PROCESOS

INTRODUCCIÓN

Ya vimos antes que un proceso es un programa en ejecución.

Nos vamos a ocupar ahora de cómo el sistema operativo se encarga de controlar todos los procesos cargados en memoria, de la asignación del procesador o CPU (Unidad Central de Procesos) a cada uno de los procesos, y, cuando se trate de procesos simultáneos, de la comunicación y sincronización entre ellos.

Sabemos que la CPU realiza ciertas actividades. En un sistema de tratamiento por lotes (*batch*) se ejecutan trabajos; en un entorno de tiempo compartido hay programas de usuarios o tareas; incluso en un ordenador personal, hoy día un usuario puede ejecutar varios programas simultáneamente, uno interactivo y otros en segundo plano. Estas actividades que realiza las denominaremos *procesos*.

1. Estados de un proceso.

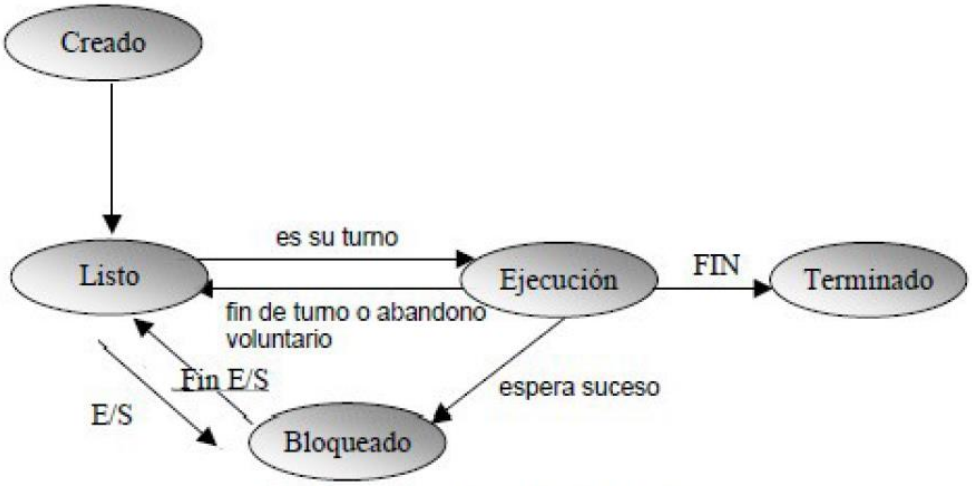

Figura 1 Estados de un proceso.

El *ciclo de vida de* un proceso podría estar caracterizado por esta secuencia:

1. Se está ejecutando (posee la CPU).
2. Realiza una operación de E/S y se pone a esperar la respuesta (abandona la CPU).
3. Cuando recibe la respuesta de la operación de E/S desea continuar la ejecución (necesita otra vez la CPU).

En un principio, un proceso no existe. En algún momento es *creado*.

El sistema operativo es el encargado de crear los nuevos procesos siguiendo las directrices del usuario. La puesta en ejecución de un nuevo proceso se produce debido a que hay un proceso en concreto que está pidiendo su creación en su nombre o en nombre del usuario. Cualquier proceso en ejecución (**proceso hijo**) siempre depende del proceso que lo creó (**proceso padre**), estableciéndose un vínculo entre ambos. A su vez, el nuevo proceso puede crear nuevos procesos, formándose lo que se denomina un **árbol de procesos.**

Una vez creado el proceso, este pasa al estado denominado **Listo**. Este estado significa que el proceso está en condiciones de hacer uso de la CPU en cuanto se le dé la oportunidad. El encargado

de darle la oportunidad de usar la CPU es el denominado **planificador de procesos o scheduler**, que suele formar parte del SO. Como lo normal es que haya más procesos que procesadores, no todos los procesos que pueden hacer uso de la CPU en un momento determinado pueden hacerlo realmente. Esos procesos permanecen en el estado listo hasta que el planificador decide darles tiempo de CPU. Cuando el planificador decide dar tiempo de CPU a un proceso, este pasa al estado de ***Ejecución*.**

Un proceso también puede pasar de **ejecución** a ***Listo***. Esta decisión la toma el planificador. El planificador sigue algún tipo de política de planificación para asignar la CPU a los distintos procesos. Una forma bastante justa y extendida de hacerlo es mediante la asignación de franjas de tiempo a cada uno de los procesos, de tal forma que cuando un proceso cumple su tiempo de permanencia en el procesador, este es desalojado y pasado al estado listo. En este estado esperará una nueva oportunidad para pasar a ejecución.

También es posible que un proceso abandone voluntariamente la CPU y pase de esta forma al estado listo.

Un proceso puede pasar de **ejecución** a ***Bloqueado*** cuando ha de esperar porque ocurra un determinado evento o suceso. Ejemplos de eventos pueden ser la espera por la terminación de una operación de Entrada/Salida, la espera por la finalización de una tarea por parte de otro proceso, un bloqueo voluntario durante un cierto periodo de tiempo, etc. Una vez que ocurre el evento por el que se está esperando, el proceso pasa al estado **Listo**.

Al acto de cambiar un proceso de estado se le denomina **cambio de contexto.**

Vistos los estados que puede tener un proceso y las transiciones entre ellos, nos interesa saber cómo un sistema operativo lleva el control de todos los procesos creados, cómo se realizan estas colas y cómo se sabe qué proceso está ejecutándose. Para ello, el sistema operativo maneja una estructura de datos que contiene toda la información de control que se requiere para ocuparse de la gestión o administración de un proceso. Nos estamos refiriendo al **Descriptor de Proceso o Bloque de Control de Proceso (BCP)**. Pues bien, entonces lo que el sistema operativo necesita es mantener la colección de todos los BCP's de los procesos existentes en cada momento. Esta colección de BCP's se puede organizar de distintas maneras: mediante una cola estática, con una cola dinámica, o simplemente utilizando una tabla, es decir, un vector o array de BCP's.

2. Comunicación y sincronización entre procesos.

Normalmente, los procesos que cooperan en la realización de un trabajo, necesitarán durante su ciclo de vida la **sincronización o comunicación** entre ellos o con el proceso maestro que les haya creado. Se hace necesario algún sistema de comunicación cuando se requiere el intercambio de información entre dos o más procesos.

En otros casos, como en los momentos en que se accede a estructuras de datos comunes a varios procesos, se requiere algún mecanismo de sincronización para que dos procesos no intenten acceder simultáneamente a dicha estructura para realizar operaciones de lectura/escritura de forma incontrolada.

A continuación, se estudiarán los problemas que pueden darse y algunos mecanismos propuestos para comunicación y sincronización entre procesos simultáneos que comparten el mismo espacio lógico de direcciones.

2.1 *Región crítica y exclusión mutua:*

Cuando dos o más procesos se ejecutan en paralelo, o e un pseudoparalelismo alternándose el uso de la CPU, pueden producirse algunos problemas al acceder a recursos compartidos.

Veamos el siguiente ejemplo.

```
Productor
Repetir
        Leer en temporalP:  contador
        Producir elemento: temporalP+1
        Asignar a contador:  temporalP+1
Siempre
```

```
Consumidor
Repetir
        Leer en temporalC: contador
        Si contador>0
                Asignar  a  contador:  temporalC-1
                Consumir elemento: temporalC
Siempre
```

Se trata de la simulación de un productor y de un consumidor, los dos procesos acceden a un recurso compartido denominado ***contador***. Esta variable esta inicialmente puesta a cero y su objetivo es contar el número de elementos que ha producido el productor y que aún no ha consumido el consumidor.

Cada vez que se ejecuta el productor, lee el contenido de la variable, produce un nuevo elemento e incrementa contador en una unidad.

Cada vez que se ejecuta el consumidor, lee el contenido de la variable y si es mayor que cero (hay algún elemento producido) decrementa en una unidad el contador y consume el elemento. Basta echar un simple vistazo para comprobar que cuando los dos procesos se alternan el uso de la CPU puede darse el caso de que se pierdan elementos producidos, por ejemplo, supongamos que contador vale 3 y que se produce la siguiente secuencia:

Proceso	***Instrucción que se ejecuta***
Productor	*Leer en temporalP: contador //le asigna 3 a temporalP*
Productor	*Producir elemento: temporalP+1 //produce el elemento 4*
Consumidor	*Leer en temporalC: contador //le asigna 3 a temporalC*
Consumidor	*Si contador >0 //es true*
Consumidor	*Asignar a contador: temporalC-1 //asigna a contador 2*
Consumidor	*Consumir elemento: temporalC //consume el elemento 3*
Productor	*Asignar a contador: temporalP+1 //asigna a contador 4*

En este caso, habiendo tres elementos, consumiendo uno y produciendo otro quedan al final cuatro elementos.

La clave para evitar problemas en las situaciones en que se comparten objetos o recursos es encontrar algún modo de prohibir que haya más de un proceso leyendo o escribiendo el dato compartido al mismo tiempo. En otras palabras, lo que se necesita es ***exclusión mutua***, es decir, asegurarse de que si un proceso está accediendo a un dato compartido, el otro o los otros procesos estarán imposibilitados para poder hacerlo también. En la vida de un proceso, parte del tiempo está ocupado con cálculos internos y otro tipo de cosas que no conducen a una situación como la anterior. Pero en otras ocasiones, el proceso accede a variables o ficheros compartidos que sí pueden conducir a este tipo de situaciones. La parte del programa en la que se accede a memoria compartida se denomina **región crítica**.

Resumiendo:

- **Exclusión mutua.** Si un proceso está accediendo a un dato o recurso compartido, los otros procesos deben estar imposibilitados. Es decir, no podrán acceder a ese dato o recurso hasta que lo libere el proceso que lo tiene en uso (¿qué pasaría si dos procesos imprimiesen a la vez, alternándose?).

- **Región crítica.** Es la parte de un programa en la que se accede a un dato o recurso compartido que requiere exclusión mutua. Es el trozo de código del que se tiene que encargar el programador para que dos procesos no lo ejecuten simultáneamente.

Para resolver el problema anterior estaría bien disponer de un mecanismo de acceso a la región crítica que se comportase como un "portero", tal que si un proceso intentara entrar en la región crítica ocupada por otro proceso, el primero quedara bloqueado por el portero hasta que el otro proceso saliese de ella.

Aunque con la exclusión mutua se evitan las condiciones de carrera, no es condición suficiente para que una serie de procesos paralelos cooperen correcta y eficientemente utilizando datos compartidos. Se requieren las siguientes condiciones para obtener una buena solución:

1. Dos procesos no pueden estar al mismo tiempo dentro de la misma región crítica.
2. No se deben hacer suposiciones sobre el hardware, es decir, sobre la velocidad o el número de procesadores que intervienen.
3. No se deben hacer suposiciones sobre las velocidades relativas de los procesos.
4. Ningún proceso fuera de su región crítica puede bloquear a otros procesos.
5. Ningún proceso deberá esperar eternamente a entrar en su región crítica.
6. No se debe posponer indefinidamente la decisión de cuál es el siguiente proceso que debe entrar.

2.2 *Mecanismos para resolver la exclusión mutua*

En este apartado examinaremos algunos de los muy diversos métodos de sincronización que existen para resolver el problema de la exclusión mutua. Estos son:

- Inhibición de interrupciones
- Cerrojos (espera activa).
- Semáforos
- Monitores
- Paso de mensajes

INHIBICIÓN DE INTERRUPCIONES

Si se pretende evitar que mientras un proceso está dentro de una región crítica, otro proceso pueda entrar también, la forma más fácil de conseguirlo es evitando que el proceso que se encuentra dentro de la región pierda el control del procesador antes de abandonarla. Un proceso pierde la posesión de la CPU solamente si realiza una operación de E/S o, si debido a una interrupción, se detecta que se le acabo su porción de tiempo o que otro proceso de mayor prioridad pasa a **Preparado** y seguidamente se apropia de la CPU. Sabido esto, por una parte, se puede hacer que un proceso que se encuentra dentro de una región crítica no realice operaciones de E/S, y por otra parte lo que falta por hacer es inhibir la aceptación de interrupciones por parte del procesador mientras el proceso está dentro de la región. Es decir, que las sentencias a ejecutar antes de entrar y al salir de la región crítica se podrían corresponder con DISABLE (inhibir interrupciones) y ENABLE (aceptar interrupciones). Así, una vez que un proceso ha inhibido las interrupciones, puede examinar y actualizar la memoria compartida sin miedo a que otro proceso intervenga, pues la secuencia de instrucciones que componen la región crítica se convierte en una única operación atómica o indivisible.

La ventaja de este sistema es su simplicidad. Sin embargo, también da lugar a varios inconvenientes:

- La región crítica debe ser corta, pues de lo contrario se podrían perder interrupciones generadas por los dispositivos de E/S si no se tratan a tiempo.
- Inhibiendo las interrupciones, no solamente se impide que otro proceso simultáneo que desee entrar en la región crítica pueda continuar su ejecución, sino también a procesos simultáneos que

no intentan acceder en la región crítica. Y lo que es peor, también se impide que prosigan el resto de los procesos del sistema, que, por no ser simultáneos, en ningún momento intentarán acceder a los datos compartidos que se están protegiendo.

- Resulta peligroso darle al usuario un medio para inhibir las interrupciones, pues podría ocurrir que por un error de programación o por cualquier otro motivo no las volviera a permitir, provocando la consiguiente pérdida de las subsiguientes interrupciones y el acaparamiento en exclusiva de la CPU.

CERROJOS

Otra posibilidad de impedir el acceso simultáneo de dos procesos a la región crítica es haciendo que una variable actúe como cerrojo o indicador de si la región crítica está ocupada o no. Esto se puede hacer con una variable booleana que tome los valores 0 (libre) y 1 (ocupada). Todos los procesos que deseen acceder a una región crítica deben cumplir un protocolo de acceso y salida de la región.

```
if (libre ==true)
        libre = false
        Región crítica
```

Cuando un proceso llegue a la entrada de la región debe consultar el valor de la variable cerrojo; si el valor es ***true***, la pone a ***false*** y entra en la región crítica; si la variable ya estaba a ***false***, el proceso debe esperar hasta que valga ***true***. Supongamos que un Proceso A lee el cerrojo y ve que está a *true*.

Antes de que pueda ponerlo a ***false***, pierde el control de la CPU y lo toma otro Proceso B que también quiere entrar en la región crítica. Este último consulta también el valor del cerrojo (que sigue a ***true***), lo pone a ***false*** y entra en la región. Antes de salir de la región crítica, finaliza su porción de tiempo y se le devuelve el control al Proceso A, que continúa lo que estaba haciendo, con lo que pone el cerrojo a ***false*** (otra vez) y entra en la región crítica, creyendo que él es el único que está dentro, porque ha entrado siguiendo el protocolo de acceso.

Como vemos, el problema se debe a que la consulta del cerrojo y su posterior puesta a ***false*** (ocupado) no se realiza como una acción atómica. Para solucionar esto han surgido diversos algoritmos, entre ellos, el de la panadería (de Lamport); el algoritmo de Dekker (para 2 procesos); el de Dijkstra (para n procesos); el de Knuth, Eisemberg y McGuire; y el de Peterson.

Se consigue una buena solución con un poco de ayuda hardware. Los procesadores de propósito general suelen ofrecer la siguiente instrucción maquina (por lo tanto, ininterrumpible en su ejecución):

TAS (Cerrojo, Valor).

Esta instrucción, ***Test And Set Lock***, permite hacer las dos instrucciones que hacíamos antes, **(*if (libre ==true)* y *libre = false*)** en un solo paso, sin ninguna interrupción. De esta forma puede controlarse el acceso exclusivo a la región crítica de la siguiente manera:

TAS(libre, false) mientras libre distinto de true

Región crítica

TAS(libre, true)

La primera instrucción se repite mientras ***libre*** es distinto de ***false*** (el otro proceso está en la región crítica). Lo que hace es leer el valor de ***libre*** y si es ***true*** le asigna ***false***. En el momento en que el otro proceso sale de la región crítica ejecuta la última instrucción poniendo ***libre*** igual a ***true***, entonces el primer proceso puede poner ***libre*** a ***false*** y entrar.

SEMÁFOROS

E.W. Dijkstra propuso en 1965 el Semáforo como mecanismo de sincronización. Una de sus principales peculiaridades es que evita la **espera activ**a. Cuando un proceso tenga que esperar lo que en realidad se hace es un bucle vacío hasta que se cumpla una condición, por lo tanto el proceso

está consumiendo recursos de CPU para "no hacer nada". La idea era tratar las regiones críticas como recursos de acceso exclusivo por los que compiten los procesos, de tal manera que el que consigue el permiso entra en la región crítica, y el resto de los procesos quedan en estado de Espera hasta que les llegue el turno de acceso.

Un Semáforo es un tipo abstracto de datos con dos operaciones básicas más una de inicialización ***(Init)***. A las dos operaciones básicas las llamaremos ***Bajar (Wait)***y ***Subir (Signal)***. Para cada región crítica se declara una variable de tipo Semáforo que identifica a la región como un recurso compartido.

Estas dos operaciones básicas se corresponden con los protocolos de acceso y salida de la región crítica, tal que para entrar en ella hay que ejecutar una operación ***Bajar*** sobre el Semáforo asociado a esa región, y a la salida debe llamarse a la instrucción ***Subir*** sobre el mismo Semáforo.

Básicamente el funcionamiento de las instrucciones es el siguiente:

- ***Init(n)*:** inicializa el Semáforo. Debe hacerse al declarar el Semáforo y se le asigna el número de procesos que pueden entrar simultáneamente en la región crítica.
- ***Bajar, Wait*:** si el Semáforo tiene un valor mayor que cero, le resta uno y el proceso puede continuar. Si el Semáforo vale cero, entonces el proceso se queda bloqueado hasta que otro haga un *Subir*.
- ***Subir, Signal*:** se le suma uno al Semáforo.

La solución para controlar el acceso exclusivo a una región crítica mediante Semáforos seria la siguiente:

```
Semáforo.Init(1)   //Se inicializa el semáforo para que sólo pueda entrar un proceso

P0:                          P1:
Semáforo.Wait()              Semáforo.Wait()
Región crítica               Región crítica
Semáforo.Signal()            Semáforo.Signal()
```

El funcionamiento es el siguiente: inicialmente el Semáforo esta puesto a uno, el primer proceso que se ejecute hará un ***Wait*** y dejará el Semáforo a cero. Mientras está en la región crítica, si el otro proceso quiere entrar tendrá que hacer un ***Wait*** y se quedará bloqueado (dormido) porque el Semáforo está a cero. Cuando el primer proceso abandona la región crítica hará un ***Signal*** que coloca de nuevo el Semáforo a uno y permite que el otro proceso se despierte.

Lógicamente, también el Semáforo necesita un acceso compartido, básicamente un Semáforo se implementa con un contador y una lista de procesos bloqueados. Las operaciones ***Wait*** y ***Signal*** sobre un Semáforo deben tener un acceso exclusivo. Los Semáforos son una estructura de datos que pueden proporcionar los sistemas operativos, y por lo tanto la forma en que se consiga esto es responsabilidad del sistema operativo, siendo el programador responsable únicamente de su utilización.

Los Semáforos tienen varias ventajas, su utilización es sencilla y además eliminan la espera activa, un proceso que se duerme no consume recursos. No obstante, se sigue manteniendo uno de los problemas de los cerrojos, esto es, sigue siendo responsabilidad del programador el uso adecuado de las primitivas de sincronización, o sea, *Bajar* para entrar en la región crítica, y *Subir* para salir de ella. Si inadvertidamente no se sigue estrictamente el protocolo de entrada y salida de la región, o no inicializa adecuadamente el Semáforo, el sistema falla.

MONITORES

Para evitar el problema de los descuidos del programador a la hora de seguir el protocolo estipulado para acceder y salir de una región crítica, surgen los ***monitores***. Básicamente, un monitor es una

colección de procedimientos y datos, agrupados en una especie de módulo muy especial conocido como módulo monitor. Los procesos pueden llamar a los procedimientos del monitor siempre que lo deseen, pero no pueden acceder directamente a las estructuras de datos internas del monitor desde procedimientos declarados fuera del monitor. Las estructuras de datos escondidas en el monitor solamente están accesibles por los procedimientos del monitor.

Los monitores tienen una propiedad especial para conseguir la exclusión mutua: ***"solamente un proceso puede estar activo a la vez dentro de un monitor"***. Dicho de otra forma, dado un proceso A que ha llamado a un procedimiento de un monitor, y mientras se está ejecutando dicho procedimiento, toma el control de la CPU otro proceso B, si este proceso B llama a cualquier procedimiento del monitor, el proceso B quedará detenido en la entrada (en estado de Espera) hasta que el proceso A abandone el monitor.

Los monitores son una construcción de los lenguajes de programación, de tal forma que los compiladores saben que las llamadas a los procedimientos de un monitor se manejan de forma distinta a las llamadas a los procedimientos convencionales. Normalmente, en la llamada de un proceso a un procedimiento de un monitor, las primeras instrucciones del procedimiento son para comprobar si algún otro proceso se encuentra dentro del monitor; si es así, el proceso llamante queda bloqueado hasta que el otro proceso salga del monitor. Si no hay ningún proceso dentro del monitor, el proceso llamante puede entrar y continuar la ejecución. La implementación de la exclusión mutua en las entradas del monitor es labor del compilador.

Ya que es el compilador, no el programador, el que se ocupa de cumplir el protocolo de acceso a la región crítica, es mucho más difícil que algo se haga mal o que se produzca algún olvido. En cualquier caso, la persona que escribe un monitor no tiene por qué saber cómo implementa la exclusión mutua el compilador. Le basta con saber que, metiendo las regiones críticas en monitores, nunca habrá más de un proceso dentro de la misma región crítica al mismo tiempo.

PASO DE MENSAJES

Aunque los Semáforos y los monitores son buenos mecanismos para la sincronización de procesos todavía tienen algunas pegas: los Semáforos son de demasiado bajo nivel y los monitores solamente están disponibles en unos pocos lenguajes de programación.

Otro problema con los monitores y los Semáforos es que están diseñados para resolver el problema de la exclusión mutua en sistemas con una o más CPU's que comparten una memoria común.

Pero cuando se trata de un sistema distribuido, formado por múltiples procesadores, cada uno con su propia memoria particular y conectados por una red de área local, estas primitivas se vuelven inservibles, pues ya no hay variables compartidas, y ninguno de estos mecanismos proporciona intercambio de información entre maquinas. Se necesita algo más. Ese algo más es el ***paso de mensajes***.

La función del paso de mensajes es permitir que dos procesos se comuniquen sin necesidad de utilizar variables compartidas.

Este método de comunicación entre procesos ofrece, al menos, dos primitivas: ***Enviar (Send)*** y ***Recibir (Receive)***, que, a diferencia de los monitores, no son construcciones del lenguaje de programación, sino Llamadas al Sistema, por lo que para utilizarlas basta con llamar a los correspondientes procedimientos de biblioteca. Mediante ***Enviar*** se expide un mensaje a un proceso destino, mientras que con ***Recibir*** se indica el deseo de recibir un mensaje de algún proceso fuente. Si no hay ningún mensaje disponible, el proceso receptor puede quedarse bloqueado hasta que llegue uno.