SISTEMAS EN TIEMPO REAL

Sistemas operativos de tiempo real

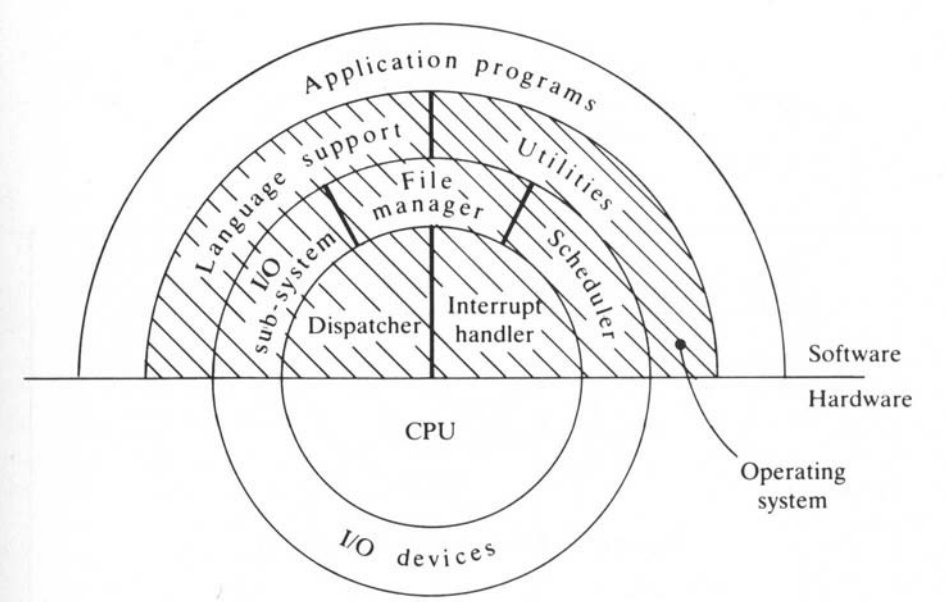
Manuel Agustín Ortiz López
Área de Arquitectura y Tecnología de Computadores
Departamento de Arquitectura de Computadores, Electrónica y Tecnología Electrónica
Universidad de Córdoba

Sistemas operativos de tiempo real

- Indice:

1. Introducción.[Bennett,94]

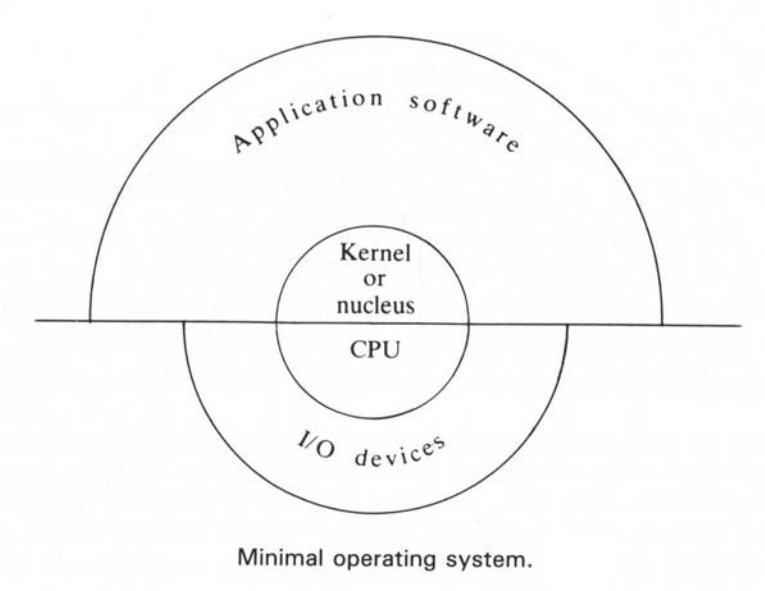
- El propósito de un sistema operativo es facilitar el acceso al hardware del sistema y a los dispositivos de entrada/salida. Un sistema operativo realiza el interfaz entre el usuario y/o las aplicaciones de usuario y el hardware del sistema computacional.
- Un sistema operativo para una máquina concreta convierte el hardware del sistema en una máquina virtual con las características definidas por él.



General purpose operating system.

- En la mayoría de los sistemas de tiempo real y multiprogramación se impide el acceso al hardware y a las interrupciones software, y el sistema operativo se construye para tal propósito.

- En los sistemas operativos que no son multitarea el acceso al hardware no suele estar prohibido, sin embargo es una buena práctica en programación, acceder a través del sistema operativo para aumentar así la portabilidad, ya que los puntos de acceso al hardware a través del sistema operativo pueden mantenerse constantes en diferentes implementaciones.
- Para soportar y controlar actividades básicas los sistemas operativos dan utilidades tales como cargadores, lincadores, ensambladores, depuradores y soporte para los lenguajes de alto nivel, aunque de tipo general, ya que si dieran facilidades particulares, solo para determinadas aplicaciones aumentarían innecesariamente el *overhead* del sistema.
- Durante la instalación del sistema operativo se pueden instalar o no ciertas características ajustándose de esta forma a las necesidades. Los sistemas operativos modernos tienen un pequeño Kernel o núcleo y añaden las características que se desean escribiéndolas en un lenguaje de alto nivel. La siguiente figura muestra un sistema operativo de este tipo:

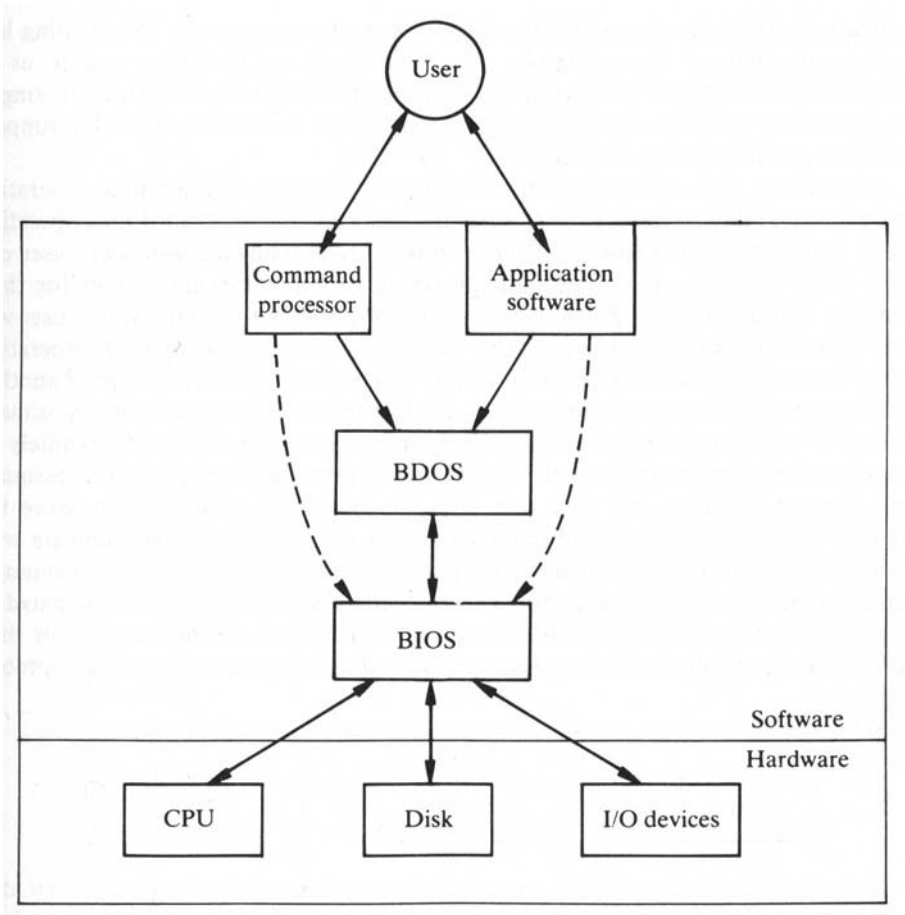


Minimal operating system.

- En este caso la distinción entre el sistema operativo y la aplicación está más difuso. Este tipo de sistema operativo presenta muchas ventajas para aplicaciones pequeñas y sistemas empotrados.

2. Sistema operativo sencillo.[Bennett,94]

- La siguiente figura muestra las distintas partes de un sencillo y simple sistema operativo entre el hardware del computador y el usuario:



General structure of a simple operating system.

  - A través del bloque de órdenes el usuario se comunica con el sistema operativo, y éste le informa de las acciones que se están realizando.
  - El procesamiento de las órdenes dadas lo realiza el BDOS (*Basic Disk Operating System*) que también realiza las operaciones de entrada y salida así como las operaciones de ficheros.

- Las aplicaciones se comunican con el hardware del sistema a través de llamadas al sistema que serán procesadas por el BDOS.
- La BIOS (*Basic Input Ouput System*) contiene los drivers de los dispositivos que son los que manipulan los dispositivos físicos. Esta parte de sistema operativo será distinta de unas implementaciones a otras ya que se opera directamente con el hardware.
- Los dispositivos se tratan como unidades lógicas o físicas. Los dispositivos lógicos son construcciones software utilizadas para simplificar el interfaz de los dispositivos físicos.
- BIOS realiza las entradas y salidas a los dispositivos lógicos y al BDOS, además de enlazar el dispositivo lógico con el físico.
- El acceso a las funciones del sistema operativo se realiza con llamadas a subrutinas y la información se intercambia a través de los registros de la CPU. Estas funciones se llaman directamente desde los lenguajes de alto nivel aislando así al programador del sistema operativo. Este aislamiento es deliberado, aunque no es completo ya que se pueden realizar llamadas utilizando rutinas en ensamblador desde el lenguaje de alto nivel y pasar parámetros entre el lenguaje de alto nivel y el código ensamblador.
- La conexión entre el lenguaje de alto nivel y el sistema operativo lo realiza el compilador a través de las rutinas de *run-time* que convierte el sistema operativo en una máquina virtual descrita por el lenguaje de alto nivel.

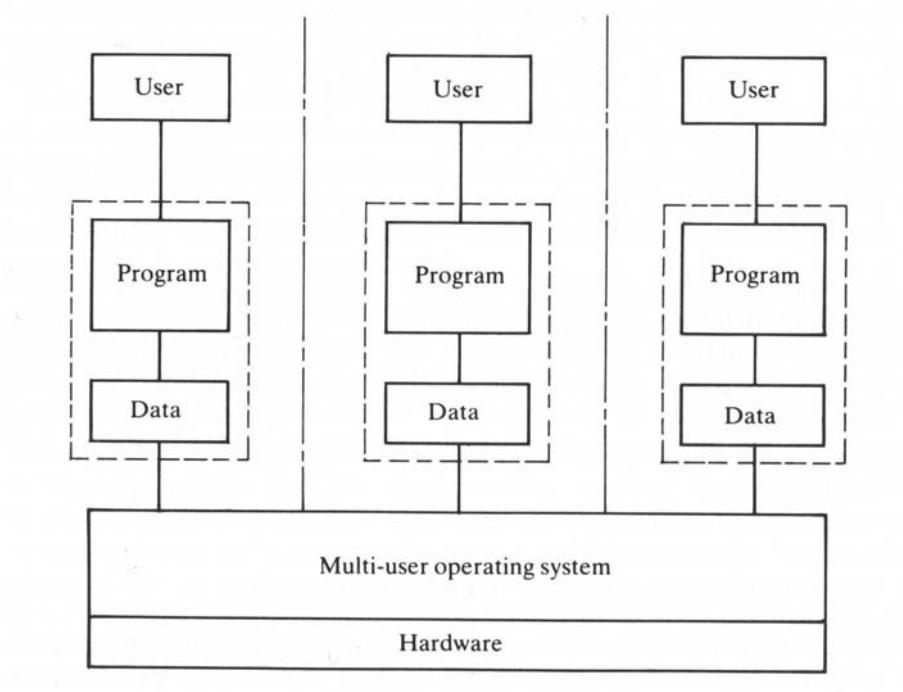
3. Sistemas operativos multitarea de tiempo real

[Bennett,94]

- Hay diferentes tipos de sistemas operativos y hasta principios de los años 80 estaba clara la distinción entre los sistemas operativos diseñados para aplicaciones de tiempo real. Sin embargo actualmente la división entre unos y otras es difusa.
- Veamos a continuación los conceptos de sistema multiusuario y sistema multitarea:

Sistema multiusuario

- La siguiente figura muestra los bloques de los que puede constar un sistema operativo multiusuario:



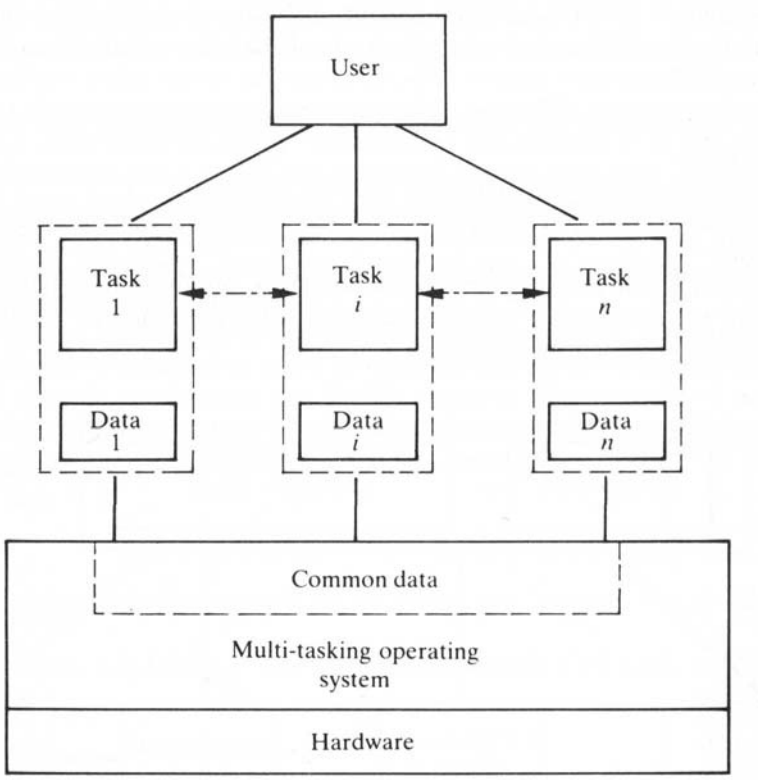
Multi-user operating system.

  - El sistema operativo asegura que cada usuario puede ejecutar su programa como si todo el sistema fuese para él

- En un instante dado no es posible predecir quien está haciendo uso de la CPU o que código de usuario estará en memoria.
- El sistema operativo asegura que el programa de un usuario no interfiere al programa de otro. Cada programa se ejecuta en un entorno protegido.

Sistema multitarea

- En un sistema multitarea se tiene un solo usuario y se ejecutan varias tareas cooperando para cumplir los requisitos del usuario.



Multi-tasking operating system.

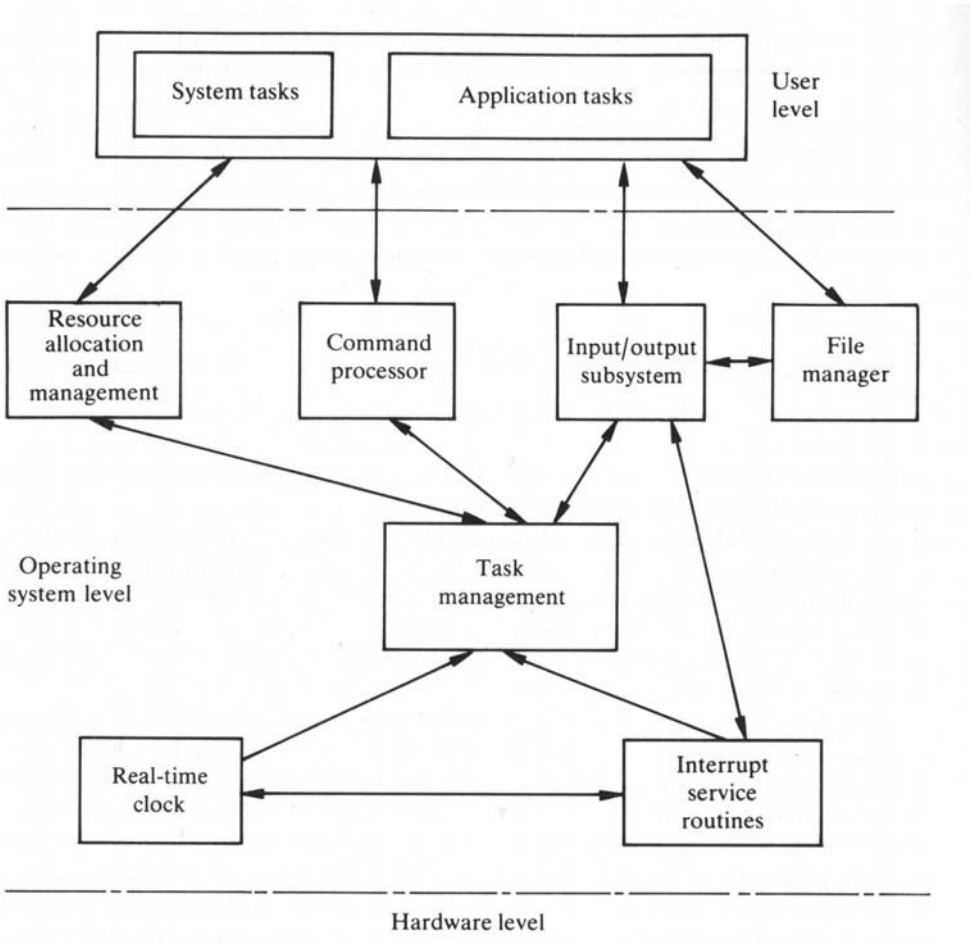
- La comunicación entre las tareas requiere que se tengan unos datos compartidos.
- La comunicación entre las tareas y los datos compartidos se regulará por el sistema operativo que debe ser capaz además de impedir

comunicaciones o accesos a datos indeseados, así como proteger los datos privados de cada tarea.

- Otro requisito fundamental que debe cumplir un sistema operativo es asignar los recursos del computador para todas las actividades que se deben realizar.
- En un sistema operativo de tiempo real esta asignación es complicada por el hecho de que algunas actividades son críticas y unas tienen más prioridad que otras.
- En los sistemas operativos de tiempo real se deben asignar prioridades a tareas y planificar el tiempo de CPU de acuerdo a algún esquema de prioridad.
- Una tarea puede utilizar otra tarea y viceversa.
- El sistema operativo deberá permitir a las tareas la utilización de memoria compartida para intercambiar datos o deberá tener un mecanismo por el que las tareas se puedan enviar mensajes.
- Las tareas pueden necesitar algún evento externo y por tanto el sistema operativo debe soportar la utilización de interrupciones.
- Las tareas pueden requerir acceso a componentes hardware y software, deberá un mecanismo para prevenir que las tareas intenten utilizar el mismo recurso al mismo tiempo.

- En resumen un sistema operativo multitarea de tiempo real debe soportar la utilización de recursos compartidos y los requisitos de tiempo de las tareas.
- Las funciones de un sistema operativo se pueden dividir en:
  - **Administrador de tareas** (*task management*): Asigna la memoria y el tiempo de proceso a las tareas.

- **Administrador de memoria** (*Memory management*): Realiza el control de la memoria asignada.
- **Gestor de recursos** (*Resource control*): Realiza el control de todos los recursos del sistema exceptuando la memoria y el tiempo de CPU.
- **Comunicación y sincronización de tareas** (*Intertask communication and synchronisation*): Da un mecanismo que permita y asegure la comunicación entre tareas así como la sincronización entre ellas.
- Además de las funciones anteriores el S. O. debería dar otras facilidades como: manejo de ficheros, drivers para las entradas y salidas básicas, etc.



Typical structure of a real-time operating system.

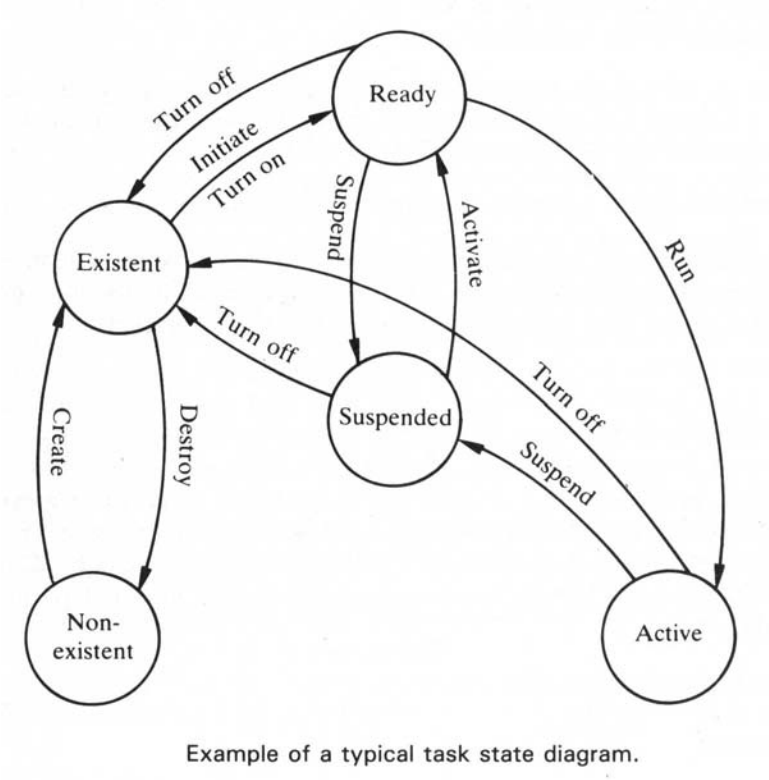
- El control del sistema lo realiza el módulo *task management*, que es responsable de asignar tiempo de CPU. Este módulo también se le llama monitor o *executive*.
- En el mismo nivel de usuario muchos sistemas operativos realizan operaciones y utilidades que se ejecutan en el mismo espacio asignado a la aplicación de usuario. A este espacio se le llama a veces memoria de trabajo.
- Desde el punto de vista del usuario las dos características más importantes del administrador de tareas son la creación de tareas, es decir, como darlas a conocer al sistema operativo de tiempo real y que estrategias de planificación soporta.
- La creación de las tareas es una función dada por el interfaz entre el S. O. y el lenguaje de alto nivel que se utilice.

4. Administrador de tareas.[Bennett,94]

- El sistema operativo es la tarea de más alta prioridad de cualquier sistema, actúa cada vez que se tiene una interrupción o cada vez que se hace una llamada al sistema para pedir un recurso. Los sistemas tienen un núcleo fundamental al que se le llama administrador de tareas o *executive* que realiza las siguientes funciones:
  - Mantener un registro del estado de cada tarea.
  - Planificar un tiempo de CPU para cada tarea.
  - Realizar el cambio de contexto, es decir grabar el estatus de la tarea que está usando la CPU y restaurar el estatus de la tarea a la que se le va a asignar la CPU.
- En la mayoría de los sistemas operativos de tiempo real el administrador de las tareas suele estar dividido en dos partes:
  - El planificador que determinará que tarea se ejecutará y que mantiene el estatus de cada tarea, y
  - El despachador o cargador, que realiza la conmutación de tareas.

Estados de las tareas

- Cuando se tiene un solo procesador solo se ejecutará una tarea, el resto de tareas se encontrará en otro estado. El nombre y el estado que se les da a las tareas que no están en ejecución varía de unos sistemas operativos a otros. La siguiente figura muestra el nombre de los estados de las tareas más usuales:



Example of a typical task state diagram.

- **Activa**: Es la tarea que se está ejecutando, normalmente será la tarea que tenga la máxima prioridad de las que puedan ejecutarse.
- **Preparada o lista**: Las tareas que están en este estado pueden ejecutarse y solo esperan a que esté disponible la CPU.
- **Suspendida o en espera**: Las tareas que se encuentran en este estado están esperando a que algún recurso este disponible o esperando a alguna señal exterior, o esperando a que pase un tiempo.
- **Existente**: El sistema operativo conoce la tarea pero aún no se le ha asignado una prioridad.
- **No Existente o dormida**: El sistema operativo no tiene conocimiento de esta tarea pero podría estar residente en la memoria del computador.
- Los estados de existente y no existente solo aparecen en aquellos sistemas operativos donde hay un número máximo de tareas.

- El estado de las tareas cambia por acciones del sistema operativo(un recurso se hace disponible) o por ordenes de la aplicación. Por ejemplo una orden típica:

  TURN ON(ID): Pone una de existente a lista. ID es el nombre por el que la tarea es conocida.

- Las ordenes clásicas se pueden ver en la siguiente tabla:

RTOS task state transition commands

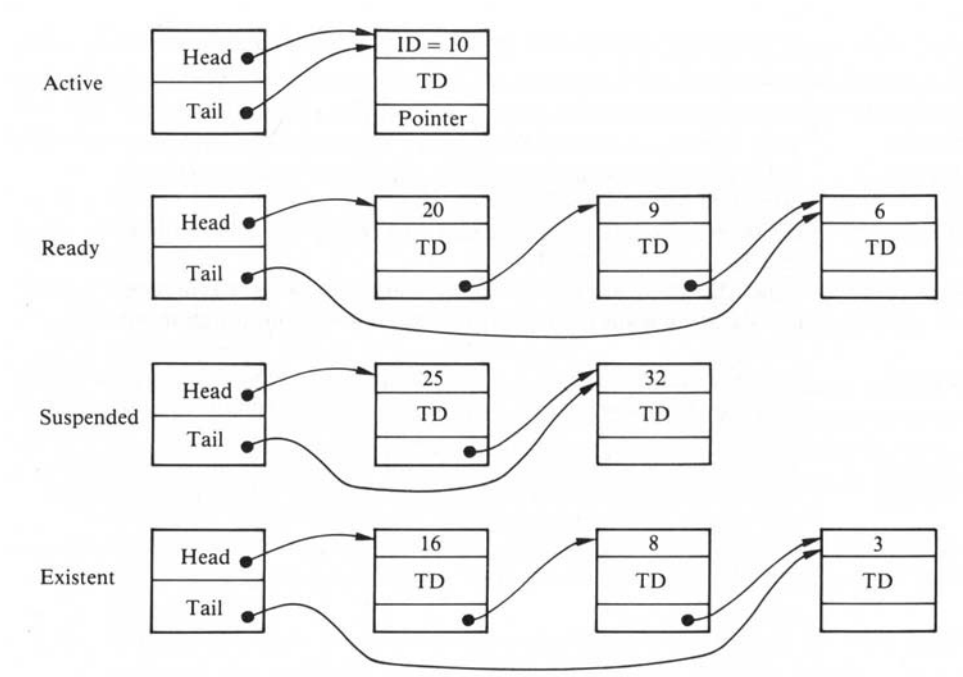
OFFC01	Turn off the task leaving the memory marked as occupied
OFFC02	Turn off the task leaving the memory marked as unoccupied
DELC01	Delay the task leaving the memory marked as occupied; delay is calculated using current value of time
DELC02	Delay the task leaving the memory marked as unoccupied; delay is calculated as for DELC01
DELC03	Delay the task leaving the memory marked as occupied; delay is calculated by adding the delay to the value of time stored in the task descriptor
TPNC01	Turn the task on; will be accepted if the task is ON, OFF or DELAYED; either the ON constant can be placed in the task descriptor or a specified turn-on time
TPNC02	Turn on the task; will only be accepted if the task is in the OFF state
TPNC03	Run the task immediately regardless of priority; will be accepted if the task is ON, OFF or DELAYED

Descriptor de tareas

- La información acerca del estatus de cada tarea la guarda el sistema operativo en un bloque de memoria. A este bloque se le da el nombre de descriptor de tareas (TD) o bloque de control de tareas (TCB).
- La información que se guarda en el TCB varía de unos sistemas a otros, pero típicamente en:
  - Identificación de la tarea ID.
  - Prioridad de la tarea P.

- Estado de la tarea.
- Área donde se deposita el contexto de la tarea o bien un puntero a otra zona.
- Puntero a la siguiente tarea de la lista.

- El puntero que punta a la siguiente tarea de la lista aparece para conseguir una estructura de listas enlazadas.
- El administrador mantiene una lista por cada tipo de estado de tareas.



List structure for holding task state information.

- Hay una tarea activa (ID=10) y tres tareas preparadas (ID=20,9 y 6).
- La ventaja de las listas enlazadas es que los descriptores de las tareas pueden residir en cualquier parte de la memoria y así el sistema operativo no tiene por qué estar restringido a un número fijo de tareas como ocurría con los sistemas operativos antiguos que tenían una tabla de descriptores con un tamaño fijo.

- El movimiento de tareas de una lista a otra, las ordenaciones dentro de cada lista, etc., se consigue fácilmente cambiando los punteros, y no se tienen que mover ni copiar los descriptores completos.

- La información que se deposita cuando una tarea se suspende por el planificador suele ser:
  - Información interna: contador de programa, registros de la CPU, stack pointer.
  - Datos de la tarea: Stack de la tarea. Área general de trabajo de la tarea (heap).

5. Administrador de memoria.[Tanenbaum,97]

- En la mayoría de las aplicaciones de control el software es estático (el software no se crea o elimina en tiempo de ejecución) por lo que el problema del manejo de la memoria es más simple que en los sistemas de multiprogramación on-line.
- Existen varias formas de administración de la memoria dependiendo de sí el sistema es monotarea o multitarea y/o dependiendo de sí hay intercambio entre las tareas que están en memoria y en otra memoria auxiliar:

5.1. Administración Básica de memoria. [Tanenbaum,97]

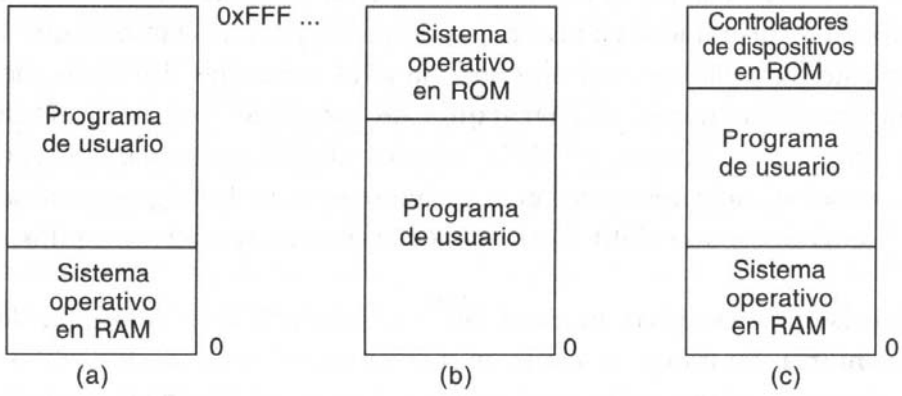


Figura 4-1. Tres formas sencillas de organizar la memoria con un sistema operativo y un proceso de usuario. También existen otras posibilidades.

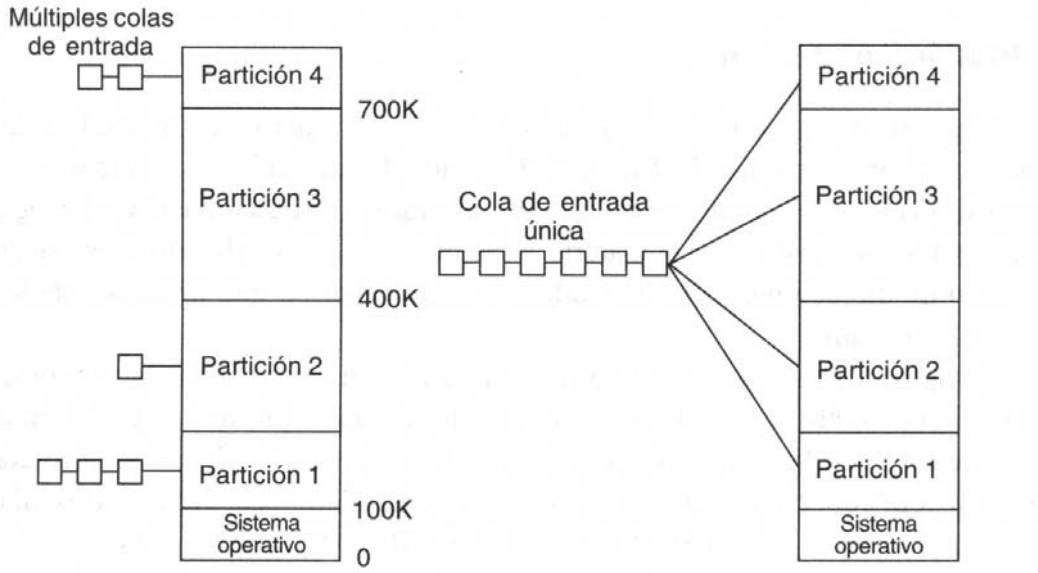


Figura 4-2. (a) Particiones de memoria fijas con colas de entrada individuales para cada partición. (b) Particiones de memoria fija con una sola cola de entrada.

5.2. Intercambio. [Tanenbaum,97]

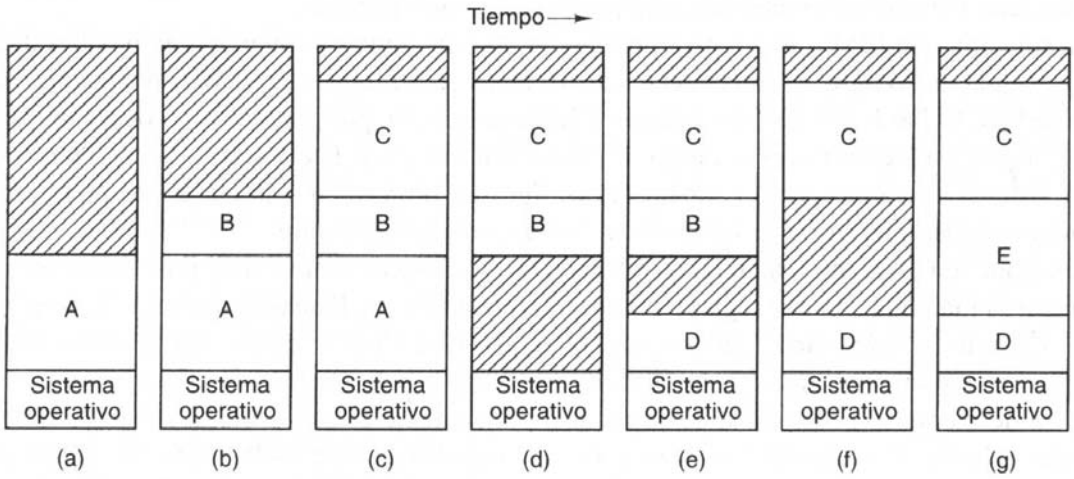


Figura 4-3. La asignación de memoria cambia conforme los procesos entran en la memoria y salen de ella. Las regiones sombreadas son memoria no utilizada.

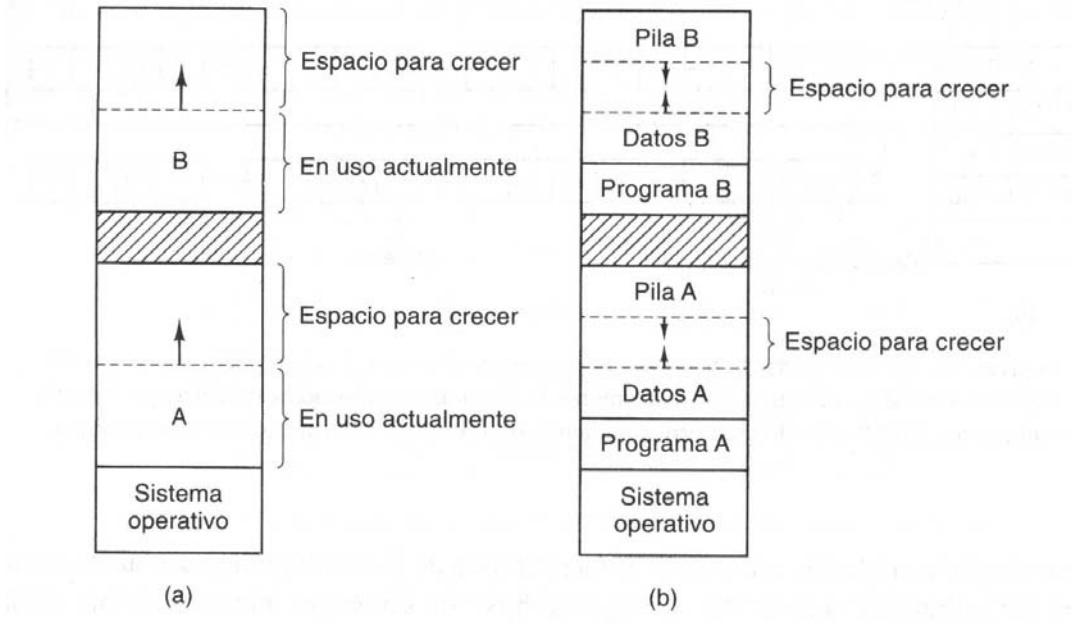


Figura 4-4. (a) Asignación de espacio para un segmento de datos que crece. (b) Asignación de espacio para una pila que crece y un segmento de datos que crece.

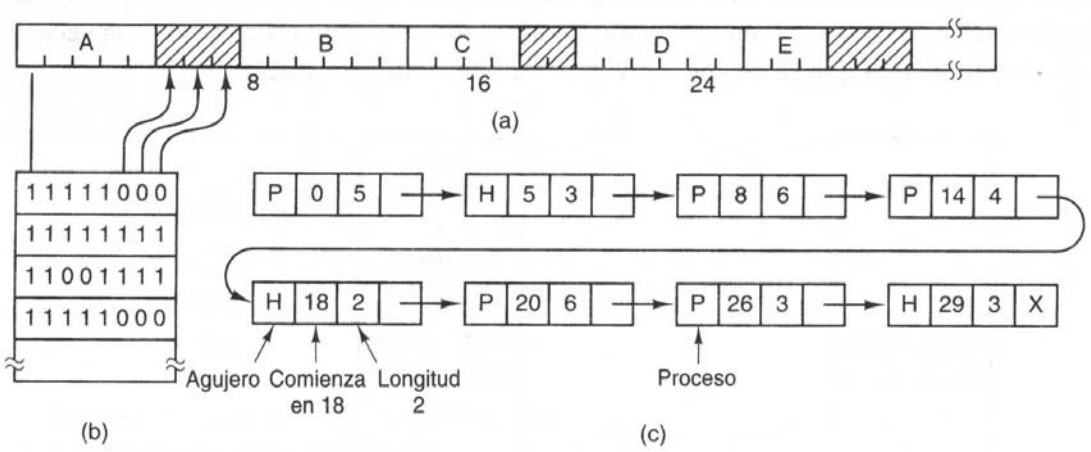


Figura 4-5. (a) Una parte de la memoria con cinco procesos y tres agujeros. Las marcas indican las unidades de asignación de memoria. Las regiones sombreadas (0 en el mapa de bits) están libres. (b) El mapa de bits correspondiente. (c) La misma información en forma de lista.

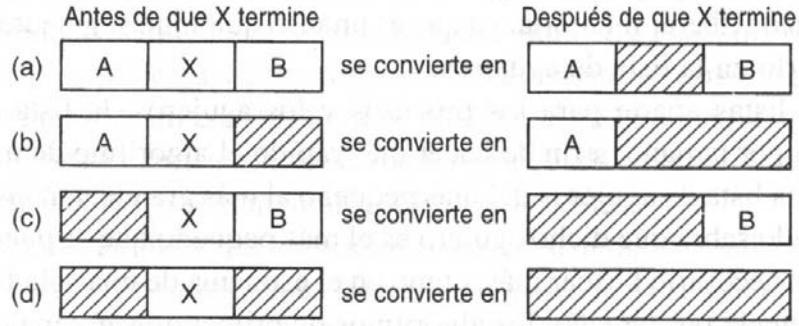
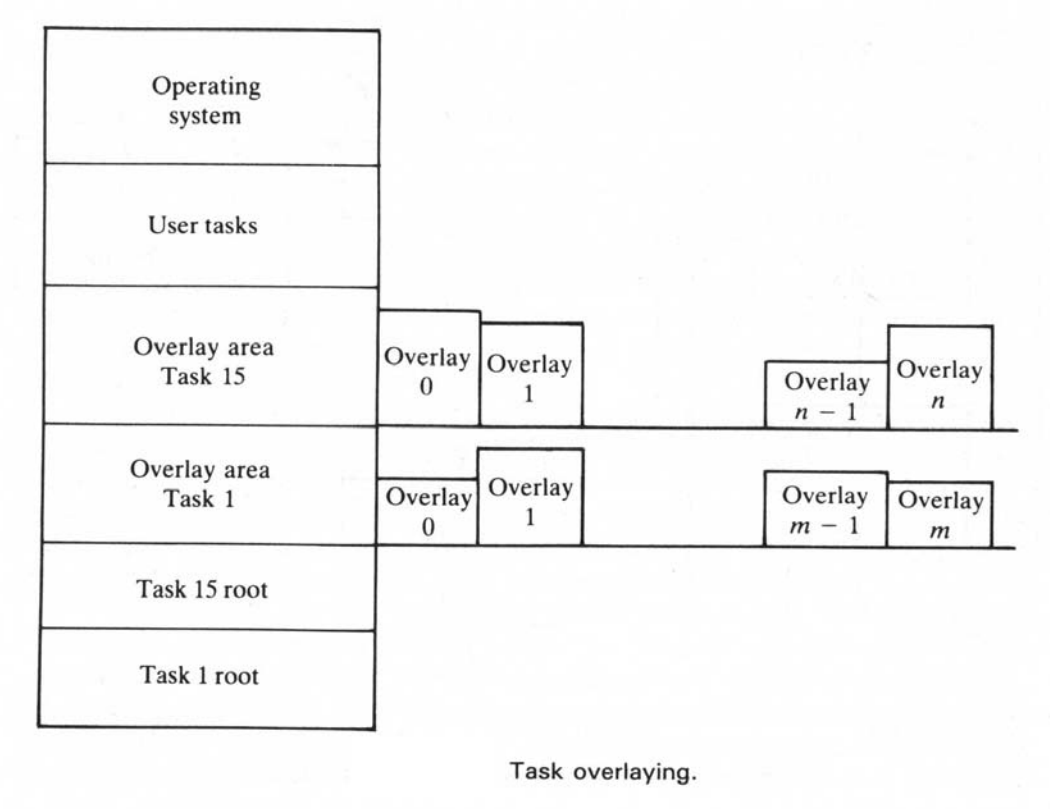


Figura 4-6. Cuatro combinaciones de vecinos para el proceso que termina, *X*.

5.3. Intercambio de segmentos de programa por las propias tareas. [Bennett,94]

- Existen sistemas de reserva dinámica de memoria que además permiten que las propias tareas controlen qué segmentos de programa se encuentran en cada momento en su zona de memoria asignada. Existen dos mecanismos de transferencia: encadenamiento (chaining) y superposición (overlaying).
- En el mecanismo de encadenamiento la tarea se divide en segmentos que se van ejecutando secuencialmente. Una vez ejecutado un segmento, el segmento siguiente se carga en el mismo lugar que éste que ha finalizado. Los datos se deben mantener o bien en disco o en un área común de la memoria.
- En el caso de superposición una parte de la tarea (llamada raíz) permanece en la memoria y los segmentos son llevados al área de memoria dedicada al solapamiento.
- En los sistemas multitarea puede haber varias áreas de superposición en la memoria las cuales pueden ser compartidas por varias tareas. En la siguiente figura se tiene un ejemplo en el que las tareas 1y 15 utilizan superposición:



Task overlaying.

- Las tareas 1 y 15 mantienen un segmento raíz y varios *Overlays* (trozos) que están fuera de las áreas de superposición en memoria auxiliar o en disco.

- La mayoría de los sistemas de tiempo real complejos que tenían facilidades para intercambio de tareas entre la memoria principal y una memoria auxiliar estaban así diseñados porque estos sistemas tenían pequeñas cantidades de memoria principal, del orden de 32K, y con un espacio de direccionamiento limitado. A medida que han mejorado estos sistemas microcomputadores, incrementando el espacio de direcciones e integrando una unidad de manejo de memoria MMU han hecho innecesario utilizar *allocation* (reserva) dinámica de memoria. En la actualidad estos mecanismos de intercambio se siguen utilizando en sistemas que tienen

microprocesadores con un espacio de direccionamiento pequeño, del orden de 64K.

- La *allocation* dinámica de memoria es compleja de manejar y debería evitarse siempre que sea posible en los sistemas empotrados. La memoria RAM es tan barata que el coste de añadir memoria extra es mucho menor que el coste de programación para dar *allocation* dinámica de memoria, y nunca debe utilizarse en aplicaciones críticas.
- Por lo que respecta a la utilización de memoria virtual debería evitarse en los sistemas de tiempo real ya es una fuente importante de incertidumbre en lo que respecta al tiempo de ejecución. El tiempo que se necesita para manejar fallos de página, por ejemplo, puede dispararse y por tanto obtener una buena velocidad en el tratamiento de los fallos de página es casi siempre imposible.[Krishna,97]

6. Procesos e Hilos.[Grehan,98]

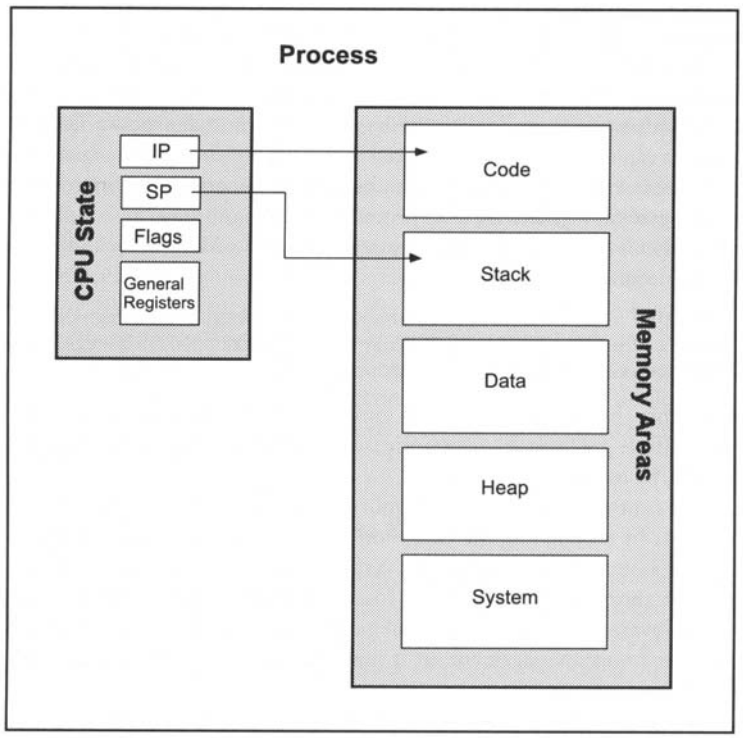


Figure 12-1: A process is defined by the CPU state (the contents of specific registers) as well as a number of memory areas that the operating system allocates and assigns to the process.

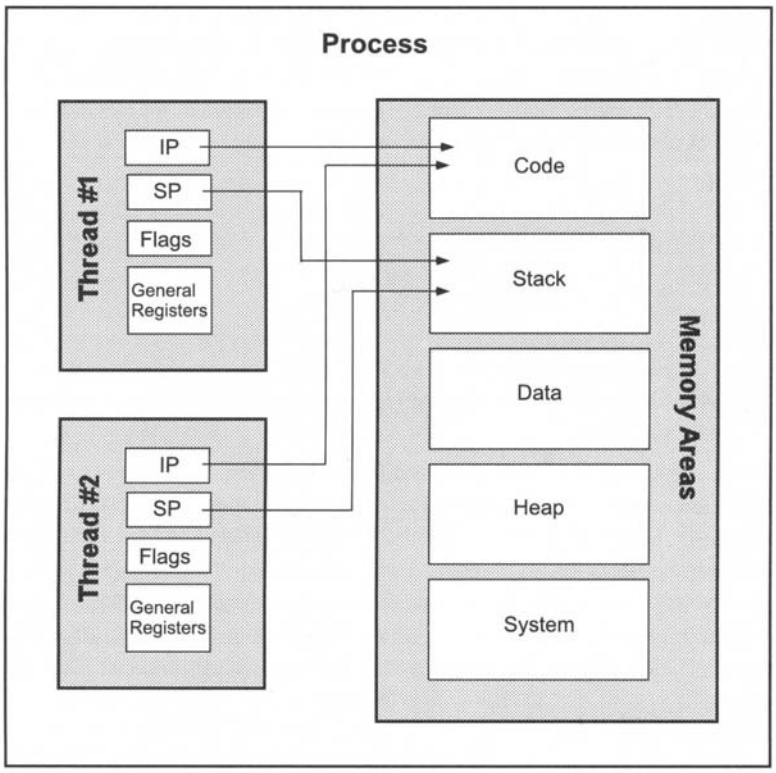


Figure 12-2: In a multithreaded system, a process can be composed of more than one thread. Each thread is a separate flow of execution. Notice that Thread #1 and Thread #2 are executing at different points in the code area.

7. Comunicación entre procesos.[Tanembaum,97]

- Los procesos dentro de un sistema pueden trabajar de forma:
  - Independiente.
  - Cooperativa.
  - Competitiva.
- Se pueden dar combinaciones de las tres formas anteriores, es decir, pueden trabajar en un momento cooperativamente y otro momento competitivamente, o pueden ser independientes y en otro momento competitivos.
- Se requiere por tanto sincronización y comunicación entre procesos. Hay dos mecanismos para que dos procesos intercambien información: Variables compartidas y paso de mensajes.
  - Variables compartidas. Este mecanismo es fácil de soportar si hay memoria compartida entre los procesos.
  - El paso de mensajes puede ser soportado vía memoria compartida o por una red física de paso de mensajes.
- La comunicación entre procesos lleva consigo tres problemas:
  - Cómo se puede pasar información de un proceso a otro.
  - Cómo asegurarse de que los procesos no interfieran al efectuar actividades críticas.
  - Como conseguir la secuencia correcta cuando existan dependencias.

7.1. Secciones críticas. [Tanembaum,97]

- Ejemplo: *X=X+1*;
- Esta sentencia de un lenguaje de alto nivel cualquiera, por ejemplo C, no se ejecutará como una operación indivisible sino:
  - 1. Se carga el valor de *X* (dirección de memoria) en un registro.
  - 2. Se incrementa en 1 el valor del registro.
  - 3. Se lleva el valor del registro a *X* (dirección de memoria).
- Tenemos por tanto, tres operaciones. Si ahora los procesos se entremezclan y quieren actualizar la variable *X* se producirá un valor incorrecto. Por ejemplo si ocurriera la secuencia:
  - X=5 al inicio
  - El proceso P1 carga el valor 5
  - Se cambia la planificación y entra el proceso P2
  - El proceso P2 carga el valor 5
  - Incrementa el proceso P2
  - Lleva el valor del registro 6 a *X*
  - En otro momento se devuelve el control al proceso P1
  - Incrementa el proceso P1
  - Lleva el valor del registro 6 a *X*

  Al final *X* valdrá 6 y no 7 como debería ser, por tanto se tiene un resultado incorrecto.

- Un conjunto de operaciones que deberían ejecutarse indivisiblemente se le llama **sección crítica**. O lo que es lo mismo aquella parte del programa que accede a memoria compartida se le llama sección crítica o región crítica (algunos autores diferencian entre sección crítica y región crítica).
- La sincronización requerida para proteger una sección crítica se le llama **exclusión mutua**.
- Las condiciones de competencia se evitarían si dos procesos nunca pudieran estar en su región crítica al mismo tiempo. Sin embargo esta condición no es suficiente para que los procesos paralelos cooperen de manera correcta y eficiente usando datos compartidos. Para solucionar estos problemas se tiene que cumplir cuatro reglas:
  - Dos procesos nunca pueden estar simultáneamente dentro de sus regiones críticas.
  - No puede suponerse nada acerca de las velocidades y el número de CPUs.
  - Ningún proceso que se ejecute fuera de su región crítica puede bloquear a otros procesos.
  - Ningún proceso deberá tener que esperar indefinidamente para entrar en su región crítica.

7.2. Exclusión mutua con espera activa (Busy Waiting). [Tanembaum,97]

- En este apartado vamos a ver las soluciones y los defectos de las propuestas para la exclusión mutua con espera activa, es decir, mientras un proceso está en su región crítica el otro espera ejecutando un ciclo de espera.

Inhabilitación de interrupciones

- Una solución al problema de la exclusión mutua es que un proceso inhabilite las interrupciones justo al entrar en su sección crítica y las habilita justo al salir. Con esto se garantiza que no se conmutará a otro proceso.
- El defecto de este método es que es arriesgado conferir a los procesos de usuario la posibilidad de actuar sobre las interrupciones porque podría este proceso no habilitarlas y el sistema se quedaría colgado.
- Por otro lado si el sistema es multiprocesador la inhabilitación de interrupciones sólo afectaría a los procesos que estuvieran en ese procesador y no los del resto.
- La utilización de las interrupciones es una técnica útil dentro del sistema operativo pero no es apropiada como mecanismo de exclusión mutua general para los procesos de usuario.

Variables candado

- Se trata de una solución software. Se define una variable compartida que indique si un proceso se encuentra o no, en su sección crítica. Supongamos una variable candado cuyo valor inicial sea por ejemplo cero. Un proceso que quiera entrar en su sección crítica realiza los siguientes pasos:
  - Comprueba si el candado está a 1, si no es así, el proceso le asigna a esta variable 1 y entra en su sección crítica.
  - Si el candado está a 1 espera a que esté a 0.
- El problema que tiene este método es que no resuelve el problema de la exclusión mutua. En el siguiente ejemplo se ha considerado dos procesos y dos variables candado: *flag1* y *flag2*

```
process P1;
  loop
    flag1 := up; (* announce intent to enter *)
    while flag2 = up do
      null     (* busy wait if the other process is in *)
    end;       (* its critical section *)
    <critical section>
    flag1 := down; (* exit protocol *)
    <non-critical section>
  end
end P1;
```

```
process P2;
  loop
    flag2 := up;
    while flag1 = up do
      null
    end;
    <critical section>
    flag2 := down;
    <non-critical section>
  end
end P2;
```

Both processes announce their intention to enter their critical sections and then check to see if the other process is in its critical section. Unfortunately, this 'solution' suffers from a not insignificant problem. Consider an interleaving that has the following progression:

```
P1 sets its flag (flag1 now up)
P2 sets its flag (flag2 now up)
P2 checks flag1 (it is up therefore P2 loops)
P2 enters its busy wait
P1 checks flag2 (it is up therefore P1 loops)
P1 enters its busy wait
```

```
process P1;
  loop
    while flag2 = up do
      null (* busy wait if the other process is in *)
    end;   (* its critical section *)
    flag1 := up; (* announce intent to enter *)
    <critical section>
    flag1 := down; (* exit protocol *)
    <non-critical section>
  end
end P1;
```

```
process P2;
  loop
    while flag1 = up do
      null
    end;
    flag2 := up;
    <critical section>
    flag2 := down;
    <non-critical section>
  end
end P2;
```

Now we can produce an interleaving that actually fails to give mutual exclusion.

```
P1 and P2 are in their non-critical sections (flag1 = flag2 = down)
P1 checks flag2 (it is down)
P2 checks flag1 (it is down)
P2 sets its flag (flag2 now up)
P2 enters critical section
P1 sets it flag (flag1 now up)
P1 enters critical section
(P1 and P2 are both in their critical sections).
```

Alternancia estricta[Tanenbaum,97]

- Veamos con un ejemplo en C esta estrategia:

```
while (TRUE) {
 while(turn != 0) /* esperar */ ;
 critical_region();
 turn = 1;
 noncritical_region();
}
```

(a)

```
while (TRUE) {
 while(turn != 1) /* esperar */ ;
 critical_region();
 turn = 0;
 noncritical_region();
}
```

(b)

Figura 2-8. Solución propuesta para el problema de la región crítica.

- La estrategia de alternancia no es una buena solución ya que aunque soluciona el problema de la exclusión mutua viola otras reglas que enunciamos al principio ya que un proceso puede bloquear al otro aunque no se encuentre en su sección crítica. Principalmente el problema de la alternancia estricta es que si un proceso es mucho más lento que otro obliga a que los dos progresen a la velocidad del más lento se encuentre o no ejecutando una sección crítica.

Solución de Peterson [Tanenbaum,97]

- En 1981 G. L. Peterson dio una solución software para resolver el problema de la exclusión mutua mejorando otras soluciones software anteriores. El algoritmo de Peterson consta de dos procedimientos:
  - Enter_region(número de proceso)
  - Leave_region(número de proceso)

```
#define FALSE       0
#define TRUE        1
#define N           2                   /* número de procesos */

int turn;                               /* ¿a quién le toca? */
int interested[N];                      /* todos los valores son inicialmente 0 (FALSE) */

void enter_region(int process);         /* proceso 0 o 1 */
{
 int other;                             /* número del otro proceso */

 other = 1 – process;                   /* lo opuesto de process */
 interested[process] = TRUE;            /* mostrar interés */
 turn = process;                        /* establecer bandera */
 while (turn == process && interested[other] == TRUE) /* instrucción nula */ ;
}

void leave_region(int process)          /* process: quién sale */
{
 interested[process] = FALSE;           /* indicar salida de la región crítica */
}
```

Solución de Peterson para lograr la exclusión mutua.

- Antes de utilizar las variables compartidas cada proceso debe invocar *enter_región* con su número de proceso como parámetro. Esta invocación le obligará a esperar hasta que pueda entrar sin peligro y manipular las variables compartidas. Para indicar que se ha finalizado, se invoca la función *leave_region*, permitiendo así que otro proceso pueda entrar.

Instrucción TSL [Tanenbaum,97]

- TEST AND SET LOCK (probar y fijar candado) que permite resolver el problema de exclusión mutua.
- TSL lee el contenido de una posición de memoria, lo coloca en un registro y almacena un valor distinto de cero. Se garantiza que las operaciones de escritura y lectura son indivisibles.
- Imprescindible en entornos multiprocesador.

- A continuación se muestra como se utiliza la instrucción *TSL* para implementar las funciones *enter_region* y *leave_region*:

```
enter_region:
        tsl register,lock       | copiar lock en register y asignarle 1
        cmp register,#0         | ¿era lock 0?
        jne enter_region        | si no era cero, se asignó 1 a lock, y se ejecuta el ciclo
        ret                     | volver al invocador; se entró en la región crítica

leave_region:
        move lock,#0            | guardar un 0 en lock
        ret                     | volver al invocador
```

Establecimiento y liberación de candados con TSL.

7.3. Dormir y despertar. [Tanembaum,97]

- La solución de Peterson y la que utiliza TSL son correctas, pero tienen el defecto de requerir espera activa.
- Esta espera activa presenta dos graves inconvenientes:
  - Se consume tiempo de CPU.
  - Se puede presentar el problema de inversión de prioridad.

Inversión de prioridad

- Supongamos dos procesos H y L. H tiene más prioridad que L.
- 1. L entra en su región crítica.
- 2. H queda listo para ejecutarse
- 3. Se produce un cambio de planificación y pasa a ejecutarse H
- 4. H inicia la espera activa
- Como L nunca se planifica, por tener menor prioiridad que H, nunca saldrá de su región crítica y H permanece en un lazo infinito.

- Primitivas de comunicación SLEEP-WAKEUP
  - Permiten la comunicación entre procesos que se bloquean sin desperdiciar tiempo de CPU.
  - SLEEP (dormir). Llamada al sistema que provoca la suspensión del proceso que lo invoca hasta que otro proceso lo despierte.
  - WAKEUP (despertar). Llamada al sistema que despierta al proceso que se indica.
  - Para enlazar SLEEP-WAKEUP se suele utilizar una dirección de memoria.

El problema del productor-consumidor. [Tanembaum,97]

- Problema clásico que se soluciona utilizando las primitivas SLEEP-WAKEUP.
- Premisas:
  - Dos procesos comparten un buffer de tamaño fijo.
  - Uno de ellos, el productor, coloca información en el buffer y el otro, el consumidor, los saca.
- Problemas:
  - 1.Buffer lleno
    - Solución: Dormir al productor y que sea despertado cuando el consumidor haya retirado uno o más elementos.
  - 2. Buffer vacio
    - Solución: Dormir al consumidor y esperar a que el productor deposite un elemento y despertar al consumidor.
- Además podrían surgir problemas de competencia que se producirían cuando se intente escribir con el buffer lleno porque el productor fuese más rápido que el consumidor o que se intentara leer del buffer vacío si el consumidor fuese más rápido que el productor.
  - Para resolver el problema se define el numero de elementos en el buffer en una variable llamada *count*. Si *count=N* entonces el buffer está lleno y si *count =0* entonces el buffer está vacío.

```
#define N 100                                   /* número de ranuras del buffer */
int count = 0;                                  /* número de elementos en el buffer */

void producer(void)
{
 while (TRUE) {                                 /* repetir indefinidamente */
        produce_item();                         /* generar el siguiente elemento */
        if (count == N) sleep();                /* si el buffer está lleno, dormir */
        enter_item();                           /* colocar elemento en el buffer */
        count = count + 1;                      /* incrementar la cuenta de elementos
                                                   en  el buffer */
        if (count == 1) wakeup(consumer);       /* ¿estaba vacío el buffer? */
 }
}

void consumer(void)
{
 while (TRUE){                                  /* repetir indefinidamente */
        if (count == 0) sleep();                /* si el buffer está vacío, dormir */
        remove_item();                          /* remover elemento del buffer */
        count = count –1;                       /* decrementar la cuenta de elementos
                                                   en el buffer */
        if (count == N–1) wakeup(producer);     /* ¿estaba lleno el buffer? */
        consume_item();                         /* imprimir elemento */
 }
}
```

El problema de productor-consumidor con una condición de competencia fatal.

- Aún con la variable *count* no se soluciona el problema de la competencia. Veámoslo:
  - 1. El buffer está vacío y el consumidor acaba de leer *count* para ver si está a cero.
  - 2. Se produce un cambio de planificación
  - 3. Se ejecuta el productor y coloca un nuevo elemento en el buffer e incrementa *count*, que al valer 1 deberá hacer despertar al consumidor (pero el consumidor no se había dormido lógicamente sino por un cambio de planificación) y este despertar se pierde.
  - 4. El consumidor se despierta y como *count=0* se duerme, pero ya no lo despertará el productor

- 5. El buffer se llenará y el consumidor se dormirá. Ambos se dormirán.
- Problema:
  - Se ha perdido la llamada para despertar un proceso que no estaba lógicamente dormido,
- Solución:
  - Definir un bit de espera de despertar.
- En este ejemplo se soluciona el problema con un solo bit pero se pueden buscar ejemplos a partir de tres procesos en los que con un solo bit de despertar no es suficiente. Es necesario introducir el concepto de semáforo.

7.4. Semáforos. [Tanembaum,97]

- E. W. DIJKSTRA en 1965 sugirió utilizar una variable entera para guardar el número de señales de despertar para uso futuro. Propuso un nuevo tipo de variable llamada semáforo:
  - Semáforo =0; No hay señales de despertar guardadas.
  - Semáforo >0 ; Hay señales de despertar.
- Propuso sobre un semáforo dos funciones DOWN y UP, que son generalizaciones de WAKEUP y SLEEP.
- DOWN(semáforo) opera de la siguiente manera:
  - Si semáforo ≠ 0 se decrementa el semáforo y continúa.
  - Si semáforo = 0 se pone a dormir, sin completar la operación de decrementar por el momento.
  - Estos dos pasos: verificar el valor del semáforo; modificar el valor o dormirse, deben realizarse como una sola acción atómica indivisible.
- UP(semáforo) incrementa el valor del semáforo, y si algunos procesos están esperando, el sistema elige uno de ellos y le permite completar la función DOWN.
- Se garantiza que una vez iniciada una acción sobre un semáforo ningún otro proceso accederá al semáforo hasta que se haya completado la acción del semáforo.
- Las funciones UP y DOWN se implementan como llamadas al sistema, el cual inhabilita las interrupciones. Si además están implementadas en sistemas con múltiples CPU's cada

semáforo debe protegerse con una instrucción TSL para asegurarse que una sola CPU examine el semáforo.

```
#define N 100                                   /* número de ranuras del buffer */
typedef int semaphore;                          /* los semáforos son un tipo especial de int */
semaphore mutex = 1;                            /* controla el acceso a la región crítica */
semaphore empty = N;                            /* cuenta las ranuras de buffer vacías */
semaphore full = 0                              /* cuenta las ranuras de buffer llenas */

void producer(void)
{
 int item;

 while (TRUE) {                                 /* TRUE es la constante 1 */
        produce_item(&item);                    /* generar algo para ponerlo en el buffer */
        down(&empty);                           /* decrementar el contador empty */
        down(&mutex);                           /* entrar en la región crítica */
        enter_item(item);                       /* colocar el nuevo elemento en el buffer */
        up(&mutex);                             /* salir de la región crítica */
        up(&full);                              /* incrementar el contador de ranuras llenas */
 }
}

void consumer(void)
{
 int item;

 while (TRUE) {                                 /* ciclo infinito */
        down(&full);                            /* decrementar el contador full */
        down(&mutex);                           /* entrar en la región crítica */
        remove_item(&item);                     /* sacar elemento del buffer */
        up(&mutex);                             /* salir de la región crítica */
        up(&empty);                             /* incrementar el contador de ranuras vacías */
        consume_item(item);                     /* hacer algo con el elemento */
 }
}
```

El problema de productor-consumidor usando semáforos.

- Los semáforos que solo pueden tomar los valores 0 o 1, y que se utilizan por dos o más procesos para que solo uno de ellos pueda entrar en su región crítica se llaman semáforos binarios
- Los semáforos tienen doble utilidad:
  - Exclusión mutua.
  - Sincronización.

- Los semáforos solucionan el problema del productor-consumidor, sin embargo una descuidada utilización de los semáforos podría crear problemas.
- Problema: Supóngase que en el código del productor el programador se hubiera confundido y hubiese invertido el orden de las funciones *down(&empty)* y *down(&mutex)* y se tuviera en un momento dado el buffer lleno y comenzando a ejecutarse el productor:
  - 1. Mutex se decrementa y se tiene exclusión mutua en el acceso al buffer.
  - 2. Como el buffer está lleno, al ejecutarse *down(&empty)* el productor se dormirá, y además con mutex a cero.
  - 3. Entra en ejecución el consumidor y cuando ejecuta *down(&mutex)* como mutex está a cero, el consumidor se dormiría también.
  - Se llega a que tanto el productor como le consumidor se encuentran suspendidos y sin posibilidad de despertar. Esta situación se llama de *bloqueo mutuo*.

7.5. Monitores. [Tanembaum,97]

- Monitor: Primitiva de sincronización de alto nivel propuesta por HOARE (1974) y BRINCH HANSEN (1975) para facilitar la escritura de programas correctos.
- Los monitores son un conjunto de procedimientos, variables y estructuras de datos agrupados en un tipo especial de módulo o paquete. Los monitores son construcciones del lenguaje de programación, y éste conoce que son especiales.
- Los procesos pueden invocar a los procedimientos de un monitor cuando lo deseen, pero no pueden acceder directamente a las estructuras de datos desde procedimientos fuera del monitor.

```
monitor example
  integer i;
  condition c;

  procedure producer(x);
  .
  .
  .
  end;

  procedure consumer(x);
  .
  .
  .
  end;
end monitor;
```

Un monitor.

- Los monitores consiguen la exclusión mutua con la propiedad de que un solo proceso puede estar activo en un monitor en un momento dado.
- Cuando un proceso llama a un monitor, lo que primero que hace éste es verificar si hay algún proceso activo dentro del monitor y si es así, se suspende hasta que el otro proceso salga.
- Con monitores es responsabilidad del compilador implementar la exclusión mutua, por lo que es mucho menos probable que algo salga mal. La exclusión mutua se realiza generalmente en los monitores con semáforos.

El problema del productor-consumidor con monitores

- Solo con monitores no se soluciona el problema del productor-consumidor, se necesita un mecanismo para bloquear un proceso cuando no pueda continuar. En el caso del productor debe bloquearse cuando el buffer está lleno, y en el caso del consumidor debe bloquearse cuando el buffer esté lleno.
- Se introducen las variables condición junto con dos operaciones que se realizan sobre ellas WAIT y SIGNAL.
- La diferencia de estas funciones con SLEEP y WAKEUP presentados anteriormente es que fallaron porque un proceso despertó a otro sin que estuviera lógicamente dormido, sino que estaba detenido por un cambio de planificación. Con monitores no puede darse esta situación ya que los monitores garantizan exclusión mutua.

- Aunque el productor descubra el buffer lleno y el planificador cambie la planificación, el consumidor no podrá entrar hasta que el monitor no complete la ejecución del WAIT.

```
monitor ProducerConsumer
 condition full, empty;
 integer count;

 procedure enter;
 begin
  if count = N then wait(full);
  enter_item;
  count := count + 1;
  if count = 1 then signal(empty)
 end;

 procedure remove;
 begin
  if count = 0 then wait(empty);
  remove_item;
  count := count - 1;
  if count = N - 1 then signal(full)
 end;

 count := 0;
end monitor;

procedure producer;
begin
 while true do
 begin
  produce_item;
  ProducerConsumer.enter
 end
end;

procedure consumer;
begin
 while true do
 begin
  ProducerConsumer.remove;
  consume_item
 end
end;
```

Bosquejo del problema de productor-consumidor con monitores. Sólo un procedimiento de monitor está activo a la vez. El *buffer* tiene *N* ranuras.

- Inconvenientes de los monitores:
  - La mayoría de los lenguajes no lo tienen definido y por tanto el compilador no los reconoce.
  - Los monitores se diseñaron para una máquina con una CPU (o varias si se usa la instrucción TSL) pero no para sistemas distribuidos.

7.6. Transferencia de Mensajes. [Burns,97] [Tanembaum,97]

- Este método de comunicación entre procesos utiliza dos primitivas SEND y RECEIVE que son llamadas al sistema como los semáforos y no construcciones del lenguaje como los monitores.
  - Send(destino,&mensaje) : Envía un mensaje a un destino.
  - Receive(origen, &mensaje) : recibe un mensaje de un origen (o de cualquiera). Si no hay mensaje disponible el receptor puede bloquearse hasta que llegue uno o puede regresar con un código de error.

Sincronización entre procesos

- La transferencia de mensajes lleva implícitamente una sincronización ya que el destino no puede procesar el mensaje si previamente no se le ha enviado ninguno. Esto es distinto al sistema de variables compartidas donde el proceso que recibe puede leer una variable sin saber si previamente la ha escrito el proceso emisor.
- La variación entre los distintos modelo de sincronización de procesos en la transferencia de mensajes está en la operación de envío. Estos modelos se pueden clasificar en:
  - Asíncrono (o sin espera). El emisor continúa procesando independientemente de sí el mensaje se recibió o no. (este mecanismo lo utiliza POSIX).

- Síncrono. El emisor continúa procesando solo si el mensaje se ha recibido.
- Invocación remota. El emisor continúa procesando sólo cuando ha recibido una respuesta del destino. (Este modelo es el que utiliza ADA).

- Para ver la diferencia entre uno y otro pongamos un ejemplo paralelo:
  - Asíncrono: correo
  - Síncrono. Conversación telefónica. El emisor espera hasta que contacte con el destinatario y confirme la autenticidad y envía el mensaje sin esperar respuesta.
  - Invocación remota. Siguiendo el ejemplo anterior si ahora el emisor espera respuesta de lo que se está hablando tenemos entonces una invocación remota.
- Cuando el emisor y el destinatario avanzan juntos en una comunicación sincronizada tenemos una estrategia de *rendezvous* o cita. Si se pueden realizar además computaciones antes de que se reciba la respuesta tenemos lo que se llama *extended rendezvous*.
- Evidentemente hay relaciones entre las tres formas de envío:
  - Dos comunicaciones asíncronas pueden constituir una relación síncrona:

P1	P2
asyn_send(mensaje) wait(reconocimiento)	wait(mensaje) asyn_send(reconocimiento)

- Dos comunicaciones síncronas pueden utilizarse para construir una invocación remota:

P1	P2
syn_send(mensaje) wait(respuesta)	wait(mensaje) ... Construye la respuesta ... syn_send(respuesta)

Nombrando a los procesos

- La llamada a los procesos tiene dos aspectos importantes: dirección e indirección; simetría o asimetría
  - Esquema de llamada directa. El emisor nombra explícitamente al destinatario:

    send<mensaje> to <nombre_proceso>
  - Esquema de llamada indirecta. El emisor utiliza una entidad intermedia (canal, mailbox, link port, pipe).

    El nombramiento directo tiene la ventaja de la simplicidad mientras que el indirecto ayuda a la descomposición del software, por ejemplo un mailbox puede verse como un interfaz entre distintas partes del programa.
  - Un esquema de llamadas es simétrico si ambos, el emisor y el receptor se nombran uno a otro.
  - Un esquema de llamadas es asimétrico si el destinatario no especifica el nombre del emisor sino que acepta mensajes de cualquier proceso o mailbox

    wait <mensaje>

El esquema asimétrico encaja en el paradigma de programación cliente-servidor en el que el servidor da respuesta a cualquier mensaje de los procesos cliente. Una implementación de este tipo debe ser capaz de soportar una cola de procesos que esperan servicio.

Estructura de los mensajes

- Idealmente los lenguajes deberían permitir que cualquier tipo de datos se pudiera transmitir en un mensaje, sin embargo este propósito es difícil, particularmente si el emisor y receptor tienen diferentes tipos de datos y sobre todo si se utilizan punteros. Por estos motivos algunos lenguajes tienen restricciones respecto al tamaño fijo de objetos y tipos definidos por el sistema. Los lenguajes modernos han acabado con estas restricciones aunque sin embargo los sistemas operativos todavía requieren que los datos se conviertan en bytes antes de ser transmitidos.

8. Control de recursos. [Tanembaum,97][Burns,97][Bennett,94]

- El control de los recursos tiene dos aspectos, uno la utilización de los recursos por las aplicaciones y otro como se implementa el sistema de entrada/salida en un sistema operativo.

8.1. Utilización de recursos. [Tanembaum,97][Burns,97]

- Cuando dos procesos cooperativos o competitivos necesitan un recurso, lo solicitan, lo utilizan y lo liberan.
- Un recurso se puede solicitar para acceder a él de dos formas:
  - Acceso compartido. El recurso puede ser compartido por dos o más procesos. Ejemplo: fichero de solo lectura.
  - Acceso exclusivo. Sólo un proceso accede a un recurso a la vez. Ejemplo: impresora.
  - Hay recursos que pueden ser accedidos unas veces en un modo y otras veces de otro.
- Si un proceso intenta acceder a un recurso en modo compartido y está siendo utilizado en modo exclusivo debe esperar. El caso contrario también obliga al proceso a esperar.
- Ya que los procesos pueden ser bloqueados cuando piden un recurso no deben pedirlo hasta que lo necesiten y deben liberarlos tan rápido como sea posible.

- Un sistema multiproceso puede estar falto de viveza por alguna de las siguientes causas:
  - DEADLOCK. Se produce cuando dos procesos no pueden progresar, permaneciendo suspendidos. Por ejemplo si un proceso *P1* tiene un recurso *R1* y pide *R2* y otro *P2* tiene *R2* y pide *R1*. Ambos procesos estarán esperando a que cada uno libere los recursos. En este caso los procesos están suspendidos indefinidamente.
  - LIVELOCK. Es similar al caso anterior pero los procesos siguen ejecutándose.
  - BLOQUEO o POSPOSICIÓN INDEFINIDA. Este problema ocurre cuando varios procesos están continuamente intentando conseguir acceso exclusivo al mismo recurso. Si la política de obtención de los recursos no es equitativa puede hacer que un proceso nunca consiga un recurso.

Condiciones necesarias para que exista un DEADLOCK

- Hay cuatro condiciones que deben darse para que ocurra un deadlock. Si no se tiene alguna de las cuatro, no se producirá un deadlock:
  - 1. Exclusión mutua. Solo un proceso puede utilizar un recurso a la vez.
  - 2. Mantenimiento y espera. Debe existir un proceso que mantenga un recurso mientras otros esperan.
  - 3. No debe haber apropiación. Un recurso solo puede ser liberado voluntariamente por el proceso que lo posee.

- 4. Espera circular. Debe existir un encadenamiento circular de procesos de tal forma que cada proceso mantenga recursos que están siendo pedidos por el siguiente proceso en la cadena.

- Un sistema de tiempo real puede abordar los deadlock de alguna de las tres maneras siguientes:
  - Prevención de deadlock.
  - Eludir los deadlock.
  - Detectar los deadlock y recuperarse.

Prevención de DEADLOCK

- Para que no se produzca un deadlock debe evitarse alguna de las cuatro condiciones y para ello se debería actuar:
  - 1. En exclusión mutua. Para que no se produzca esta condición se deberían restringir la utilización de los recursos.
  - 2. En mantenimiento y espera. Una manera simple de evitar los deadlock es que los procesos pidan recursos antes de que comiencen a ejecutarse o en los instantes de ejecución en los que no tengan recursos pedidos. Desgraciadamente esta manera de operar conduce a una ineficiente utilización de los recursos e incluso puede conducir al bloqueo.
  - 3. En la no apropiación. Si se permitiera apropiación se solucionaría el problema de los deadlock. Sin embargo posibilitar la apropiación en cuanto recursos, presenta el inconveniente de que se debe guardar y restaurar posteriormente el estado del recurso y en muchos casos esto no puede conseguirse.

- 4. En cuanto a la espera circular. Para evitar esperas circulares se debe imponer un orden lineal en todos los recursos. Para ello veamos como se opera. Supongamos R el conjunto de los tipos de recursos

  $R=\{r_1, r_2, r_3, ..., r_n\}$

  Supongamos una función F que aplicada a un tipo de recurso devuelve su posición en el orden lineal. Para evitar la espera circular se debe actuar de la siguiente forma:

  - Si un proceso tiene un recurso r_j y pide el recurso r_k la petición se considerará solo sí $F(r_j) < F(r_k)$
  - O alternativamente desde el punto de vista de los procesos, un proceso que pide un recurso r_j debería liberar todos los recursos r_i tales que $F(r_i) < F(r_j)$

  La función F se crearía de acuerdo a la utilización del recurso.

Eludir los DEADLOCK: Algoritmo del banquero

- Un sistema es seguro si puede reservar recursos a cada proceso en algún orden que permita evitar los deadlock. Un algoritmo sencillo que permite eludir los deadlock es el algoritmo del banquero.
- El algoritmo del banquero fue diseñado por Dijkstra (1965). Se trata de un algoritmo de planificación que permite evitar el bloqueo mutuo. Veamos en que consiste este algoritmo para un solo tipo de recurso del que se tiene varias unidades.

- Para comprender el algoritmo del banquero para un solo recurso se utiliza el símil de un banquero que trata con un grupo de clientes a los que ha concedido una línea de crédito.
  - La siguiente tabla muestra las unidades de crédito concedidas para cuatro clientes

Nombre	Utilizados	Máximo
Andrés	0	6
Bárbara	0	5
Miguel	0	4
Susana	0	7

Disponibles: 10

  - El banquero sabe que no todos los clientes van a necesitar un crédito máximo de inmediato, por lo que ha reservado 10 unidades en lugar de 22.
  - Los clientes hacen sus peticiones de crédito y en un momento dado la situación es:

Nombre	Utilizados	Máximo
Andrés	1	6
Bárbara	1	5
Miguel	2	4
Susana	4	7

Disponibles: 2

Seguro.

  - Se denomina estado del sistema, respecto a la asignación de recursos, a una lista donde aparezca los clientes junto con las unidades prestadas y el crédito máximo disponible.

- Se dice que un estado es seguro si existe una secuencia de otros estados que conduzca a una situación en la que todos los clientes obtienen prestamos hasta sus límites de crédito.
- El estado de la tabla anterior es seguro, porque aunque solo tiene disponible dos unidades puede posponer todas las peticiones excepto la de Miguel, y una vez que esta finalice, tendrá disponible cuatro unidades para otro cliente.
- En la siguiente tabla se muestra un estado inseguro, ya que el banquero no podría atender ninguna petición de utilizar su crédito máximo.

Nombre	Utilizados	Máximo
Andrés	1	6
Bárbara	2	5
Miguel	2	4
Susana	4	7

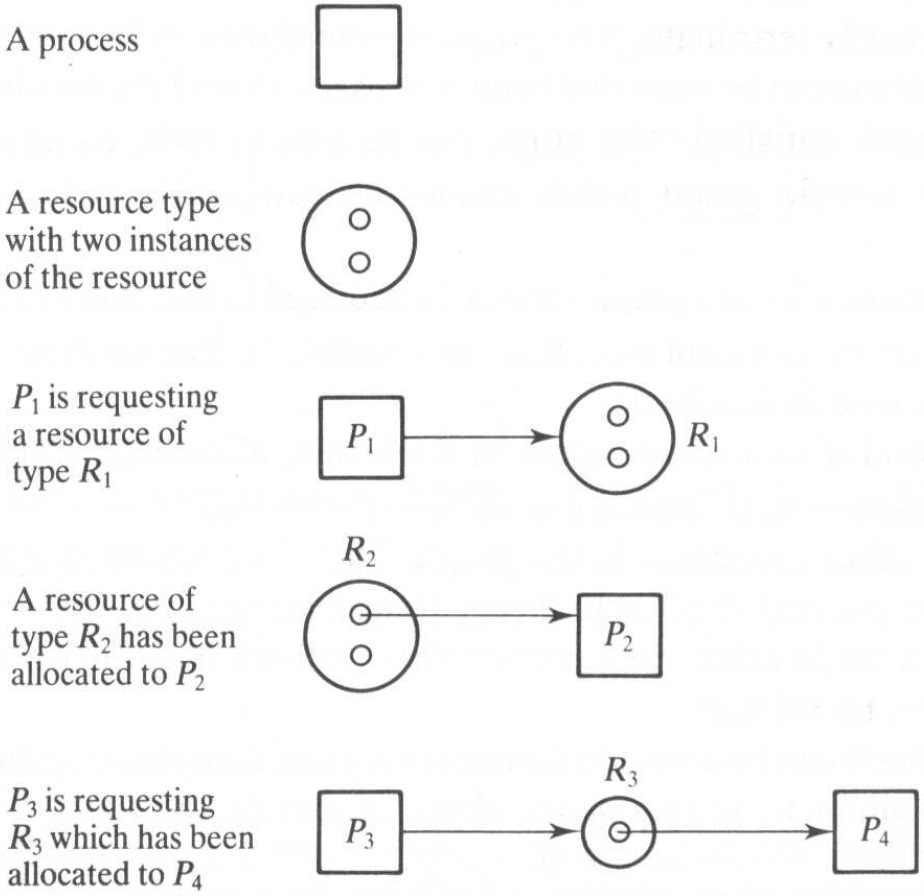
Disponibles: 1

Inseguro.

Detección de DEADLOCK y recuperación

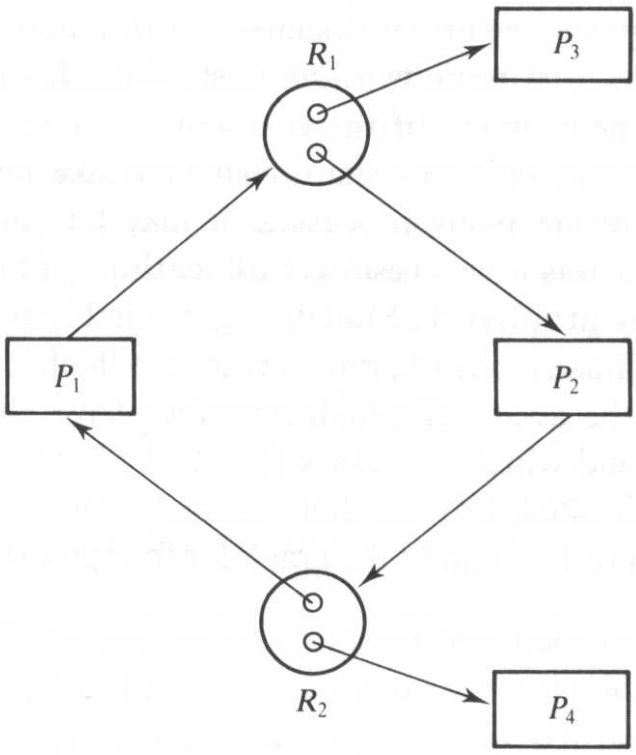
- En muchos sistemas concurrentes de propósito general la reserva o apropiación de recursos es a priori desconocido, y aunque fuese conocido el coste de evitar el deadlock lo hace prohibitivo. Consecuentemente muchos de estos sistemas ignoran el problema del deadlock hasta que entran en ese estado y entonces toman una medida correctiva.

- Una alternativa es la utilización de grafos de reserva recursos:



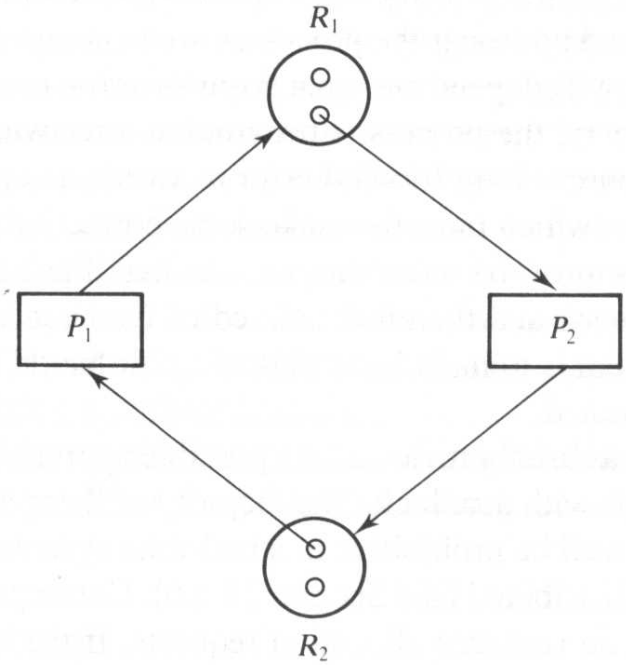
Notation for resource allocation graphs.

- Para detectar si un grupo de procesos están en deadlock se requiere que el sistema de reserva de recursos sea capaz de conocer que recursos han sido reservados y a que procesos, y que procesos están bloqueados esperando recursos que han sido reservados. Con esta información se construye un grafo.
  - La siguiente figura muestra el grafo de 4 procesos y 2 tipos de recursos:



A resource allocation graph.

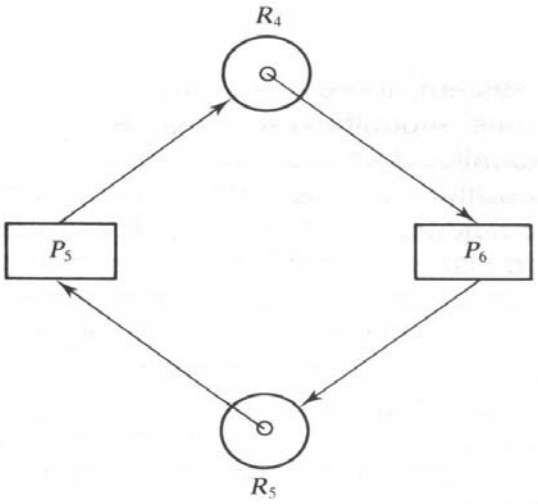
- Utilizando el grafo del sistema se pueden determinar los deadlock:
  - Se examinan los procesos que no están bloqueados y se eliminan del grafo liberando sus recursos. Se supone que si continúan procesándose ellos terminaran:



A partially reduced graph.

  - Se obtiene el grafo reducido del ejemplo anterior. Se puede observar que los procesos tendrán cubierta su necesidad de recursos ya que en cada tipo de recurso hay dos unidades.

- El siguiente grafo no puede ser reducido ya que se tiene una cola circular. La diferencia con el grafo anterior es que en este hay una sola unidad por cada tipo de recursos:

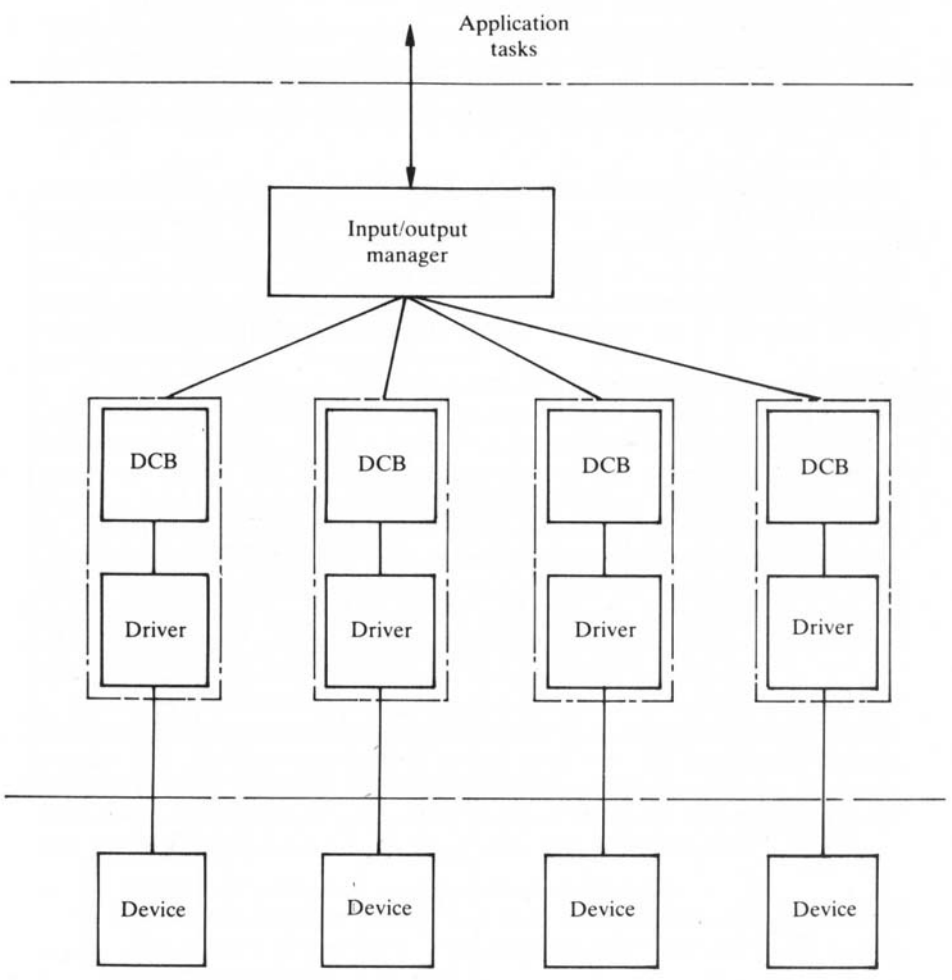


A resource allocation graph.

- Cada vez que se solicita un recurso el grafo se actualiza y analiza. Si por motivo de alguna petición se pudiera producir un deadlock entonces se rechazaría.
- Si se produce un deadlock y se intenta un reestablecimiento entonces se podría optar por alguna de las siguientes opciones:
  - Ruptura de la exclusión mutua. Tendría la ventaja de que es fácil de conseguir pero tendría el inconveniente de que el sistema se podría quedar en un estado inconsistente.
  - Abortar uno o más procesos. Sería la medida más drástica y dejaría el sistema en un estado inconsistente.
  - Apropiarse de los recursos de un proceso en deadlock. El inconveniente sería seleccionar que procesos van a perder los recursos, ya que habría de tenerse en cuenta las cuestiones de prioridad.

8.2. Subsistema de Entrada/salida [Bennett,94]

- Este subsistema (IOSS) del sistema operativo permite al programador realizar operaciones de entrada/salida mediante llamadas al sistema bien desde un lenguaje de alto nivel o incluso desde ensamblador.
- El subsistema de entrada/salida debe manejar los detalles de cada dispositivo y si además el sistema es multitarea debe sincronizar el acceso de varias tareas a un mismo dispositivo.
- La siguiente figura muestra un IOSS típico:

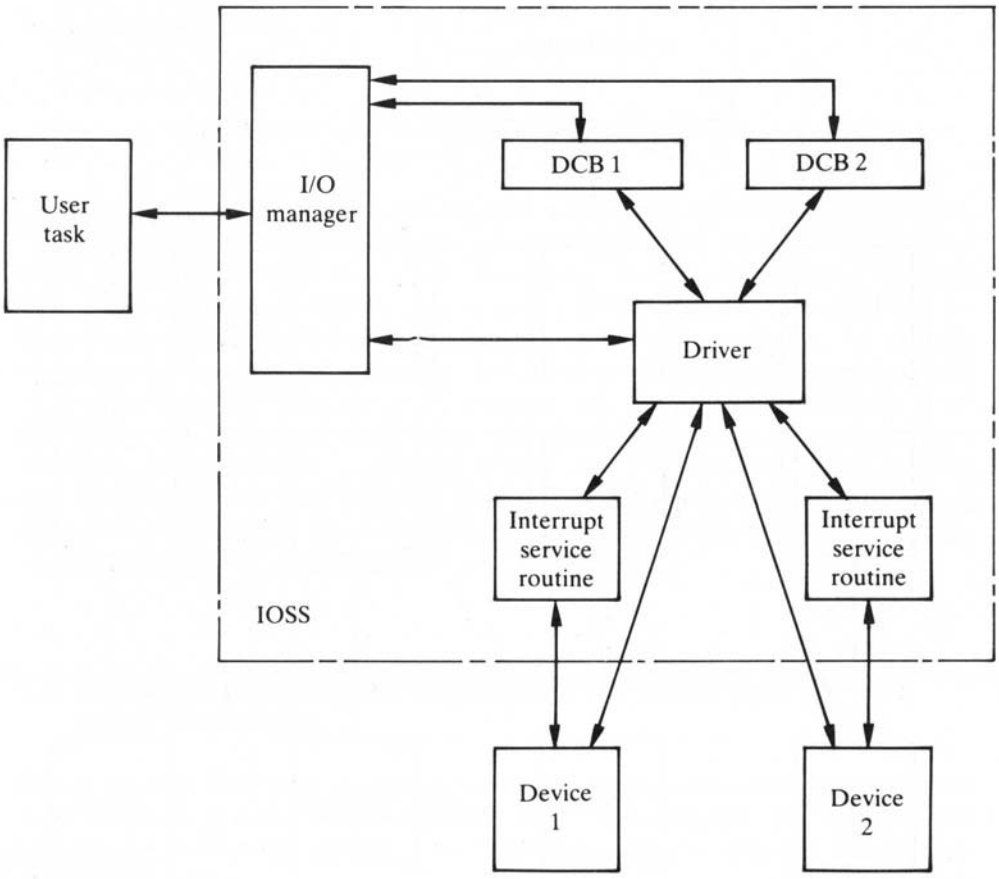


General structure of IOSS.

- El IOSS está dividido en dos niveles. El manejador de entrada/salida acepta las llamadas al sistema que realiza la tarea de usuario y transfiere la información que contiene la llamada al *bloque de control de dispositivo* (DCB) de un dispositivo concreto.
- La información que debe contener la llamada de la tarea de usuario podrá ser:
    - La situación del área del buffer donde se depositarán los datos que se transferirán, en el caso de salida o donde se irán depositando en el caso de entrada.
    - La cantidad de datos.
    - El tipo de datos: ASCII, binario, etc.
    - Dirección de la transferencia.
    - Dispositivo que se utilizará.

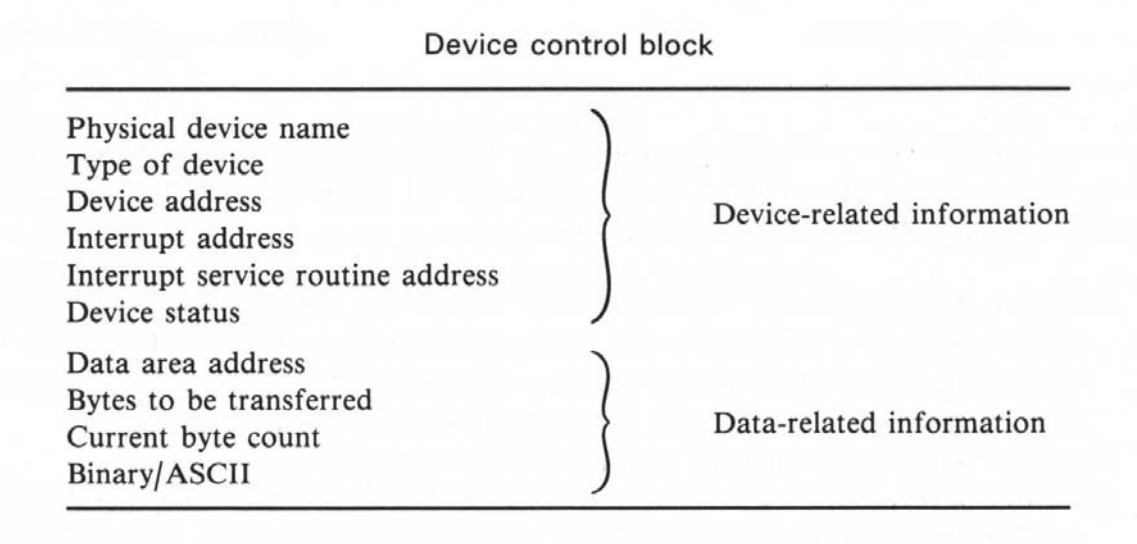
- La transferencia de datos entre la tarea de usuario y el dispositivo la llevará a cabo el *driver* del dispositivo utilizando la información depositada en la DCB. La transferencia de datos en general suele llevarse a cabo bajo control de interrupciones.

- Cada dispositivo debe tener su propia DCB, mientras que el *driver* (código) puede compartirse para varios dispositivos:



Detailed arrangement of IOSS.

- La siguiente tabla muestra la información que contiene una DCB típica:



  - El nombre del dispositivo físico es el nombre por el que el S. O. reconoce al dispositivo y el tipo de dispositivo se tiene habitualmente como un código reconocido por él. Los *drivers* de dispositivos más comunes suelen acompañar al S.O.
  - Los DCB pueden variar en lo que respecta a las direcciones de un sistema particular aunque muchos fabricantes suelen adoptar direcciones e interrupciones y rutinas de interrupción estándar. Cuando se quieran añadir dispositivos no estándar, el propio usuario debe crear su código.