Tema II

Administración de Procesos

Concepto de proceso

- Un proceso es un programa en memoria principal.
- Un proceso está compuesto de seis componentes:
  - Id
  - Código del programa
  - Datos
  - Recursos
  - Pila
  - Estado del CPU

Concepto de proceso

- Un proceso necesita ciertos recursos para realizar satisfactoriamente su tarea:
  - Tiempo de CPU.
  - Memoria.
  - Archivos.
  - Dispositivos de E/S.
- Los recursos se asignan a un proceso:
  - Cuando se crea.
  - Durante su ejecución.

Bloque de Control de Proceso

Cada proceso se representa en el Sistema Operativo con un Bloque de Control de Proceso (PCB, *Process Control Block*).

Estados de un Proceso

- El estado de proceso es un indicador de la naturaleza de la actividad actual en un proceso.[1]
  - Nuevo. Se solicitó al sistema la creación de un proceso, y sus recursos y estructuras están siendo creadas
  - Listo. Está listo para ser asignado para su ejecución.
  - En ejecución. El proceso está siendo ejecutado.
  - Bloqueado. En espera de algún evento para poder continuar
  - ejecutándose
  - Terminado. El proceso terminó de ejecutarse; sus estructuras están a la espera de ser limpiadas por el sistema

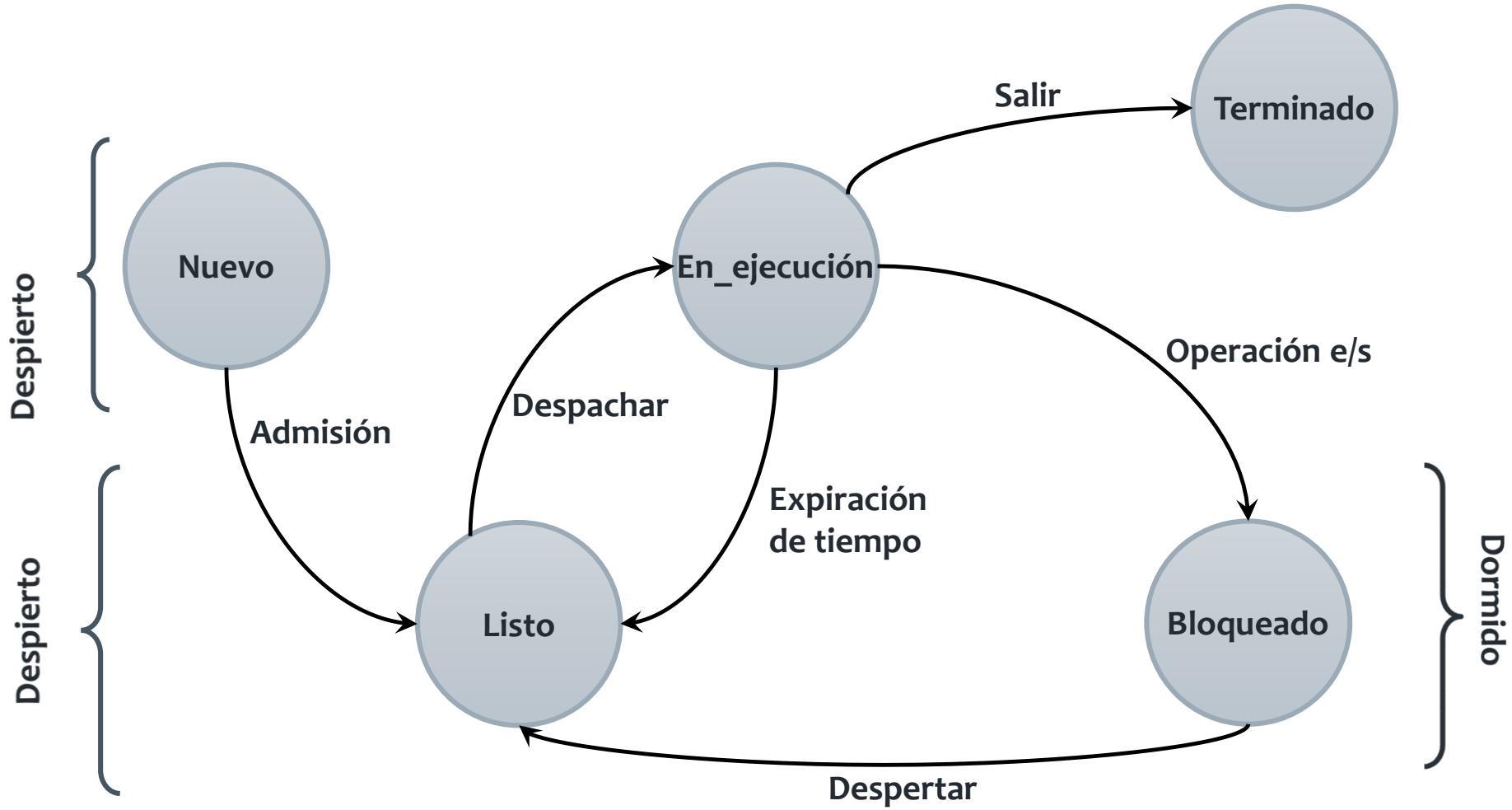
[1] Wolf, Gunnar. Fundamentos de Sistemas Operativos.

Transición de estado de un Proceso

Una transición de estado para un proceso es un cambio en su estado, y se debe a la ocurrencia de algún evento en el sistema.

- admisión (nombre_del_proceso): nuevo→ listo
- despachar (nombre_del_proceso): listo → en_ejecución
- expiración_de_tiempo (nombre_del_proceso): en_ejecución → listo
- operación_e/s (nombre_del_proceso): en_ejecución → bloqueado
- despertar (nombre_del_proceso): bloqueado → listo
- salir (nombre_del_proceso): en_ejecución→ terminado

Transición de estado de un Proceso



Cinco Estados de un Proceso

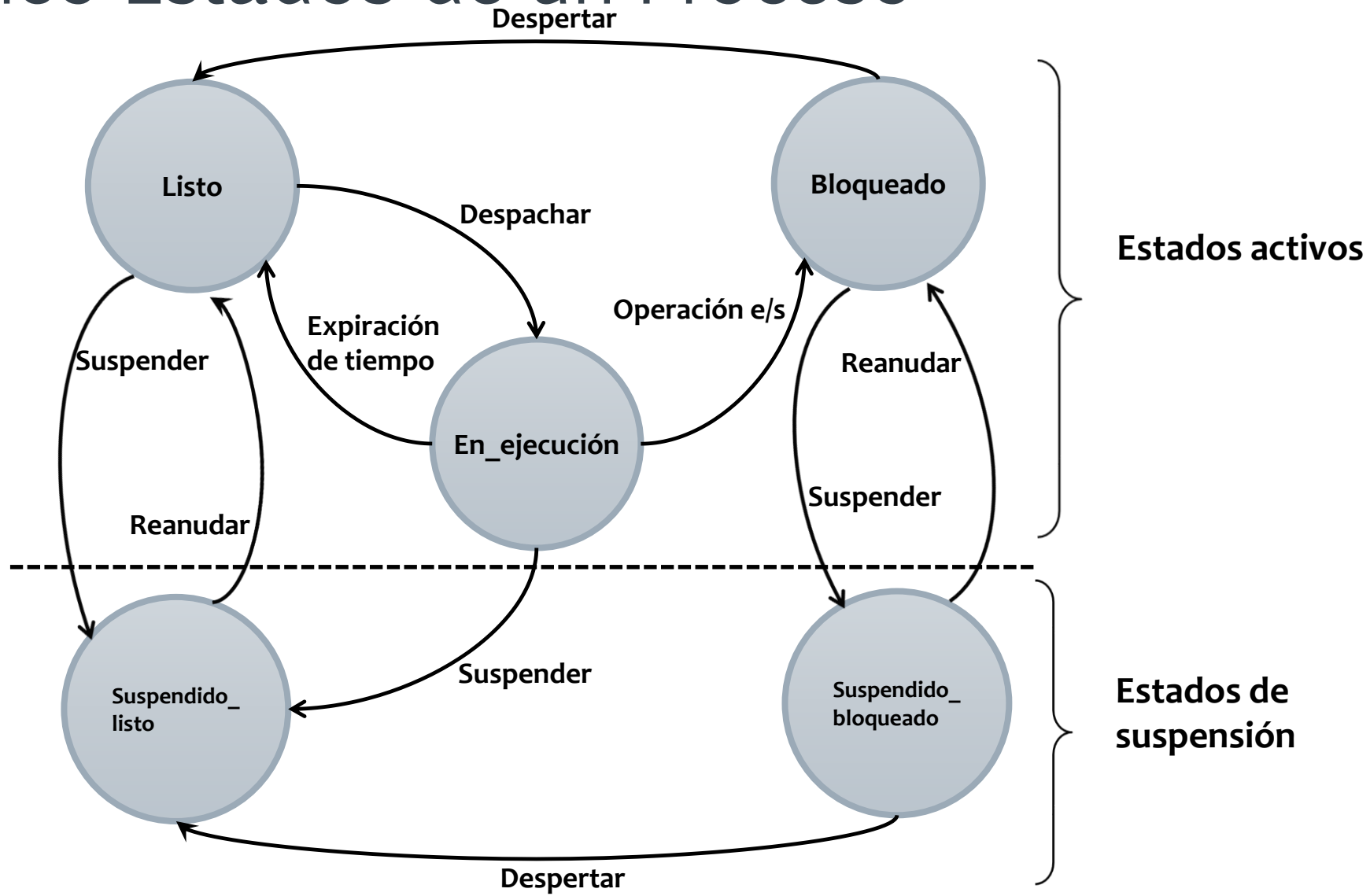
- Conceptos independientes:
  - Si un proceso está esperando un suceso (bloqueado o no)
  - Si un proceso ha sido expulsado de la memoria principal (suspendido o no).

Cinco Estados de un Proceso

- La suspensión es externa a la actividad del proceso.
- El bloqueo es interna a la propia actividad del proceso.

- Dos causas típicas de suspensión son:
  - Un proceso sale de la memoria, es decir, se intercambia.
  - El usuario que inició un proceso especifica que el proceso no deberá planificarse hasta que se satisfaga alguna condición.

Cinco Estados de un Proceso



Cinco Estados de un Proceso
Estados y Transiciones

- suspendido_listo: El proceso está en memoria secundaria pero está disponible para su ejecución tan pronto como se cargue en la memoria principal.

suspender (nombre_del_proceso): en_ejecución → suspendido_listo

suspender (nombre_del_proceso): listo → suspendido_listo

reanudar (nombre_del_proceso): suspendido_listo → listo

Cinco Estados de un Proceso
Estados y Transiciones

- suspendido_bloqueado: El proceso está en la memoria secundaria esperando un suceso.

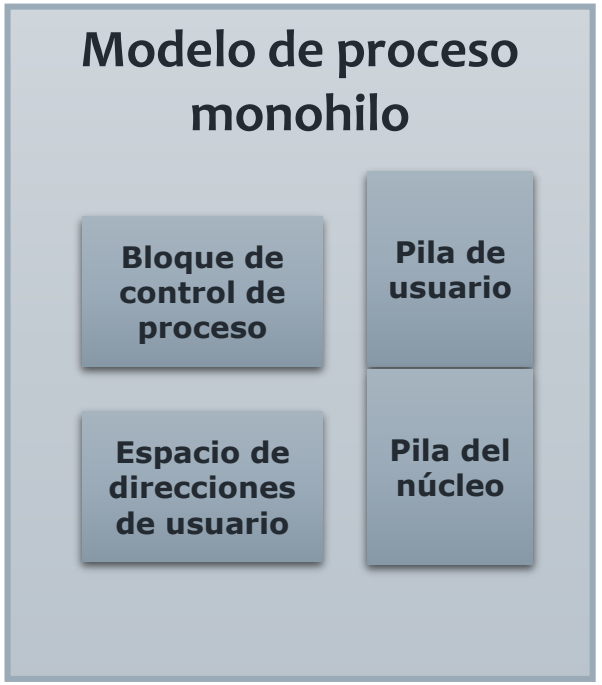
suspender (nombre_del_proceso): bloqueado → suspendido_bloqueado

reanudar (nombre_del_proceso): suspendido_bloqueado → bloqueado

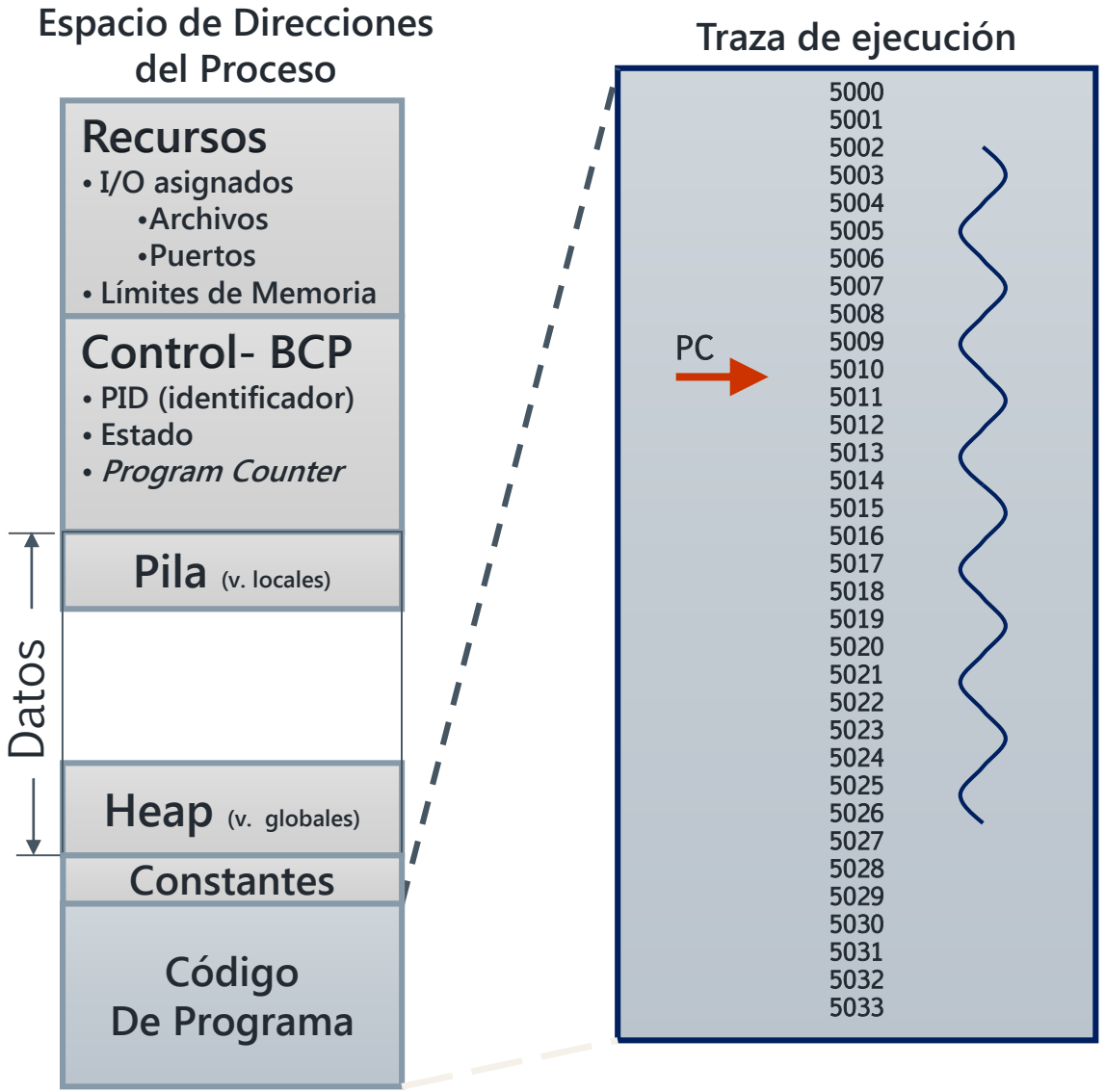
despertar(nombre_del_proceso): suspendido_bloqueado → suspendido_listo

Proceso Monohilo

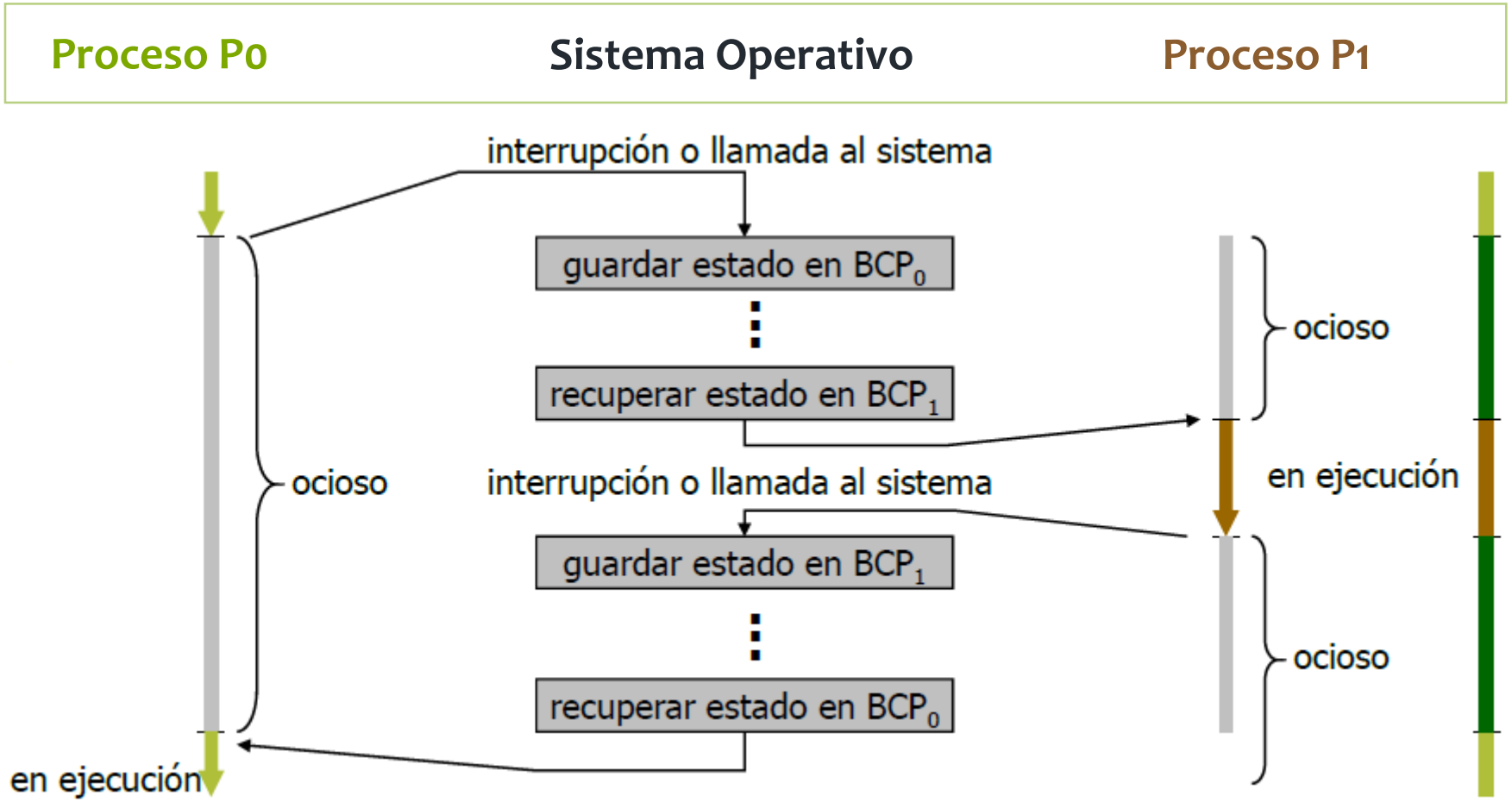
Un proceso es un hilo de ejecución. La representación de un proceso incluye su PCB, un espacio de direcciones del proceso, una pila de proceso y una pila núcleo.



Ejecución típica de un Proceso

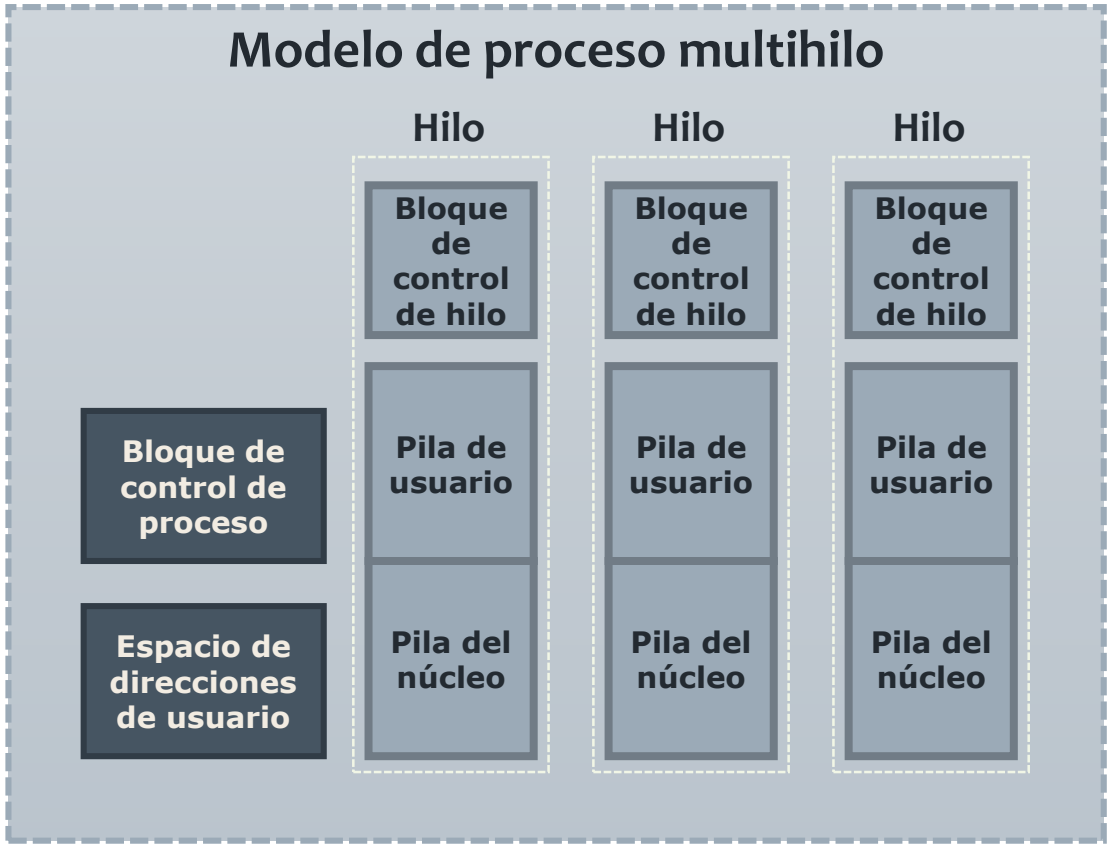


Cambio de contexto (*context switch*)



Proceso Multihilo

Un proceso tiene múltiples hilos de ejecución. Sólo hay un PCB y un espacio de direcciones asociados al proceso. Hay pilas separadas y bloques de control para cada hilo.



Hilo

Un hilo (*thread*) llamado Proceso Ligero (LWP, *Light Weight Process*). El Bloque de Control de Hilo (TCB, *Thread Control Block*), tiene información como:

- Estado del proceso.
- Su propio contador de programa y pila de llamadas (*stack*).
- Información de planificación interna.
- Conjunto de variables locales.

Los hilos[1]:

- Comparten memoria, descriptores de archivos y dispositivos.
- Pueden tener variables locales y globales.

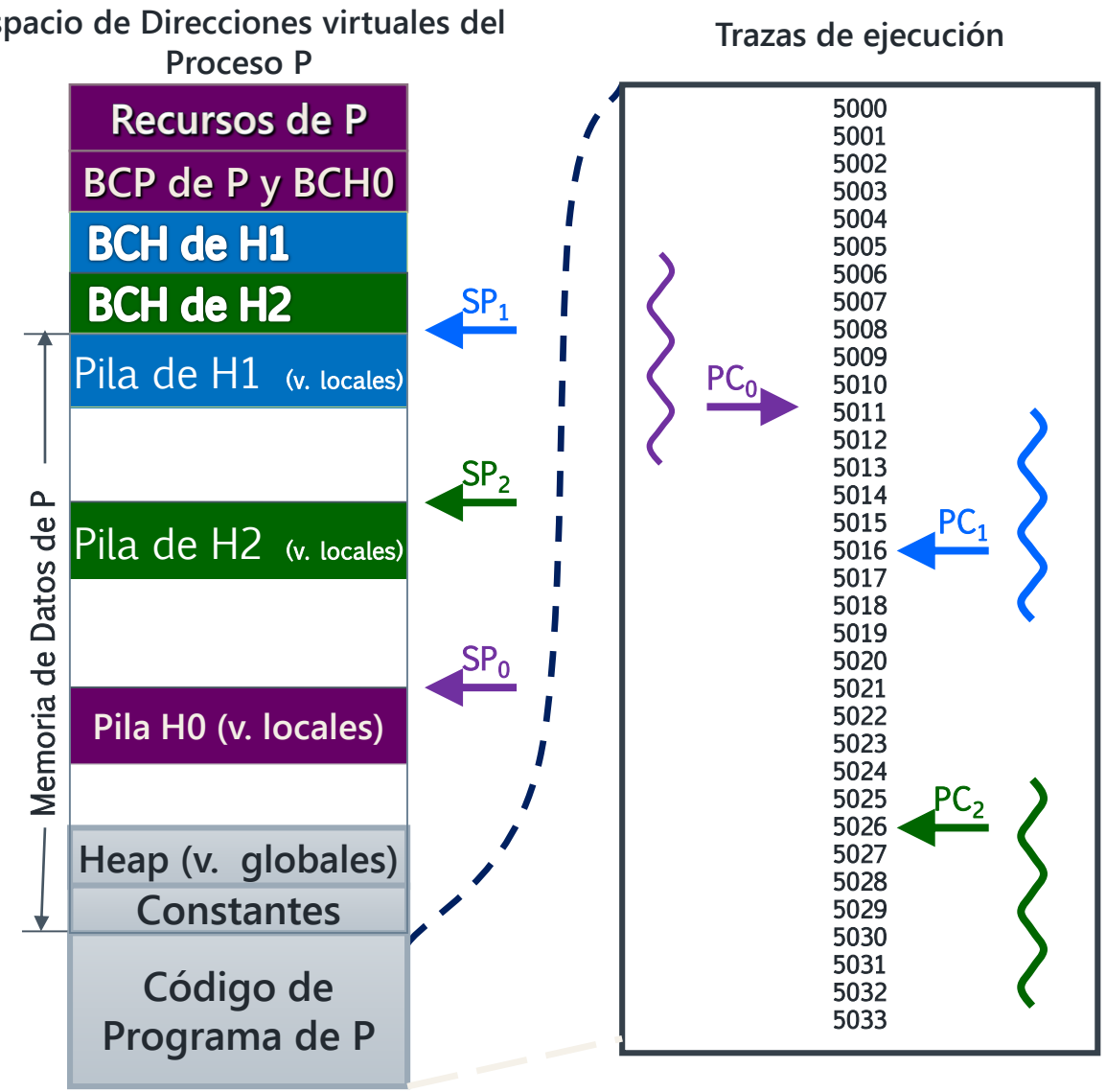
[1] Wolf, Gunnar. Fundamentos de Sistemas Operativos.

Estados de los Hilos

- Los principales estados de un hilo son: ejecución, listo y bloqueado.

- El estado del Proceso P es la combinación de los estados de sus Hilos.
  - Cuando cualquiera de los Hilos está en estado “Ejecución”, el estado del Proceso P será “Ejecución”.
  - Si ningún Hilo está en “Ejecución”, si alguno está en “Listo”, el estado del Proceso P será “Listo”.
  - El estado del Proceso P es “Bloqueado” sólo si todos sus Hilos están en estado “Bloqueado”.

Ejecución típica de un Proceso



Ejemplos de Hilos

- Navegadores.
- Procesador de texto.
- Servidores Web.
- Servidor de Archivos.
- Video juegos.
- ...

Hilos de Usuario

ULT (*User Level Thread*)

- Conocidos como "*green thread*".
- Mayor Portabilidad, a través de alguna biblioteca del lenguaje/entorno de programación.[2]
- Típicamente multitarea interna cooperativa.[3]
- El SO no es consciente de la existencia de hilos a nivel de usuario.

Ejemplos:

- *GNU Portable Threads*.
- *FSU Threads*.
- *Apple Computer Thread Manager*.
- *REALbasic's cooperative threads*.

[2][3] Wolf, Gunnar. Fundamentos de Sistemas Operativos.

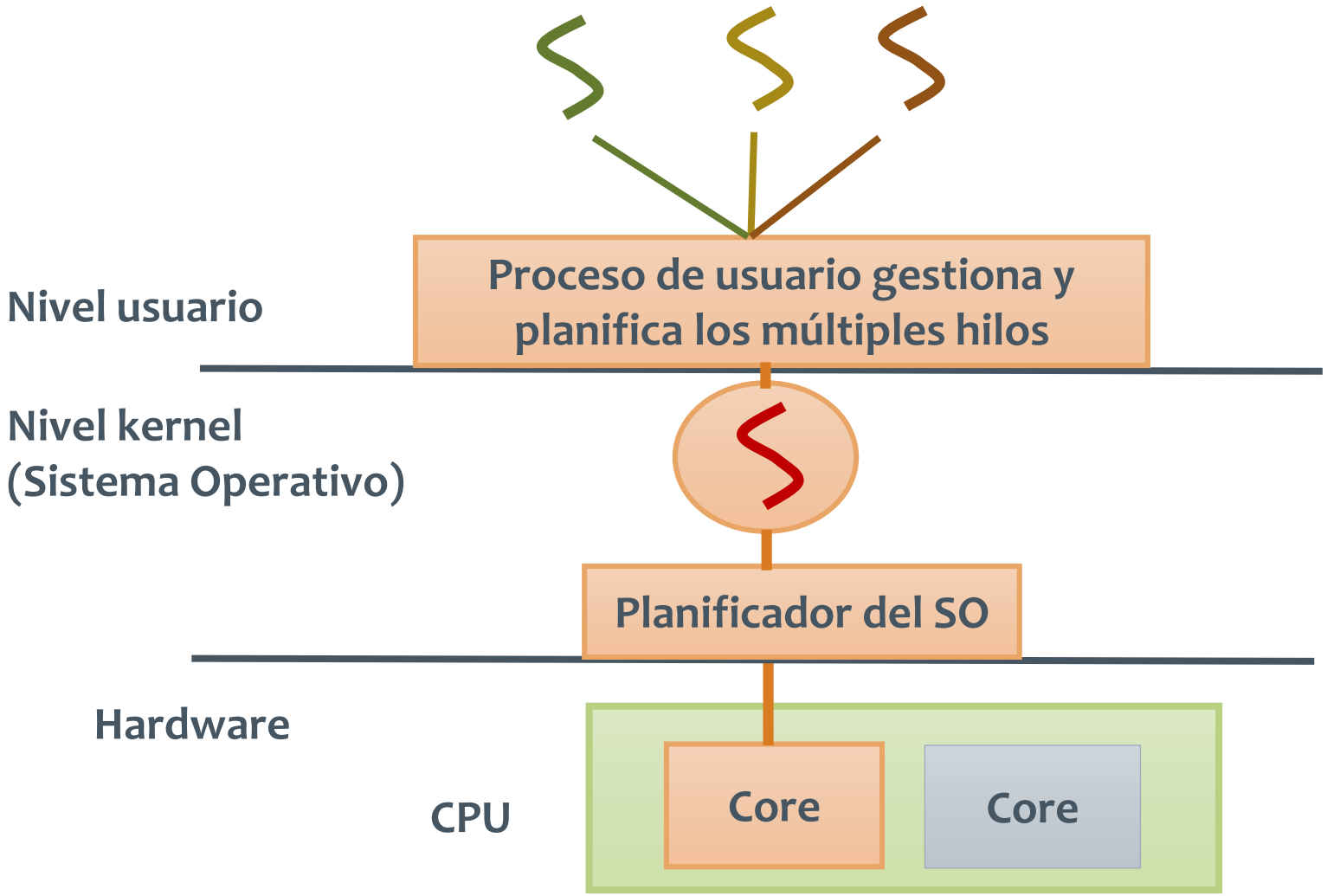
Ventajas y desventajas de los Hilos de Usuario[4]

- Ventajas
  - Espacio de memoria compartido sin intervención del SO.
  - Mayor rapidez para realizar trabajos cooperativos.

- Desventajas
  - Cualquier llamada al sistema interrumpe a todos los hilos.
  - No aprovechan el multiprocesamiento.

[4] Wolf, Gunnar. Fundamentos de Sistemas Operativos.

ULT (CPU con dos núcleos)



Hilos de Kernel

KLT (*Kernel Level Thread*)

- Conocidos como “hilos nativos”.
- La planificación de los hilos es realizada por el *kernel*, a través de bibliotecas de sistema que informan al SO.

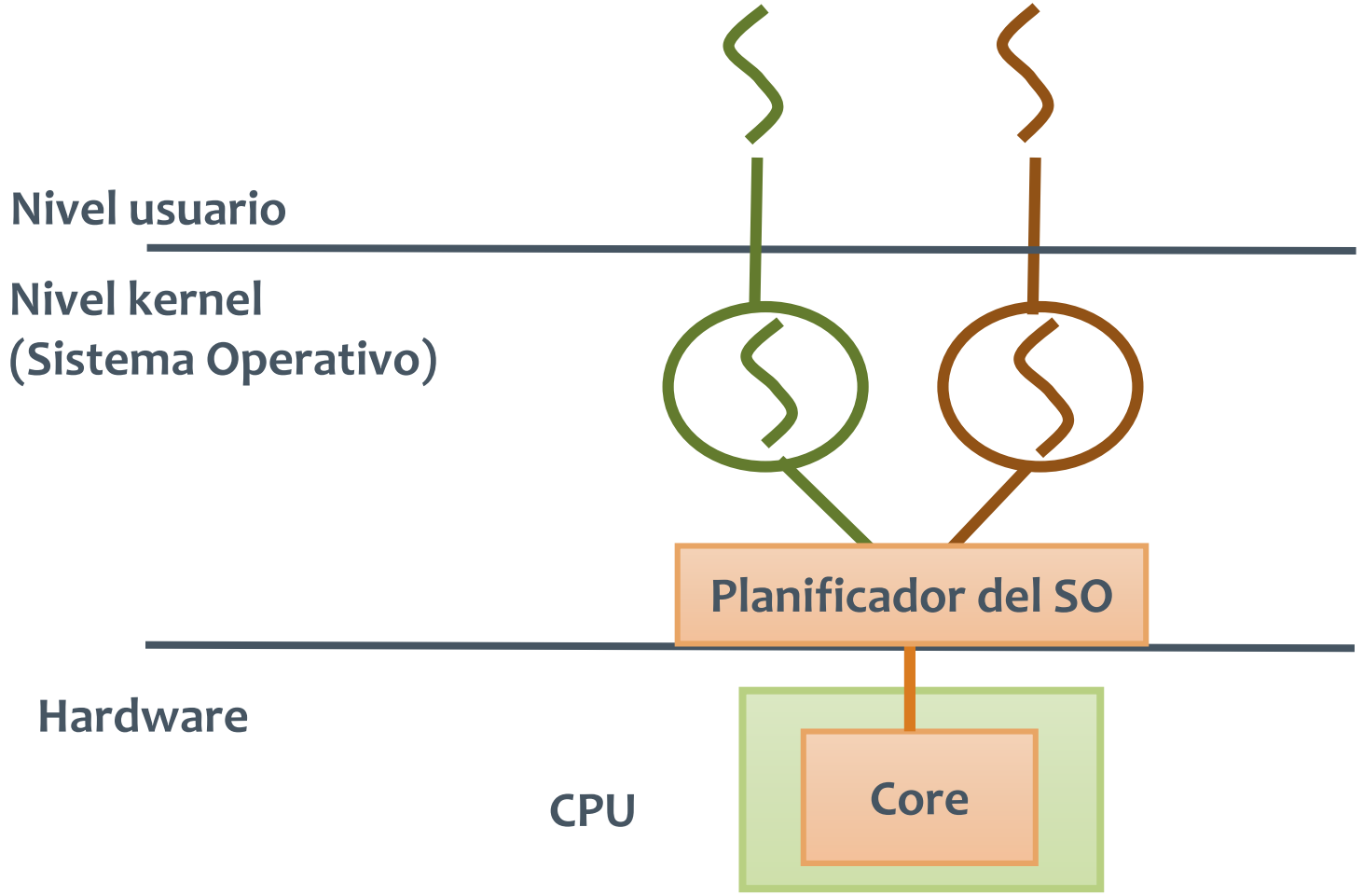
Ejemplos:

- *Light Weight Kernel Threads*.
- M:N *threading*.
- Native POSIX Thread Library para Linux.

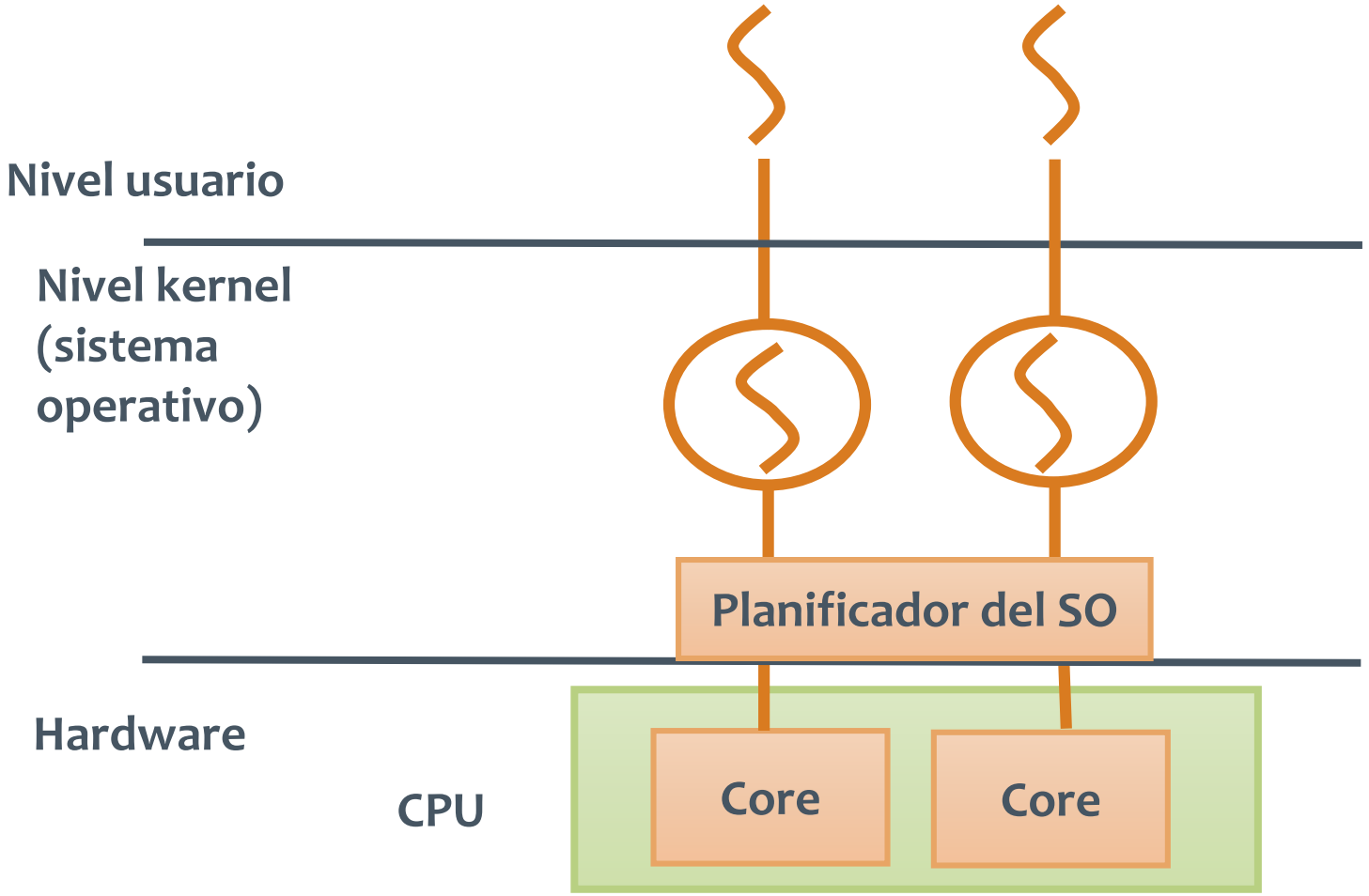
Ventajas y desventajas de los Hilos de Kernel

- Ventajas
  - Si tenemos un procesador con más de un núcleo, el SO puede planificar los hilos en diferentes núcleos.
  - Si se bloquea un hilo, puede planificar otro hilo del mismo proceso.
- Desventajas
  - Si varios hilos acceden al mismo conjunto de datos, este acceso debe ser sincronizado.

KLT (CPU con un núcleo)



KLT (CPU con dos núcleos)



Identificación de procesos

POSIX (*Portable Operating System Interface*) identifica cada proceso por medio de un entero único denominado identificador de proceso de tipo pid_t. Servicios relativos a la identificación de los procesos son:

a) Obtener el identificador de proceso

```
pid_t    getpid (void);
```

b) Obtener el identificador del proceso padre

```
pid_t    getppid (void);
```

Creación de procesos

Para crear un proceso se invoca la llamada al sistema fork().

El SO trata esta llamada al sistema realizando una clonación del proceso que la solicita. Además, es la única que devuelve dos valores al ser solicitada.

Su prototipo es:

```
#include <unistd.h>
pid_t   fork();
```

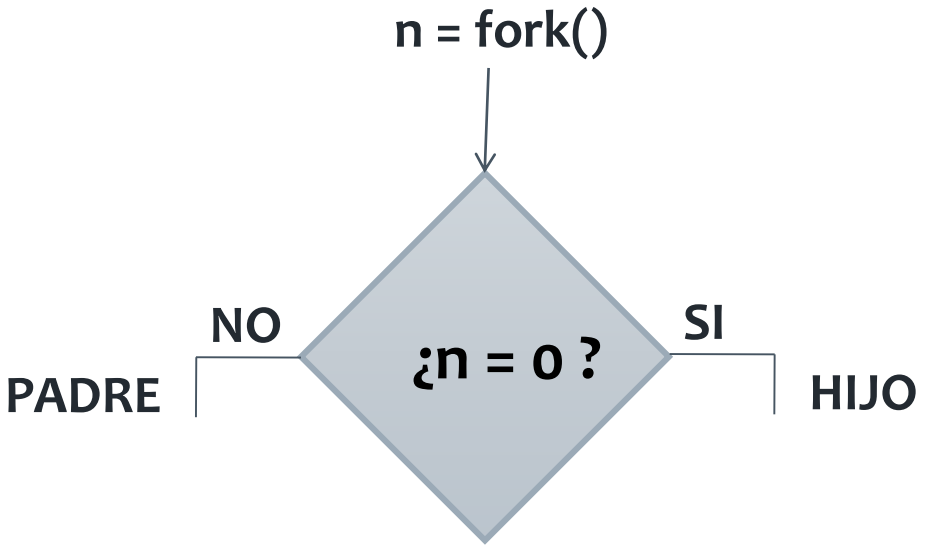
- fork() es invocada una sola vez (por el proceso padre).
- fork() causa que el proceso actual se divida en dos procesos: un proceso padre y un proceso hijo.

Creación de procesos

- A la salida de fork(), los dos procesos tienen una copia idéntica del contexto de nivel de usuario excepto el valor de PID.
  - En el proceso padre, fork() regresa el PID del nuevo proceso creado: proceso hijo.
  - En el proceso hijo, fork() regresa el valor de cero.
  - Si por alguna razón fork() falla (no hay suficiente memoria, hay demasiados procesos, etc.), no crea el nuevo proceso y devuelve el valor de -1.

El único proceso que no se crea con la llamada a fork() es el proceso 0 creado por el núcleo del sistema.

Uso de la llamada al sistema **fork()**



Función **main()** con argumentos

int main (int argc, char **argv)

donde:

- *argc* representa el número de argumentos que se pasan al programa, incluido el propio nombre del programa.
- *argv* es un vector de cadenas de caracteres, conteniendo cada elemento de este vector un argumento pasado al programa. El primer componente de este vector (argv [0]) representa el nombre del programa.

Terminación de procesos

- Un proceso puede terminar su ejecución de forma normal:
  - Ejecutando una sentencia *return* en la función *main*.
  - Ejecutando la llamada *exit*.

Cuando un programa ejecuta dentro de la función *main* la sentencia *return(valor)*, es similar a *exit(valor)*. El prototipo de la función *exit* es:

```
#include <stdlib.h>
void exit (int status);
```

Estos servicios tienen por función finalizar la ejecución de un proceso. Ambos reciben como parámetro un valor que sirve para que el proceso dé una indicación de cómo ha terminado.

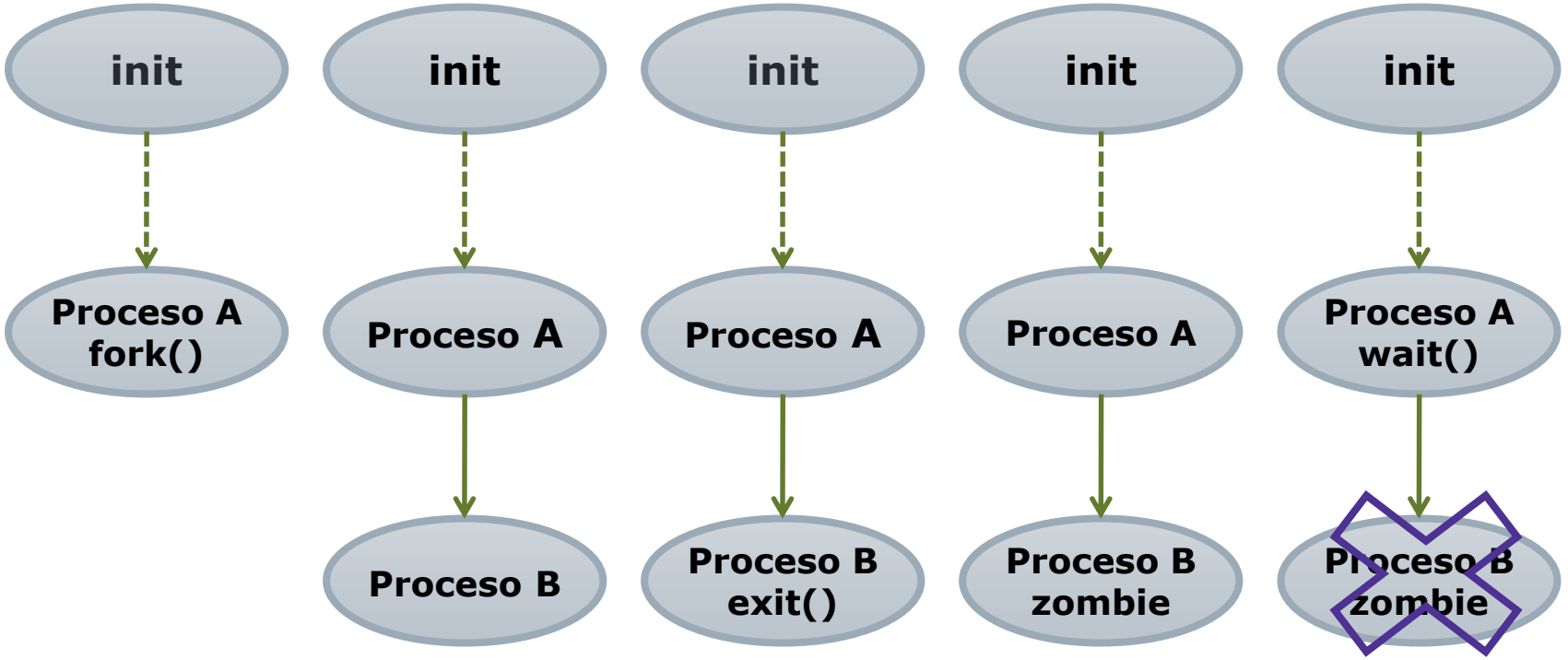
Terminación de procesos

- Esperar por la finalización de un proceso hijo.

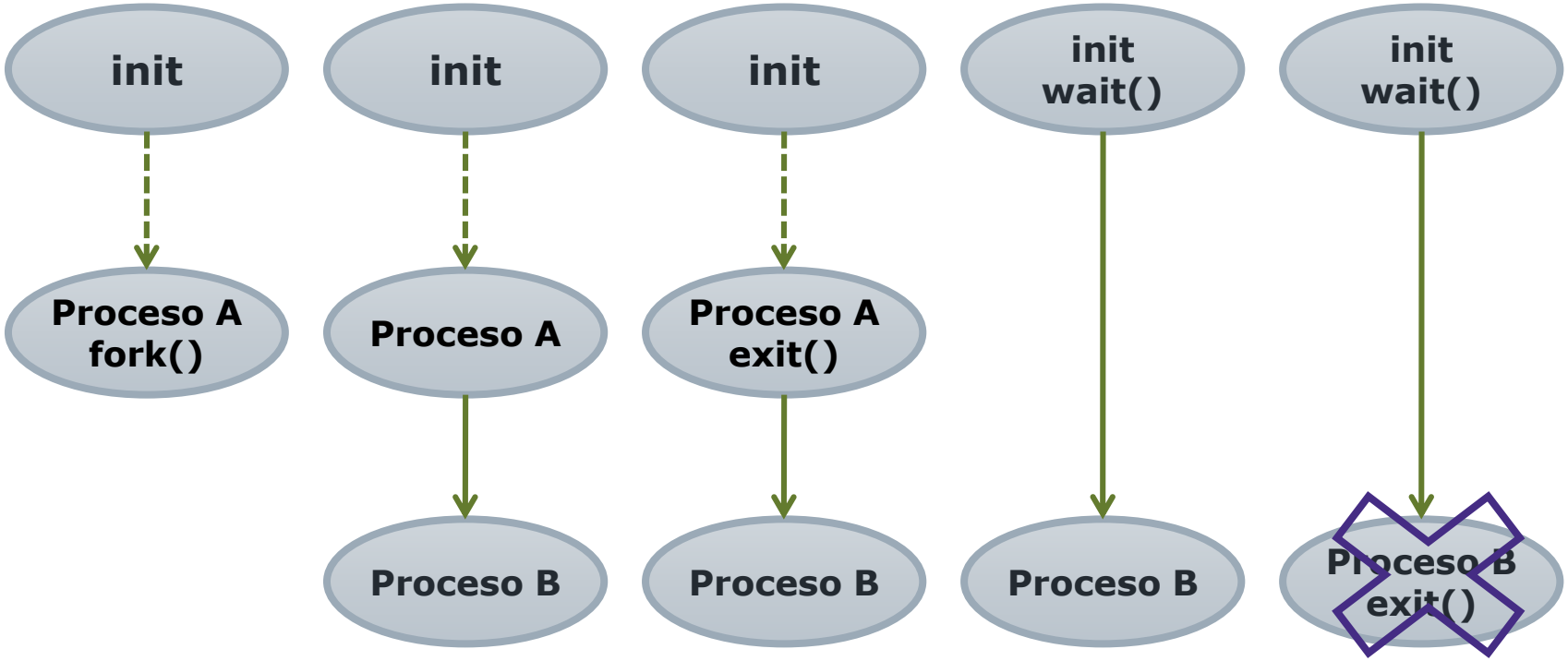
```
pid_t wait (int *status);
pid_t waitpid (pid_t, int *status, int options);
```

Permite a un proceso padre quedar bloqueado hasta que termine la ejecución de un proceso hijo, obteniendo información sobre el estado de terminación del mismo.

Procesos zombies



Procesos huérfanos



Identificación del hilos

Regresa el ID del hilo que realiza la llamada:

```
pthread_t pthread_self (void);
```

Creación y destrucción de atributos

Los atributos se almacenan en un objeto atributo de tipo pthread_attr_t.

Para la creación y el inicio de un objeto atributo (utilizado en la creación de un proceso ligero) se utiliza la siguiente función:

```
int pthread_attr_init (pthread_attr_t *attr);
```

Para la destrucción del objeto de tipo atributo se utiliza:

```
int pthread_attr_destroy (pthread_attr_t *attr);
```

Establecimiento del estado de terminación

El servicio para el establecimiento del atributo correspondiente al estado de terminación es:

```
int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate);
```

El valor del argumento *detachstate* puede ser:

- **PTHREAD_CREATE_DETACHED**: El proceso ligero que se cree con este estado de terminación se considerará independiente y liberará sus recursos cuando finalice su ejecución.
- **PTHREAD_CREATE_JOINABLE**: El proceso ligero que se cree con este estado de terminación se considerará como no independiente y no liberará sus recursos cuando finalice su ejecución. En este caso, es necesario que otro proceso espere por su finalización utilizando *pthread_join*.

Creación de hilos

- **pthread_create**, devuelve 0 si todo salió bien o -1 en caso error.

```
int pthread_create (pthread_t *thread, pthread_attr_t *attr,
                    void *(*routine) (void *), void *arg);
```

- **pthread_t *thread**: identificador del hilo.
- **const pthread_attr_t *attr**: atributos del hilo.
- **void *(*routine)(void *)**: función llamada por el hilo cuando éste comienza ejecución.
- **void *arg**: argumento(s) de la función invocada.

Terminación de hilos

Únicamente se puede solicitar el servicio **pthread_join** sobre hilos creados como no independientes.

```
int pthread_join (pthread thid, void *value);
```

Este servicio permite esperar a que termine un hilo. La función suspende la ejecución del hilo llamante hasta que el hilo con identificador **thid** finaliza su ejecución. Devuelve en el segundo argumento el valor devuelto por el hilo que termina (mediante **pthread_exit**, comentado a continuación).

Terminación de hilos

Para finalizar la ejecución de un hilo se utiliza:

```
void pthread_exit (void *value);
```

El argumento es un apuntador a una estructura que es devuelta al hilo que ha ejecutado la correspondiente llamada a **pthread_join**.

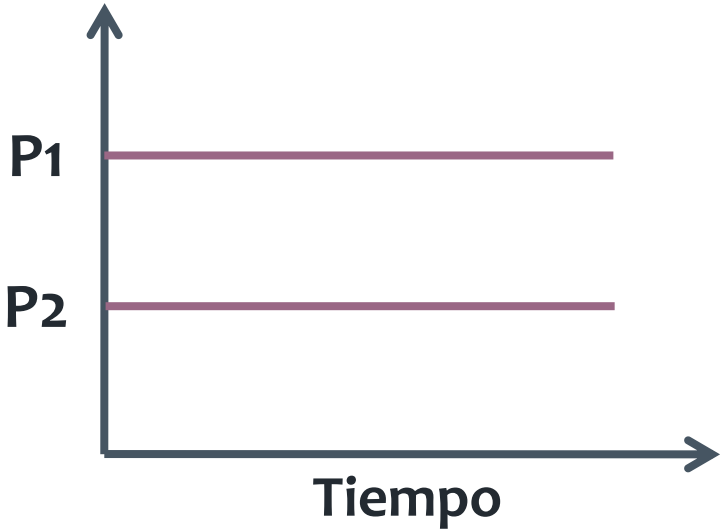
Concurrencia

- La concurrencia es la existencia simultánea de varios procesos en ejecución.

  "**Existencia simultánea**" no implica "**Ejecución simultánea**"

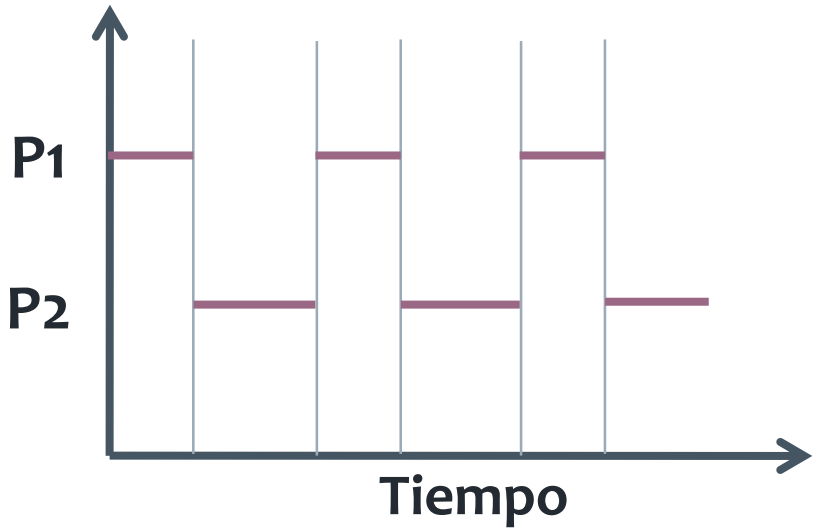
- La concurrencia se consigue haciendo que:
  - Los procesos se asignen a distintos procesadores.
    - **paralelismo real**
  - Se alternen varios procesos en un mismo procesador.
    - **pseudo-paralelismo**

Concurrencia



Concurrencia real

N° procesadores >= N° procesos



Concurrencia aparente

N° procesadores < N° procesos

Exclusión mutua

Cuando un proceso tiene el uso exclusivo del recurso.

Hacer que se cumpla la exclusión mutua puede dar lugar a dos problemas:

- Interbloqueo
- Inanición

Problema de la sección crítica

La sección crítica es un segmento de código donde un proceso realiza el acceso de datos compartidos.

Cualquier solución al problema de la sección crítica debe satisfacer los siguientes requisitos:

- **Exclusión mutua**: mientras que un proceso esté en su sección crítica, ningún otro podrá acceder a los recursos compartidos con los que se está ejecutando el primero.
- **Progreso**: Un proceso que NO está en su sección crítica no puede impedir a otro proceso entrar en su sección crítica.
- **Espera limitada**: Todo proceso obtiene el recurso al cabo de un tiempo finito.

Problema de la sección crítica Algoritmo Versión Uno

```
int processnumber;
processone()
{
   while(1)
   {
       while(processnumber == 2);
       criticalsection;
       processnumber = 2;
       otherthings;
   }
}
```

```
processtwo()
{
   while(1)
   {
       while(processnumber == 1);
       criticalsection;
       processnumber =   1;
       otherthings;
    }
}
```

```
main()
{
    processnumber = 1;
    co{
            processone();
            processtwo();
    co}
}
```

Problema de la sección crítica Algoritmo Versión Dos

```
int p1inside, p2inside;
processone()
{
   while(1)
   {
        while( p2inside);
        p1inside = 1;
        criticalsection;
        p1inside = 0;
        otherthings;
   }
}
```

```
processtwo()
{
   while(1)
   {
        while( p1inside);
        p2inside = 1;
        criticalsection;
        p2inside = 0;
        otherthings;
    }
}
```

```
main()
{
    p1inside = 0;
    p2inside = 0;
    co{
            processone();
            processtwo();
    co}
}
```

Problema de la sección crítica Algoritmo Versión Tres

```
int p1inside, p2inside;
processone()
{
   while(1)
   {
        p1inside = 1;
        while( p2inside);
        criticalsection;
        p1inside = 0;
        otherthings;
   }
}
```

```
processtwo()
{
   while(1)
   {
        p2inside = 1;
        while( p1inside);
        criticalsection;
        p2inside = 0;
        otherthings;
    }
}
```

```
main()
{
     p1inside = 0;
     p2inside = 0;
     co{
               processone();
               processtwo();
     co}
}
```

Sincronización

Es cuando un proceso espera ser notificado sobre la ocurrencia de un evento que será detectado por otro proceso.

Semáforos

- Definidos por Dijkstra en 1965.
- Los semáforos generales o contadores son útiles cuando un recurso será asignado, tomándolo de un conjunto de recursos idénticos.
- Es una variable entera cuyo valor se modifica solamente por tres procedimientos:

INIT (S, int)
P(S)
V(S)

Operaciones con Semáforos

Semáforo es **un entero e inicializado con el número de recursos existentes**:

```
INIT (S, int)
{
    S = int;
}
```

Operaciones con Semáforos

P(S) decrementa S en 1; indicando que un recurso ha sido ocupado.

Si S = 0 → no hay más recursos y el proceso se bloquea

```
P(S)
        if ( S > 0 ) then
                S = S – 1;
        else
                WAIT;
```

Operaciones con Semáforos

V(S) incrementa S en 1; indicando que un recurso ha sido liberado.

Si un proceso esperaba por un recurso, éste se despierta.

```
V(S)
        if ( alguien esperando por S ) then
                liberalo;
        else
                S = S + 1;
```

Operaciones atómicas

- Se garantiza que al iniciar una operación con un semáforo, ningún otro proceso puede tener acceso al semáforo hasta que la operación termine o se bloquee.
- En ese tiempo ningún otro proceso puede simultáneamente modificar el mismo valor de semáforo.

Monitores

- Módulos de programación de alto nivel de abstracción que resuelven internamente, el acceso de forma segura a una variable o a un recurso compartido por múltiples procesos concurrentes.
- Los procesos entran al monitor en exclusión mutua.

Monitores

Para sincronizar las tareas del monitor, una **variable de condición debe ser declarada como condición** para permitir que mas de un proceso estén en el mismo monitor al mismo tiempo, aunque sólo uno de ellos actualmente será el que esté activo dentro del monitor.

Variable de condición puede utilizarse únicamente con las operaciones:

- **wait()**
- **signal()**

Monitores

- **wait()**

El proceso que invoca la operación **wait()** queda bloqueado/suspendido si una determinada condición es verdadera. Entonces el monitor se desbloquea y permite que otra tarea utilice el monitor.

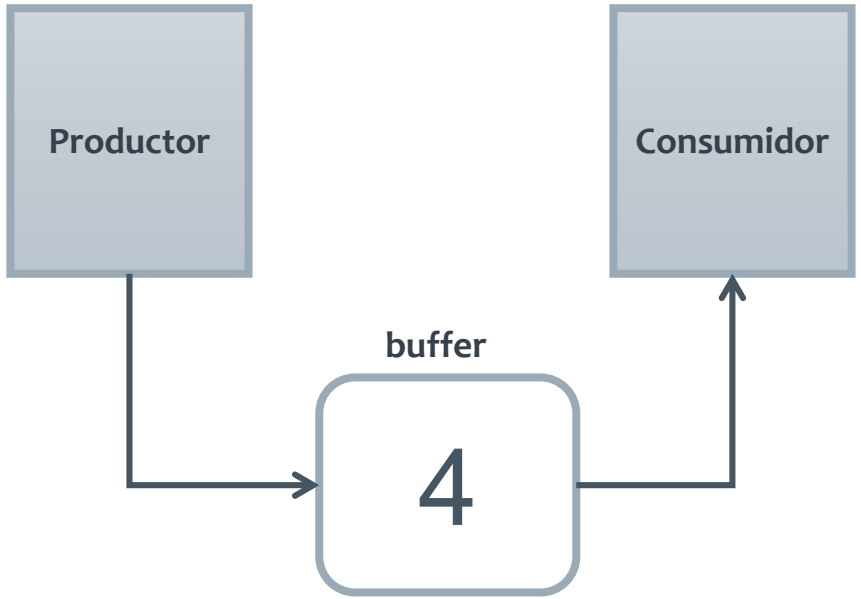
- **signal()**

Cuando la misma condición se convierte en falsa, entonces la operación **signal()** reanuda la ejecución de algún proceso suspendido después de la condición **wait()**, colocándolo en la cola de listos del procesador. Si hay varios de estos procesos, elije uno de ellos; si no hay procesos en espera, la operación **signal()** es ignorada.

Monitores

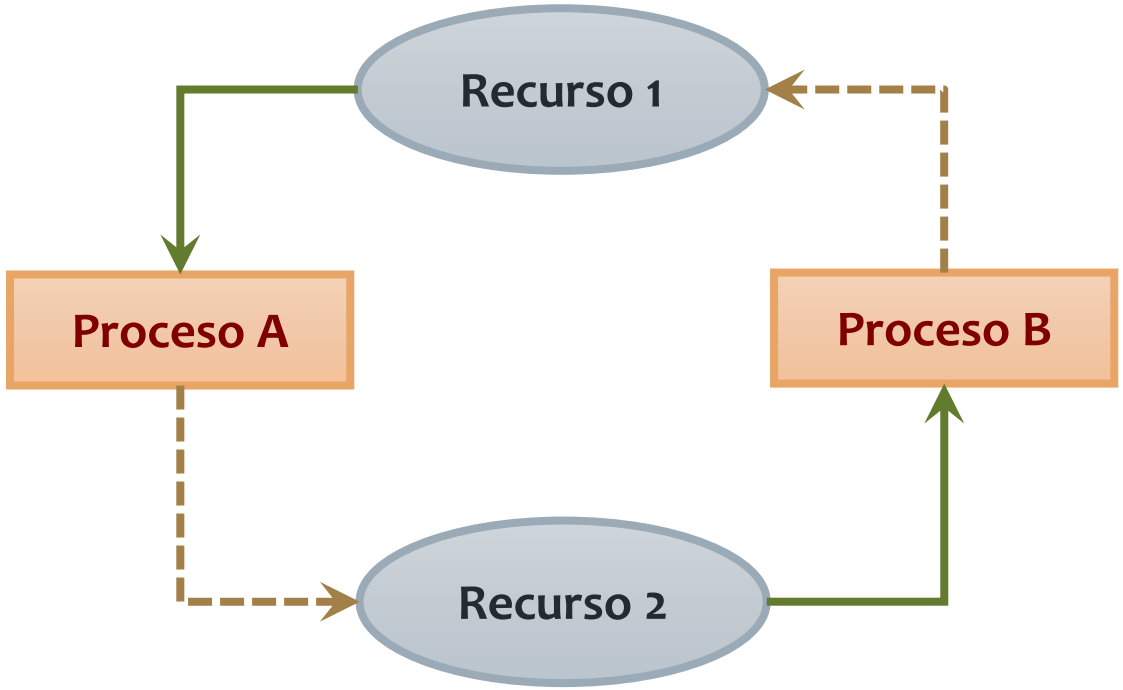
```
Monitor bounded_buffer
{
      Resource buffer[N];
      //  Variables para buffer de indexación
      Condition not_full, not_empty;

      void put_resource (Resource R)
      {
         while (buffer array is full)
               wait (not_full);
         Add R to buffer array;
         signal (not_empty);
      }
```

```
      Resource get_resource ()
      {
            while (buffer array is empty)
                  wait (not_empty);
            Get resource R from buffer array;
            signal (not_full);
            return R;
      }
} //  Fin del monitor
```



Interbloqueo (*deadlocks*)

Ocurre cuando los procesos son incapaces de continuar su ejecución; principalmente por no poder acceder a un recurso.



Condiciones necesarias

Coffman, Elphick y Shoshani establecieron las **cuatro condiciones necesarias** para que se produzca interbloqueo:

- **Exclusión mutua**: los procesos reclaman acceso exclusivo de los recursos que piden.
- **Esperar por**: los procesos mantienen los recursos que ya les habían sido asignados mientras esperan por recursos adicionales.
- **No apropiatividad**: los recursos no pueden ser extraídos de los procesos que los tienen hasta su completa utilización.
- **Espera circular**: existe una cadena circular de procesos en la cual cada uno de ellos mantiene a uno o más recursos que son requeridos por el siguiente proceso de la cadena.

Modelado de Bloqueo Mutuo

Holt en 1972 mostró como pueden modelarse estas cuatro condiciones usando grafos dirigidos.

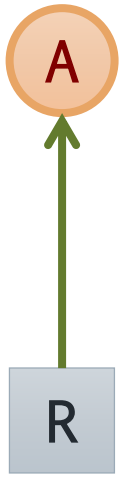
Los grafos tienen dos clases de nodos:

- **Procesos**: se indican con círculos.

- **Recursos**: se indican con cuadrados.

Modelado de Bloqueo Mutuo

Un **arco** que va de un **nodo de recurso** (cuadrado) a uno de **proceso** (círculo) indica que el recurso fue solicitado previamente por el proceso, le fue concedido y actualmente está en su poder.



El recurso **R** está asignado actualmente al proceso **A**.

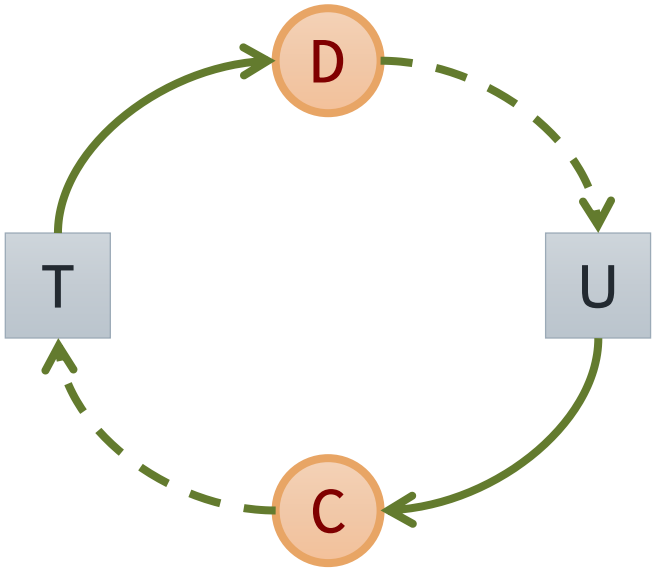
Modelado de Bloqueo Mutuo

Un **arco** de un proceso a un **recurso** indica que el proceso está bloqueado esperando ese recurso.

El proceso **B** solicita el recurso **S**.

Modelado de Bloqueo Mutuo

Un **ciclo** en el grafo implica que hay un **bloqueo mutuo** en el que intervienen los procesos y recursos del ciclo.



Estrategias de Bloqueo Mutuo

- **Prevención.** Modela el comportamiento del sistema para eliminar toda posibilidad de un bloqueo.
- **Evasión.** Impone condiciones menos estrictas. No puede evitar todas las posibilidades de un bloqueo; cuando éste se produce busca *evitar* sus consecuencias.
- **Detección y recuperación.** Permite que ocurran los bloqueos, pero busca determinar si ha ocurrido y actuar para eliminarlos.

Recuperación de un Bloqueo Mutuo

- Terminación de los procesos.
- Arrebato de los recursos: un número de recursos son tomados a la fuerza.
- En función a un COSTO de terminación del Proceso.

Estructura de control de Paralelismo

- Construcción de paralelismo **parbegin/parend**.

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

Estructura secuencial

Cálculo de una raíz de una ecuación cuadrática.

$$X_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2 \times a}$$

$x = (-b + (b^2 - 4 * a * c)^{0.5})/(2 * a)$

Se puede evaluar en un procesador secuencial:

# Paso	Almacena	Operación
1	T1	b * b
2	T2	4 * a
3	T2	T2 * c
4	T1	T1 – T2
5	T1	T1 ** 0.5
6	T2	-1 * b
7	T2	T2 + T1
8	T1	2 * a
9	x	T2 / T1

9 operaciones se ejecutan de una en una, esta secuencia es determinada por las reglas de precedencia de operadores del sistema.

Estructura de control de Paralelismo

En un sistema que maneja procesamiento paralelo, la expresión puede evaluarse como sigue:

1 **parbegin**
```
    temp1 = -1 * b;
    temp2 = b * b;
    temp3 = 4 * a;
    temp4 = 2 * a;
```
parend;

2 temp5 = temp3 * c;

3 temp5 = temp2 – temp5;

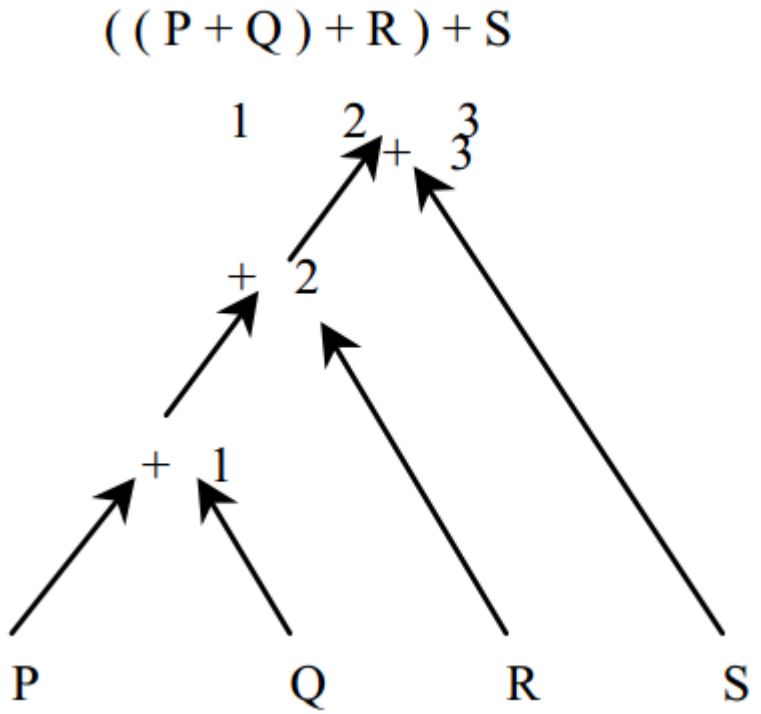
4 temp5 = temp5 ** 0.5;

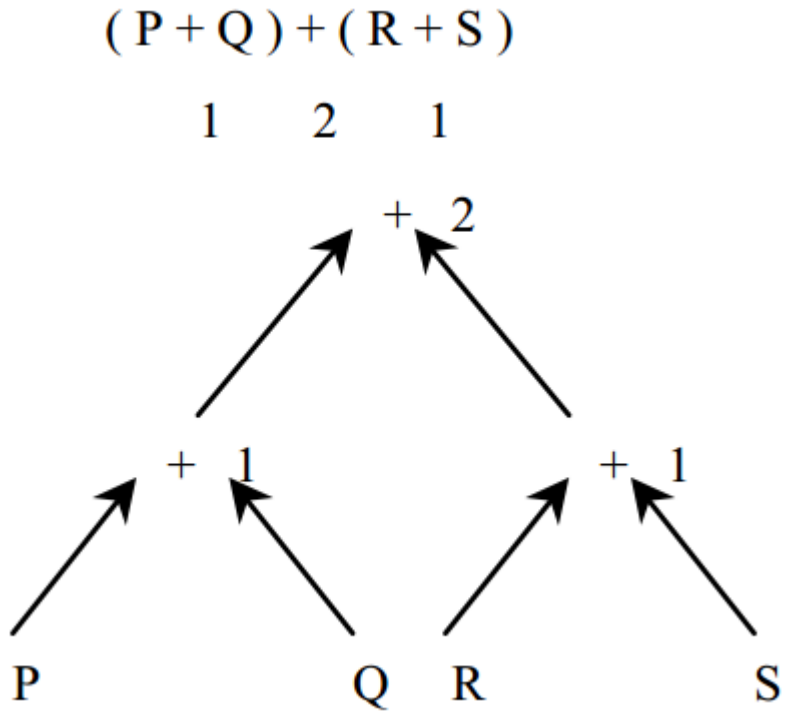
5 temp5 = temp1 + temp5;

6 x = temp5 / temp4

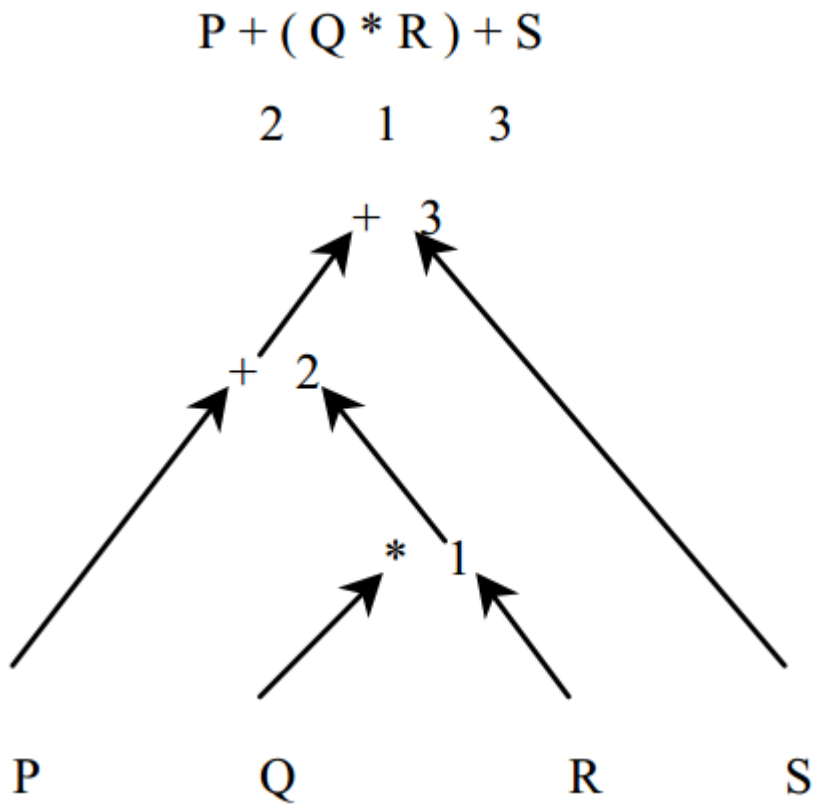
6 operaciones teniendo cuatro procesadores ejecutando instrucciones en paralelo.

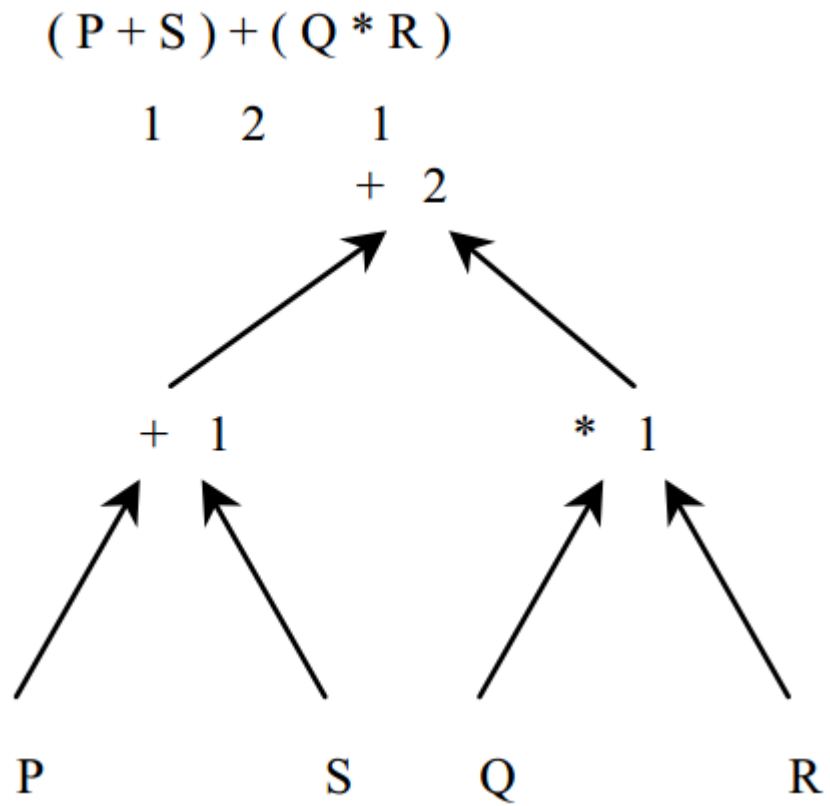
Reducción del Árbol por Asociatividad



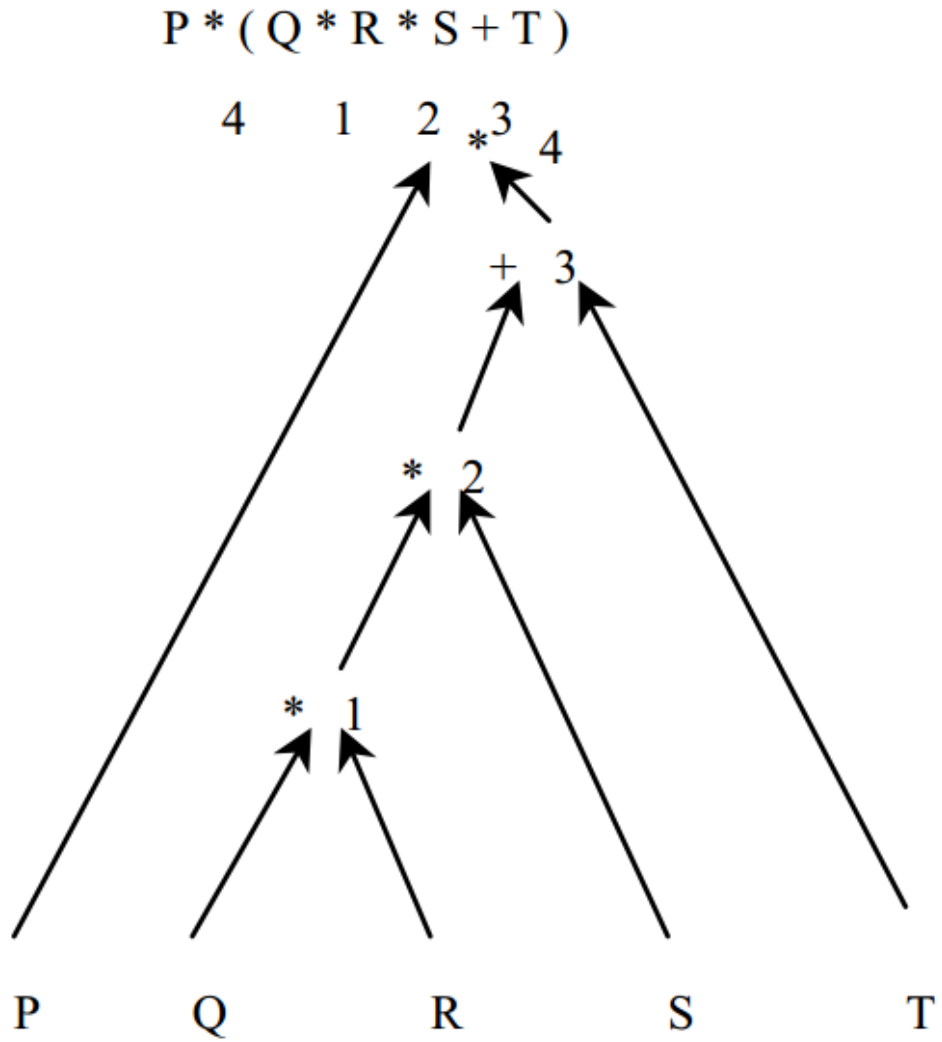


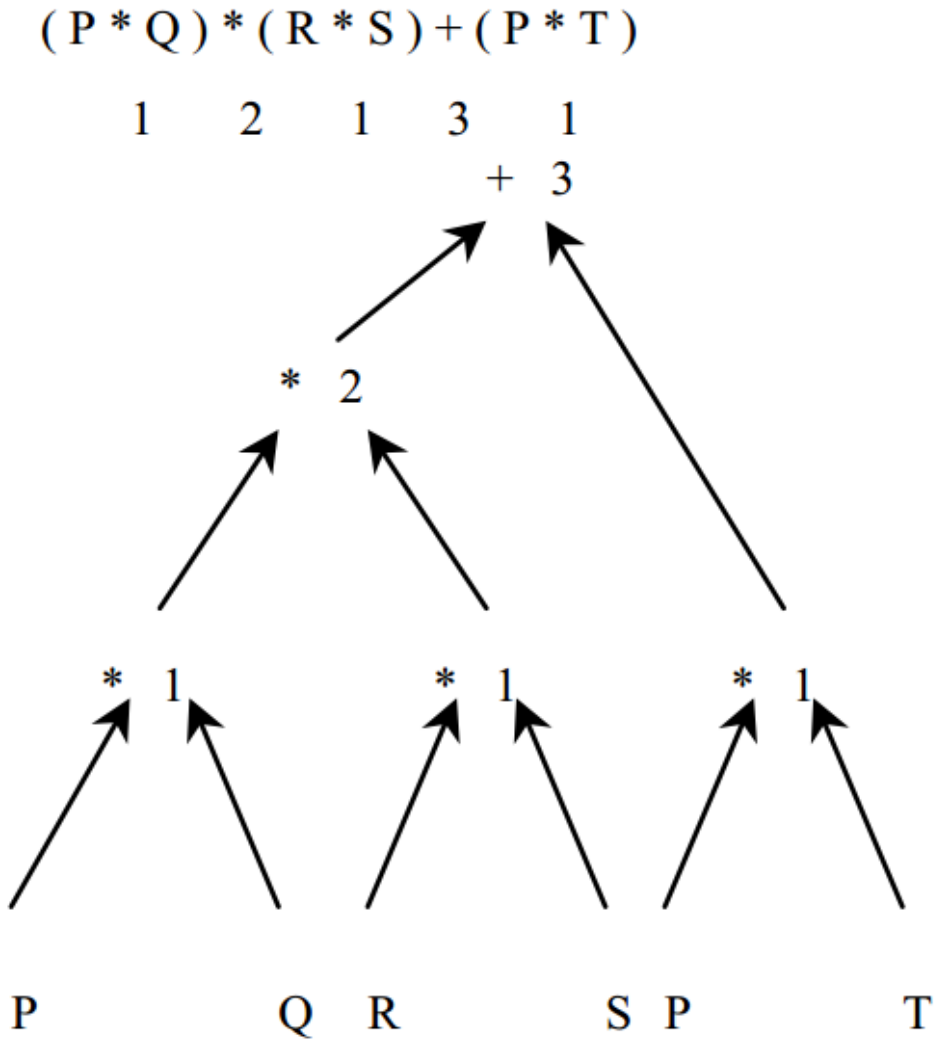
Reducción del Árbol por Conmutatividad





Reducción del Árbol por Distributividad





Algoritmo del Banquero

- El algoritmo del banquero ofrece una forma de asignar los recursos que evita el interbloqueo.
- Permite ejecutar procesos que tendrían que esperar seguramente con alguna de las estrategias de prevención.
- El algoritmo del banquero permite la asignación de recursos a los usuarios solamente cuando la asignación conduzca a estados seguros, y no a estados inseguros.
  - Un **estado seguro** es una situación tal en la que todos los procesos son capaces de terminar en algún momento.

    La clave de un estado seguro es que exista al menos una forma en la que terminen todos los procesos.
  - Un **estado inseguro** es aquel en el cual puede presentarse un **bloqueo mutuo**.

Defectos del Algoritmo del Banquero

El algoritmo tiene varios defectos importantes :

- Requiere un número fijo de recursos asignables.
- Requiere una población de procesos constantes.
- El algoritmo requiere que el SO satisfaga todas las solicitudes de los procesos en un tiempo finito.
- Requiere que los procesos devuelvan sus recursos en un tiempo finito.
- El algoritmo requiere que los procesos declaren por anticipado sus necesidades máximas.

Ejemplo de Estado Seguro

Supóngase que un sistema tiene doce unidades de cinta y tres procesos que las comparten.

Estado I

	Préstamo actual	Necesidad máxima
Proceso 1	1	4
Proceso 2	4	6
Proceso 3	5	8
Disponibles	2	

Ejemplo de Estado Inseguro

Supóngase que un sistema las doce unidades de cinta están asignadas de la siguiente manera.

Estado II

	Préstamo actual	Necesidad máxima
Proceso 1	8	10
Proceso 2	2	5
Proceso 3	1	3
Disponibles	1	

Ejemplo de transición de un Estado Seguro a un Estado Inseguro

Supongamos la situación que se muestra en la siguiente tabla:

Estado III

	Préstamo actual	Necesidad máxima
Proceso 1	1	4
Proceso 2	4	6
Proceso 3	5	8
Disponibles	2	

Ejemplo de transición de un Estado Seguro a un Estado Inseguro

Si el proceso 3 solicita un recurso más. Si el sistema satisface esta petición, el nuevo estado será:

Estado IV

	Préstamo actual	Necesidad máxima
Proceso 1	1	4
Proceso 2	4	6
Proceso 3	6	8
Disponibles	1	

Algoritmo del Banquero para múltiples recursos

Préstamo Actual.- Matriz de n x m que muestra el número de recursos asignados en ese instante a cada uno de los procesos.

Necesidad Máxima.- Matriz de n x m elementos que define la máxima demanda de recursos de cada proceso.

Solicitud actual.- Matriz de n x m que muestra el número de recursos que todavía necesita cada proceso para llevar a cabo su función (petición).

Solicitud Actual = Necesidad Máxima – Préstamo Actual

Recursos Disponibles.- Es un vector de longitud m que indica el número de recursos disponibles de cada tipo.

Recursos Disponibles = Recursos Disponibles + Préstamo Actual

Recursos Totales.- Recursos existentes en el sistema.

Préstamo Actual + Recursos Disponibles (al inicio)

Procedimiento del Algoritmo del Banquero

El algoritmo de comprobación de estado seguro consiste en:

- Buscar un vector de la **Solicitud Actual** cuyas necesidades de recursos sean menores o iguales a los **Recursos Disponibles**. Si no existe, el sistema puede presentar un ***deadlock***, ya que ningún proceso puede ejecutarse hasta el final.
- Suponer que el proceso seleccionado solicita todos los recursos que necesita y termina. Marcar el proceso como terminado y añadir su **Préstamo Actual** al vector de **Recursos Disponibles**.
- Repetir los pasos anteriores hasta que se vayan marcando todos los procesos como terminados o hasta que se llegue a una situación de ***deadlock***.

Ejemplo 1

Procesos	Préstamo Actual	Necesidad Máxima	Solicitud Actual	Recursos Disponibles
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3		
P1	2 0 0	3 2 2		
P2	3 0 2	9 0 2		
P3	2 1 1	2 2 2		
P4	0 0 2	4 3 3		

Recursos Totales

A	B	C
10	5	7

Recursos Disponibles

A	B	C
3	3	2

¿Es un Estado seguro?

Ejemplo 2

Si la solicitud actual de **P1** es (**1 0 2**) ¿Es un Estado seguro?

Procesos	Préstamo Actual	Necesidad Máxima	Solicitud Actual	Recursos Disponibles
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3		
P1	2 0 0	3 2 2		
P2	3 0 2	9 0 2		
P3	2 1 1	2 2 2		
P4	0 0 2	4 3 3		

Recursos Totales

A	B	C
10	5	7

Recursos Disponibles

A	B	C
3	3	2

Bibliografía

- CARRETERO PÉREZ JESÚS
Sistemas Operativos. Una visión aplicada
Mc. Graw Hill/Interamericana de España, 2001

- DEITEL, H. M.
Introducción a los Sistemas Operativos
2a. Edición
Addison Wesley Iberoamericana, 2000

- FLYNN, Ida y McIver A.
Sistemas Operativos
3a. Edición
Thomson Learning, 2001

Bibliografía

- SILBERSCHATZ, GALVIN, GAGNE
Sistemas Operativos.
6a. Edición
Limusa - Wiley, 2002

- STALLINGS, William
Sistemas Operativos
5a. Edición
Prentice Hall, 2005

Ing. Yesenia Carrera Fournier

sofiunam at gmail dot com