el caso de las condiciones de bajo nivel, cuando un procesador invoca la operación wait, tendrá que esperar haciendo

busy-waiting en un spin-lock. En cambio para los semáforos de alto nivel, la operación wait retomará otro proceso para hacer un uso eficiente de la CPU.

## Mutex y condiciones de bajo nivel

Esta herramienta de sincronización supone la inexistencia de un sistema operativo y por lo tanto no hay scheduler de proceso, no hay malloc, no hay semáforos, etc. La API a implementar es la siguiente:

| | |
|---|---|
| `LLMutex m;` | Un mutex de bajo nivel |
| `LLCond c;` | Una condición de bajo nivel |
| `void llm_init(LLMutex *m);` | Inicializa el mutex *m* |
| `void llc_init(LLCond *c);` | Inicializa la condición *c* |
| `void llm_lock(LLMutex *m);` | Solicita la propiedad del mutex *m*. |
| `void llm_unlock(LLMutex *m);` | Libera el mutex *m* |
| `void llc_wait(LLCond *c, LLMutex *m);` | Libera el mutex *m* y espera hasta la notificación de *c* y reobtención de la propiedad de *m* |
| `void llc_signal(LLCond *c);` | Notifica *c* |

Observe que esta API es equivalente a los mutex y condiciones presentes en pthreads. En su implementación la exclusión mutua entre *llm_lock* y *llm_unlock* se logra con un spin-lock y por tanto hace busy-waiting. Dado que no existe un scheduler de procesos, la operación *llm_wait* no puede retomar otro proceso, no le queda otra que esperar en un spin-lock haciendo busy-waiting.

```
typedef struct llnode {  // Se usa para implementar llc_wait
  int w;                 // El spin-lock en el que espera el procesador
  struct llmutex *m;     // El mutex que se debe readquirir
  struct llnode *next;   // Sirve para encadenar los nodos
} LLNode;

typedef struct llmutex {  // La estructura de un mutex
  int sl;                 // spin-lock para la exclusión mutua
  LLNode *pend;           // Nodos de procesadores que esperan obtener el spin-lock
} LLMutex;

typedef struct {          // La estructura de una condición
  LLNode *head;           // Lista de nodos de procesadores que esperan notificación
} LLCond;

void llm_init(LLMutex *m) {  // inicialización de un mutex
  m->sl= OPEN;               // parte liberado
  m->pend= NULL;
}

void llc_init(LLCond *c) {   // inicialización de una condición
  c->head= NULL;
}

void llm_lock(LLMutex *m) {  // cierra el mutex
  spinLock(&m->sl);          // cierra el spin-lock sl
}

void llm_unlock(LLMutex *m) {  // abre el mutex
  LLNode *pnode= m->pend;
  if (pnode==NULL)             // caso fácil: no hay condiciones notificadas pendientes
```

```
    spinUnlock(&m->sl);          // se abre el spin-lock
  else {                         // caso interesante: hay condiciones notificadas pendientes
    m->pend= pnode->next;        // se cede la propiedad directamente al procesador notificado
    spinUnlock(&pnode->w);       // retoma la ejecución en el procesador que espera en w
  }
}

void llc_wait(LLCond *c, LLMutex *m) {  // libera el mutex m y espera la condición c
  LLNode node; // Contiene el spin-lock w en donde se hará espera al procesador
               // Como no hay malloc el nodo es una variable local
  node.w= CLOSED;
  node.m= m;

  node.next= c->head; // agrega node al comienzo de la lista encabezada por c->head
  c->head= &node;

  llm_unlock(m);       // abre el mutex m
  spinLock(&node.w);   // espera en w la notificación de la condición c y la reobtención de m
}

void llc_wait_intr(LLMutex *m, LLCond *c) { // libera el mutex m y espera la condición c
  // equivalente a llc_wait pero habilita la interrupciones durante la espera
  LLNode node;
  node.w= CLOSED;
  node.m= m;
  node.next= c->head;
  c->head= &node;
  llm_unlock(m);
  enable();
  spinLock(&node.w);
  disable();
}

void llc_signal(LLCond *c) {  // notifica la condición c
  // No se puede continuar la ejecución en el procesador que espera la condición ya que se debe
  // readquirir el mutex primero.  Por lo tanto la condición queda pendiente en m->pend hasta que
  // se reabra el mutex.
  LLNode *pnode= c->head;
  if (pnode!=NULL) {
    c->head= pnode->next;  // extrae pnode de la lista encabezada por  c->head
    LLMutex *m= pnode->m;
    pnode->next= m->pend;  // agrega pnode a la lista encabezada por m->pend
    m->pend= pnode;
  }
}
```

Observe que el nodo declarado en llc_wait y llc_wait_intr no se ocupa después de que estas funciones retornan. Esto es importante porque el nodo se destruye cuando estas funciones retornan.

## Scheduler para un núcleo moderno

El scheduler implementa Round Robin. Cada proceso dispone de una tajada de tiempo (*slice*). Si pasa a estado de espera (termina su ráfaga) se descuenta el tiempo de CPU ocupado de su tajada. Cuando pasa nuevamente a estado READY, recibe la CPU de inmediato. Cuando su tajada se agota, se envía al final de *ready_queue* pero recupera la tajada de tiempo inicial (*slice*).

Observe que cuando hay múltiples procesadores, no se puede tener una variable global que apunta al proceso en ejecución. En su lugar, la función `getCurrentProcess` entrega el puntero al descriptor

del proceso en ejecución el procesador que lo invoca. Además, el scheduler debe tener un candado asociado a la cola de procesos ready que maneja el scheduler, para evitar que dos procesadores extraigan o encolen al mismo tiempo. Como la cola es manipulada por `kResume`, este procedimiento se debe invocar siempre con el lock de la cola tomado.

La exclusion mutua del scheduler y la *ready_queue* se logra de dos formas: (i) se inhiben las interrupciones para evitar un cambio de contexto mientras se manipula por ejemplo la cola de procesos ready, y (ii) con un mutex de bajo nivel: *sched_mutex*. Típicamente una operacion con el scheduler (*kResume*) implica agregar procesos a *ready_queue* y luego invocar *kResume*. Antes de hacerlo se debe invocar *disable()* para inhibir las interrupciones y *llm_lock(&sched_mutex)*. Después se termina con *llm_unlock(&sched_mutex)* e invoca *enable()* para reactivar las interrupciones.

Cuando se invoca *kResume* desde un procesador y *ready_queue* esta vacía, el procesador correspondiente espera con *llm_wait_intr(&queue_not_empty)*. Cuando se agrega un nuevo proceso a la *ready_queue* se despierta un eventual procesador ocioso invocando *llm_signal(&queue_not_empty)*.

La implementación requiere de las siguientes funciones y definiciones:

| | |
|---|---|
| `int coreId();` | Entrega un entero que identifica el procesador que hace la invocación (varía entre 0 y el número de procesadores disponibles) |
| `void setAlarm(int delay, void (*handler)());` | Configura el cronómetro regresivo para que interrumpa en *delay* milisegundos |
| `void disable();` | inhibe las interrupciones |
| `void enable();` | reactiva las interrupciones |
| `void ChangeContext(Process *out, Process *in);` | Realiza el cambio de contexto del proceso *out* al proceso *in* |
| `typedef struct {`<br>`  int status;`<br>`  int slice;`<br>`  int start_time;`<br>`  …`<br>`} Process;` | El descriptor de proceso |
| `enum { RUN, READY, WAIT_SEM, WAIT_COND, … };` | Los estados de un proceso |
| `typedef struct queue Queue;`<br>`Queue *makeQueue();`<br>`int emptyQueue(Queue *q);`<br>`void putProcess(Queue *q, Process *p);`<br>`void pushProcess(Queue *q, Process *p);`<br>`Process *getProcess(Queue *q);` | La API para el manejo de colas FIFO |

Se implementarán las siguientes funciones:

| | |
|---|---|
| `Process *currentProcess()` | Entrega el proceso en ejecución en el procesador que lo invoca |
| `void setCurrentProcess(Process *p);` | Estable *p* como el proceso en ejecución |
| `void kResume();` | El scheduler: se invoca para retomar un nuevo proceso |
| `void timerHandler();` | Rutina de atención para el cronómetro regresivo |

Variables globales del scheduler:

| | |
|---|---|
| `LLMutex sched_mutex` | Para asegurar la exclusión mutua de los procesadores al acceder al scheduler |
| `Queue *ready_queue;` | La cola de procesos en estado READY |
| `LLCond queue_not_empty;` | Para que un procesador espere cuando encuentra *ready_queue* vacía |
| `int slice;` | El tamaño de la tajada de CPU para Round Robin |
| `Process *processes[NCORES];` | Arreglo de punteros a los descriptores de proceso en estado RUN |
| `int avail= 0;` | Número de procesadores que esperan en *queue_not_empty* |

```
Process *currentProcess() {
  return processes[coreId()];
}

void setCurrentProcess(void *p) {
  processes[coreId()]= p;
}

void kResume() { // El scheduler
  Process *this= currentProcess();
  Process *next= NULL;

  // se descuenta el tiempo ocupado de su tajada
  this->slice -= getTime() - this->start_time;

  // Ojo: slice puede resultar <= 0 cuando se invoca desde timerHandler o si la tajada de this expiró
  // cuando las interrupciones estaban inhibidas y entonces se llama a kResume

  while (next==NULL) {
    next= getProcess(ready_queue);
    if (next==NULL) {
      setAlarm(0, NULL); // desactiva el timer para las tajadas de tiempo
      setCurrentProcess(NULL);
      avail++;
      llc_wait_intr(&sched_mutex, &queue_not_empty);
      avail--;
    }
    else if (next->slice<=0) {
      next->slice= slice;              // se estable una tajada completa
      putProcess(ready_queue, next); // se lanza al final de la cola
      next= NULL;                      // para que se considere un nuevo proceso de la cola
    }
  }

  ChangeContext(this, next);

  this->start_time= getTime();
  setCurrentProcess(this);
  this->status= RUN;
```

```
  setAlarm(this->slice, timerHandler); /* 1 alarma x core */
}

void timerHandler() { // Rutina de atención para el cronómetro regresivo
  // se invoca con interrupciones inhibidas
  Process *this= currentProcess();
  if (this==NULL) {
    // Nunca se debe llamar a scheduler cuando this==NULL, porque ya estamos
    // en el scheduler.  Sería una invocación recursiva con resultados impredecibles.
    // Hay que tener esta precaución en toda rutina de atención de interrupciones.
    return; // Solo ocurre cuando la interrupción del timer quedo pendiente
  }
  llm_lock(&sched_mutex);           // se adquiere el mutex del scheduler
  this->status= READY;
  this->slice= 0;                   // asegura que se le acabó la tajada
  pushProcess(ready_queue, this);   // se coloca en 1er. lugar

  kResume();                        // se elige un nuevo proceso para ejecutar
    // El primer proceso que kResume obtendrá es este mismo proceso, pero como su tajada estará
    // agotada, lo enviará al final de la cola, con una nueva tajada completa.

  llm_unlock(&sched_mutex);         // se devuelve el mutex del scheduler
}
```

Advertencia: este scheduler nunca ha sido probado. Puede tener bugs. Si Ud. encuentra alguna, me avisa por favor.

## Implementación de semáforos en un núcleo multi-threaded

Un semáforo del núcleo se implementa a partir de un monitor de bajo nivel. Se necesita para controlar el acceso exclusivo a la estructura de datos que representa el semáforo. Las operaciones sobre semáforos las llamaremos `kInitSem`, `kWaitSem` y `kSignalSem` para indicar que se llaman en el *kernel* (núcleo). El semáforo se representa mediante la siguiente estructura:

```
typedef struct {
  LLMutex m;    // controla la exclusión mutua del semáforo
  int count;    // número de tickets del semáforo
  Queue *queue;
} kSem;

void kInitSem(kSem *sem, int tickets) {
  llm_init(&sem->m);
  sem->count= tickets;
  sem->queue= makeQueue();
}
```

Esta es la implementación de la solicitud de un ticket:

```
void kWaitSem(kSem *sem) {
  disable();              // inhibe las interrupciones
  llm_lock(&sem->m);      // se adquiere el mutex del semáforo
  if (sem->count>0) {
    sem->count--;         // caso fácil: quedan tickets
    llm_unlock(&sem->m); // se devuelve el mutex del semáforo
  }
  else {
    // caso interesante: hay que suspender el proceso
    Process *this= currentProcess();
```

```
    this->status= WAIT_SEM;
    putProcess(sem->queue, this);
    llm_unlock(&sem->m); // se devuelve el mutex del semaforo

    // Invoca al scheduler para retomar otro procesos
    llm_lock(&sched_mutex);
    kResume();           // se retoma un nuevo proceso
    llm_unlock(&sched_mutex);
  }
  enable();              // activa las interrupciones
}
```

El principio es que el mutex *m* permanece cerrado sólo cuando un procesador está manipulando el semáforo. Si el contador está en 0, el proceso se coloca en la cola del semáforo, se abre nuevamente el mutex y el procesador llama al scheduler para retomar un nuevo proceso. En ningún caso, el procesador espera en un *spin-lock* hasta que el contador del semáforo se haga positivo. De esta forma, el tiempo que permanece cerrado el mutex es de una centena de nanosegundos, muy inferior al tiempo que puede permanecer cerrado el candado único de un núcleo clásico.

El código para *kSignalSem* es:

```
void kSignalSem(kSem *sem) {
  disable();             // inhibe las interrupciones
  llm_lock(&sem->m);     // se adquiere el mutex del semáforo
  if (emptyQueue(sem->queue)) {
    sem->count++;        // caso fácil: nadie espera este ticket
    llm_unlock(&sem->m); // se devuelve el mutex del semáforo
  }
  else {
    Process *w= getProcess(sem->queue);
    Process *curr= currentProcess();
    llm_unlock(&sem->m);
    w->status= READY;
    curr->status= READY;

    // Agregamos el proceso actual y w al comienzo de la cola ready
    llm_lock(&sched_mutex);      // se devuelve el mutex del semáforo
    pushProcess(ready_queue, curr);
    pushProcess(ready_queue, w); // A

    // Invoca al scheduler para retomar w
    llc_signal(&queue_not_empty);
    kResume();
    llm_unlock(&sched_mutex);
  }
  enable();              // activa las interrupciones
}
```

Observe que en A hay al menos 2 procesos ready: el procesos w, que está siendo despertado, y el que invocó `kSignalSem`, que entrega `kCurrentProcess`. Entonces se invoca el scheduler, el que elige alguno de los 2 para continuar ejecutándose. Sin embargo, en ese momento puede existir un core ocioso, porque la cola de procesos estaba vacía antes de invocar `kSignalSem`. Entonces la llamada a llc_signal despertará el core ocioso para que retome el segundo proceso.

Observe que en este diseño una herramienta de sincronización debe obtener el candado del scheduler explícitamente antes de invocar `kResume`, y liberarlo a su retorno. Se hace así porque normalmente

junto con invocar `kResume` también se necesitará agregar procesos a `ready_queue`, como ocurre en `kSignalSem`. En otros diseños de núcleos `kResume` sí podría obtener y liberar el candado del scheduler por sí mismo. En este caso sería un error pedir el candado antes de llamar a `kResume` y por lo tanto si se necesita agregar procesos a `ready_queue` se debería obtener el candado y liberarlo antes de invocar `kResume`.

El problema de esta implementacion es que si hay un core adicional disponible, el proceso que invoca kSignal migrará a ese core adicional, porque el core en el que se ejecutaba previamente se ocupara de w. Esto es ineficiente, porque al migrar perdera su datos en el cache L1 (y L2 en algunas arquitecturas).

***Ejercicio***: modifique el semáforo para que no ocurra esa migración. Use *avail* para averiguar si existe o no ese core adicional. En la implementación de monitores que viene a continuación se programó la misma optimización.

## Implementación de monitores en un núcleo multi-threaded

Estos monitores se diferencian de los mutex/condiciones vistos previamente en que cuando un proceso invoca la operación wait, se retoma otro proceso en vez de hacer busy-waiting. Las operaciones que se implementarán son equivalentes a los monitores de nSystem:

| | |
|---|---|
| `kMonitor m;` | Un monitor |
| `void kInitMonitor(kMonitor *m);` | Inicializa el monitor *m* |
| `void kEnter(kMonitor *m);` | Obtiene la propiedad del monitor *m* |
| `void kExit(kMonitor *m);` | Devuelve el monitor *m* |
| `void kWait(kMonitor *m);` | Espera hasta la notificación en *m*. |
| `void kNotifyAll(kMonitor *m);` | Notifica todos los procesos en espera de *m* |

Se usa un LLMutex para garantizar la exclusion mutua entre *kEnter* y *kExit*. Por lo tanto, la operacion realizada entre *kEnter* y *kExit* debe ser de corta duración (menos de 10 microsegundos). Entre *kEnter* y *kExit* se desactivan las interrupciones porque el scheduler requiere que estén desactivadas. Además un cambio de contexto podría alargar el tiempo en que el monitor permanece tomado a muchos milisegundos.

Cuando un proceso invoca *kEnter* y el monitor está tomado, se espera invocando *llm_lock*, y por lo tanto hay busy-waiting. Cuando se libera el monitor y hay un proceso esperando adquirir el monitor despues de un kWait, se retoma inmediatamente ese proceso.

La implementación que se verá para un monitor interno al núcleo usará la siguiente estructura:

```
typedef struct {
  LLMutex m;    // exclusion mutua del monitor
  Queue *wq;    // procesos que invocaron kWait
  Queue *mq;    // procesos que esperan adquirir el monitor
} kMonitor;

void kInitMonitor(kMonitor *mon) {
  llm_init(&mon->m);
  mon->wq= makeQueue();
  mon->mq= makeQueue();
}
```

El principio es que el tiempo entre las invocaciones de `kEnter` y `kExit` debe ser breve y por lo tanto

la exclusión mutua en el acceso al monitor se implementa mediante un mutex de bajo nivel, representado por la variable `m`. Esto significa que si hay contención, habrá busy-waiting, pero se espera que el tiempo de espera sea inferior al tiempo de CPU que cuesta hacer un cambio de contexto y por eso es más eficiente el busy-waiting.

Los procesos que invocaron `kWait` esperan en la cola `wq`. En un `kNotifyAll` se mueven todos los procesos en `wq` a la cola de procesos `mq`. En `kExit` se revisa la cola `mq` y si existen procesos pendientes se le cede el monitor directamente al primero de ellos.

La operación `kEnter` es la más simple de implementar:

```
void kEnter(kMonitor *mon) {
  disable(); // inhibe las interrupciones
  llm_lock(&mon->m);
}
```

La implementación de `kExit` es más delicada porque si hubo un notify, habrán 2 procesos listos para ejecutarse. Si hay 2 procesadores libres para ejecutarlos, se procurará que el proceso actual continúe ejecutándose en el mismo procesador previo.

```
void kExit(kMonitor *mon) {
  if (!emptyQueue(mon->mq)) { // caso fácil: nadie espera el monitor
    llm_unlock(&mon->m);
  }
  else {
    // caso interesante: hay un proceso esperando el monitor
    Process *this= currentProcess();
    Process *w= getProcess(mon->mq);
    this->status= READY;
    w->status= READY;

    llm_lock(&sched_mutex);

    if (avail>=1) {
      // si hay un core disponible, el llc_signal lo despertará
      // y ejecutará w. El core actual continuará ejecutado this.
      pushProcess(ready_queue, w);
      llc_signal(&queue_not_empty);
    }
    else {
      // Si no hay otro core disponible, este mismo core deberá
      // ejecutar w y dejar this en la cola ready.
      pushProcess(ready_queue, this);
      pushProcess(ready_queue, w);
      kResume();
    }
    llm_unlock(&sched_mutex);
  }
  enable(); // activa las interrupciones
}
```

La operación *kWait* se usa para esperas de tiempo prolongado y por lo tanto se implementa retomando otro proceso. He aquí la implementación de *kWait*:

```
void kWait(kMonitor *mon) {
  Process *this= currentProcess();
  this->status= WAIT_MON;
```

```
  putProcess(mon->wq, this);

  if (!emptyQueue(mon->mq)) { // caso fácil: nadie espera el monitor
    llm_unlock(&mon->m);
    llm_lock(&sched_mutex);
  }
  else { // caso interesante: hay un proceso esperando el monitor
    Process *w= getProcess(mon->mq);
    w->status= READY;

    llm_lock(&sched_mutex);
    pushProcess(ready_queue, w);
    // el kResume de más abajo retomará w!
  }

  kResume(); // Retorna cuando se emitió el notify
  llm_unlock(&sched_mutex);
}
```

Cuando se emite el notify, no tiene sentido tratar de retomar ningún proceso todavía porque no podrán continuar hasta que el proceso actual libere el monitor con `kExit`. Esta es la implementación de `kNotifyAll`:

```
void kNotifyAll(kMonitor *mon) {
  // se transfieren todos los procesos de mon->wq a mon->mq
  while (!emptyQueue(mon->wq)) {
    Process *w= getProcess(mon->wq);
    pushProcess(mon->mq, w);
  }
}
```

Por lo tanto solo agregamos los procesos en espera a la cola de procesos *mon->mq*. Uno de estos comenzarán a ejecutarse después de *kExit*.

## Ejercicios

- Implemente el tipo `kCondition` para representar las condiciones de los monitores de Hoare y las operaciones `kWaitCondition` y `kSignalCondition`.
- Resuelva el control 2 del semestre Otoño de 2012 publicado en: http://users.dcc.uchile.cl/~lmateu/CC4302/controles/c2-121.pdf.
- Resuelva el control 2 del semestre Otoño de 2013 publicado en: http://users.dcc.uchile.cl/~lmateu/CC4302/controles/c2-131.pdf.

## Resumen

Los núcleo modernos permiten que múltiples procesos ligeros se ejecuten en paralelo dentro del núcleo. Para garantizar la exclusión mutua de estructuras de datos de acceso rápido se usan spin-locks y para estructuras más complejas se recurre a semáforos o monitores, implementados a partir de spin-locks. Ejemplos de sistemas operativos que usan este esquema son Solaris 2, Unix System V.4, Window NT y Mach (Next y OSF/1) y todos los sistemas operativos más recientes. Por eso se les llama *núcleos modernos*. También se les llama *núcleos multi-threaded* porque pueden correr en paralelo varios threads en el núcleo (y pueden ser interrumpidos), a diferencia de un núcleo clásico en donde los threads no pueden correr en paralelo.