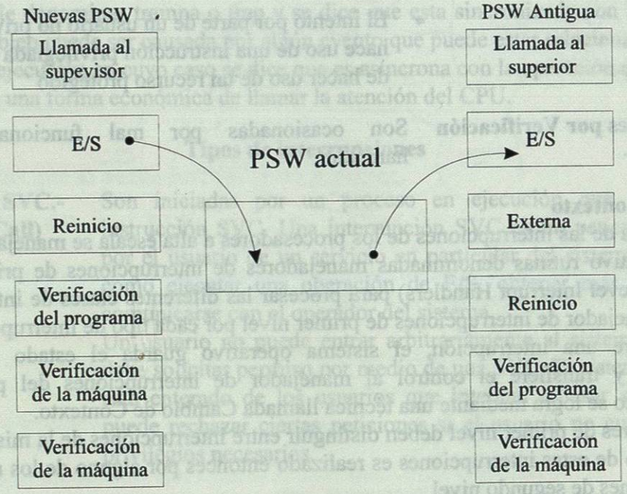
La PSW nueva para algún tipo de interrupción contiene la dirección permanente de la memoria principal en la que reside el manejador de interrupciones correspondiente.
Cuando ocurre una interrupción, si el procesador no esta inhabilitado para este tipo de interrupción, el hardware automáticamente cambia la PSW como sigue:

1. Se guarda la PSW actual en la PSW antigua de ese tipo de interrupción.
2. Se guarda la PSW nueva de ese tipo de interrupción en la PSW actual.

Después de este intercambio de PSW, la PSW actual contendrá la dirección del manejador de interrupciones apropiada y este ejecutara y procesara la interrupción.



Núcleo del Sistema Operativo (Nucleus/Kernel/Core).
Todas las operaciones en las que participan procesos son controladas por la parte del sistema operativo llamada núcleo. El núcleo normalmente representa solo una pequeña parte de lo que por lo general se piensa que es todo el sistema operativo pero también es el código que más se utiliza. Reside por lo regular en la memoria principal mientras que otras partes del sistema operativo son cargadas solo cuando se necesitan.

Una de las funciones mas importantes incluidas en ese núcleo es el procesamiento de interrupciones. El núcleo inhabilita las interrupciones cuando atiende una interrupción y las habilita nuevamente una vez que ha completado el procesamiento de la interrupción. Cuando se da un flujo continuo de interrupciones, es posible que el núcleo deje inhabilitadas las interrupciones por un periodo largo; esto puede ocasionar un elevado

tiempo de respuesta a las interrupciones. Por esta razón los núcleos son diseñados para realizar el "mínimo" posible de procesamiento en cada interrupción y dejar que el resto lo realice el proceso apropiado del sistema, que puede operar mientras el núcleo se habilita para atender otras operaciones. Lo que se traduce en que las interrupciones permanecen habilitadas un porcentaje mucho mayor del tiempo y que el sistema responde mejor.

Resumen de las funciones del núcleo del Sistema Operativo:

1. Manejo de Interrupciones.
2. Crear y Destruir procesos.
3. El cambio de los estados de los procesos.
4. El despacho de los procesos.
5. Suspender y reanudar un proceso.
6. Sincronización de los procesos.
7. La comunicación entre procesos.
8. El manejo de los bloques de control de procesos.
9. Apoyo a las actividades de entrada/salida.
10. Asignación y liberación de memoria.
11. Es apoyo para el sistema de archivos.
12. Apoyo para el mecanismo de llamada y retorno de un procedimiento.
13. Apoyo para ciertas funciones de contabilidad del sistema

Habilitación e Inhabilitación de Interrupciones

Al núcleo del sistema operativo solo puede llegarse mediante una interrupción. El núcleo inhabilita las interrupciones mientras responde a la interrupción que esta procesando. Una vez que se determina la causa de la interrupción, el núcleo pasa el tratamiento de la interrupción a un proceso específico del sistema que se ha diseñado para manejar ese tipo de interrupciones.

En algunos sistemas el procesamiento de cada interrupción es realizado por un sistema operativo de una sola pieza.

Procesos Concurrentes Asíncronos

Proceso Concurrente.- Los procesos son concurrentes si existen simultáneamente. Los procesos concurrentes pueden funcionar en forma totalmente independiente unos de otros o pueden ser asíncronos, lo cual significa que en ocasiones requieren cierta sincronización y cooperación.

Procesamiento en Paralelo

A medida que el hardware siga disminuyendo en tamaño y costo irán apareciendo tendencias definidas hacia el multiprocesamiento, el procesamiento distribuido y el paralelismo a gran escala. Si algunas operaciones pueden ejecutarse de manera lógica en paralelo, los computadores también podrán realizarlas físicamente en paralelo, aun sí el nivel de

paralelismo es de miles o tal vez de millones de actividades concurrentes. Ello puede significar mejoras en el rendimiento muy superiores a las que son posibles en los computadores secuenciales.

El procesamiento paralelo resulta interesante y complejo por diversas razones. Resulta más fácil para la gente concentrar su atención en una sola actividad a la vez que pensar en paralelo. Ejemplo.- Intente leer dos libros al mismo tiempo, leyendo una del primero y una del segundo y así sucesivamente.

Determinar qué actividades pueden realizarse en paralelo es difícil y lleva mucho tiempo. Los programas en paralelo son mucho mas difíciles de depurar que los secuenciales. Ejemplo.- Después de corregir un error, puede resultar imposible recinstruir la secuencia de eventos que provocaron su aparición original, por lo que sería impropio asegurar que ya se ha corregido.

Una Estructura de Control para Indicar Paralelismo: Parbegin/Parend

- Existen muchas construcciones de lenguajes de programación para indicar paralelismo. Normanlmente se trata de parejas de enunciados como a continuación se describen:
- Un enunciado para indicar que la ejecución secuencial se dividirá en varias secuencias de ejecución paralelas (hilos de control).
- Un enunciado para indicar la fusión de ciertas secuencias de ejecución paralelas y la reanudación de la ejecución secuencial.

Estos enunciados aparecen en pareja y se conocen como parbegin/parend (para la ejecución en paralelo) o cobegin/coend (para la ejecución concurrente) para indicar el principio y fin.

```
parbegin
        enunciado1;
        enunciado2;
            .
            .
            .
        enunciadon
parend
```

Ejemplo.- Supongamos que un programa se esta ejecutando una sola secuencia de instrucciones encuentra la contrucción parbegin. Eso hace que el hilo de control único se divida en *n* hilos separados; uno para cada enunciado de la construcción parbegin/parend. Puede tratarse de enunciados simples, llamadas a procedimientos, bloques de enunciados secuenciales delimitados por un par *begin/end*, o combinaciones de ellos. Cada uno de los hilos de control terminará en algún momento y llegará a *parend*. Cuando terminen todos los hilos de control, reanudará un hilo de control único y el sistema ejecutará el enunciado que sigue al *parend*.

Ejemplo.- Considérese el cálculo de una raíz de una ecuación cuadrática:

x:= (-b + (b ** 2 - 4 * a * c) ** .5)/(2 * a)

Procesamiento Secuencial	Procesamiento Paralelo
1 b ** 2	1 parbegin
2 4 * a	temp1: = -b;
3 (4 * a) * c	temp2: = b ** 2;
4 (b ** 2) - (4 * a * c)	temp3: = 4 * a;
5 (b ** 2 - 4 * a * c) ** .5	temp4: = 2 * a
6 - b	parend;
7 (-b) + ((b ** 2 - 4 * a * c) **.5)	2 temp5: = temp3 * c;
8 2 * a	3 temp5: = temp2 - temp5;
9 (-b + (b **2 - 4 * a * c) **.5) / (2 * a)	4 temp5: = temp5 ** .5;
	5 temp5: = temp1 + temp5;
	6 x: = temp5 / temp4

Las cuatro operaciones contenidas dentro del parbegin/parend se evalúan en paralelo y las cinco restantes en secuencia, reduciendo considerablemente el tiempo.

Exclusión Mutua.- Se llama exclusión mutua a la situación en la cual los procesos cooperan de tal forma que, mientras un proceso obtiene acceso a datos compartidos modificables, los demas procesos no pueden hacer lo mismo.

Secciones Críticas.- Cuando un proceso obtiene acceso a datos compartidos modificables se dice que se encuentra en una sección critica o región critica.

Las secciones críticas deben ejecutarse lo más rápido posible; un proceso no debe bloquearse en su sección crítica y las secciones críticas deben ser codificadas cuidadosamente para evitar la posibilidad de ciclos infinitos por ejemplo.

Algoritmo de Dekker.- Es una realización en software de las primitivas de exclusión mutua y tiene las siguientes propiedades:

- No requiere ninguna instrucción especial de hardware.
- Un proceso que se encuentra fuera de su sección crítica no puede evitar que otro proceso entre en su propia sección crítica.

Un proceso que desea entrar en su sección crítica puede hacerlo sin que haya posibilidad de un aplazamiento indefinido.

```
program algoritmo_dekker;
var proceso_favorecido: (primero, segundo);
    p1deseaentrar, p2deseaentrar: boolean;
procedure proceso_uno;
    begin
      while true do
        begin
          p1deseaentrar := true;
          while p2deseaentrar do
            if proceso_favorecido = segundo then
              begin
                p1deseaentrar:= false;
                while proceso_favorecido = segundo do;
                p1deseaentrar := true
              end;
            sección_crítica_uno;
```

```
                proceso_favorecido:= segundo;
                p1deseaentrar:= false;
                otras_tareas_uno
            end
    end;
procedure proceso_dos;
    begin
        while true do
            begin
                p2deseaentrar := true;
                while p1deseaentrar do
                    if proceso_favorecido = primero then
                        begin
                            p2deseaentrar:= false;
                            while proceso_favorecido = primero do;
                            p2deseaentrar := true
                        end;
                sección_crítica_dos;
                proceso_favorecido:= primero;
                p2deseaentrar:= false;
                otras_tareas_dos
            end
    end;
begin
    p1deseaentrar := false;
    p2deseaentrar := false;
    proceso_favorecido := primero;
    parbegin
        proceso_uno;
        proceso_dos
    parend
end.
```

El algoritmo Dekker se aplica para la exclusión mutua de n pasos; estas suelen ser soluciones muy complejas. Sin embargo durante muchos años representó el último adelanto en cuestión de algoritmos de espera activa para manejar la exclusión mutua.

En 1981, G.L. Peterson publicó un algoritmo mucho más sencillo para manejar la exclusión mutua de dos procesos con espera activa.

```
program algoritmo_peterson;
var proceso_favorecido: (primero, segundo);
    p1deseaentrar, p2deseaentrar: boolean;
procedure proceso_uno;
    begin
        while true do
            begin
                p1deseaentrar := true;
                proceso_favorecido := segundo;
                while p2deseaentrar
                    and proceso_favorecido = segundo do;
                sección_crítica_uno;
                p1deseaentrar:= false;
                otras_tareas_uno
            end
    end;
procedure proceso_dos;
    begin
        while true do
```



```
            begin
                p2deseaentrar := true;
                proceso_favorecido := primero;
                while p1deseaentrar
                    and proceso_favorecido = primero do;
                sección_crítica_dos:= false;
                otras_tareas_dos
            end
    end;
begin
    p1deseaentrar := false;
    p2deseaentrar := false;
    proceso_favorecido := primero;
    parbegin
        proceso_uno;
        proceso_dos
    parend
end.
```

Semáforos

Dijkstra extrajo los conceptos fundamentales de la exclusión mutua en su concepto de semáforos. Un semáforo es una variable protegida cuyo valor solo puede ser leído y alterado mediante las operaciones P y V y una operación de asiganción de valores iniciales.

Los *semáforos binarios* pueden tener solamente los valores 0 o 1. Los semáforos contadores o semáforos generales pueden tener tan solo valores enteros no negativos.

La operación P sobre el semáforo S, escrita P(S), opera como sigue:

```
si S > 0
    entonces S := S - 1
    si no (esperar  S)
```

La operación V sobre el semáforo S, escrita V(S), opera como sigue:

```
si (uno o mas procesos esperan S )
entonces (dejar que prosiga uno de esos procesos)
si no S:=S + 1
```

Se adoptará una disciplina de colas de primeras entradas-primeras salidas para los procesoso que esperan terminar una operación P(S).

P y V son indivisibles. La exclusión mutua sobre el semáforo S se implanta dentro de P(S) y V(S). Si varios procesos desean ejecutar una operación P(S) de manera simultánea, solamente se permitirá la ejecución de uno de ellos.. Los otros quedaran en espera, pero la manera como se realizan P y V garantiza que los procesos no se aplazarán en forma indefinida.

Los semáforos pueden llevarse a la práctica en software o hardware. Se suelen incluir en el núcleo del sistema operativo donde se controla la comunicación de los estados de los procesos.

Los semáforos pueden servir para llevar a la práctica un mecanismo de sincronización bloquear/despertar: un proceso se bloquea a sí mismo mediante P(S), con S=0 inicialmente, para esperar que ocurra un evento; otro proceso detecta el evento y despierta al proceso bloqueado mediante V(S).
En una relación productor-consumidor, un proceso productor, genera información que utiliza un segundo proceso consumidor (comunicación entre procesos). Si estos procesos se comunican por medio de un buffer compartido, el productor no debe producir cuando el buffer esta lleno y el consumidor no debe consumir cuando el buffer esta vacío (sincronización de procesos).
Los semáforos contadores son particularmente útiles cuando un recurso debe asignarse a partir de un banco de recursos idénticos. Cada operación P indica que se ha asignado un recurso, en tanto que una operación V indica que se ha devuelto un recurso al banco.
Las operaciones de los semáforos pueden llevarse a la práctica con espera activa, pero esto puede ocasionar desperdicio (pueden incluirse en el núcleo para evitar la espera activa).

Programación Concurrente

Los métodos anteriores poseen en su estructura varios defectos. Son lo bastante primitivos que es difícil expresar las soluciones a los problemas de concurrencia más complejos aumentando el ya de por sí difícil problema de probar la corrección de los programas. De hecho el mal uso de estas primitivas podría corromper la operación de un sistema concurrente.
En el método de los semáforos, si omitimos P, no obtenemos la exclusión mutua; si omitimos V, las tareas que están esperando por causa de operaciones P podrían quedar en bloqueo permanente. Una vez que comienza una operación P, el usuario no puede arrepentirse y seguir otro curso de acción mientras el semáforo esta en uso. Una tarea puede esperar solo un semáforo a la vez, lo cual pudiera originar un bloqueo mutuo en situaciones de asiganción de recursos.
Ha surgido gran interés en los lenguajes de programación concurrente porque permiten expresar de una forma natural las soluciones a ciertos problemas de carácter inherentemente paralelo, y también gracias al verdadero paralelismo de hardware que es posible lograr con los multiprocesadores y sistemas distribuidos. Los programas concurrentes son más difíciles de escribir, depurar, modificar y probar que son correctos, sin embargo, tienen muchas aplicaciones posibles. Los sistemas operativos en sí son ejemplos importantes de sistemas concurrentes, como también los son los sistemas de control de tráfico aéreo, los sistemas de control de tiempo real (los que controlan las refinerías, plantas de productos químicos, procesadoras de alimentos, etc.). Los robots son sistemas altamente concurrentes, cuando un robot camina debe ser capaz de ver, oír, tocar, saborear y oler al mismo tiempo. De hecho, el proceso de caminar, es preciso coordinar con cuidado la operación de cada motor y cada parte de los miembros para lograr incluso los movimientos más sencillos.
ADA es el primer lenguaje concurrente de uso generalizado, pero pasará algún tiempo antes de que haya un número significativo de sistemas realizados por completo en ADA.

Regiones Críticas y Regiones Críticas Condicionales

La noción de *región crítica* se desarrollo para expresar de forma sencilla la exclusión mutua. Por ejemplo para indicar que alguna acción se realizará con acceso exclusivo a ciertos datos_ compartidos se escribe la siguiente instrucción en Pascal:

región datos_compartidos **do** acción (Región Crítica)

La noción de *región crítica condicional* permite especificar la sincronización, además de la exclusión mutua cuyo ejemplo es:

región datos_compartidos **do begin await** condición; acción **end**

Es posible realizar acción con acceso exclusivo a datos_compartidos cuando se cumple condición.

MONITORES

Monitor.- Es una construcción para concurrencia que contienen tanto los datos como los procedimientos necesarios para asignar un recurso compartido específico y reutilizable en serie o en grupos de estos recursos.

Para ejecutar una función de asignación de recursos, un proceso debe llamar a una entrada del monitor especifica. Muchos procesos pueden desear entrar al monitor en diversos momentos, pero la exclusión mutua se mantiene de manera estricta en los límites del monitor ya que solo se permite la entrada de un proceso a la vez. Los procesos que desean entrar en el monitor cuando ya esta en uso deben esperar; el monitor controla automáticamente esta espera, de esta forma la exclusión mutua esta garantizada.

Los datos que se encuentran dentro del monitor pueden ser *globales* con respecto de todos los procedimientos del monitor o *locales* con respecto de un procedimiento específico, Todos esos datos son solo accesibles dentro del monitor; un proceso fuera del monitor no puede tener acceso a esos datos, a esto le llamamos *ocultación de información*, que es una técnica de estructuración de sistemas que facilita mucho el desarrollo de sistemas de software confiables.

Si un proceso que esta llamando a la entrada del monitor encuentra que el recurso ya ha sido asignado, el procedimiento monitor llama a la función *esperar*. El proceso podría permanecer dentro del monitor pero ello violaría la *exclusión mutua* si otro proceso entrara en el monitor. Por lo tanto el proceso que llamó a *esperar* debe aguardar fuera del monitor hasta que se libere el recurso.

En algún momento, el proceso que tiene el recurso llamará a una entrada al monitor para devolver el recurso al sistema. Esta entrada podría limitarse a aceptar el recurso devuelto y esperar a que llegue otro proceso solicitante. Pero puede haber procesos esperando el