Sistemas Operativos

Introducción, procesos, hilos, gestión de memoria y RTOS

Julian

TECNICAS DIGITALES III

SISTEMAS OPERATIVOS

Capa de software que proporciona un modelo de computador mejor y más simple y se encarga de la administración de los recursos.

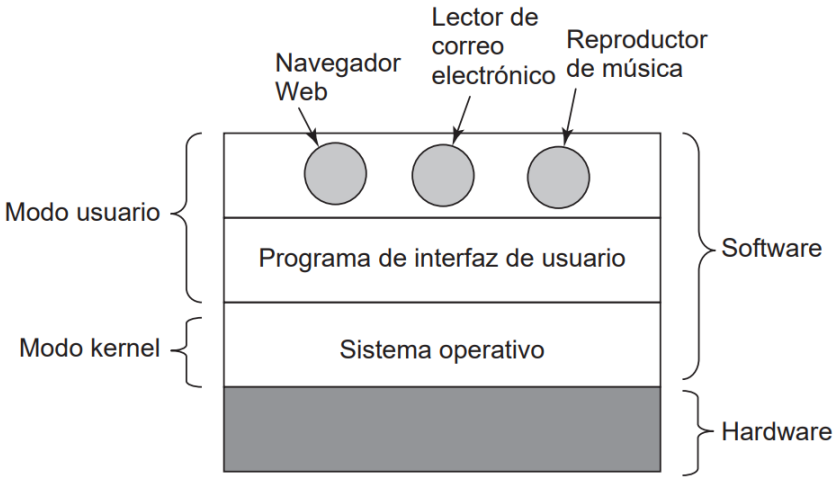


Figura 1-1. Ubicación del sistema operativo.

El programa de interfaz de usuario, Shell o GUI es el nivel más bajo del software en modo usuario y permite la ejecución de otros programas. El software que es parte del SO está protegido por el hardware contra cualquier intento de modificación por parte de los usuario.

El SO se ejecuta en modo kernel, por lo que tiene acceso completo a todo el hardware. El resto del software se ejecuta en modo usuario, donde el control de la máquina o de las I/O no están permitidas.

Los SO difieren de los programas de usuario en que son enormes, complejos y de larga duración. Realizan dos funciones básicas:

- Proporcionar a programadores de aplicaciones y a aplicaciones, un conjunto abstracto de recursos simples.
- Administrar recursos de hardware.

Existen entonces dos enfoque para analizar un SO:

- **SO como máquina extendida:** el trabajo del SO es crear abstracciones. Debe ocultar el hardware y presentar a los programadores abstracciones agradables y simples.
- **SO como administrador de recursos:** da una asignación ordenada y controlada de los recursos de hardware entre los diversos programas que compiten por ellos. El SO registra qué recurso está usando cada programa y otorga peticiones de recursos. Esta administración incluye el multiplexado de los mismos en espacio y tiempo.

No hay una definición completamente adecuada de SO, pero puede decirse que es aquel programa que brinda un entorno para que se ejecuten otros programas, que se generalmente se ejecuta permanentemente.

Los procesos están encargados de gestionar:

- Procesos: crear, terminar, suspender, reanudar, etc.
- Memoria: asignar, liberar.

- Almacenamiento: archivos y directorios, creación y eliminación de archivos, etc.
- Protección y seguridad: evitar sobreescrituras, monopolización de CPU, permisos de acceso, etc.

Conceptos y abstracciones de un SO

- **Procesos:** programa en ejecución con un espacio de direcciones asociado que contiene el programa ejecutable, los datos del programa y su pila.
- **Espacio de direcciones:** conjunto de direcciones que cada proceso puede utilizar.
- **Archivos:** abstracción para ocultar peculiaridades de los dispositivos I/O. La forma de agruparlos da surgimiento al sistema de archivos.
- **Entrada/Salida:** cada SO tiene un subsistema de E/S para administrar los dispositivos E/S. Parte del software de E/S es independiente de los dispositivos, se aplica a la mayoría por igual.
- **Protección:** para administrar la seguridad del sistema el código de protección consta de 3 campos de 3 bits cada uno, uno para el propietario, otro para el grupo del mismo y otros para los demás usuarios. Cada campo tiene un bit para lectura, escritura y ejecución (bits rwx).
- **Shell:** intérprete de comandos. No forma parte del SO pero utiliza muchas características del mismo. Es la interfaz entre usuario y SO.

Llamadas a sistema

Realizar una llamada a sistema es como realizar un tipo especial de llamada a procedimiento, solo que en este caso las llamadas entran al kernel. Las llamadas a sistema llevan a cabo una serie de pasos.

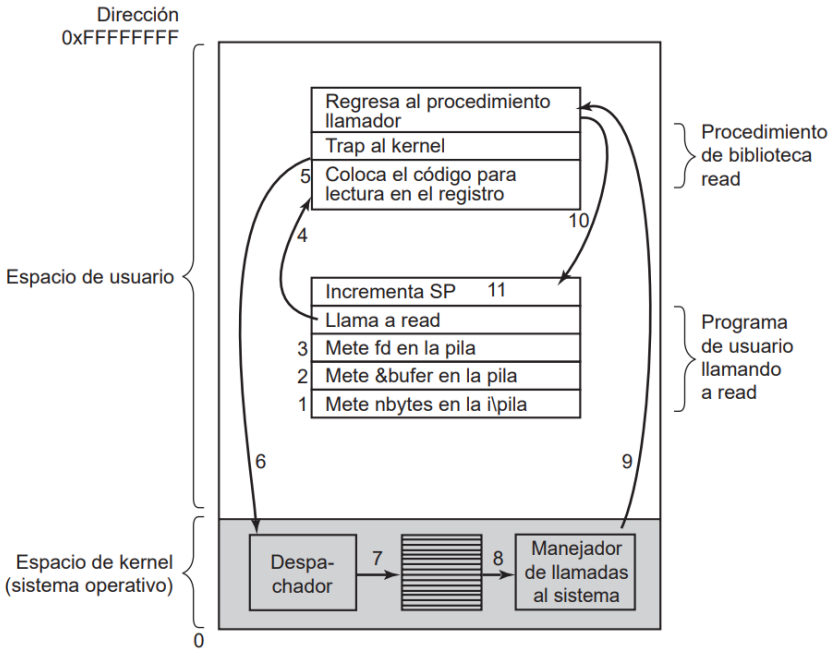


Figura 1-17. Los 11 pasos para realizar la llamada al sistema `read(fd, bufer, nbytes)`.

Por ejemplo, para la llamada $read$:

- El programa llamador primero mete los parámetros en la pila (1 a 3).
- Llamada al procedimiento de biblioteca (4).
- El procedimiento de biblioteca coloca por lo general el número de la llamada a sistema en un lugar en el que el SO lo espera, como en un registro (5).
- Ejecuta un TRAP para cambiar de modo usuario a kernel y empezar la ejecución en una dirección fija dentro del núcleo (6).

- El código de kernel examina el número de llamada a sistema y la pasa al manejador correspondiente de llamadas a sistema (7).
- Se ejecuta el manejador de llamadas a sistema (8).
- Cuando el manejador termina, el control puede regresar al procedimiento de biblioteca en espacio de usuario en la instrucción siguiente al TRAP (9).
- Luego este procedimiento regresa al programa de usuario (10).
- Para terminar, el programa de usuario limpia la pila, (11).

Estructuras de SO

- **Estructura monolítica:** todo el SO se ejecuta como un solo programa en modo kernel, todos los procedimientos son visibles entre sí. Esta organización sugiere una estructura básica para el SO:
  - Un programa principal que invoca el procedimiento de servicio solicitado.
  - Un conjunto de procedimientos de servicio que llevan a cabo las llamadas a sistema.
  - Un conjunto de procedimientos utilitarios que ayudan a los procedimientos de servicio.

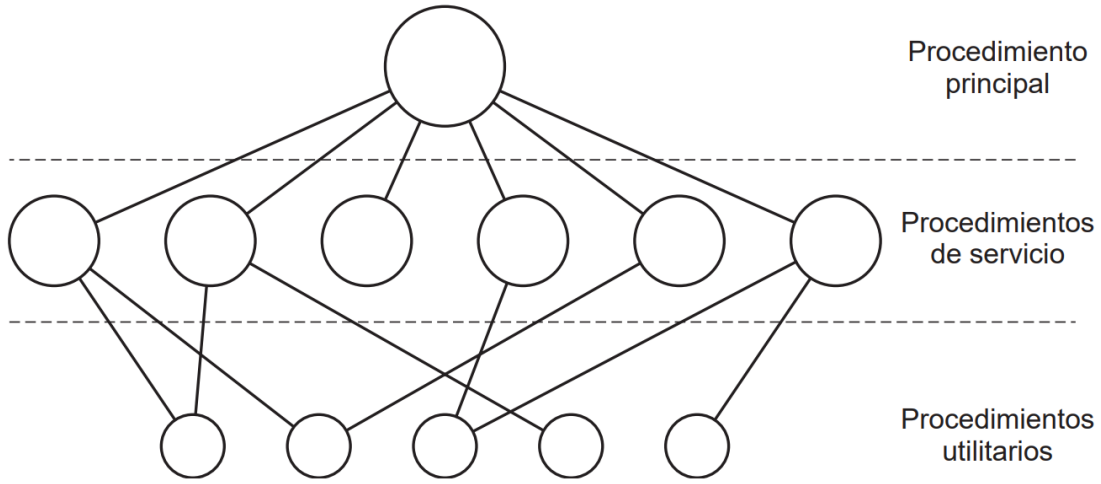


Figura 1-24. Un modelo de estructuración simple para un sistema monolítico.

- **Estructura de capas:** se organiza el SO en una jerarquía de capas.
- **Microkernel:** para lograr una alta confiablidad divide el SO en módulo pequeños bien definidos, solo uno de los cuales (el microkernel) se ejecuta en modo kernel y el resto en modo usuario.
- **Máquinas virtuales:** copias exactas de maquinas reales generadas por software. El kernel se denomina "monitor de la máquina virtual" y se ejecuta sobre el hardware.
- **Cliente – servidor:** ligera variación de la idea del microkernel diferenciando dos procesos: servidor, que proporciona cierto servicio y cliente, que utiliza estos servicios.

PROCESOS

Instancia de un programa en ejecución que tiene asociado los registros utilizados, el contador de programa, el puntero de pila, las variables y otros registros de hardware.

El modelo del proceso

Todo el software se organiza en procesos secuenciales. Cada uno tiene su propia CPU virtual (la real se conmuta).. La rápida conmutación entre procesos se denomina **multiprogramación**.

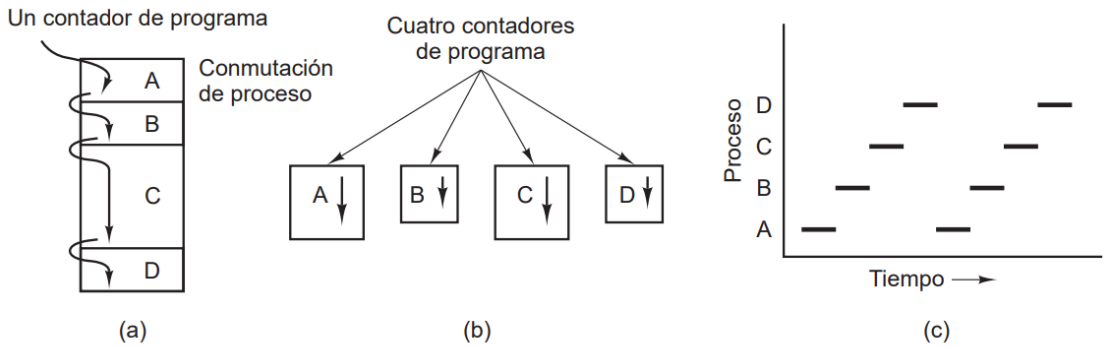


Figura 2-1. (a) Multiprogramación de cuatro programas. (b) Modelo conceptual de cuatro procesos secuenciales independientes. (c) Sólo hay un programa activo a la vez.

Solo hay un contador de programa físico, por lo que cuando se ejecuta cada proceso, se carga su contador de programa lógico en el contador de programa real. Cuanto termina, el contador de programa físico se guarda en el contador de programa lógico almacenado.

Puede entenderse al programa como una receta (algoritmo) y al proceso como la actividad que tiene un programa, una entrada, una salida y un estado. Dos o más procesos en ejecución con el mismo programa son procesos distintos.

Varios procesos pueden compartir un procesador mediante un algoritmo planificador.

Creación y finalización de procesos

Existen 4 eventos que provocan la **creación** de procesos:

1. Arranque del sistema.
2. Ejecución, desde un proceso, de una llamada a sistema para crear procesos.
3. Petición de usuario para crear otro proceso.
4. Inicio de un trabajo por lotes.

Siempre un proceso existente debe ejecutar una llamada a sistema de creación de procesos. Ésta indica al SO que cree un proceso y le indica cuál programa debe ejecutar. En UNIX esta llamada es `fork()`, la cual crea un clon exacto del proceso llamador.

Padre e hijo tienen la misma imagen de memoria y archivos abiertos. El hijo luego ejecuta una llamada `execve()` para cambiar su imagen de memoria y ejecutar un nuevo programa.

Un proceso **finaliza** por alguna de las siguientes razones:

1. Salida normal, voluntaria (termina su trabajo).
2. Salida por error, voluntaria (descubre error y finaliza).
3. Error fatal, involuntaria (el error provoca la finalización).
4. Eliminado por otro proceso, involuntaria (llamada a sistema para finalizar otro proceso, kill).

Memoria de un proceso

La memoria de un proceso se divide en segmentos de tamaño fijo y variable.

- Segmentos de tamaño fijo:
    - `text`: instrucciones.

    - `initialized data`.
    - `uninitialized data`.
- Segmentos de tamaño variable:
    - `stack` (pila): marco de datos por cada llamada a función. Crece con cada llamada y decrece al retornar.
    - `heap`: variables creadas en tiempo de ejecución.

`stack` y `heap` crecen en sentido opuesto, nunca deben solaparse.

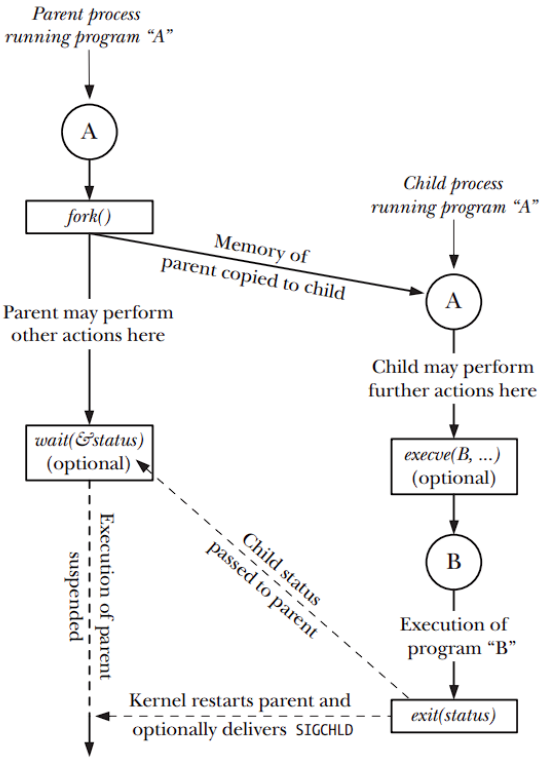


Figure 24-1: Overview of the use of *fork()*, *exit()*, *wait()*, and *execve()*

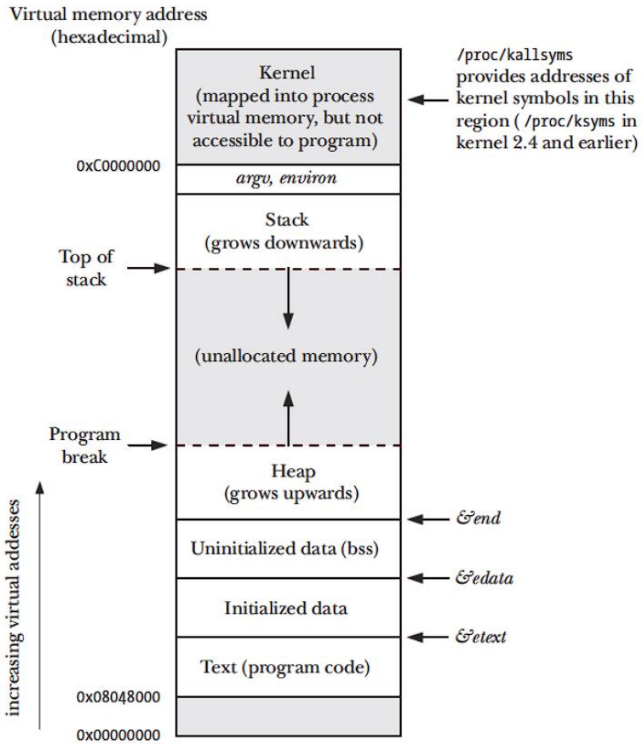


Figure 6-1: Typical memory layout of a process on Linux/x86-32

Jerarquía de procesos

En UNIX, un proceso y sus hijos forman un grupo de procesos. Todos los procesos del sistema pertenecen a un solo árbol con INIT en la raíz. Los procesos no pueden desheredar a sus hijos.

Los procesos se identifican con un ID de proceso, denominado PID. Al crear un proceso, este obtiene copias de la pila, datos, heap y segmento text de su padre. Padre e hijo se diferencian en el resultado de la llamada a `fork()`, al padre le devuelve el PID del hijo y al hijo 0.

Estados de un proceso

El 3 estados posibles son:

1. **En ejecución:** usando la CPU.
2. **Listo:** ejecutable, se detuvo para que otro se ejecute.
3. **Bloqueado:** no puede ejecutarse hasta que ocurra cierto evento externo.

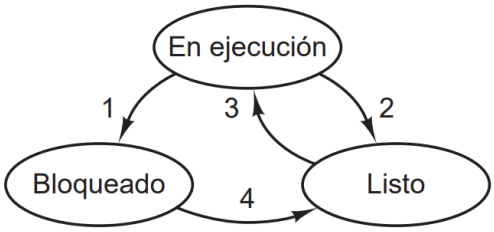


Figura 2-2. Un proceso puede encontrarse en estado "en ejecución", "bloqueado" o "listo". Las transiciones entre estos estados son como se muestran.

- **Transición 1:** el SO descubre que el proceso no puede continuar y se bloquea.
- **Transición 2:** el planificador decide que el proceso en ejecución debe dejar que otro se ejecute.
- **Transición 3:** el planificador decide que el proceso actual debe ejecutarse.
- **Transición 4:** ocurre cuando se produce el evento externo esperado por el proceso.

El nivel más bajo del SO es el planificador, se ocupa el manejo de interrupciones y procesos.

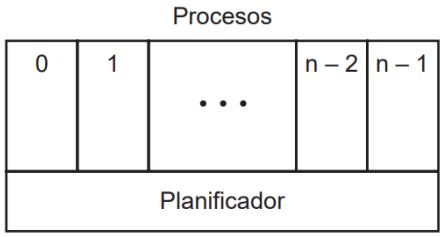


Figura 2-3. La capa más baja de un sistema operativo estructurado por procesos se encarga de las interrupciones y la planificación. Encima de esa capa están los procesos secuenciales.

Implementación de procesos

El SO mantiene una tabla de procesos con una entrada para cada uno, con información de su estado, contador de programa, puntero de pila, asignación de memoria, estado de archivos abiertos y todo lo que se necesite para que cuando el proceso cambie de estado se pueda reanudar.

Creación de un proceso

```
#include <sys/types.h>  // define typedef pid_t
#include <unistd.h>     // define fork

pid_t fork(void);
```

Devuelve al padre el PID del proceso hijo, al hijo un cero y en caso de error un -1.

Obtención del PID

```
#include <sys/types.h>  // define typedef pid_t
#include <unistd.h>

pid_t getpid(void);     // devuelve el pid del proceso que la ejecuta
```

```
#include <sys/types.h>  // define typedef pid_t
#include <unistd.h>

pid_t getppid(void);    // devuelve el pid del padre
```

Finalización de un proceso

```
#include <unistd.h>

void exit(int status);
```

Termina el proceso que la ejecuta, todos los recursos utilizados por este quedan disponibles. El argumento de la función es un entero que determina el estado de finalización del proceso. Mediante la llamada `wait()` el padre puede obtener este valor de su hijo.

Llamada a sistema wait

```
#include <sys/types.h>  // define typedef pid_t
#include <unistd.h>

pid_t wait(int *status);
```

Si el proceso hijo no ha terminado, lo espera. Una vez terminado, el padre puede obtener el estado de finalización en `status`. Retorna el PID del hijo terminado o -1 en caso de error.

Procesos huérfanos y zombies

Un proceso se convierte en **huérfano** cuando su proceso padre termina antes que él. Al huérfano se le asigna como nuevo padre un proceso superior o INIT (PID = 1).

Un proceso se convierte en **zombie** cuando termina antes que su proceso padre. Se borra de memoria el proceso pero no su PID ni su valor de retorno (`status`). Un zombie no puede ser matado por kill, cuando el padre hace un `wait`, el zombie es sacado completamente de memoria.

HILOS

Cada proceso tiene un espacio de direcciones y un hilo de control. Hay situaciones en las que conviene tener varios hilos de control que se ejecuten en (cuasi) paralelo, como procesos separados.

Un proceso puede tener múltiples hilos que se ejecutan de forma independiente en el mismo programa.

Cada hilo comparte con otros hilos del mismo proceso:

- ✓ Memoria global.
- ✓ Sección de código y datos.
- ✓ Pila de proceso.
- ✓ Archivos abiertos.
- ✓ PID de proceso y de proceso padre.
- ✓ Credenciales de proceso.
- ✓ IPC.

Algunos atributos que no comparten son:

- ⊘ ID de hilos (TID).
- ⊘ Datos específicos del hilo.
- ⊘ Pila local de hilo.

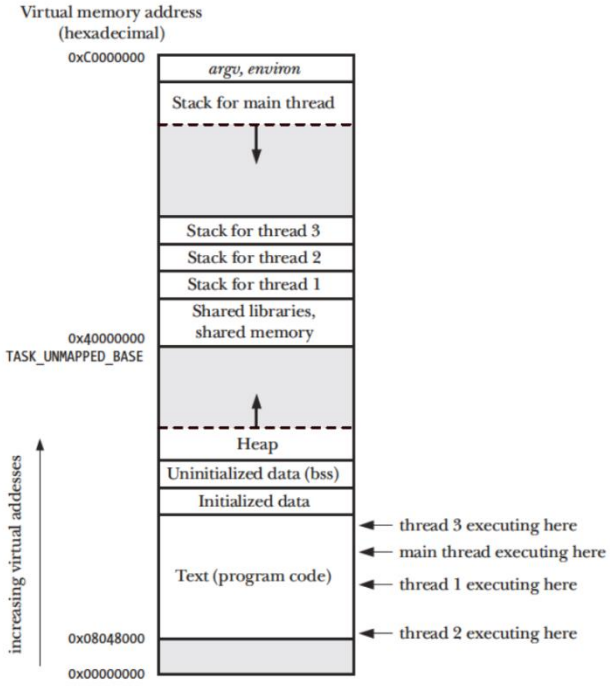


Figure 29-1: Four threads executing in a process (Linux/x86-32)

Para una ejecución independiente de un hilo debe almacenarse el ID del hilo, el estado, los registros (incluyendo contador de programa del hilo) y la pila del hilo.

Uso de los hilos

Un hilo es un flujo secuencial de instrucciones dentro de un proceso. Permiten a las aplicaciones realizar múltiples tareas concurrentemente. Agregan 3 ventajas:

- ✓ Habilidad de compartir espacio de direcciones y datos.
- ✓ Mayor velocidad, más simples de crear y destruir (10 a 100 veces).
- ✓ Agiliza la velocidad de las aplicaciones con gran cantidad de operaciones E/S.

Algunas desventajas son:

- ⊘ Un error en un hilo puede afectar la ejecución de los restantes hilos del proceso.
- ⊘ Cada hilo compite por el espacio de memoria con otros hilos.
- ⊘ Se requieren mecanismos de sincronización para un correcto funcionamiento.

Modelo clásico de hilo

Todo proceso tiene un hilo de ejecución, el cual contiene:

- Contador de programa (registro de la próxima instrucción a ejecutar).
- Registros con variables de programa actual.
- Pila con el historial de ejecución.

El mulithilamiento permite varios hilos en el mismo proceso, estos toman turnos para ejecutarse mientras la CPU conmuta entre ellos dando sensación de paralelismo.

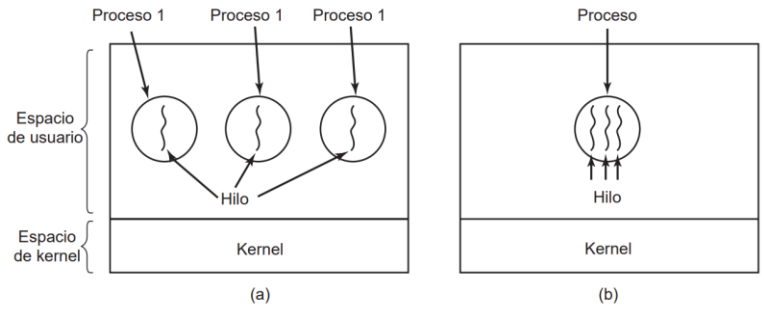


Figura 2-11. (a) Tres procesos, cada uno con un hilo. (b) Un proceso con tres hilos.

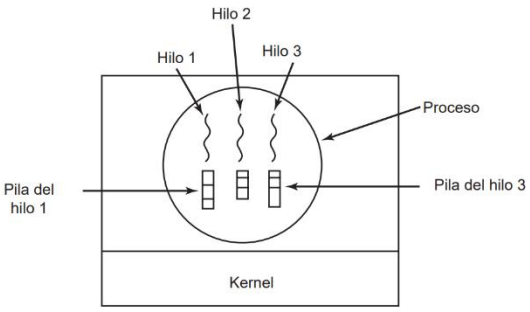


Figura 2-13. Cada hilo tiene su propia pila.

Un hilo puede leer, escribir o borrar la pial de otro del mismo proceso, no hay protección entre ellos.

El proceso es la unidad administradora de recursos. Varios hilos en ejecución da la habilidad de trabajar en conjunto para realizar tareas, ya que comparten un conjunto de recursos.

Los hilos pueden estar en los siguientes estados:

- **Ejecución**
- **Bloqueado**
- **Listo**
- **Terminado**

Cada hilo posee su pila con valores para cada procedimiento llamado del que todavía no se ha regresado. Este conjunto de valores contiene las variables locales y direcciones de retorno.

Hilos en posix

Creación y finalización de hilos

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start)(void*), void *arg);

// (puntero a variable hilo, atributos del hilo, función, argumentos función).
```

Devuelve 0 si tuvo éxito y un valor positivo en caso de error.

El nuevo hilo se inicia llamando a la función en el argumento `start`, con el argumento `arg`. Si el argumento `attr` es definido como `NULL` se toman los atributos por defecto.

La ejecución de un hilo puede terminar por alguno de los siguientes motivos:

- La función `start` retorna con un valor.
- Llamada a la función `pthread_exit()`.
- Hilo cancelado con `pthread_cancel()`.
- Cualquier hilo termina con la llamada `exit()`, lo cual finaliza todos los hilos del proceso.
- Llamada `return` desde el hilo main.

Función `pthread_exit()`.

```
#include <pthread.h>

void pthread_exit(void *retval);
```

El argumento `retval` indica el valor de retorno del hilo. Esta llamada solo finaliza al hilo que la realizó.

ID de hilos (TID)

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Devuelve el TID del hilo que realiza la llamada.

Unión de hilos

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

Espera a que el hilo indicado en el primer argumento, `thread`, termine. Devuelve 0 en caso de éxito y un valor positivo en caso de error. Si `retval` es un puntero no nulo, recibe el valor de retorno del hilo terminado.

A diferencia de `waitpid()`, como en los procesos no hay jerarquía cualquier hilo puede esperar a otro.

Atributos de los hilos

El argumento `attr` de `pthread_create()` es del tipo `pthread_attr_t` y especifica los atributos usados al crear el hilo. Esta estructura incluye:

- Localización y tamaño de la pila del hilo.
- Prioridad de los hilos.
- Si el hilo es o no unible (join).
    - `PTHREAD_CREATE_JOINABLE`.
    - `PTHREAD_CREATE_DETACHED`.

Liberación de hilos

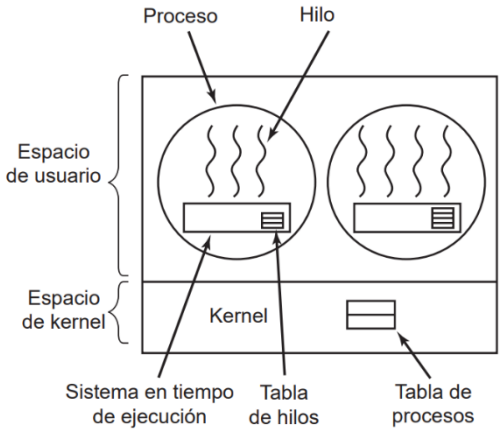
```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Por defecto, los hilos son unibles. Podemos liberar un hilo con la función `pthread_detach()`, así el sistema limpia y remueve el hilo automáticamente cuando termina. La función devuelve 0 en caso de éxito y un valor positivo en caso de error.

Implementación de hilos en el espacio de usuario

El kernel no es consciente de los hilos. La ventaja es que puede implementarse en un SO que no acepte hilos, mediante una biblioteca. Cada proceso necesita su propia tabla privada de hilo, similar a la tabla de procesos del kernel. Cuando un hilo pasa de listo a bloqueado, la información necesaria para reanudarlo se almacena en esta tabla.

Cuando un hilo realiza una acción que puede ponerlo en estado bloqueado llama a un procedimiento que comprueba si el hilo debe bloquearse, en cuyo caso almacena sus registros en la tabla de hilos. Luego busca en la misma tabla un hilo listo y carga los registros en él. La conmutación de hilos es más rápida y es la gran ventaja de los hilos de nivel de usuario.



Tanto el procedimiento que guarda el estado del hilo como el planificador son procedimientos locales, lo que lo hace rápido y eficiente.

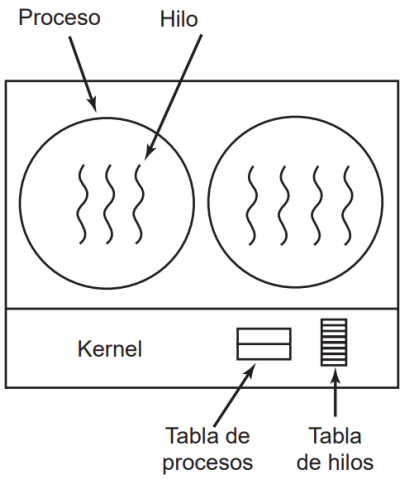
Además permite que cada proceso tenga su propio algoritmo de planificación

Las desventajas son:

- Las llamadas al sistema de bloqueo detienen todos los hilos del proceso en ejecución.
- Si un hilo produce un fallo de página, el kernel bloquea todo el proceso.
- Si un hilo empieza a ejecutarse, ningún otro hilo en el proceso se ejecutará a menos que el primero renuncie de manera voluntaria a la CPU. Como dentro de un proceso no hay interrupciones de reloj, no se puede planificar tomando turnos.

Implementación de hilos en kernel

El kernel sabe de los hilos y los administra, la tabla de hilos ahora se ubica en kernel. Si un hilo desea crear o destruir otro hilo realiza una llamada a kernel, el cual se encarga de ello mediante la modificación de la tabla. Si un hilo se bloque, el kernel puede ejecutar otro del mismo proceso o de un proceso distinto.



Si un hilo produce un fallo de página, el kernel puede comprobar si el proceso tiene otro hilo que pueda ejecutar, en cuyo caso es ejecutado.

La desventaja es que la llamada a sistema implica un costo considerable.

PLANIFICACIÓN

Cuando se multiprograma, varios procesos o hilos compiten por la CPU cada vez que más de uno se encuentra en estado listo. Si lo hay una CPU disponible debe decidirse quién la tomará, decisión tomada por el planificador, el cual forma parte de SO.

El planificador además debe hacer un uso eficiente de la CPU, ya que la conmutación de procesos es costosa. Primero debe hacer un cambio de modo usuario a kernel, después guardar el estado del proceso actual. Luego hay que seleccionar un nuevo proceso mediante la ejecución del algoritmo de planificación. Después volver a cargar en la MMU el mapa de memoria del nuevo proceso. Finalmente, se debe iniciar el nuevo proceso.

Generalmente, los proceso alternan ráfagas de cálculo con peticiones de E/S. Dos tipos comunes de comportamiento de procesos son los siguientes:

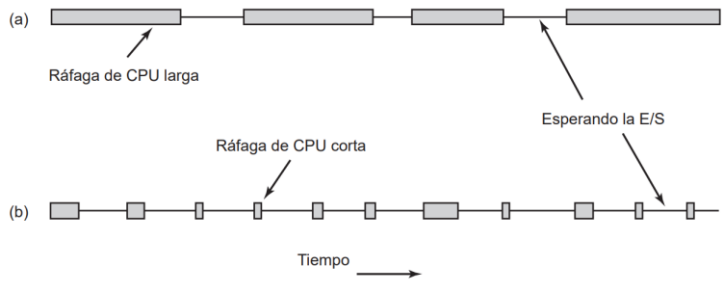


Figura 2-38. Las ráfagas de uso de la CPU se alternan con los periodos de espera por la E/S. (a) Un proceso ligado a la CPU. (b) Un proceso ligado a la E/S.

a. **Limitado a cálculos:** el proceso invierte la mayor parte del tiempo en realizar cálculo.
b. **Limitado a E/S:** el proceso pasa la mayor parte del tiempo esperando E/S.

Se deben tomar decisiones de planificación en 4 casos:

1. Creación de un nuevo proceso.
2. Finalización de un proceso.
3. Bloqueo de un proceso.
4. Interrupción de E/S.

Los algoritmos de planificación, respecto de la forma en que manejan las interrupciones de reloj, pueden dividirse en dos categorías:

- **No apropiativos:** seleccionan un proceso y dejan que se ejecute hasta que se bloquee solo. No toman decisiones de planificación durante las interrupciones de reloj.
- **Apropiativos:** selecciona un proceso y deja que se ejecute por un tiempo fijo. Requiere de una interrupción de reloj en cada intervalo para que el control de la CPU regrese al planificador.

Además, dependiendo de qué se desee optimizar, pueden dividirse en 3 categorías más:

1. **Procesamiento por lotes:** sin usuarios en espera de respuestas rápidas a peticiones cortas. Acepta algoritmos no apropiativos o apropiativos con largos periodos, reduce la conmutación de procesos.
2. **Interactivo:** usuarios interactivos, la apropiación es esencial. Son de propósito general y pueden ejecutar programas arbitrarios no cooperativos.
3. **De tiempo real:** restricciones de tiempo real, realizan su trabajo y se bloquean con rapidez.

Las metas de los algoritmos dependen del entorno, pero algunas se desean en todos los casos.

- En todos los sistemas se desea **equidad, aplicación de políticas y balance.**
- En procesamiento por lotes **rendimiento, tiempo de retorno mínimo y utilización de CPU elevada.**
- En sistemas interactivos **tiempos de respuesta cortos y proporcionalidad.**
- En sistemas de tiempo real **cumplimiento de plazo y predictibilidad.**

Planificación de sistemas de procesamiento por lotes

PRIMERO EN LLEGAR PRIMERO EN SER ATENDIDO – FCFS

No apropiativo. La CPU asigna a los procesos en el orden que solicitan, no se interrumpen. Mientras ingresan se colocan al final de la cola. Si un proceso en ejecución se bloquea, el primer proceso listo en la cola se ejecuta. Si luego pasa a listo se coloca al final de la cola.

Este sistema presenta una desventaja en procesos limitados a E/S.

TRABAJO MÁS CORTO PRIMERO – SJF

Es no apropiativo, supone que los tiempos de ejecución se conocen de antemano. El planificador selecciona el trabajo más corto primero.

EL MENOR TIEMPO A CONTINUACION – SRTN

Versión apropiativa del anterior. El planificador selecciona el proceso cuyo tiempo restante de ejecución sea más corto. Al llegar un trabajo se compara su tiempo con el tiempo restante del proceso en ejecución.

Planificación de sistemas interactivos

PLANIFICACION POR TURNO CIRCULAR – ROUND ROBIN

A cada proceso se le asigna un tiempo (quantum). Si la ejecución del proceso sigue al final del quantum la CPU es apropiada por el planificador para dársela a otro proceso. Si termina antes, se conmuta la CPU. Requiere una lista de procesos ejecutables.

La conmutación requiere un tiempo aproximado de $1ms$ por lo que un quantum demasiado corto produce demasiadas conmutaciones reduciendo la eficiencia de la CPU. Por otro lado, un quantum muy elevado produce una mala respuesta a peticiones interactivas cortas. Un valor razonable se encuentra entre 20 y $50ms$.

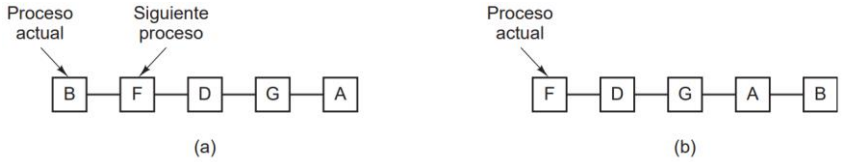


Figura 2-41. Planificación de turno circular. (a) La lista de procesos ejecutables. (b) La lista de procesos ejecutables una vez que *B* utiliza su quántum.

PLANIFICACIÓN POR PRIORIDAD

A cada proceso se le asigna una prioridad y se ejecuta el proceso ejecutable con la prioridad más alta. Para evitar una ejecución indefinida el planificador puede reducir la prioridad del proceso en ejecución en cada pulso de reloj.

Es conveniente agrupar los procesos en clases por prioridad y usar round robin entre procesos de la misma clase (misma prioridad).

PROCESO MÁS CORTO A CONTINUACIÓN – SPN

Los procesos interactivos suelen seguir el patrón "en espera de comandos – ejecución". Si se considera a cada comando como un trabajo separado, podemos minimizar el tiempo de respuesta total mediante la ejecución del más corto primero. El problema es averiguar cuál de los procesos ejecutables es el más corto. Un método es realizar estimaciones tomando comportamientos anteriores, técnica conocida como envejecimiento.

PLANIFICACIÓN GARANTIZADA

Un método es dar a n usuarios conectados $1/n$ del poder de CPU, o en un sistema con 1 usuario y n procesos, $1/n$ ciclos de CPU a cada proceso.

El sistema debe llevar la cuenta de cuanta potencia de CPU ha tenido cada proceso desde su creación. Se calcula la proporción de tiempo de CPU que consumió y se la compara con la que debería tener cada proceso, luego se ejecuta el proceso con la menor proporción.

PLANIFICACIÓN POR SORTEO

Cada proceso tiene un boleto para diversos recursos. Se selecciona al azar uno correspondiente al ganador que ocupará el recurso. Procesos cooperativos pueden intercambiar boletos para modificar sus probabilidades.

PLANIFICACIÓN POR PARTES EQUITATIVAS

Se le asigna a cada usuario cierta fracción de CPU según la cantidad de procesos que ejecuten.

Planificación de sistemas en tiempo real

Se dividen en:

- **Tiempo real duro:** los límites temporales son absolutos.
- **Tiempo real suave:** se permiten dispersiones de tiempo.

El comportamiento en tiempo real se logra dividiendo el programa en varios procesos. Los eventos en estos sistemas pueden ser periódicos o aperiódicos. Un sistema es planificable siempre y cuando los eventos periódicos en él cumplan la siguiente ecuación:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Donde m es la cantidad de eventos periódicos, C_i el tiempo que toma el evento i y P_i el periodo de dicho evento. Este calculo supone que la sobrecarga por conmutación es despreciable.

Los algoritmos de planificación en tiempo real pueden ser:

- **Estáticos:** toman decisiones de planificación antes de que el sistema empiece a ejecutarse. Funcionan solo si hay información perfecta de antemano.
- **Dinámicos:** toman sus decisiones durante la ejecución.

Planificación de hilos

Cuando varios procesos tiene múltiples hilos se tienen dos niveles de paralelismo: de procesos y de hilos. La planificación depende de si los hilos son a nivel usuario o kernel.

Algoritmos round robin y por prioridad son los más comunes. La única restricción es la ausencia de un reloj para interrumpir un proceso en hilos de nivel de usuario.

Para la conmutación de hilos nivel usuario se requiere de muchas instrucciones de máquina mientras que hilos nivel kernel requieren un cambio de contexto total, lo cual es más lento. En estos, cuando un hilo se bloquea no se bloquea todo el proceso, cosa que si sucede en hilo nivel usuario. Sin embargo, los hilos nivel usuario pueden emplear un planificador de hilos especifico para la aplicación, que por lo general puede optimizar mejor una aplicación que el kernel.

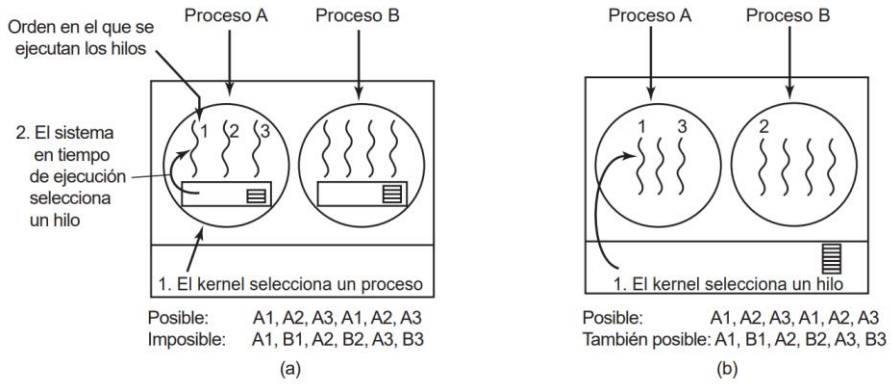


Figura 2-43. (a) Posible planificación de hilos a nivel usuario con un quántum de 50 mseg para cada proceso e hilos que se ejecutan durante 5 mseg por cada ráfaga de la CPU. (b) Posible planificación de hilos a nivel kernel con las mismas características que (a).

COMUNICACIÓN ENTRE PROCESOS – IPC

IPC es una función básica de los SO, permite enviar información entre procesos. Los 3 puntos a solucionar por IPC son:

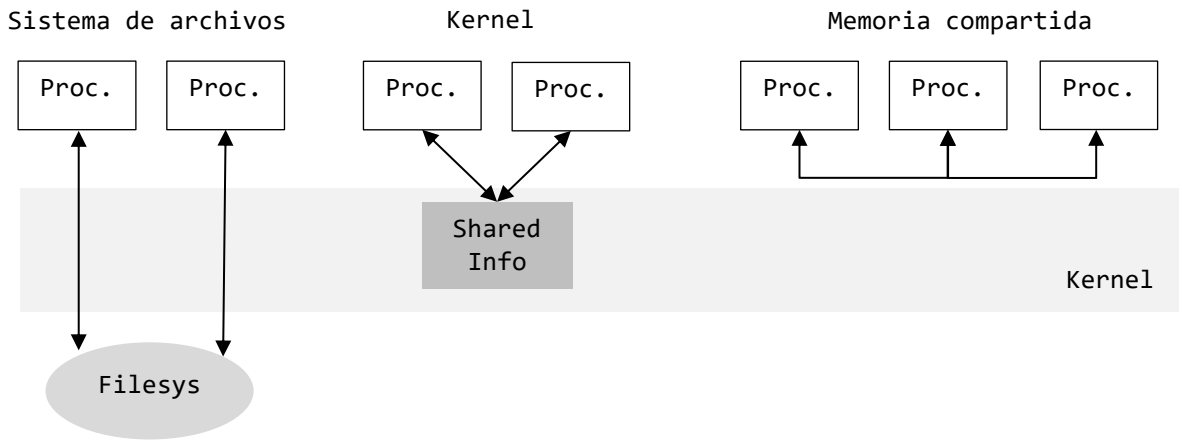
- Pasar información entre procesos, preferentemente de forma estructurada y sin interrupciones.
- Asegurar que dos procesos no se interfieran en tareas críticas.
- Secuencias correctamente ante existencia de dependencias.

Clasificaciones de IPC

- **Paso de mensajes:** llamadas a kernel para comunicar pequeñas cantidades de información. Sencillo de implementar. Se divide en 6 categorías:
  - **PIPE – FIFO – Queue – RPC – MPI – Sockets**
- **Memoria compartida:** comparte un espacio de memoria, es más rápida, requiere sincronizar.

SEGÚN LOS CAMINOS PARA COMPARTIR INFORMACIÓN

1- **Sistema de archivos:** dos procesos comparten información que reside en un archivo. Cada proceso debe pasar por kernel (`read`, `write`). Requiere sincronización si hay varios escritores. Es el caso de archivos y sockets.
2- **Kernel:** la información reside en el núcleo (`PIPE`, `FIFO` Y `MSG QUEUE`). Necesitan una llamada a sistema en el kernel.
3- **Memoria:** región de memoria compartida, no hay participación del kernel. Se requiere sincronización (`memorias compartidas`).



SEGÚN SU PERSISTENCIA

Tiempo que permanece en existencia una comunicación IPC. Se distinguen 3 tipos:

- **Persistencia de proceso:** persiste hasta que el último proceso con el objeto abierto se cierra.
  - **PIPE, FIFO, Socket**
- **Persistencia de kernel:** persiste hasta que el kernel se reinicia.
  - **Queue, semáforos**
- **Persistencia del sistema de archivos:** persiste hasta que se elimina el archivo (persisten ante reinicios del kernel).
  - **Semáforos**

Si bien la información de un PIPE está dentro del kernel, el PIPE tiene persistencia de proceso.

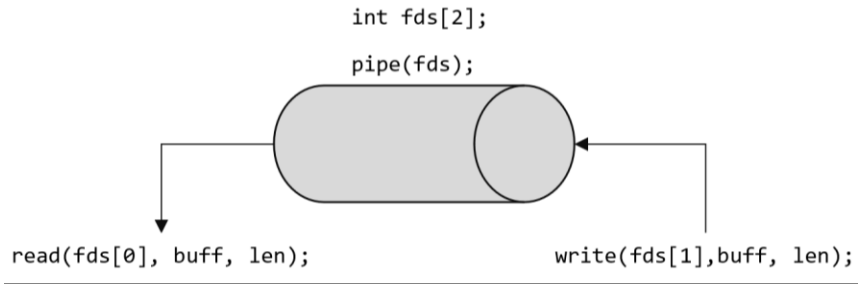
SEGÚN SU USO DE LOS OBJETOS DE IPC

- **Comunicación entre procesos en una misma máquina:** PIPE, FIFO, Queue.
- **Comunicación en una misma máquina o en una misma red:** Socket.

PIPE: tubería anónima

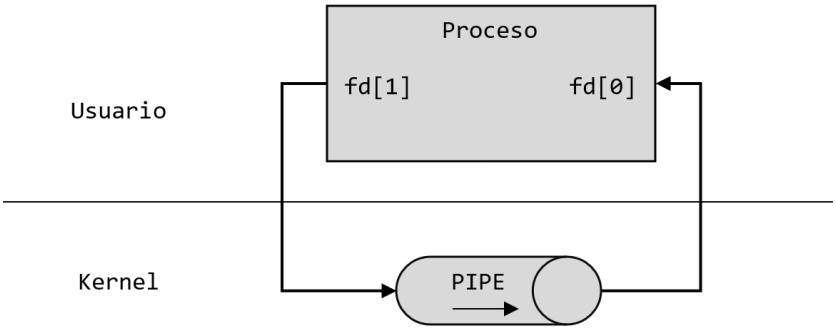
Permite comunicación entre procesos relacionados por herencia. Utiliza mensajes stream, pueden abrirse como bloqueantes o no bloqueantes. Une la salida estándar de un proceso a la entrada estándar de otro. Utiliza los denominados "descriptores de archivos":

0- entrada estándar (stdin).
1- salida estándar (stdout).
2- salida de error (stderr).

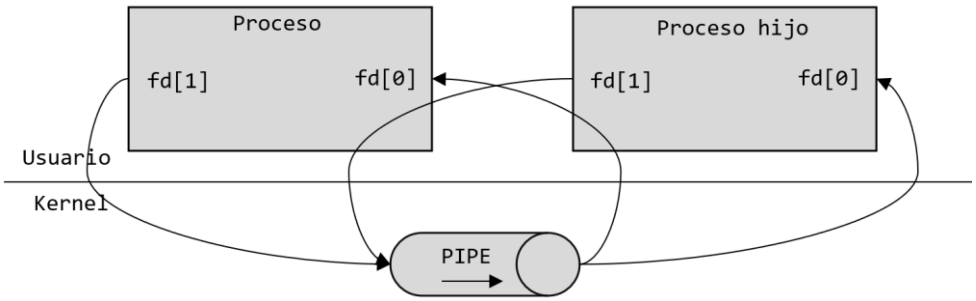


La creación de una tuberías sigue los siguientes pasos:

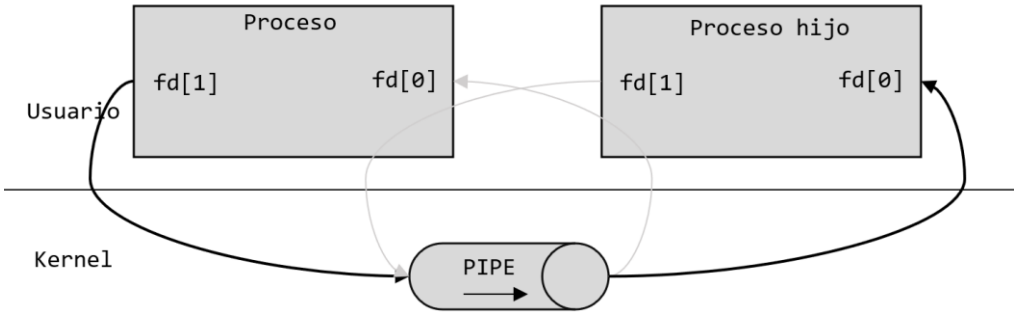
1. Padre crea `PIPE`.



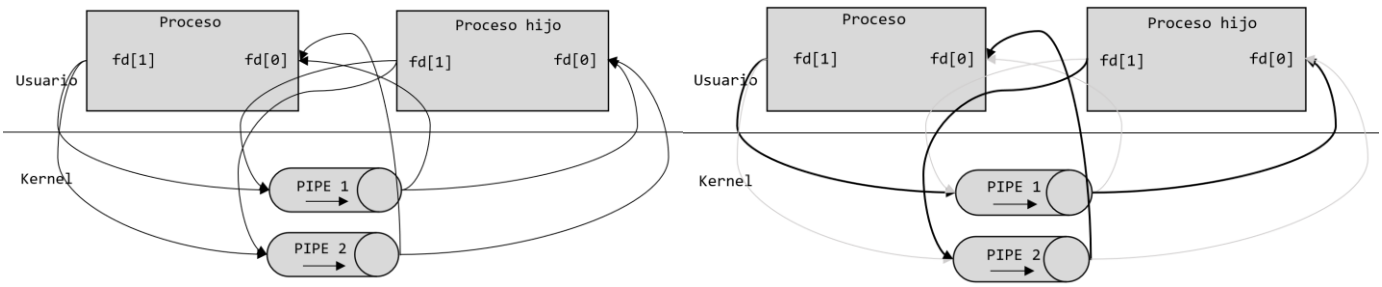
2. Llama a `fork()`.



3. Padre cierra extremo de lectura, hijo cierra escritura (o viceversa).



Para comunicación bidireccional deben crearse dos pipes y cerrar descriptores de la siguiente forma:



PIPES EN POSIX

Creación de pipes

```
#include <unistd.h>

int fd[2];
int pipe(int fd);
```

Devuelve -1 en caso de error y 0 si tuvo éxito. El argumento es un arreglo de dos enteros, en caso de éxito la tabla de descriptores contendrá dos nuevos descriptores de archivos:

- `fd[0]`: lectura
- `fd[1]`: escritura

Cierre de pipes

```
#include <unistd.h>

int close(int fd[x]);
```

Cierra un extremo de la tubería. Si se cierran ambos la tubería es eliminada. Devuelve 0 en caso de éxito y -1 en caso de error.

Escritura de pipes

```
#include <unistd.h>

int write(int fd[1], void *buffer, size_t count);
```

Devuelve el número de bytes escritos o -1 en caso de error. Se bloquea hasta que los datos hayan sido escritos (escritura atómica). Si un proceso intenta escribir en un descriptor cerrado el kernel envía la señal `SIGPIPE`.

Lectura de pipes

```
#include <unistd.h>

int read(int fd[0], void *buffer, size_t count);
```

Devuelve el número de bytes leídos y el contenido se almacena en `buffer`.

Situaciones conflictivas

1. Proceso que lee un pipe vacío, o escribe en uno lleno, se bloquea.
2. Si dos procesos leen un pipe no se puede determinar cuál leyó primero.
3. Proceso que escribe en una tubería sin descriptores de lectura abiertos para otros procesos recibe señal SIGPIPE, que por defecto mata al proceso.

FIFO: tubería con nombre

Permite comunicación entre procesos no relacionados. Es unidireccional, tiene una ruta de acceso al sistema de archivos, utiliza mensajes stream y puede abrirse como bloqueante o no bloqueante.

Se crea mediante mkfifo() y puede abrirse para lectura o escritura. Utiliza las mismas llamada `read` y `write` para lectura y escritura respectivamente.

FIFOS EN POSIX

Creación de fifo

```
#include <sys/stat.h>
#include <sys/types.h>

int mkfifo(const char *pathname, mode_t mode);
```

Devuelve 0 en caso de éxito y -1 en caso de error. El argumento `pathname` indica el nombre o ruta al fifo. El argumento `mode` indica los permisos (lectura, escritura, ejecución), se colocan en octal.

```
$ mkfifo [ -m mode ] pathname
```

Crea un fifo desde el shell. Al listar con `ls -l`, el fifo se muestra como tipo "`p`" en la primer columna.

Apertura de fifo

```
#include <sys/stat.h>
#include <fcntl.h>

int open(nombre, int flags, … /*mode_t mode/*);
```

Abre un archivo existente, o crea y abre uno nuevo. Devuelve el descriptor del archivo o -1 en caso de error. El campo mode indica los permisos al crear el archivo, si ya está creado se coloca en 0.

El campo flags indica el modo de apertura, puede ser:

- `O_RDONLY – O_WRONLY – O_RDWR – O_NONBLOCK –`

Estas constantes son máscaras de bits definidas en la biblioteca `fcntl.h`.

Un fifo puede ser abierto por cualquier proceso que tenga los permisos. La apertura para lectura bloquea al proceso hasta que otro lo abra para escritura y viceversa. Esto significa que los fifo sincronizan los procesos de lectura y escritura.

Para utilizarlo como no bloqueante debe usarse `O_NONBLOCK` o `O_RDWR` (no recomendado).

Eliminación de fifo

```
#include <unistd.h>

int unlink(const char *pathname);
```

Devuelve 0 en caso de éxito y -1 en caso de error.

PROBLEMA DEL STREAM DE DATOS

Al trabajar con stream de datos, el receptor no discrimina quién envió un mensaje, ni entre más de 1 mensaje. Algunas soluciones son:

- Utilizar una secuencia especial in – band.
- Utilizar un largo explícito y/o fijo.
- Comunicar solo 1 mensaje por conexión. Solo Applicable a sockets.

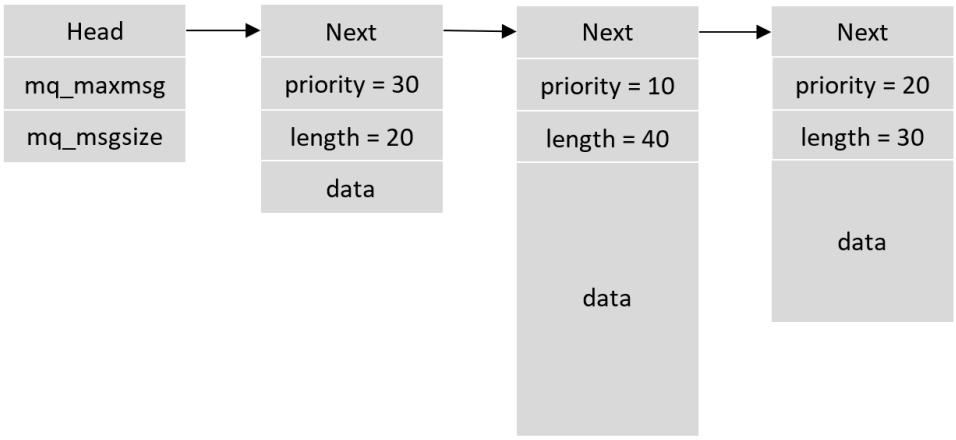
Colas de mensajes

Lista enlazada de mensajes segmentados, cada uno es un registro con una prioridad. Pueden abrirse como bloqueantes o no bloqueantes. Sirve para procesos no relacionados.

Al escribir mensajes se van almacenando en la cola y al leerlos se van eliminando. Se ordenan por prioridad y dentro de una misma prioridad por antigüedad.

Los mensajes poseen la siguiente estructura:

- Prioridad
- Longitud de la parte de datos
- Datos



Poseen persistencia de kernel y sufren el inconveniente de no poder identificar el emisor de cada mensaje en el receptor.

COLAS DE MENSAJES EN POSIX

Para compilar un programa que utilice colas de mensajes debe añadirse la librería `lrt`:

- `gcc -o mq01 mq01.c -lrt`

Apertura de cola de mensajes

```
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag, … mode_t mode, struct mq_attr *attr);

// mq_des = mq_open(nombre, banderas, permisos, atributos);
```

Devuelve un descriptor de cola de mensajes o -1 en caso de error. Requiere como mínimo dos argumentos, `name` y `oflag`. Este último es una máscara de bits que controla la operación de mq_open:

`O_CREAT – O_RDONLY – O_WRONLY – O_RDWR – O_NONBLOCK`

El argumento `mode` especifica los permisos al crear una nueva cola de mensajes, solo se tiene en cuenta si el `oflag` es `O_CREAT`.

El argumento `attr` es un puntero a estructura `mq_attr` que especifica los atributos de la cola de mensajes. Si `attr` es `NULL` utiliza los atributos por defecto.

Atributos de colas de mensajes

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
```

La función devuelve 0 en caso de éxito y -1 en caso de error. Copia una estructura con información de la descripción de la cola `mqdes` a la estructura `attr`. Los parámetros de esta estructura son:

- `attr.mq_curmsgs`: cantidad actual de mensajes en la cola.
- `attr.mq_maxmsg`: número máximo posible de mensajes.
- `attr.mq_msgsize`: tamaño máximo de los mensajes.
- `attr.mq_flags`: banderas para la descripción de la cola.

Envío de mensajes

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, uint msg_prio);
```

Agrega el mensaje apuntado por `msg_ptr` a la cola `mqdes`. El argumento `msg_len` indica la longitud del mensaje y `msg_prio` la prioridad del mismo, siendo `0` la menor. En caso de éxito retorna 0, sino -1.

Recepción de mensajes

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, uint *msg_prio);
```

Devuelve el número de bytes leídos o -1 en caso de error. Elimina el mensaje más antiguo y con mayor prioridad de `mqdes` y lo envía al buffer apuntado por `msg_ptr`. El argumento `msg_len` indica el número de bytes de espacio disponible, debe ser mayor o igual a `mq_msgsize`. La función devuelve el número de bytes leídos y -1 en caso de error.

Cierre de cola de mensajes

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);
```

Devuelve 0 en caso de éxito o -1 en caso de error. Cierra la cola cuyo descriptor se pasa como argumento. Este cierre no la elimina.

Eliminación de una cola de mensajes

```
#include <mqueue.h>

int mq_unlink(const char *name);
```

Marca la cola para ser destruida cuando todos los procesos dejen de utilizarla. Devuelve 0 en caso de éxito o -1 en caso de error.

SINCRONIZACIÓN

En ocasiones debe respetarse el orden de ejecución de los eventos para un funcionamiento correcto. Esta relación entre dos o más eventos requiere de sincronización.

Modelos de ejecución

- **Sistemas monoprocesador monotarea:** no hay problemas de sincronización, nunca se tiene dos eventos en simultáneo.
- **Sistemas monoprocesador multitarea:** permiten múltiples eventos, el planificador debe decidir cuál sigue. Puede haber problemas de sincronización por el pseduoparalelismo.
- **Sistemas multiprocesador multitarea:** no se conoce el orden de la sentencia a ejecutar, también existen problemas de sincronización.

Relación entre eventos

- **Concurrencia:** dos eventos son concurrentes cuando pueden ejecutarse en simultáneo.
    - Viendo el código no puede saberse cual se ejecuta primero (no determinístico).
    - Si no modifican recursos compartidos, el resultado final es independiente de orden de ejecución.
    - Si se modifican recursos compartidos, deben imponerse restricciones de software para independizar el resultado del orden de ejecución.
- **Serialización – secuenciación:** ante múltiples eventos, un evento debe ocurrir antes que otro. Esto suele deberse a que comparten recursos. Deben imponerse restricciones de software.
- **Punto de encuentro:** dos eventos deben esperarse mutuamente. Suele requerirse cuando ambos comparten y utilizan recursos del otro, requiere restricciones por software.
- **Exclusión mutua:** dos eventos no deben producirse al mismo tiempo, modifican recursos compartidos, siendo el resultado dependiente del orden de ejecución. Requiere restricciones.
- **Barrera:** punto de encuentro para el caso de más de dos eventos.
- **Condición de competencia o carrera:** dos o más procesos acceden a un recurso compartido sin control, de modo que el resultado depende del orden de llegada. La zona donde se modifican los recursos compartidos se denomina **sección crítica**.

Semáforos

Variables que pueden considerarse un entero, cuentan con las siguientes características:

- Pueden inicializarse con cualquier valor positivo.
- Para incrementarse y decrementarse utilizan primitivas atómicas (TSL).
- Si al decrementar el valor del semáforo resulta negativo, el hilo se bloquea (no lo decrementa).
- SI se incrementa el valor de un semáforo negativo, el hilo bloqueado se despierta.
- Cuando un hilo incrementa un semáforo y despierta a otro hilo, ambos continúan ejecutándose.
- Nunca puede leerse el valor actual del semáforo.
- Da acceso a recursos compartidos a un solo proceso o hilo a la vez.

El valor del semáforo es mantenido en kernel y se utiliza en los siguientes casos:

- Cuando 2 o más procesos quieren acceder a un recurso compartido.
- Para sincronizar accesos a memoria o recursos compartidos.

SEMÁFOROS EN POSIX

Inicializar semáforo

```
#include <semaphore.h>

sem_t sem;
int sem_init(&sem, int pshared, int value);
```

El argumento `pshared` indica si es para ser compartido entre hilos (0) o procesos (1). El argumento `sem` es la variable semáforo, debe declararse en una región de memoria compartida. Finalmente `value` indica el valor inicial del semáforo.

La función devuelve 0 en caso de éxito, -1 en caso de error.

Decrementar semáforo

```
#include <semaphore.h>

sem_t sem;
int sem_wait(&sem);
```

Decrementa en 1 el valor de semáforo `sem`. Si el valor del semáforo es mayor a 0, luego de `sem_wait()` vuelve inmediatamente. En caso contrario se bloquea hasta que otro evento lo incremente.

La función retorna 0 en caso de éxito o -1 en caso de error.

```
#include <semaphore.h>

sem_t sem;
int sem_trywait(&sem);
```

Esta función es una versión no bloqueante de la anterior. En lugar de bloquearse falla con EAGAIN.

Incrementar semáforo

```
#include <semaphore.h>

sem_t sem;
int sem_post(&sem);
```

Si el valor antes de la llamada era 0 y otro evento está bloqueado, esta llamada lo desbloqueará y hará que su respectivo `sem_wait()` lo decremente.

Recuperación del valor actual de un semáforo

```
#include <semaphore.h>

sem_t sem;
int sem_getvalue(&sem, int *sval);
```

Almacena el valor actual del semáforo en la variable pasada como argumento `sval`. Retorna 0 en caso de éxito o -1 en caso de error.

Destrucción de un semáforo

```
#include <semaphore.h>

sem_t sem;
int sem_destroy(&sem);
```

Es segura solo si no hay eventos bloqueados en espera. Retorna 0 en caso de éxito o -1 en caso de error.

Mutex – Semáforo binario

Similar a un semáforo pero puede tomar solo dos valores, 0 y 1. Se utiliza para sincronizar variables compartidas, asegurando que solo un hilo a la vez tengo acceso a ellas. Sus funciones también son atómicas.

MUTEX EN POSIX

Inicialización de un mutex estático

```
#include <pthread.h>

pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

Al inicializarlo de forma estática solo puede tomar atributos por defecto.

Inicialización de un mutex dinámico

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mtx, const pthread_mutexattr_t *attr);
```

El argumento `mtx` es un puntero al mutex a inicializar y `attr` son los atributos asignados al mismo. Si este argumento es `NULL` se toman los atributos por defecto.

Atributos de mutex

```
#include <pthread.h>

int pthread_mutexattr_t mtxattr;

int pthread_mutexattr_init(pthread_mutexattr_t *mtxattr);
```

Seteo de atributos

```
#include <pthread.h>

int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

El argumento `type` puede ser:

- `PTHREAD_MUTEX_ERRORCHECK`.
- `PTHREAD_MUTEX_DEFAULT`.
- `PTHREAD_MUTEX_NORMAL`.

Decrementar mutex

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mtx);
```

Resta 1 al valor de mutex, si previamente se encuentra en 0 se bloquea hasta ser incrementado por otro evento. La función retorna 0 en caso de éxito y -1 en caso de error.

```
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mtx);
```

Versión no bloqueante de la función anterior. En lugar de bloquearse falla y retorna EBUSY.

Incrementar mutex

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mtx);
```

Suma 1 al valor del mutex. Retorna 0 en caso de éxito y -1 en caso de error.

Destrucción de mutex

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mtx);
```

Destruye un mutex inicializado en forma dinámica. Deben ser destruidos antes de la liberación de dicha memoria dinámica. Solo es seguro destruirlos si se encuentra en estado 1 y ningún evento intenta decrementarlo. Un mutex inicializado de forma estática no necesita ser destruido.

Interbloqueos de mutex

En ciertas ocasiones, un hilo necesita acceder a más de un recurso compartido. En dicha ocasión utiliza un mutex para cada recurso. Los lock y unlock de los mutex pueden producir interbloqueos.

```
/* Hilo A */

pthread_mutex_lock(mtx1);

pthread_mutex_lock(mtx2);

pthread_mutex_unlock(mtx2);

pthread_mutex_unlock(mtx1);
```

```
/* Hilo B */

pthread_mutex_lock(mtx2);

pthread_mutex_lock(mtx1);

pthread_mutex_unlock(mtx1);

pthread_mutex_unlock(mtx2);
```

Para evitarlo deben hacerse los locks y unlocks en el mismo orden en todos los hilos.

Señales

Una señal es una notificación a un proceso de que ha ocurrido un evento. Suelen ser descriptas como interrupciones de software, son análogas a interrupciones de hardware ya que interrumpen el flujo normal de ejecución de un programa y en muchos casos no es posible predecir exactamente cuándo llegará una señal.

Un proceso puede enviar una señal a otro. Este uso puede ser empleado como una técnica de sincronización, o incluso como una forma primitiva de IPC. La fuente emisora usual de muchas señales es el kernel.

Tras la entrega de una señal, un proceso lleva a cabo una de las siguientes acciones por defecto:

- Ignorar la señal, es decir, (descartada por el kernel, el proceso nunca se entera).
- Terminar (killed).
- Suspenderse.
- Reanudarse luego de haberse suspendido.

Un programa puede cambiar la acción que ocurre cuando una señal llega por una de las siguientes:

- Ignorar la señal.
- Ejecutar un manejador de señal, el cual es una función, escrita por el programador, que realiza tareas apropiadas en respuesta de la llegada de una señal.

GESTIÓN DE MEMORIA

La RAM principal debe administrarse. Los SO crean abstracciones de la memoria y las administran. Mediante un registro de las partes en uso, asignan y desasignan memoria a los procesos. El SO abstrae la jerarquía de memoria en un modelo útil y la administra.

Sin abstracciones de memoria

Denominado también monoprogramación sin intercambio ni paginación. Cada programa ve la memoria física. No es posible tener dos programas ejecutándose, uno podría escribir en el otro.

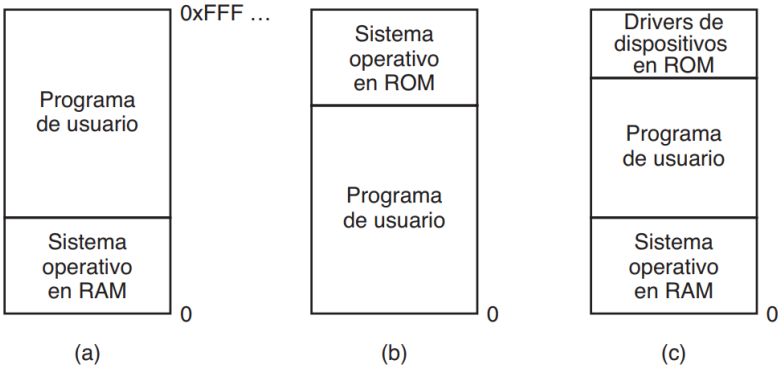


Figura 3-1. Tres formas simples de organizar la memoria con un sistema operativo y un proceso de usuario. También existen otras posibilidades.

a) Usado antes de los mainframes, ya no se usa.
b) Usando en PC de bolsillo y sistemas integrados.
c) Usado en al primer computadora personal, la porción ROM es la BIOS.

Los modelos a y c tienen la desventaja de que un error en el programa de usuario puede borrar el SO.

Al ser monoprogramación, el SO copia el programa solicitado del disco a la memoria y lo ejecuta, cuando termina muestra un carácter indicador de comendo y espera otro comando. Al recibirlo carga un nuevo programa, sobrescribiendo el primero. Una forma de obtener paralelismo es mediante hilos, lo cual tiene al limitación de requerir que los programas estén relacionados.

EJECUCIÓN DE MÚLTIPLES PROGRAMAS SIN ABSTRACCION DE MEMORIA

Denominado también multiprogramación con partes fijas. El SO guarda el contenido de la memoria en un archivo en disco, luego trae y ejecuta el siguiente programa. Esto se denomina intercambio.

Añadiendo hardware es posible la multiprogramación sin intercambio al dividir la memoria y asignando a cada bloque una llave de protección guardada en registros especiales. El hardware mediante un TRAP controla cualquier intento de un proceso de acceder a la memoria con un código de protección diferente. El SO puede modificar las llaves, controlando a los procesos para que no se interfieran.

Las desventajas de este esquema son:

- Los programas hacen **referencia a la memoria física absoluta**, no a su bloque de direcciones locales privado. Una solución es modificar el programa a medida que se carga en memoria, lo cual se denominó reubicación estática. Esta técnica requiere información adicional.
- Existe **fragmentación interna** debido a que generalmente cada programa no ocupa todo su bloque de direcciones.

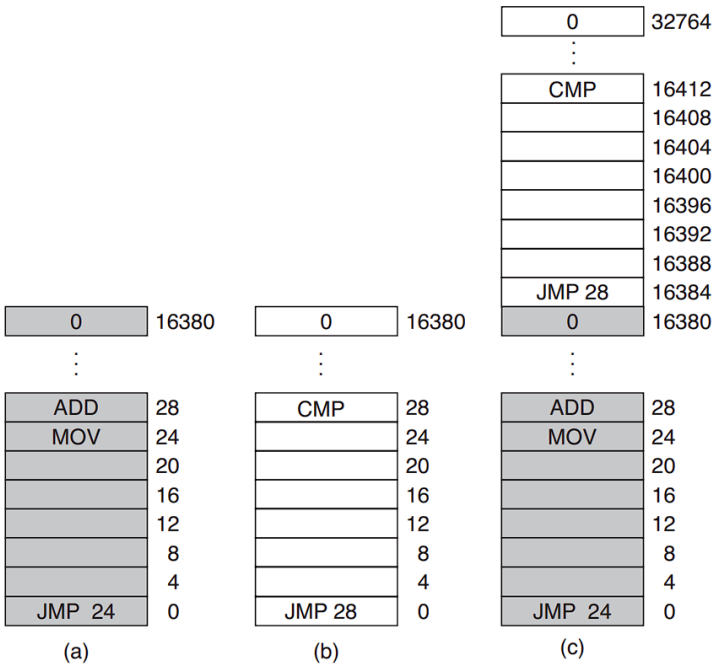


Figura 3-2. Ilustración del problema de reubicación. (a) Un programa de 16 KB. (b) Otro programa de 16 KB. (c) Los dos programas cargados consecutivamente en la memoria.

Una abstracción de memoria: espacio de direcciones

Deben resolverse 2 problemas del esquema anterior para permitir múltiples programas en memoria:

1. **Protección:** el método anterior lo soluciona mediante la llave de bits. Sin embargo este método no escala bien, ya que la cantidad de bits depende del espacio de memoria.
2. **Reubicación:** el método anterior lo soluciona reubicando los programas al momento de cargarlos, lo cual es lento y complicado.

Se solucionaron estos problemas mediante la abstracción del **espacio de direcciones**. Este es un conjunto de direcciones que puede utilizar un proceso para direccionar memoria. Cada proceso tiene su propio espacio de direcciones.

REGISTROS BASE Y LIMIT

Los programas son cargados en ubicaciones contiguas de memoria. Al ejecutar un proceso, el registro base se carga con la dirección física donde empieza el programa en memoria y el registro limit con la longitud de mismo. Cuando posteriormente un proceso hace referencia a la memoria, el hardware de la CPU suma el valor de base a la dirección generada antes de enviarla al bus de memoria. Estos dos registros dan una forma fácil de proporcionar a cada proceso su espacio de direcciones.

Una desventaja es la necesidad de realizar una suma y una comparación en cada referencia a memoria, sobre todo porque las sumas son lentas debidas a la velocidad de propagación del acarreo.

INTERCAMBIO – MULTIPROGRAMACIÓN CON PARTICIONES VARIABLES

Normalmente el tamaño de la RAM es menor al necesario para tener todos los programas. Para resolver esto se utilizan dos esquemas:

1. **Intercambio:** cada proceso se lleva por completo a memoria, se ejecuta cierto tiempo y luego vuelve al disco.

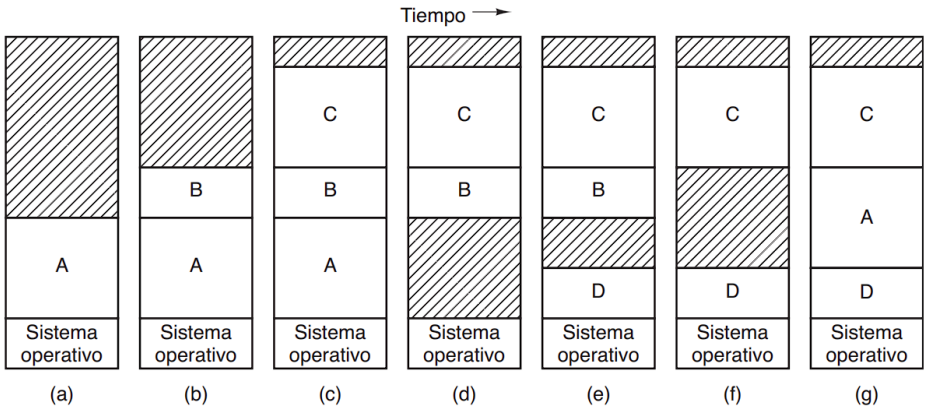


Figura 3-4. La asignación de la memoria cambia a medida que llegan procesos a la memoria y salen de ésta. Las regiones sombreadas son la memoria sin usar.

En la figura anterior, en a y g se observa que A reingresa en una ubicación distinta, por lo que sus direcciones deben reubicarse (modificar registro base). Además en g se observa la compactación de memoria para que no queden huevos por fragmentación externa.

2. **Memoria virtual:** permite que los programas se ejecuten encontrándose de forma parcial en la memoria.

ADMINISTRACIÓN DE MEMORIA LIBRE

Cuando la memoria se asigna dinámicamente, el SO debe administrarla. Existen 2 formas de registrar el uso de la memoria:

- **Mapas de bits:** la memoria se divide en unidades de asignación. A cada unidad corresponde 1 bit en el mapa, donde 0 indica que está libre y 1 que está ocupada. Para cargar un proceso de k unidades, el administrador de memoria debe buscar en el mapa un k bits consecutivos en 0. Debe seleccionarse un tamaño de unidad de asignación acorde al tamaño de la memoria. Un valor pequeño requeriría un mapa muy extenso, un valor muy grande podría no dividir la memoria de forma adecuada.
- **Listas enlazadas:** lista enlazada de segmentos de memoria asignada y libre. Cada segmento contiene un proceso o es un hueco entre dos procesos. Cada entrada en la lista especifica un hueco (H) o proceso (P), la dirección de inicio, la longitud y un puntero a la siguiente entrada. La actualización de la lista requiere cambiar un P por un H y unir los elementos adyacentes.

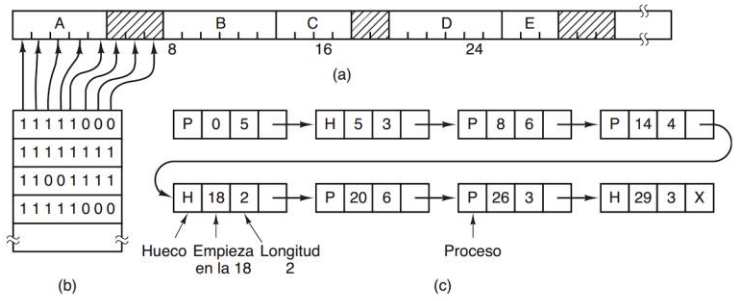


Figura 3-6. (a) Una parte de la memoria con cinco procesos y tres huecos. Las marcas de graduación muestran las unidades de asignación de memoria. Las regiones sombreadas (0 en el mapa de bits) están libres. (b) El mapa de bits correspondiente. (c) La misma información en forma de lista.

El mapa de bits es más eficiente en la liberación de programas, pero menos en la búsqueda de espacios libres. Por otro lado, la lista enlazada es mejor en la exploración pero menos eficiente en la liberación de memoria, debido a la necesidad de exploración de los elementos adyacentes.

Algunos algoritmos para la asignación de memoria son:

- **Primer ajuste:** el administrador explora la lista hasta encontrar un hueco donde quepa el proceso. Puede tener el inconveniente de nunca usar posiciones altas de memoria.
- **Siguiente ajuste:** funciona como el anterior, solo que lleva un registro de donde se encontró la última vez que asignó memoria. La próxima búsqueda comienza en dicha posición.
- **Mejor ajuste:** explora toda la lista y toma el hueco más pequeño donde quepa el programa. Es más lento debido a esta exploración y además genera huecos demasiado pequeños e inutilizables.
- **Peor ajuste:** toma siempre el hueco más grande disponible, así deja huecos grandes reutilizables.
- **Ajuste rápido:** acelera los 4 anteriores mediante una lista separada de procesos y huecos, con los tamaños más comúnmente utilizados Posee la misma desventaja que los esquemas que se ordenan por tamaño de hueco, si no se realiza fusión de procesos cercanos cuando un proceso termina, la memoria se fragmenta en grandes huecos. La fusión es un proceso costoso.

Memoria virtual

Provoca que el procesador crea tener una memoria diferente a la que realmente tiene. La paginación de la ilusión de una memoria mayor y la segmentación de tener varios espacios de direcciones separados.

La idea central es que cada programa tenga su espacio de direcciones dividido en trozos llamados páginas. Cuando el programa hace referencia a una porción de su espacio de direccione en memoria, el hardware hace la asociación al instante. Si hace referencia a una parte que no está en memoria, el SO recibe una alerta, la busca y vuelve a ejecutar la instrucción fallida.

PAGINACIÓN

Los programas hacen referencia a direcciones de memoria (direcciones virtuales) que forman el espacio de direcciones virtuales. Este espacio se divide en unidades de tamaño fijo llamadas **páginas**. Las unidades correspondientes en la memoria física se dividen en **marcos de página**. Marcos y páginas son del mismo tamaño, cada una con 4096 direcciones. La MMU asigna paginas a marcos de página.

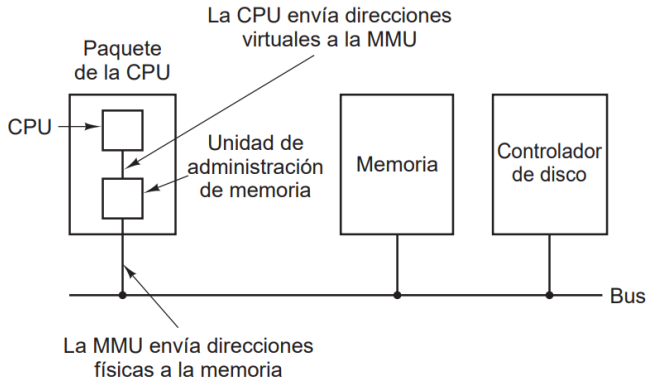


Figura 3-8. La posición y función de la MMU. Aquí la MMU se muestra como parte del chip de CPU, debido a que es común esta configuración en la actualidad. Sin embargo, lógicamente podría ser un chip separado y lo era hace años.

Por ejemplo, si un programa intenta acceder a la dirección virtual 0, ésta se envía a la MMU. La MMU ve que dicha dirección se encuentra en la página 0, que de acuerdo a su asociación es el marco de

página 2. Transforma la dirección y la envía al bus, asociando todas las direcciones virtuales entre la pagina 0 y el marco 2.

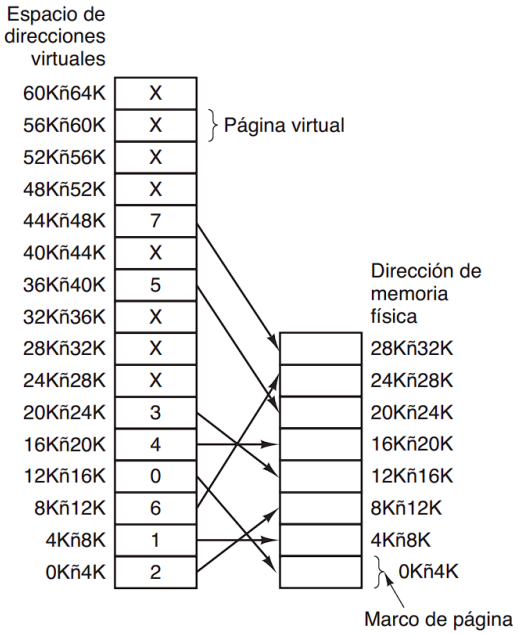
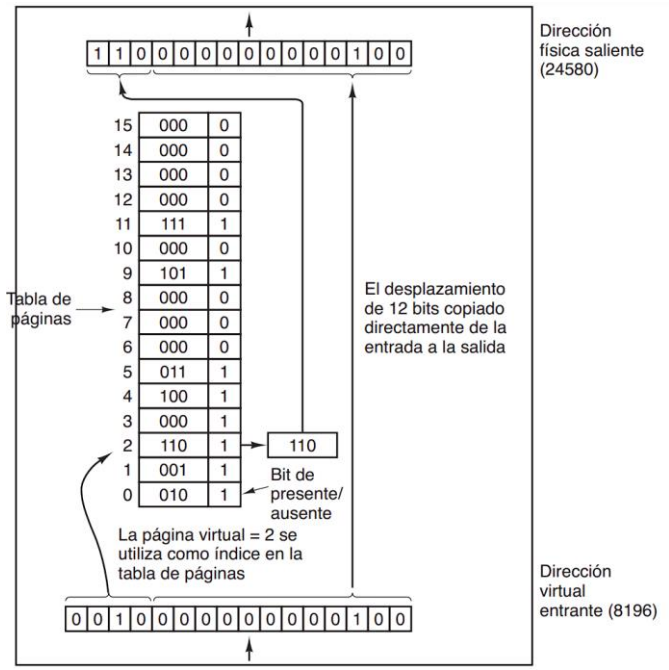


Figura 3-9. La relación entre las direcciones virtuales y las direcciones de memoria física está dada por la tabla de páginas. Cada página empieza en un múltiplo de 4096 y termina 4095 direcciones más arriba, por lo que de 4 K a 8 K en realidad significa de 4096 a 8191 y de 8 K a 12 K significa de 8192 a 12287.

La cantidad máxima de páginas virtuales que se asocian a la memoria física es igual a la cantidad de marcos de página. Un bit presente/ausente lleva el registro de cuáles páginas están presentes en memoria física. Si un programa referencia direcciones no asociadas, la MMU lo detecta y hace que la CPU haga un TRAP al SO denominado **fallo de página**. El SO selecciona un marco de página que se utilice poco y escribe su contenido de vuelta al disco. Luego obtiene la pagina a la que se acaba de referenciar, la asigna al marco recientemente liberado cambiando la asociación en la tabla de páginas y reinicia la instrucción que originó el TRAP. En la siguiente figura se observa la operación interna de la MMU.



Los pasos efectuados son:

1. Utiliza el numero de página como índice en la tabla de páginas.
2. Si el bit presente/ausente está en 0 genera un TRAP al SO.
3. Si es 1, el número de marco se copia a los 3 bits de mayor orden del registro de salida, los 12 bits de desplazamiento se copian sin modificación de la dirección virtual entrante. Se forma así la dirección de 15 bits.
4. El registro de salida se coloca en el bus de memoria como la dirección de memoria física.

TABLA DE PÁGINAS

Su propósito es asociar páginas virtuales a los marcos de página. La tabla es una función cuyo argumento es el número de página y su resultado es el número de marco.

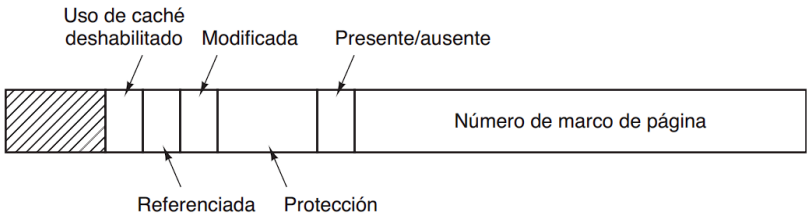


Figura 3-11. Una típica entrada en la tabla de páginas.

El tamaño de la página varía de una máquina a otra, suele ser de 32 bits, sus campos son:

- **Bit presente/ausente**: indica si la entrada es o no válida.
- **Bits de protección**: indican qué tipo de acceso está permitido, la forma más simple es 1 bit$(0: w, 1: r/w)$, pero puede tener 3 (rwx).
- **Bits modificada M y referenciada R**: registran el uso de la página para ayudar al SO a elegir una página para desalojar ante un fallo de página.
    - **bit modificada** indica si la página ha sufrido modificaciones en la ejecución, en cuyo caso debe guardarse antes de eliminarse.
- **Uso de caché:** habilita o deshabilita el uso de caché de página.

ACELERACIÓN DE PAGINACIÓN

En los sistemas de paginación deben abordarse dos cuestiones:

1. **La asociación de páginas debe ser rápida.** Se realizan varias asociaciones por instrucción.
2. **El tamaño de páginas debe ser adecuado.** Paginas muy chicas generan muchas páginas, páginas muy grandes generan fragmentación externa.

Un esquema implementado para la aceleración de paginación es el uso de **búferes de traducción adelantada**. Por la paginación, cada instrucción requiere una referencia a memoria adicional, lo cual reduce el rendimiento. Una solución, gracias a que la mayoría de los programas tienden a hacer gran cantidad de referencias a un pequeño conjunto de páginas, es equipar un dispositivo de hardware que asocie direcciones virtuales a físicas, sin pasar por la tabla de páginas. Este se denomina **TLB** o memoria asociativa y se encuentra por lo general en la MMU.

Al presentarse una dirección virtual en la MMU, el hardware comprueba si el número de página está presente en el TLB, en cuyo caso, si el acceso no viola la protección, se toma el marco de página.

Si el número de página no se encuentra en el TLB, la MMU lo detecta y realiza la búsqueda en la tabla de páginas, luego desaloja una entrada del TLB y la reemplaza con la página que acaba de encontrar.

ALGORITMOS DE REEMPLAZO DE PÁGINAS

Luego de un fallo de página el SO debe elegir una página a desalojar y hacer espacio para la página entrante. Algunos algoritmos para tomar dicha decisión son:

- **Reemplazo de páginas óptimo:** no implementable, se utiliza como referencia para comparar con las demás. Cada página se etiqueta con el número de instrucciones que se ejecutarán antes de que se la referencie por primera vez.
- **NFU – No usadas frecuentemente:** utiliza los bits R y M. Ante una falla, el SO inspecciona las páginas y las divide en 4 categorías:

Clase	bit R	bit M
0	0	0
1	0	1
2	1	0
3	1	1

  Elimina una página de la clase de menor numeración que no esté vacía.
- **FIFO – Primera en entrar, primera en salir:** el SO mantiene una lista de las páginas en memoria, la menos recientemente llegada se ubica en la parte frontal y es eliminada ante un fallo de página. Luego se coloca la nueva al final.
- **Segunda oportunidad:** modificación del FIFO. Inspecciona el bit R de la página más antigua, si es 0 se reemplaza por la nueva página, si es 1 se borra el bit y se coloca al final de la cola como si recién llegara.

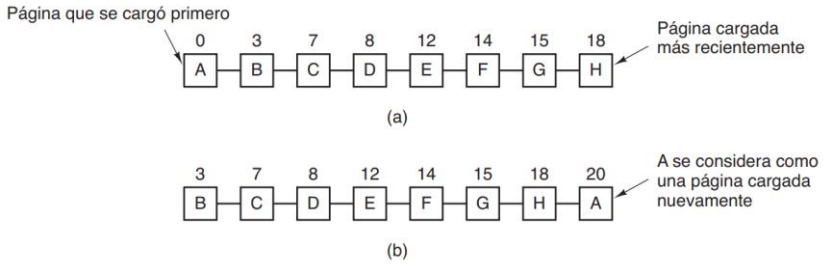


Figura 3-15. Operación del algoritmo de la segunda oportunidad. (a) Páginas ordenadas con base en FIFO. (b) Lista de las páginas si ocurre un fallo de página en el tiempo 20 y *A* tiene su bit *R* activado. Los números encima de las páginas son sus tiempos de carga.

- **Reloj:** los dos algoritmos anteriores son ineficientes por estar moviendo páginas en su lista. Una mejora implica mantener los marcos en una lista circular. Ante un fallo de página, la pagina apuntada por la aguja se inspecciona, si R = 0 la pagina es desalojada, se inserta la nueva y se apunta con la aguja a la siguiente. Si R = 1, se borra el bit y se mueve la aguja al siguiente marco.

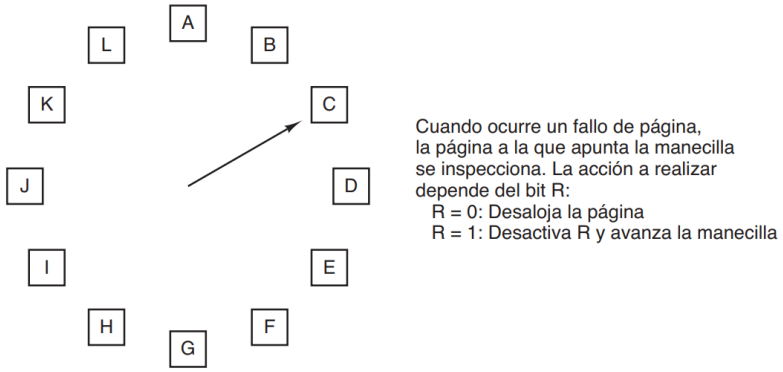


Figura 3-16. El algoritmo de reemplazo de páginas en reloj.

- **LRU – Menos usada recientemente:** se aproxima a un algoritmo óptimo. En una máquina con n marcos, el hardware de LRU mantiene una matriz de $n \times n$ (todos ceros). Cuando se referencia a la página en el marco k se establecen todos los bits de la fila k en 1 y luego todos los de la columna k en 0. La fila con menor cantidad de unos indica la página que menos se usó.

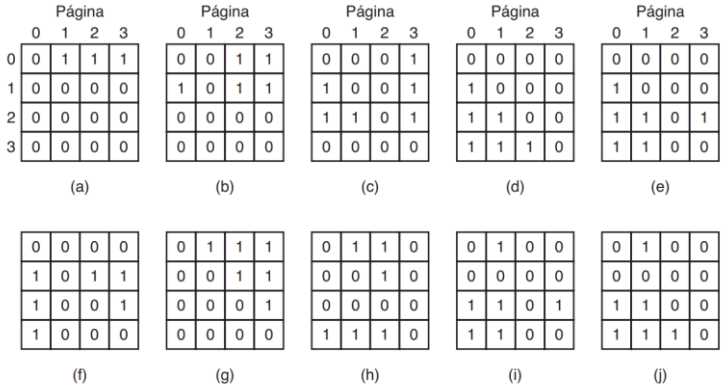


Figura 3-17. LRU usando una matriz cuando se hace referencia a las páginas en el orden 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

- **Conjunto de trabajo:** el conjunto de páginas utilizado por un proceso en un momento dado se denomina conjunto de trabajo. Muchos sistemas de paginación tratan de llevar la cuenta del conjunto de trabajo de cada proceso y se aseguran de que esté en memoria antes de que el proceso se ejecute. Este proceso de cargar páginas antes de la ejecución se denomina prepaginación. Ante un fallo de página se busca una página que no esté en el conjunto de trabajo y se la desaloja.

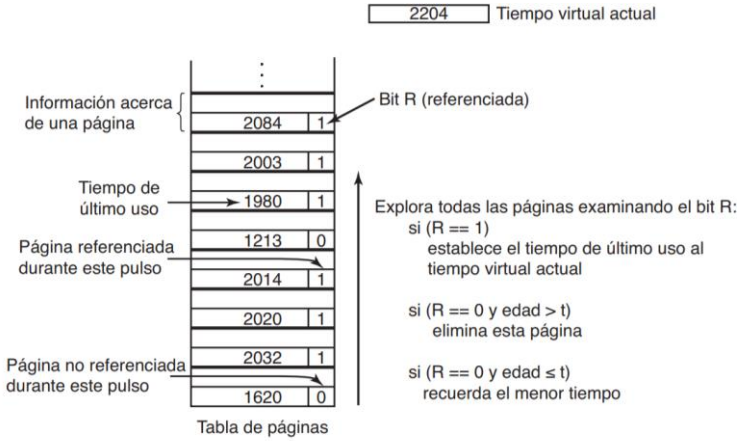


Figura 3-20. El algoritmo del conjunto de trabajo.

- **WSClock:** se basa en el algoritmo de reloj y utiliza la información del conjunto de trabajo.
  - Al principio la lista está vacía y a medida que se agregan páginas forman un anillo.
  - Cada entrada tiene el campo **tiempo de ultimo uso** del algoritmo anterior y los bits R y M.
  - Ante un fallo de página se examina la página apuntada por la aguja del reloj.
  - Si R = 1, la página no es eliminada, se establece R en 0 y la aguja avanza.
  - SI R = 0:
    - Si la edad es mayor que τ (no pertenece al conjunto de trabajo) y la página está limpia (copia válida en disco), el marco de página es reclamado y la nueva página se coloca allí.
    - Si la página está sucia, el marco no puede ser reclamado de inmediato ya que on existe una copia válida de la página en disco. Se planifica la escritura en disco de dicha página pero la aguja del reloj avanza.
    - La primer página limpia con edad mayor a τ es desalojada.

Resumen de los algoritmos

- El óptimo solo sirve de referencia.
- NFU divide las páginas en 4 clases según los bits M Y R.
- FIFO registra el orden en que se cargan las páginas en una lista enlazada.
- Segunda oportunidad es un FIFO que comprueba el bit R.
- LRU requiere hardware especial, es difícil de implementar.
- Los 2 últimos utilizan el conjunto de trabajo. WSClock es el mejor.

Segmentación

Hasta ahora la memoria virtual es unidimensional. El problema de ello es que existen tablas que modifican su tamaño en forma continua y/o impredecible mientras avanza la compilación. Se necesita entonces de una abstracción que libere al programador de administrar las tablas en expansión y contracción. Una solución simple es proporcionar a la máquina muchos espacios de direcciones independientes, llamados **segmentos**, cada cual es una secuencia lineal de direcciones.

Cada segmento constituye un espacio de direcciones separado, pueden crecer o decrecer de forma independiente sin afectar a otro. Pueden llenarse pero por lo general son muy grandes. Para especificar una dirección el programa debe suministrar una dirección en 2 partes, un número de segmento y una dirección dentro del segmento.

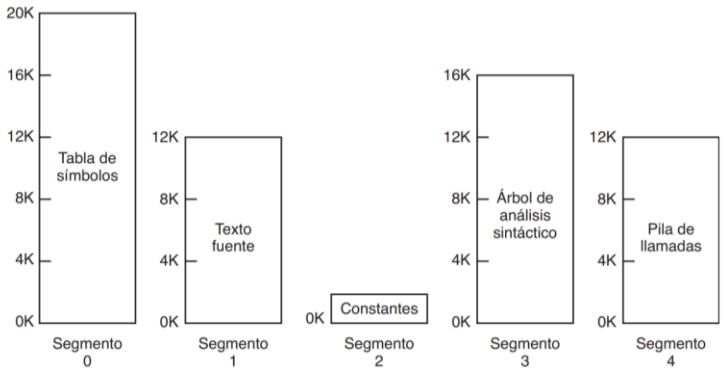


Figura 3-32. Una memoria segmentada permite que cada tabla crezca o se reduzca de manera independiente a las otras tablas.

Un segmento es una entidad lógica de la que el programador está consciente. Puede contener arreglos, pilas, constantes, etc. pero generalmente no una mezcla de estos.

Facilita la compartición de procedimientos o datos entre varios procesos, además distintos segmentos pueden tener diferentes tipos de protección.

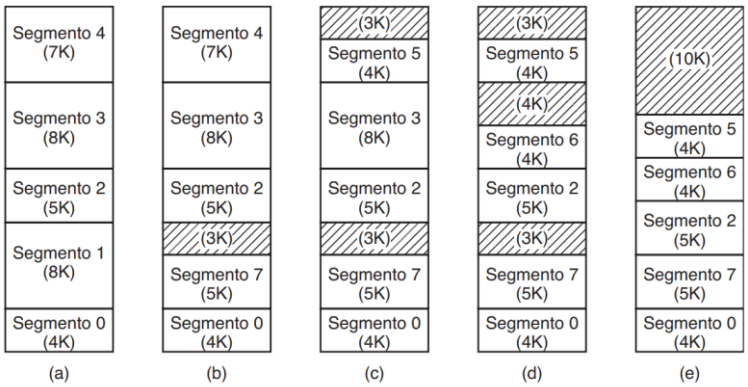


Figura 3-34. (a)-(d) Desarrollo del efecto de tablero de ajedrez. (e) Eliminación del efecto de tablero de ajedrez mediante la compactación.

IMPLEMENTACIÓN DE SEGMENTACIÓN PURA

Consideración	Paginación	Segmentación
Programador consciente de su existencia	No	Si
Cantidad de espacios de dirección lineal	1	Muchos
El espacio total puede exceder al de la memoria física	Si	Si
Pueden diferenciarse procesos y datos para protegerse por separado	No	Si
Facilita compartición de procedimientos entre usuarios	No	Si
Razón de su invención	Obtener un gran espacio de direcciones lineal sin tener que comprar más memoria física.	Permitir a los programas y datos dividirse en espacios de direcciones lógicamente independientes, ayudando a la compartición y protección.

Si un segmento se desaloja y otro más pequeño se coloca en su lugar, entre segmentos adyacentes queda un espacio sin uso. Si se repite en reiteradas ocasiones se produce **fragmentación externa**. Esto puede mejorarse mediante compactación o paginación segmentada, técnica que requiere cargar los segmentos de programa por partes.

Fragmentación externa: espacios de memoria inutilizables generados al colocar o eliminar procesos. Estos espacios se encuentran entre dos procesos adyacentes.
Fragmentación interna: espacio de memoria inutilizable dentro de un proceso, debido a una asignación de espacio mayor requerida por el mismo.

SISTEMAS OPERATIVOS DE TIEMPO REAL – RTOS

Sistemas de tiempo real – RTS

Un sistema informático en tiempo real es aquel en el que la corrección del resultado depende tanto de su validez lógica como del instante en que se produce. Es decir, para que un resultado sea completamente correcto debe dar el valor correcto en el momento correcto.

Para garantizar el tiempo de respuesta un RTS necesita no solo velocidad de respuesta, sino determinismo. El tiempo de ejecución debe ser acotado al caso más desfavorable, de modo que se garantice que se cumplirán siempre las restricciones temporales. No necesariamente necesitan ejecutarse rápido, su velocidad debe ser acorde a la aplicación en que se lo implementa.

Los métodos para implantar un sistema en tiempo real son:

- Procesamientos secuencial (bucle scan).
- Foreground/Background (interrupciones).
- RTOS:
    - Multitarea cooperativa
    - Multitarea expropiativa

PROCESAMIENTO SECUENCIAL

Se trata de un bucle scan que se repite indefinidamente, ejecutando tareas en un orden predefinido. Para trabajar como RTS debe cumplir lo siguiente:

1. No bloquearse en espera de eventos externos asíncronos.
2. La suma de todos los tiempos máximos debe ser menor al tiempo de muestreo de bucle Ts.
3. Si se requiere un Ts exacto debe usar temporizadores o tareas que se bloqueen cierto tiempo.

Los inconvenientes son la latencia (Ts en el peor de los casos) y la necesidad de hacer entrar todas las tareas en un ciclo. Para este último inconveniente dos posibles soluciones son:

1) Cambiar el procesador
2) Dividir las tareas

También surgen problemas cuando se tienen tareas con periodos diferentes al del bucle.

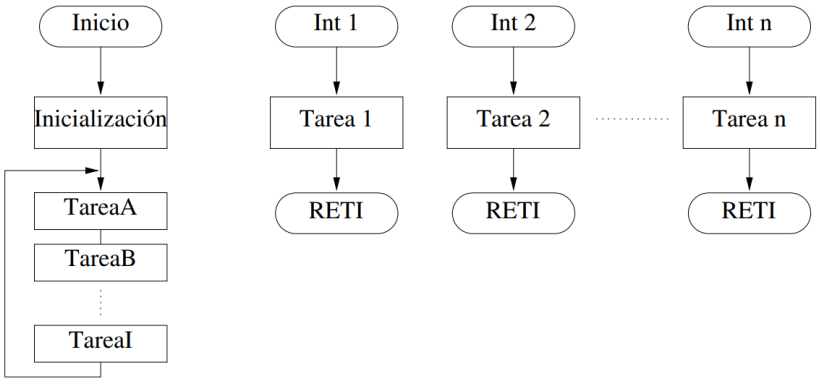
Su ventaja es la sencilla implementación y depuración, la facilidad para compartir datos y la eficiencia.

SISTEMA FOREGROUND/BACKGROUND

Utiliza interrupciones para esperar eventos externos asíncronos. Posee dos tipos de tareas:

- Foreground (bucle scan).
- Background (rutina de atención de interrupción – ISR).

Ejecuta el bucle scan hasta la llegada de una interrupción, momento en el cual ejecuta la ISR y retorna al bucle scan. Su latencia depende de la latencia máxima de las tareas en segundo plano.



Desventajas:

- Necesita soporte de interrupciones de hardware.
- Cada tarea background debe asociarse a una interrupción.
- Las rutinas de background deben ser cortas (se lleva parte de background a foreground).
- Problemas de concurrencia.
- Depuración compleja.

RTOS – SISTEMAS OPERATIVOS DE TIEMPO REAL

Si bien existe gran cantidad de RTOS, todos tienen los siguientes componentes en común:

1. Mecanismos para permitir compartir datos entre tareas.
2. Mecanismos para sincronizar tareas.
3. Planificador.

Planificador

Mantiene estructuras de datos para conocer qué tareas pueden ejecutarse. Si existe más de una tarea lista usa información adicional para decidir cual ejecutar. Puede ejecutar estas tareas periódicamente o cuando no tenga nada mejor que hacer.

Existen 2 tipos de planificadores:

- **Cooperativos:** las tareas en primer plano llaman al planificador, realizan una cesión de CPU (yield). El problema es la falta de control sobre la latencia de las tareas de primer plano, lo que dificulta garantizar su temporización. La ventaja es la cesión explicita de la CPU, así no existen problemas de incoherencia de datos entre tareas de primer plano, aunque si entre tareas e ISRs.
- **Expropiativos:** mejoran la latencia de las tareas en primer plano. El planificador se ejecuta periódicamente de forma automática, mediante una interrupción generada por un temporizador. Como su ejecución lleva tiempo, no puede elegirse un intervalo muy pequeño.

FreeRTOS

TAREAS EN FREERTOS

Las tareas no deben terminar nunca, siempre deben ser bucles infinitos.

Creación de tareas

```
#include “FreeRTOS.h”
#include “task.h”

BaseType_t xTaskCreate(TaskFunction_t pvTaskCode,
                       constchar * constpcName,
                       uint16_t usStackDept,
                       void *pvParameters,
                       UBaseType_t uxPriority,
                       TaskHandle_t *pxCreatedTask);
```

- `TaskFunction_t pvTaskCode`: nombre de la función. Pueden crearse múltiples tareas con una misma función.
- `constchar * constpcName`: nombre utilizado para depuración.
- `uint16_t usStackDept`: tamaño de pila donde cada tarea almacena su información. Se le debe asignar un tamaño de pila en bytes. Una macro comúnmente utilizada es `configMINIMAL_STACK_SIZE`, la cual asigna el tamaño mínimo de pila.
- `void *pvParameters`: parámetros pasados a la tarea.
- `UBaseType_t uxPriority`: prioridad de la tarea. El rango va de `tskIDLE_PRIORITY` a `configMAX_PRIORITIES-1`.
- `TaskHandle_t *pxCreatedTask`: ID de tarea.

La función retorna un 0 en caso de éxito y 1 en caso de error (pdPASS/pdFAIL o pdTRUE/pdFALSE).

Planificador de FreeRTOS

Puede ser configurado mediante las macros. Un modo comúnmente utilizado es `Fixed Priority Preemptive with Time Slicing`, es decir, expropiativo de prioridad fija con quantum de tiempo definido. Las macros a configurar para ello son:

- configUSERPREEMPTION = 1.
- configUSETIME_SLICING = 1.

Este planificador funciona de la siguiente manera:

- Ante dos tareas de igual prioridad ejecuta una y al terminar el quantum ejecuta la otra, son tratadas por igual compartiendo tiempo de ejecución.
- No da tiempo de ejecución a tareas de menor prioridad, a no ser que una de mayor prioridad se bloquee.
- El quantum se define en la macro `configTICK_RATE_HZ`.

Tipos de tareas

- **Periódicas:** se ejecutan con un frecuencia determinada.
  - Mayormente bloqueadas.
  - Alta prioridad para que se respete su frecuencia (baja latencia).
  - Usan timer y contador.
- **Tareas aperiódicas:**
  - Mayormente bloqueadas, su desbloqueo depende de un evento externo.
  - Prioridad intermedia.
- **Tareas continuas:**
  - Mayormente en ejecución.
  - Baja prioridad, así pueden ser interrumpidas por tareas periódicas y aperiódicas.

Recursos para el manejo de tareas

```
void vTaskDelay(TickType xTicksToDelay);
```

Delay de duración que depende del valor de `xTicksToDelay`. Para poder utilizar esta función debe definirse en 1 la macro INCLUDE_vTaskDelay. El periodo de cada tick depende de la configuración de la macro de `configTICK_RATE_HZ`. Para trabajar en unidades de tiempo en lugar de Ticks, pueden usarse dos funciones:

- `time_msec/`**`porTICK_RATE_MS`**.
- **`pdMS_TO_TICKS`**`(time_msec)`.

```
void vTaskDelayUntil(TickType_t * pxPreviousWakTime, TickType_t * pxTimeIncrement);
```

Es otro recurso temporal. Sus parámetros son:

- `TickType_t *` **`pxPreviousWakTime:`** número de ticks al momento en el que se bloqueó la tarea. Este valor se obtiene por primera vez desde la función `xTaskGetTickCount()` y luego es actualizado automáticamente por la función `vTaskDelayUntil`.
- `TickType_t *` **`pxTimeIncrement`**: tiempo de bloqueo. La tarea se desbloquea en **`pxPreviousWakTime + pxTimeIncrement`**.

Para usar esta función debe setearse la macro INCLUDE_vTaskDelayUntil.

```
UBaseType_t uxTaskPriorityGet(TaskHandle_t pxTask);
```

Recurso no temporal para obtener la prioridad de la tarea `pxTask`.

```
void vTaskPrioritySet(TaskHandle_t pxTask, UBaseType_t uxNewPriority);
```

El SO no las cambia por lo que es trabajo del programador hacerlo. Esta función realiza el cambio de

prioridad de la tarea `pxTask` a un valor `uxNewPriority`. Tanto en esta función como en la anterior, si se pasa como argumento `pxTask` un puntero a `NULL` se apunta a la tarea que ejecuta la función.

RECURSOS PARA SINCRONIZAR TAREAS

Se utilizan semáforos. FreeRTOS posee los siguientes tipos de semáforos:

- Binarios: pueden interpretarse como una cola de mensajes con un solo mensaje.
- Mutex: la diferencia con los binarios es que la tarea que lo toma es la que debe devolverlo.
- Counting: puede ser tomado varias veces.

Todos estos semáforos restringen el acceso a una sección (exclusión mutua), ordenan cronológicamente (serializan) y sirven como punto de encuentro.

Poseen 2 primitivas, incrementar y decrementar. Se define un tiempo máximo de bloqueo, el cual una vez cumplido se desbloquea la tarea.

Creación de semáforo

```
#include "semphr.h"

SemaphoreHandle_t xSemaphoreCreateBinary(void);
```

Retorna un semáforo binario. Debe setearse la macro `INCLUDE_vTaskSuspend`.

Decrementar semáforo

```
BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);
```

- `SemaphoreHandle_t` **`xSemaphore`**: semáforo creado previamente.
- `TickType_t` **`xTicksToWait`**: tiempo máximo de bloqueo. Para el infinito se usa portMAX_DELAY.

Incrementar semáforo

```
BaseType_t xSemaphoreGive(SemaphoreHandle_t xSemaphore);
```

RECURSOS PARA COMUNICACIÓN ENTRE TAREAS – COLA DE MENSAJES

Las variables globales pueden generar problemas de incoherencia entre tareas que las accedan. Por eso se utilizan colas de mensajes, que además proveen sincronismo sin semáforos.

Las colas de mensajes deben:

- Ser globales.
- Bloquearse al leer/escribir en ellas.
- Definir un tamaño máximo y cantidad de mensajes (tamaño limitado de heap).
- Opcionalmente definir un tiempo máximo de bloqueo.
- Ser LIFO o FIFO.

Creación de cola de mensajes

```
#include "queue.h"
QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);
```

Retorna la cola de mensajes creada. Sus argumentos son el máximo tamaño de cola y el tamaño de cada mensaje.

Escritura en cola de mensajes

```
BaseType_t xQueueSend(QueueHandle_t xQueue,
                      const void * pvItemToQueue,
                      TickType t xTicksToWait);
```

Retorna pdPASS en caso de éxito o pdFAIL en caso de error. El primer argumento indica la cola, el segundo el mensaje y el tercero el tiempo máximo de bloque en caso de estar llena la cola.

`xQueueSend` es una abreviación de `xQueueSendToBack`, función que usa la cola como una FIFO. Para usarla como LIFO se utiliza `xQueueSendToFront`.

Lectura de cola de mensajes

```
BaseType_t xQueueReceive(QueueHandle_t xQueue,
                         void * const pvBuffer,
                         TuckType_t xTicksToWait);
```

El mensaje leído se almacena en `pvBuffer`. Puede usarse `xQueuePeek` para leer sin eliminar de la cola.

ADMINISTRACIÓN DE RECURSOS EN FREERTOS

En lugar de usar semáforos, puede hacerse lo siguiente para compartir recursos sin problemas entre tareas.

- Deshabilitar interrupciones:
  - Aumenta latencia de sistema.
  - Sencillo de implementar y válida para exclusión Task – Task y Task – ISR (los ticks son ISR).
- Suspender planificador:
  - Aumenta latencia de Foreground.
  - Solo excluye Task – Task.
- Aumenta prioridad de tarea en ejecución.
  - Solo excluye Task – Task.

El uso de semáforos solo afecta a las tareas Foreground que comparten el recurso. Sin embargo, su implementación es más compleja y es propensa a errores de programación.

INVERSIÓN DE PRIORIDAD

Se produce al usar semáforos, ya que cuando una tarea toma un semáforo, la prioridad no es relevante, una tarea de mayor prioridad puede quedarse bloqueada por dicho semáforo hasta que la de menor prioridad lo libere. Algunas soluciones a este problema son:

- Usar planificador no apropiativo.
- Herencia de prioridad (mutex). Solo disminuye el problema.
- Configurar adecuadamente el máximo tiempo de bloqueo del semáforo.

MANEJO DE EVENTOS Y/O TAREAS APERIÓDICAS

El manejo de tareas aperiódicas puede realizarse mediante:

- Pooling (costoso).
- Interrupciones:
  - Deben ser cortas. FreeRTOS solo planifica tareas foreground.
  - Sus prioridades dependen del hardware, no del SO. La interrupción de menor prioridad interrumpe a la tarea más prioritaria y ejecuta la ISR correspondiente.

Consideraciones de ISR

Para que las ISR sean cortas debe realizarse el mayor procesamiento posible en foreground. Para ello se necesita enviar o sincronizar variables desde la ISR, lo cual se hace mediante funciones especiales para ISR. Estas funciones tienen la misma nomenclatura que las anteriores pero con el sufijo `FromISR`.

- FreeRTOS no distingue ISR y tarea, por lo que podría realizar un cambio de tarea en medio de una ISR.
- Dentro de la ISR deben usarse estas funciones ISR – safe para saber desde donde se invocan.
- Las funciones para bloquear no deben usarse en la ISR. No existen la funciones de Delay y debe tenerse cuidado con los semáforos y colas.

CAMBIOS DE CONTEXTO DESDE ISR

Las funciones `FromISR` tienen un argumento adicional que indica si debe hacerse un cambio de contexto al finalizar la ISR. Este se inicializa en `pdFALSE`.

```
portBASE_TYPE xSempahoreGiveFROMISR(xSemaphoreHandle Semaphore,
                                    portBASE_TYPE pxHigherPriorityTaskWoken);
```

Si al ejecutar la función se despierta una función de mayor prioridad, la variable `pxHigherPriorityTaskWoken` pasa a ser `pdTRUE`. Así el planificador sabe si debe seguir en la ISR o conmutar a la tarea de mayor prioridad que acaba de despertar.

```
if (xTasKWokenByPost == pdTRUE){
      portYIELD_FROM_ISR( xTaskWokenByPost );
}
```

Otra forma de hacerlo es:

```
if( xTaskWokenByPost == pdTRUE){
      taskYIELD();
}
```