Sistemas Operativos I

Iría Álvarez, Nerea Nieto, Sonia Rey

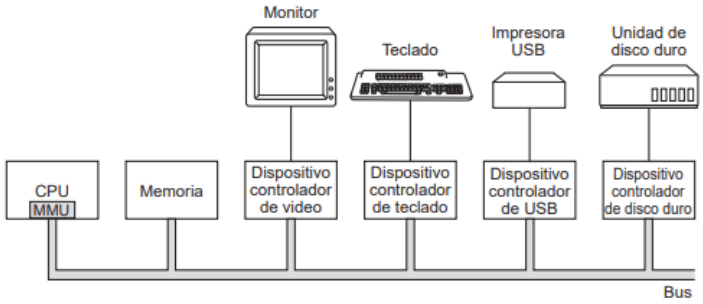
Tema 1

Introducción a los Sistemas Operativos

1.1. ¿Qué es un Sistema Operativo?

Un sistema operativo es un **programa** (un componente puramente software) cuyas funciones son dos. La primera es la de **administrar/gestionar** de manera eficiente **el hardware** de la máquina, incluyendo todos sus componentes (los distintos componentes de la memoria, dispositivos entrada/salida, …), haciéndole fácil la vida del usuario. La segunda consiste en **proporcionar** al programador **un conjunto abstracto de recursos**, por ejemplo, la abstracción de memoria o la interacción con el teclado (la cuál en MIPS se realiza con llamadas al sistema y no es tan sencilla).

El sistema operativo gestiona la memoria a través del movimiento de información entre los diferentes niveles de la jerarquía. Más concretamente se responsabiliza de la información transmitida entre la memoria caché y la RAM (o memoria principal) o entre la RAM y disco duro. Sin embargo, no gestiona las cachés (esa es una gestión hardware).



El sistema operativo ignora las características de los dispositivos de E/S, ya que solo interactúa con sus controladoras. Por definición una **controladora** es un elemento hardware electrónico y dentro de cada una, hay circuitos digitales para activar las líneas de control que unen las controladoras al dispositivo que controlan (por ejemplo, la controladora de monitor le indica a este que muestre por pantalla cierta información). A mayores, también se encuentran aquí unos registros dónde se escriben y se almacenan las operaciones que hay que realizar codificadas. Si el dispositivo mueve mucha cantidad de información, en vez de registros habrá buffers.

La memoria está organizada en una estructura jerárquica. En el nivel más bajo está el disco duro y tiene dos funcionalidades: ser un dispositivo de E/S con su propia controladora y estar en ese nivel más bajo de la memoria como respaldo de almacenamiento. En un programa de código ejecutable, todos los datos y código máquina están almacenados ahí, en la zona de **swapping**. A partir de ahí, esa información se mueve en la jerarquía a niveles más altos (a RAM, caché L2, caché L1 y registros de la CPU).

La CPU, incluye a la ALU y unidad de control (no es gestionada por el SO, pero sí accede a ella para ejecutar código). La **MMU** (Unidad de Manejo de Memoria), es un elemento hardware situado dentro de la CPU que tiene la función de ayudar al sistema operativo a manejar la memoria, realizando un papel bastante importante, ya que agiliza muchas situaciones.

El **bus** es un conjunto de cables eléctricos conductores que conectan los distintos dispositivos. También se organizan en jerarquías (no es lo mismo un bus que conecta un teclado, dónde la respuesta del usuario será muy lenta, que uno para el disco). Es el encargado de conectar la CPU con todo lo demás. Cualquier movimiento entre la controladora y la CPU debe pasar por el bus. Hay un único bus compartido. Es controlado por la misma CPU y en cada ciclo de reloj se tiene que decidir que 2 componentes se conectan, porque los demás deben esperar (en principio dos controladoras no podrán conectarse entre sí, aunque más adelante veremos como con DMA sí).

1.2. Niveles de abstracción

En la parte más baja de la abstracción está el **hardware**, con los circuitos. Todo lo que está más arriba es **software** (organizado a su vez en distintas capas). Entre ambos la comunicación se hace a través del **leguaje máquina**, que son todo 0 y 1.

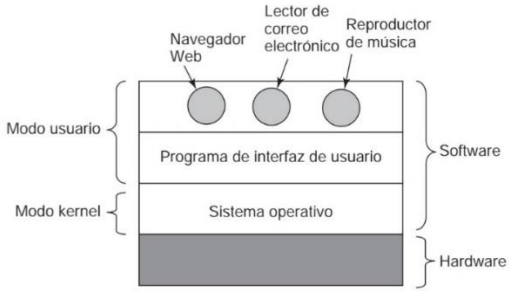


Figura 1-1. Ubicación del sistema operativo.

En la siguiente capa está el **software de base** (nivel abstracción aún bajo), encargado de manejar directamente las funcionalidades y gestión del hardware. Aquí es dónde está el sistema operativo. Esto es el **kernel** o núcleo del sistema operativo (que es el software que tiene la funcionalidad del manejo de las capas más bajas de los niveles de abstracción), un conjunto de funciones que sirven para manejar las llamadas al sistema, interrupciones, dispositivos entrada salida o memoria (es decir, resuelve varios problemas).

Cuando se arranca la máquina, una de las primeras cosas que se hace es coger ese software (código) del disco duro y se meterlo en la memoria principal, ya que todos los programas lo usan y no se borra hasta que se apaga.

La siguiente capa pertenece al **entorno de ventanas** (GUI) y **aplicaciones** por encima del sistema operativo. A partir de aquí el software es de **medio nivel**. Ayuda a una funcionalidad del sistema operativo, ofreciendo comodidad y sencillez al usuario.

Al **último nivel** (el más alto) pertenecen las aplicaciones (juegos, browsers...).

Existen dos tipos de **comparticiones de los recursos** a tener en cuenta:

- **Compartición de tiempo:** en un sistema con una única CPU, esta tiene que repartirse entre todos los programas y cada uno de ellos tiene asignados unos periodos de tiempo en los que puede estarse ejecutando y cuando esto termina se pasa a ejecutar otro (quantums).
- **Compartición espacial:** trozos de recurso simultáneamente. Dentro de la memoria del computador tiene que estar todo lo que necesiten todos los programas, es decir, su codigo y datos de cada uno. De esta forma cada uno de ellos tiene un trozo propio en la memoria.

1.3. Modos de ejecución de un programa

En la RAM, o **memoria principal**, se meten datos e instrucciones mezclados. Todo el código del sistema operativo se mete en parte de la memoria principal y de ahí no se mueve hasta que se apaga el sistema, de forma que a esa zona de la memoria principal se la llama **kernel** o núcleo del sistema.

La zona que queda libre en la memoria principal está disponible para los datos y programas propios del usuario. Cuando se está ejecutando alguna instrucción de la zona que no es el sistema operativo, se ejecuta lo que se llama **modo de usuario**. Y cada vez que se hace una llamada al sistema, solo se hace una instrucción de salto de esa zona de usuario a una zona del sistema operativo, que es la zona de kernel y su ejecución es en **modo núcleo**, porque tienen ciertos privilegios como el acceso a zonas de memoria o dispositivos que el usuario no tiene.

Cuando un usuario quiere hacer algo en una de esas partes restringidas no tiene más remedio que pedirle al sistema operativo que lo haga a través de un **syscall** o un **trap** (llamada al sistema, salto a una instrucción que está en modo kernel, el usuario obtiene los servicios del SO).

Las **interrupciones** son avisos que llegan del exterior a la CPU para que se detenga y pase al modo núcleo y resuelva la interrupción con salto al modo núcleo.

El **ciclo de ejecución** de las instrucciones son las diferentes fases por las que pasan durante su ejecución, empezando siempre leyendo el contador del programa, que apunta a la siguiente instrucción a ejecutar (fase de carga).

Tema 2

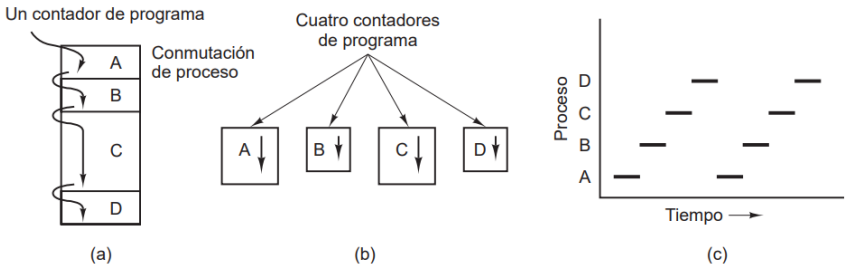
Procesos e Hilos

2.1. Concepto de Proceso

Un **proceso** es la abstracción de un programa en ejecución. Cada proceso tiene sus propias direcciones virtuales (que no son direcciones de la RAM), en las que tiene su código, datos, pila, registros y más, a los que solo él tiene acceso.
Es por eso por lo que cuando se lanza a ejecutar un programa ejecutable, este pasa a ser un proceso. Son difíciles de manejar porque en un sistema suele haber muchos procesos vivos sin importar el número de CPUs que tenga, ya que todos se pelean por ejecutarse.

Cuando solo hay una CPU, no hay ejecución concurrente debido a que solo puede haber un proceso ejecutándose a la vez. Puede parecer que es paralelo ya que el usuario no es consciente de que cada muy pocos milisegundos se ejecutan varios trozos de diferentes procesos, de forma que todos están avanzando, por poco que sea, pero no de forma paralela. A esto se le llama **pseudoparalelismo**.

A lo largo del tiempo se van ejecutando partes de los procesos (a) de forma secuencial (uno detrás de otro). En el caso de que un proceso no haya terminado su ejecución antes de haber perdido la CPU, cuando sea posible la volverá a reanudar(c) en el mismo punto en el que estaba antes del **cambio de contexto**. El usuario va a tener la sensación de que están siendo ejecutados en paralelo (b) por lo que los cambios de contexto se hacen de forma transparente al usuario.



Cuando hay un cambio de contexto, el sistema operativo tiene que almacenar el contenido de todos los registros en memoria del proceso que tenía la CPU, entre otras cosas. De esta forma hay varios contadores de programa (PC) lógicos en la memoria, aunque solo haya un registro físico. De esta forma cuando el proceso tiene la oportunidad de continuar su ejecución, esta se retoma en el punto exacto en el que se quedó cuando perdió la CPU, además de recuperar todos los valores de los registros. Otra cosa que tiene que hacer el sistema operativo es planificar cuál va a ser el siguiente proceso en obtener la CPU y recuperar de memoria todo lo necesario para su ejecución.

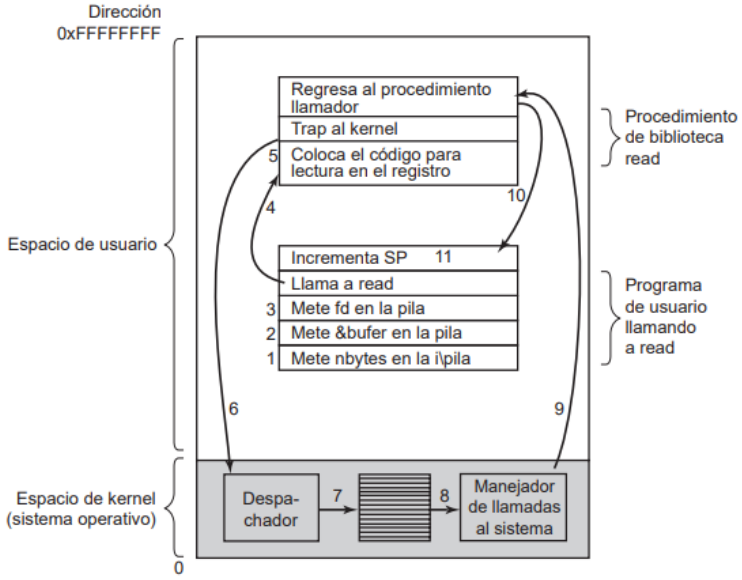
Un **programa** es un fichero que contiene las instrucciones y estructuras necesarias para la ejecución, que es un ejecutable (que está en código máquina). Un **proceso** es ese programa en ejecución con toda la información del fichero cargada en el espacio virtual que se creó para él.

En la zona de kernel de la memoria hay una tabla llamada **tabla de procesos**. Esta almacena toda la información sobre todos los procesos que están vivos en el sistema: su estado, identificador, archivos, donde están sus datos en memoria (mientras no tiene la CPU), … Puede que en la

entrada de la tabla no estén todos los datos sobre el proceso, pero si un puntero al sitio donde está toda su información. Un ejemplo es el siguiente:

Administración de procesos	**Administración de memoria**	**Administración de archivos**
Registros	Apuntador a la información del segmento de texto	Directorio raíz
Contador del programa	Apuntador a la información del segmento de datos	Directorio de trabajo
Palabra de estado del programa	Apuntador a la información del segmento de pila	Descripciones de archivos
Apuntador de la pila		ID de usuario
Estado del proceso		ID de grupo
Prioridad		
Parámetros de planificación		
ID del proceso		
Proceso padre		
Grupo de procesos		
Señales		
Tiempo de inicio del proceso		
Tiempo utilizado de la CPU		
Tiempo de la CPU utilizado por el hijo		
Hora de la siguiente alarma		

En la imagen siguiente se muestra un ejemplo de los pasos que realiza un proceso en su ejecución:



Cada una de las cajas representa cada una de las instrucciones del proceso, leyéndose estas de abajo a arriba. En el **stack** se van almacenando los datos que va usando el proceso y cuando se hace un salto en la rutina se debe guardar también la dirección de retorno para cuando se termine la subrutina (no se muestra en la foto).

A continuación, se mete en un registro el código, que en realidad es un número (OJO con ser número y no código en sí), de la función que se ha llamado para después hacer la llamada al sistema o trap. Esa llamada implica pasar a la **zona de núcleo** o **kernel**.

Una vez allí, primero se pasa por el **despachador** o *handler* del kernel, encargado de ver que código (recordemos que es un número) hay en el registro. Usando esto se accede a una línea de una tabla donde están los punteros a las funciones. A través del puntero se va al código concreto (este sí que es código como tal) seleccionado. Al terminar de ejecutarse el código vuelve al modo usuario y se hacen los retornos necesarios.

A pesar de que los procesos son entidades independientes y no comparten información entre sí, se pueden establecer interacciones entre ellos, bien a través de señales, mensajes o utilizando memoria compartida declarada expresamente. Normalmente se utilizan ficheros en el disco duro a los que los procesos pueden acceder, de forma que actualizan información en el fichero y avisan a los procesos interesados del cambio para que lo lean.

2.2. Gestión de Procesos

Muchos de los procesos se crean en el arranque de sistema y la mayoría de ellos se ejecutan en segundo plano (no tienen salida visible para el usuario). Entre ellos están los **demonios**, que son procesos de kernel con permisos que los de usuario no tienen.
Otras formas de crear procesos son creándolos desde otro proceso o siendo creados por el usuario (este podría ser considerado un caso particular del primero).

Algunas de las funciones más comunes a la hora de manejar procesos son:

- pid = fork( ): Crea un proceso hijo idéntico al padre
- pid = waitpid(pid, &statloc, opts): Espera a que un hijo termine
- s = execve(name, argv, envp): Reemplaza la imagen del núcleo de un proceso
- exit(status): Termina la ejecución del proceso y devuelve el estado
- s = sigaction(sig, &act, &oldact): Define la acción a realizar en las señales
- s = sigreturn(&context): Regresa de una señal
- s = sigprocmask(how, &set, &old): Examina o cambia la máscara de la señal
- s = sigpending(set): Obtiene el conjunto de señales bloqueadas
- s = sigsuspend(sigmask): Reemplaza la máscara de la señal y suspende el proceso
- s = kill(pid, sig): Envía una señal a un proceso
- residual = alarm(seconds): Establece el reloj de alarma
- s = pause( ): Suspende el proceso que hizo la llamada hasta la siguiente señal

---

2.2.1. Creación de un proceso

La llamada al sistema **fork** se usa como una función para crear un proceso que es una copia exacta (mismas posiciones relativas, mismos datos e incluso mismo estado) del proceso padre (proceso que ejecutó el fork).

Desde que se ejecuta el fork, ambos procesos tienen las mismas variables, pero son valores independientes, lo que implica que, si uno de los procesos modifica la variable, el otro proceso no ve el cambio en el dato porque son direcciones virtuales (es decir, son iguales, pero no son lo mismo). El proceso padre no puede leer nada de ningún otro proceso, sin importar si es su hijo o no, solo puede acceder a sus propias direcciones.

La única diferencia entre el proceso padre y el hijo es que para el primero el fork devuelve el **PID** (identificador de proceso) del hijo y para el segundo devuelve un 0. Esa diferencia es necesaria para saber separar los caminos y las funciones de cada proceso. Es por eso por lo que se hace:

```
pid = fork()
  if (pid == 0) {/*código del hijo*/}
  else {/*código del padre*/}
```

Ejemplo del uso de procesos y de la no compartición de datos:

```
main() {
  int par, x=0;
  if ((par=fork()) == -1){
     printf("Error en la ejecución del fork");
     exit(0);
  } else if (par==0) { // proceso hijo
     x=x+2;
     printf("\nProceso hijo, x= %d\n", x);
  } else // proceso padre
      printf("\nProceso padre, x= %d\n", x);
      printf("Finalizar\n");
}
```

Cuando se crea al proceso hijo, ambos procesos tienen en una de sus direcciones la variable x con el valor 0. Cuando el hijo cambia el valor de x a 2, solo él ve el cambio, mientras que el padre sigue con x = 0. A la hora de imprimir los resultados, el orden es impredecible porque ambos pelean por el Shell (donde se imprime) y es el planificador quien decide quien imprime antes. Fuera del if hay un print que ambos van a imprimir y no se sabrá quien escribió cada uno.

Puede darse el caso de que salga primero la frase del hijo, la frase del padre, uno de los mensajes de finalizar, el prompt (la parte del Shell donde se escribe un nuevo comando) y después el otro finalizar. Esto se debe a que el Shell solo espera a que el padre termine (el proceso principal) y no tiene en cuenta si el hijo terminó también o no.

En estos casos, se dice que el proceso hijo se ha quedado **huérfano** y presenta un problema ya que no tiene padre, condición que es obligatoria para todos menos para el **init**. Cuando esto ocurre, el proceso hijo es "adoptado" por otro proceso que normalmente suele ser el init (el proceso que se crea durante el arranque del sistema y que es el proceso padre principal o proceso raíz). También puede ser que init cree un nuevo proceso que haga de padrastro en lugar de ser adoptados por el propio init.

```
int fichero1, fichero2;
char caracter;

main(int argc, char* argv[]) {
   if (argc !=3) exit(1)
   fichero1=open(argv[1], 0444); //Lectura
   fichero2=open(argv[2], 0666); //Escritura
   if (fichero1==-1) printf("\nError al abrir el fichero 1\n");
   if (fichero2==-1) printf("\nError al abrir el fichero 2\n");
   fork();
   leer_escribir();
   exit(0);
}

int leer_escribir() {
   for (;;) {
      if(read(fichero1,&caracter,1)!=1)
         return(0);
      write(fichero2,&caracter,1);
    }
}
```

El padre abre ambos ficheros, uno para leer y otro para escribir. Al hacer el fork, ambos procesos tienen los dos ficheros. Ambos tienen el mismo código a partir del fork y ejecutan la función. Con el lazo infinito se leen los caracteres del fichero y se van copiando en el otro.

Los ficheros tienen un puntero que marca el carácter al que se están apuntando, solo hay un puntero por archivo, no uno por proceso. Los dos procesos quieren hacer lo mismo, ambos leen y

escriben, pero como el orden es impredecible se puede dar el siguiente caso. El padre lee un carácter y almacena el dato e incrementa el puntero del archivo, pero justo ahí pierde la CPU. El hijo va a leer el siguiente carácter, y escribe en el segundo fichero el carácter. De esta forma el segundo archivo no será igual al primero. Se dice que se ha producido una **carrera crítica** entre ambos procesos (materia de Sistemas Operativos II).

2.2.2. Terminación de un proceso

Un proceso puede finalizar de las siguientes maneras:

- **Salida normal**: es una terminación voluntaria. Lo suele hacer cuando termina su trabajo y en la última línea que ejecuta se utiliza la llamada al sistema **exit(entero)**. El entero que se emplea sirve para ver cómo terminó su ejecución desde el proceso padre, que lo puede estar esperando (es el &status que se utiliza en waitpid). Si no se coloca esta llamada al sistema en el código se ejecuta por defecto, que es lo que hace el main de forma implícita normalmente.
- **Salida por error**: es una salida voluntaria, que puede darse por un uso erróneo del programa, como esperar un valor del usuario y que introduzca uno incorrecto.
- **Salida por error fatal**: es involuntaria, de forma inesperada ocurre una excepción que no permite continuar con la ejecución como una división entre 0 o un acceso a una dirección no válida. Son errores del propio programa.
- **Eliminado por otro proceso**: otro proceso le envía una señal que lo obliga a terminar (lo mata) sin importar que le quede trabajo aún por hacer. Esto se puede hacer con la llamada al sistema de kill.

La forma más recomendable para terminar un proceso es invocando a la llamada al sistema **exit** que tiene como argumento un entero que puede ser tomado por el padre para controlar si todo a salido según lo esperado. Cuando se ejecuta un exit el proceso se desconecta del árbol de procesos y no produce ninguna salida, pero sigue presente en la tabla de procesos (explicado en el punto 2.2.3).

2.2.3. Espera a la terminación de un proceso

El resultado de la finalización de un proceso puede ser recogido por el proceso padre, con wait o waitpid, por ejemplo.

Cuando un proceso ejecuta la llamada al sistema **wait** lo que hace es esperar a que cualquiera de sus hijos termine (solo hijos directos, no descendientes posteriores) y con que finalice uno de ellos esta llamada ya retorna y se puede leer el valor que proporcionó el proceso al terminar (valor del exit). En el caso de ejecutar un **waitpid** se espera a un hijo en concreto, quedándose el padre bloqueado hasta que el hijo finalice. Si se pone un -1 en el argumento pid, funciona como un wait porque espera a cualquiera de sus hijos.

Un proceso **zombi** es un proceso que ya ha terminado su tarea, es decir, que está muerto, pero no ha sido eliminado del sistema, es decir, sigue en la tabla de procesos (situada en el kernel). Esto se debe a que los procesos solo se borran de la tabla cuando su padre finaliza o hasta que su padre ejecute un wait o un waitpid asociado a él. Mientras tanto, se mantiene en la tabla por si el padre quiere algo de él, decirle que ha muerto. Todos los procesos al acabar pasan a ser zombies hasta que se realice una de las dos acciones que provocan su eliminación de la tabla de procesos.

Es posible que un padre cree a un hijo y finalice antes el padre que el hijo, creando así un proceso **huérfano** (explicado en el punto 2.2.1). Cuando un huérfano finaliza, **no** se convierte en zombi porque ya no tiene un padre que esté esperando por él, a pesar de haber sido adoptado, por lo que una vez terminado se elimina directamente de la tabla de procesos.

En el código siguiente se usa el fork de forma que el proceso padre crea un hijo que tiene unas tareas propias. El hijo espera 4 segundos antes de ejecutar su exit con el 3 como valor de salida. El padre lo único que hace es esperar al hijo. No es posible saber si el padre ejecuta antes el wait o el proceso hijo imprime por pantalla, pero lo que es seguro es que se ejecutan con muy poca diferencia de tiempo, pareciendo paralelas para el humano. El padre se queda esperando a que el hijo finalice por lo que el mensaje del padre se va a ver con una diferencia de 4 segundos desde que se imprimió el mensaje del hijo.

```
main() {
   int pid, estado;
   if (fork()==0){
      printf("\nMensaje 1\n");
      sleep(4);
      exit(3);
   } else {
      pid = wait(&estado);
      printf("\nFinalizar\n");
   }
}
```

2.2.4. Cambiar la imagen de un proceso

En el siguiente ejemplo se vuelve a mostrar el código de un proceso padre y su hijo (representando el funcionamiento del Shell), para hablar de las llamadas al sistema de **cambio de imagen** de los procesos, como es el caso de las de la familia **execv**.

Dado un proceso, el que las invoca, borran toda su memoria menos el kernel y lo cambian por otro ejecutable, ejecutando este nuevo código desde el principio. En el caso particular de abajo, el hijo hace un cambio de imagen utilizando la función **execve** donde el primer parámetro es el fichero ejecutable, dirección donde está el código que se va a colocar ahora en el proceso. Cuando termine, se ejecuta el exit correspondiente haciendo que el padre pueda continuar.

Esto implica que el padre solo espera a su hijo, así que en el caso de que el hijo con su nueva imagen cree nuevos procesos, el padre no será consciente de ello (esperando solo por su hijo). Es posible que alguno de los descendientes del hijo aún no haya terminado cuando el hijo si termina, implicando también la finalización del padre y haciendo que se mezclen elementos del padre y de alguno de los descendientes.

```
#define TRUE 1
while (TRUE) { /* se repite en forma indefinida */
   type_prompt(); /* muestra el indicador de comando en la pantalla */
   read_command(command, parameters); // lee la entrada de la terminal
   if (fork() !=0) { /* usa fork para el proceso hijo */
      /* Codigo del padre. */
       waitpid(-1, &status, 0); /* espera a que el hijo termine */
   } else { /* Codigo del hijo. */
         execve(command, parameters, 0); /*ejecuta el comando de cambio de imagen */
    }
}
```

2.3. Señales

Las **señales** son avisos que reciben los procesos acerca de eventos que pueden ocurrir en el sistema, y se suelen utilizar como mecanismo de comunicación y/o sincronización entre procesos. Estas están asociadas a unos pocos números enteros de forma que, cuando un proceso recibe una de ellas, la identifica para determinar cuál es la causa del aviso. Son similares a las interrupciones, pero con la diferencia de que las interrupciones son avisos que se realizan al S.O. mientras que las señales están dirigidas expresamente a un proceso determinado.

Una señal puede ser generada por: otros procesos (kill), sucesos en el hardware, interrupciones del terminal (CTRL+C), notificaciones de E/S, alarmas, etc.

Cuando se recibe una señal se debe identificar y cada una de ellas tiene asociada un comportamiento por defecto. Esta acción se realiza en el proceso receptor de la señal formando parte de su espacio virtual.

```
#include <signal.h>
main(){
   int a ;
   a = fork();
              if (a == 0) {
      while (1) {
         printf("pid del proceso hijo = %d \n", getpid());
         sleep(1);
      }
    }
    sleep(10);
     printf("Terminación del proceso con pid = %d\n, a);
    kill(a, SIGTERM);
  }
```

En este ejemplo hay dos procesos. El hijo tiene un bucle infinito en el que imprime y después se bloquea durante un segundo. El padre se bloquea 10 segundos, imprime y envía al hijo una señal utilizando el pid del hijo para marcar el destinatario en la función kill y la señal que desea enviarle (puede ir como constante de C o como número entero).

El uso de kill también se puede hacer desde la consola solo que su formato pasa a ser **kill -Señal pid.**

Es posible que se desee cambiar el comportamiento del proceso cuando recibe una señal, es por ello por lo que se utiliza la función **signal**. Esta función tiene 2 argumentos: la señal de la que se varía el comportamiento y un puntero a la función que se ejecutará cuando se reciba la señal (**signal(señal, puntero a la función manejadora);**) sin importar quién le envía la señal.

En el caso de emplear signal y después querer reestablecer la acción por defecto solo hay que ejecutar signal con SIG_DFL en el campo de la función manejadora. Lo mismo con SIG_IGN si se desea ignorar ese tipo de señal.

También se puede emplear la llamada al sistema **sigaction**, que da mayor flexibilidad ya que además de recibir los mismos parámetros que signal, también permite especificar otros parámetros y/o banderas para la gestión de la señal. Un ejemplo de uso de esta llamada al sistema es el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
main(){
   int codigo_error = 0;
   struct sigaction gestion;
   gestion.sa_handler = gestor; /* Manejador de la señal */
```

```
    gestion.sa_mask = 0; /* Máscara para tener en cuenta la señal o no */
    gestion.sa_flags = SA_ONESHOT; /* Bandera, solo resuelve la primera señal con el
                                      manejador y el resto de veces que la reciba
                                      actuará por defecto */
   codigo_error = sigaction (SIGINT, gestion, 0);
   if (codigo_error == SIG_ERR) {
      perror ("Error al definir el gestor de SIGINT");
      exit(-1);
   }
   /*Código del programa*/
   while(1);
}

void gestor (int señal){
   /*Código de manejo de la señal*/
}
```

Algunas de las señales más comunes son:

- SIGABRT: Se envía para abortar un proceso y forzar un vaciado de núcleo
- **SIGALRM**: Se activó el reloj de alarma
- SIGFPE: Ocurrió un error de punto flotante (por ejemplo, división entre 0)
- SIGHUP: La línea telefónica que utilizaba el proceso se colgó
- SIGILL: El usuario oprimió SUPR para interrumpir el proceso
- SIGQUIT: El usuario oprimió la tecla que solicita un vaciado de núcleo
- **SIGKILL**: Se envió para eliminar un proceso (no se puede atrapar ni ignorar)
- SIGPIPE: El proceso escribió en una tubería que no tiene lectores
- SIGSEGV: El proceso hizo referencia a una dirección de memoria inválida
- **SIGTERM**: Se utiliza para solicitar que un proceso termine de forma correcta
- **SIGUSR1**: Disponible para propósitos definidos por la aplicación
- **SIGUSR2**: Disponible para propósitos definidos por la aplicación
- **SIGINT:** Se pulsó CTRL + C y por defecto se interrumpe el programa
- **SIGCHLD:** Señal que envía un proceso hijo al padre cuando finaliza

2.4. Implementación de Procesos

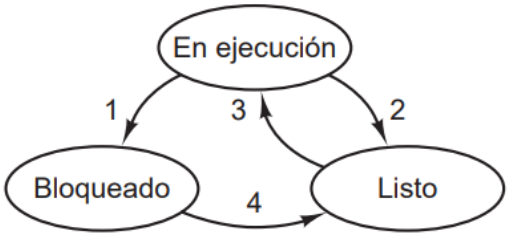
Un **grupo de procesos** es un proceso con sus hijos y sus respectivos descendientes. Es importante hablar de grupos porque en ocasiones es posible enviar señales al grupo en su totalidad, aunque las señales son tratadas de forma individual por cada uno de los procesos.

El proceso **init** es especial porque es el único que no tiene padre en el sistema y el resto de los procesos son descendientes de él. El proceso Shell se crea cada vez que se abre una terminal (esto puede variar según el S.O., pudiendo crearse un proceso terminal entre el proceso init y el proceso Shell) y este crea más procesos según los comandos que se ejecuten. En Linux, los procesos siempre se pueden representar en forma de árbol debido a la jerarquía entre procesos, aunque en Windows no es así.

Cada proceso tiene un estado en el cual se encuentra, que pueden estar en uno de los siguientes estados, representado con 2 bits en cada proceso:

- **En ejecución:** el proceso está haciendo uso de la CPU en este momento. Solo puede haber tantos procesos en este estado como CPUs tenga el sistema.

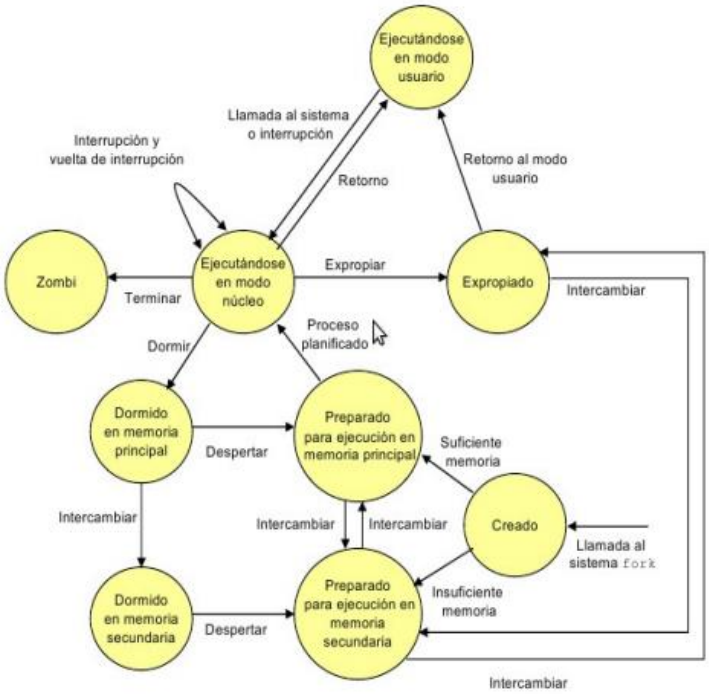
- **Bloqueado:** el proceso no puede avanzar porque puede estar esperando alguna condición. No es candidato para utilizar la CPU, por lo que el planificador no puede elegirlo.
- **Listo/Preparado:** el proceso no está haciendo uso de la CPU por lo que no está avanzando, pero el planificador puede ponerlo a ejecutar en cualquier momento.



Un proceso en ejecución puede pasar a estado bloqueado (1) cuando espera una condición que le impide seguir su ejecución (cambiando su valor de estado en la tabla de procesos). El S.O. lo que hace ahora es ir a la tabla de procesos para trabajar con uno de los procesos que está listo (3), que ejecuta el que decide el planificador. Nunca va a ocurrir que un proceso bloqueado pase a ejecutarse, ya que debe pasar a estado listo (4) para que el planificador lo elija para ejecutarse, ni que un proceso listo pase a bloqueado.

Un proceso puede pasar de estar ejecutándose a estar en estado listo, esto se debe a que el S.O. puede quitarle la CPU para dársela a otro proceso (2).

En Unix el diagrama de los estados en los que se puede encontrar un proceso es mucho más complejo que el anterior y se ve de la siguiente manera:



No todos los procesos son iguales: unos pueden ser del **usuario** y otros del S.O. (demonios). Además, los procesos tienen una **prioridad** asignada en la tabla de procesos, obteniendo mayor

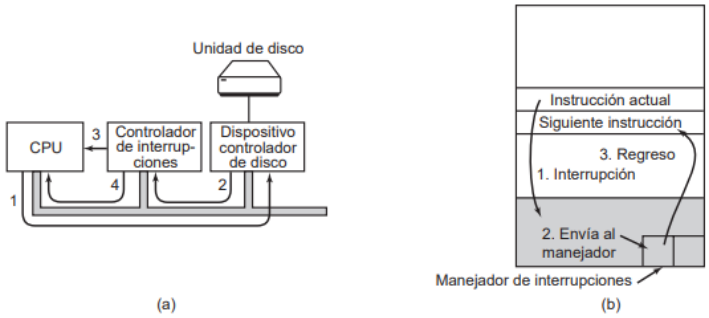
preferencia sobre otros cuando el planificador selecciona que proceso va a ir antes. Solo se le puede variar la prioridad a los procesos de usuario. La prioridad suele disminuir cuando mayor sea su uso de la CPU.

Cuando un proceso ejecuta un *fscanf* pasa de estarse ejecutando a estado bloqueado porque hay que ir al disco, bloqueando así el proceso. Cuando finaliza la operación de acceso al disco se lanza una interrupción que provoca que el proceso que tenía ahora la CPU (estaba en estado de ejecución) pase a estado listo. El manejador de la interrupción mira a qué proceso afecta la interrupción y lo desbloquea para pasarlo a estado listo. Una vez completado esto, el sistema ejecuta el planificador para ver a qué proceso se le concede la CPU disponible. Al menos hay dos candidatos listos (el que ejecutó el *fscanf* y era receptor de la interrupción y el proceso que al que se le quitó la CPU cuando llegó la interrupción). El planificador debe elegir entre ellos.

2.4.1. Interrupciones

(Esto tiene que ver más con el tema 5 pero él lo metió aquí)

Esta parte solo se centra en la captura y resolución de las interrupciones (b), obviando como se gestionan globalmente (a). Las interrupciones normalmente vienen lanzadas desde las controladoras de los dispositivos de E/S aunque también están las de reloj (quantums).



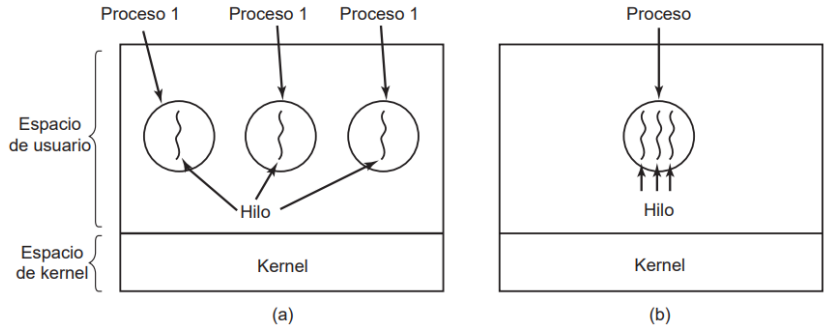
Cuando se recibe una interrupción habrá un proceso en ejecución. Cuando esta llega, hay que saltar a una rutina de la zona de kernel, por lo que hay que guardar en el stack la dirección de retorno (que es la de la siguiente instrucción del proceso). Después se salta a una rutina del sistema operativo llamada **manejadora de interrupciones** que detecta cual es la causa de la interrupción y salta a la rutina concreta para resolver ese tipo de interrupción. Esta es tomada de una lista de direcciones de memoria, llamada **vector de interrupciones**.

Una vez identificada la rutina, se ejecuta y se retorna a la instrucción del proceso que se almacenó al inicio. Normalmente estas interrupciones van dirigidas a algún proceso en concreto que estaba esperando, (en estado bloqueado), y la interrupción funciona como aviso para cambiarle el estado a listo. Puede ser necesario volver a lanzar el planificador.

2.5. Hilos

Un **hilo** es una secuencia de instrucciones que se ejecutan en un proceso (a). Por defecto un proceso tiene un único hilo y este solo puede acceder a los mismos datos a los que puede acceder el proceso. Un proceso puede tener varias partes de su código ejecutándose al mismo tiempo empleando varios hilos (b).

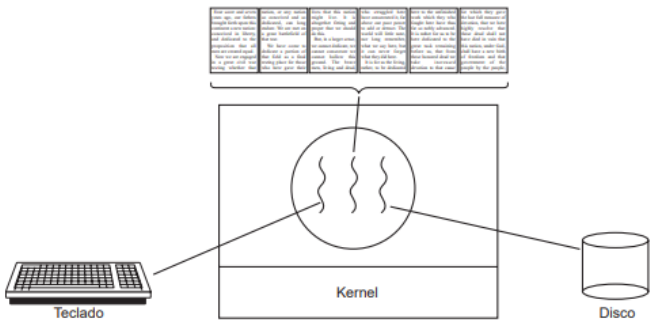
(El dibujo (a) no es el más correcto porque da a entender que los 3 procesos tienen la misma zona de memoria. Debería haber una caja por proceso).



Cuando hay varios hilos dentro de un mismo proceso, todos los hilos tienen el mismo espacio de direcciones. Cuando hay una única CPU pasa lo mismo que con los procesos, no se pueden ejecutar dos al mismo tiempo (pseudoparalelo), aunque pueda parecerlo para el humano. En el caso de haber varias CPUs se puede ejecutar cada uno de los hilos al mismo tiempo en una CPU diferente. A diferencia de los procesos, los hilos cooperan y no compiten entre sí.

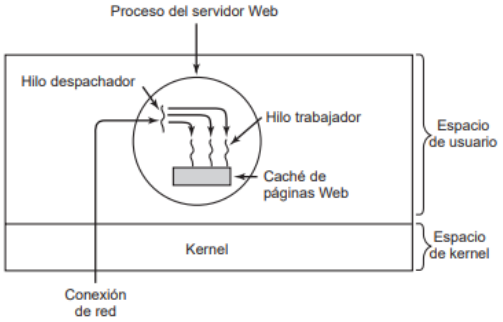
Su uso se destaca porque **comparten el espacio de direcciones** (a diferencia de los procesos), permitiendo una ejecución mucho más rápida, ya que a la hora de hacer cambios de contexto entre hilos de un mismo proceso no es necesario copiar toda la información del que se estaba ejecutando y copiar la que se tiene del que se va a ejecutar ahora. También es más rápido crear y destruir los hilos.

Es útil utilizar hilos cuando un programa tiene que realizar diferentes actividades simultáneas que colaboran entre sí, ya que se pueden colocar los datos a compartir en la memoria del proceso a la que todos tienen acceso. Esta opción es mucho mejor que utilizar las opciones de colaboración entre procesos (señales, archivos compartidos o bloques de memoria compartida).



Este ejemplo representa un procesador de texto, con un solo proceso con 3 hilos ejecutándose dentro del espacio de memoria. Uno de ellos es el encargado de controlar que escribe el usuario, otro está formateando y el último hace copias en el disco de vez en cuando. El documento es común a todos ya que está en la memoria principal y así todos pueden modificarlo sin problemas.

Tanto los hilos correspondientes a la atención del teclado como a la creación de las copias de seguridad están haciendo operaciones de E/S mientras que el hilo encargado del formateo está utilizando la CPU, de forma que se están aprovechando todos los recursos del sistema de forma eficiente.



Otro ejemplo puede ser un servidor web conectado a la red, que se encarga de atender peticiones. En el proceso hay varios hilos encargados de buscar la página solicitada por la petición. Hay varios hilos porque el proceso puede atender varias solicitudes al mismo tiempo y resolverlas por separado a pesar de ser el mismo proceso.

Los hilos, a pesar de que comparten la memoria del proceso al que pertenecen, tienen elementos propios de cada uno. Todos aquellos que tienen que ver con la ejecución de cada uno de los hilos son propios de cada uno, como los elementos de la siguiente tabla:

ELEMENTOS POR PROCESO	ELEMENTOS POR HILO
- **Espacio de direcciones** - **Variables globales** - **Archivos abiertos** - **Procesos hijos** - Alarmas pendientes - Señales y manejadores de señales - Información contable	- Contador de programa - **Registros** - **Pila** - **Estado**

En la zona de memoria puede haber datos del proceso, que pueden ser vistos por todos los hilos de este, y también datos que solo son accesibles por el hilo que los posee. Sin embargo, si otro hilo del mismo proceso conoce la dirección de la variable, pueden acceder a ella debido a que están en el mismo espacio de direcciones.

El **hilo principal** es el main y cuando se llama a una función este queda esperando, pero si se crea un hilo desde el main y se llama a la función desde este, el main sigue avanzando al mismo tiempo que se ejecuta la función.

Los hilos se pueden ver como varias ejecuciones en el entorno del proceso. Los procesos comparten, aunque más bien compiten por, la memoria física y otros recursos, mientras que los hilos comparten el espacio de direcciones. Cuando un hilo trae una línea del disco y la coloca en la memoria principal, todos los demás hilos del mismo proceso también tienen acceso a ella, pero ningún otro proceso.

2.5.1. Funciones para el uso de los hilos

Para crear los hilos se utiliza la llamada al sistema:

pthread_create(&idHilo, NULL, funcion, &variable).

El primer argumento es un puntero a una variable de tipo **pthread_t**, donde la llamada almacenará el identificador del hilo que se crea. El segundo argumento es un puntero a una estructura de los atributos del hilo, aunque se suelen utilizar las características por defecto y para ello se coloca un NULL en ese argumento en la llamada. Después se coloca la función que tiene que ejecutar ese hilo, que debe estar en el mismo ejecutable. Por último, se manda un puntero a una variable que va a ser leída por la función ejecutada por el hilo. Es obligatorio que las funciones que se usan para los códigos de los hilos tengan un único argumento: un puntero. En este caso se manda el valor del puntero del valor de la variable i.

Una vez creado el hilo, este ejecuta desde el inicio su función mientras que el hilo que lo creó también sigue avanzando, ambos de forma simultánea.

Para finalizar los hilos, es necesario ejecutar la llamada al sistema **pthread_exit( )**, similar a la llamada a exit( ) de los procesos. Si alguno de los hilos ejecuta un exit( ), el proceso se termina, incluyendo los hilos activos, sin importar su estado ni si han terminado sus ejecuciones.

Con la llamada a **pthread_join( )** se provoca que el hilo que la ejecuta deba esperar hasta que el proceso indicado (que debe pertenecer al mismo proceso), ejecute su pthread_exit( ).

En la llamada de **pthread_yield( )** el hilo que la ejecuta cede la CPU voluntariamente a otros que la necesiten. Esta se suele usar cuando un hilo se va a bloquear y mientras va a estar bloqueado deja que otros utilicen la CPU.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMERO_DE_HILOS 10
/* Esta funcion imprime el identificador del hilo y después termina. */
void *imprimir_hola_mundo(void *tid) {
   printf("Hola mundo. Saludos del hilo %d\n", tid);
   pthread_exit(NULL);
}
/* El programa principal crea 10 hilos y después termina. */
int main(int argc, char *argv[]) {
   pthread_t hilos[NUMERO_DE_HILOS];
   int estado, i;
   for(i=0; i < NUMERO_DE_HILOS; i++) {
      printf("Aqui main. Creando hilo %d\n", i);
      estado = pthread_create(&hilos[i], NULL, imprimir_hola_mundo, (void *)i);
      if (estado != 0) {
         printf("Ups. pthread_create devolvió el codigo de error %d\n", estado);
         exit(-1);
      }
   }
   exit(NULL);
}
```

En este código se crean 10 hilos que ejecutan la misma función. Al ejecutarlo (ignorando los warnings que se producen) se puede observar que los índices (i) se hacen un lío cuando se imprimen. Esto se debe a que mientras se ejecutan los hilos, el hilo principal (el main) sigue ejecutándose, lo que implica que el valor de i sigue aumentando. Entonces cuando el hilo lee el valor del puntero que se le pasó como argumento, es muy probable que el valor del índice ya haya cambiado desde que se hizo la llamada a pthread_create.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMERO_DE_HILOS 4
/* Esta funcion suma la parte correspondiente a cada hilo del vector*/
void *func(void *i) {
   if(i == 0)
```

```
      //Suma el primer cuarto del vector
   if(i == 2)
      //Suma el segundo cuarto del vector
   (…)
   pthread_exit(NULL);
}

/* El programa principal crea 10 hilos y después termina. */
int main(int argc, char *argv[]) {
   pthread_t hilos[NUMERO_DE_HILOS];
   int estado, i;
   for(i=0; i < NUMERO_DE_HILOS; i++) {
      estado = pthread_create(&hilos[i], NULL, func, (void *)i);
      if (estado != 0) {
         printf("Ups. pthread_create devolvió el codigo de error %d\n", estado);
         exit(-1);
      }
   }
   exit(NULL);
}
```

En este código se crean 4 hilos, para que cada uno de ellos ejecute la suma de una cuarta parte de los elementos de un vector. Cada uno suma su parte de forma simultánea y al final se suman todos los resultados, reduciendo considerablemente el tiempo de ejecución de la operación.
Para evitar el problema con i del ejemplo anterior, se hace un **casteo** de la variable **(void *)i**, que pasa el puntero a un entero y ese valor no va a verse afectado por los cambios en el main. Es una solución que provoca varios warnings en la compilación porque normalmente el tamaño de los punteros no es el mismo que el de los enteros, pudiendo provocar algún problema.

Otra posible solución es crear un array de índices y pasar el índice a la llamada que crea los hilos, quedando: `pthread_create(&hilos[i], NULL, func, *ind[i]);`.

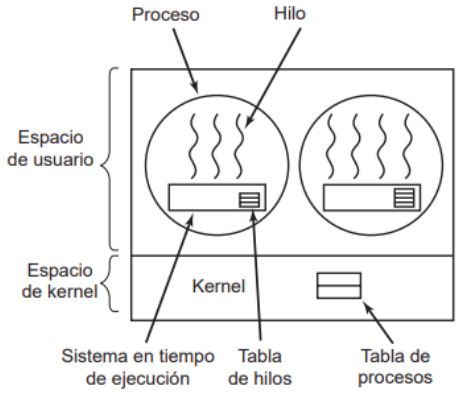
Llamadas al sistema de hilos:

- Pthread_**create**: crea un nuevo hilo
- Pthread_**exit**: termina el hilo invocador
- Pthread_**join**: espera a que un hilo especifico termine
- Pthread_**yield**: libera la CPU para dejar que otro hilo se ejecute
- Pthread_**attr_init**: crea e inicializa la estructura de atributos de un hilo
- Pthread_**attr_destroy**: elimina la estructura de atributos de un hilo

---

2.5.2. Hilos en el espacio de usuario

Si se trabaja en el **espacio del usuario**, el kernel conoce sobre los procesos lo almacenado en la tabla de procesos, pero no tiene información acerca de los hilos, ya que la información sobre ellos se almacena en el espacio de usuario, lugar donde está todo el código que gestionan los hilos y una tabla con todos los hilos del proceso (llamada **tabla de hilos)** equivalente a la de procesos, pero que solo hace referencia a los hilos del proceso.



El código que hay en el espacio de usuario tiene unas funcionalidades muy parecidas a las que tiene el kernel con los procesos, como puede ser su propio planificador. De esta forma el planificador del kernel elige que proceso se va a ejecutar en la CPU, pero es el planificador de dicho proceso el encargado de elegir que hilo va a obtener la CPU utilizando la tabla de hilos de ese proceso.

Es importante tener en cuenta que cuando se produce un cambio de contexto, solo se borra el espacio de usuario, dejando el kernel intacto. De esta forma cuando se hace un cambio entre procesos, el kernel solo sabe a qué proceso le está dando la CPU, pero no qué hilo va a ser el que se ejecute.

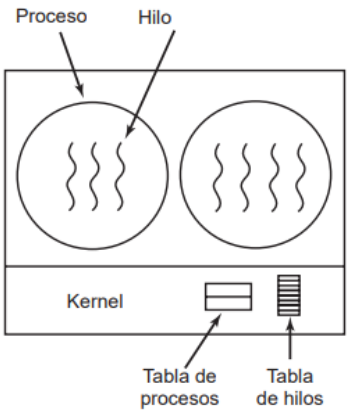
En este esquema, en el caso de que un hilo en ejecución se bloquee implica también el bloqueo de su proceso. Este bloqueo lo percibe el kernel, forzando un cambio de contexto y haciendo que los demás hilos del proceso también se bloqueen, aunque no tengan nada que ver con el bloqueo. Para evitar esto, antes de las llamadas bloqueantes como un *scanf*, se suele ejecutar un pthread_yield o se realizan ciertas comprobaciones para evitar afectar a un gran número de hilos del mismo proceso.

Al utilizar hilos en el espacio de usuario se pueden planificar estos de forma personalizada, además de que permite una **fácil escalabilidad**, ya que en memoria solo están los hilos del proceso actual.

2.5.3. Hilos en el espacio de kernel

En el kernel está la tabla de procesos y la tabla de hilos que contiene a todos los hilos de todos los procesos. Si se trabaja en el **espacio del kernel**, toda la gestión de los hilos puede ser realizada por este, pudiendo asignar quantums a hilos concretos y no a procesos. El planificador puede ir de cualquier hilo a cualquier hilo, cosa que no era posible cuando la tabla de hilos estaba en el espacio de usuario. La tabla de hilos se puede hacer muy grande, siendo su tamaño mínimo igual al tamaño de la tabla de procesos.

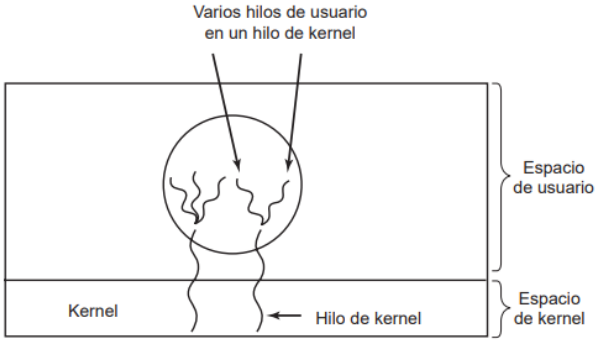
En este caso, el bloqueo de un hilo no implica el bloqueo de su proceso ni del resto de los hilos de ese proceso. Esta solución tiene un coste adicional asociado al salto a la zona de kernel (trap) haciendo así que la gestión de los hilos sea más lenta.



2.5.4. Implementaciones Híbridas

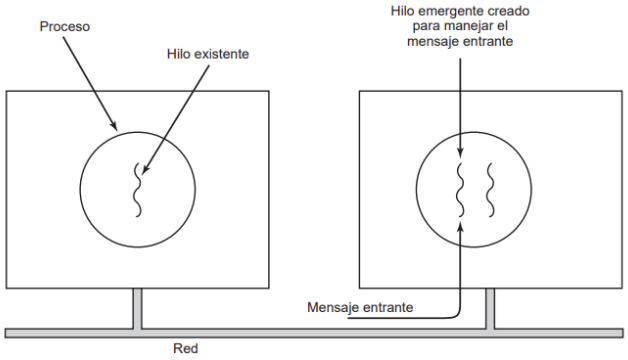
Una alternativa a las implementaciones anteriores es la **implementación híbrida**. Esta consiste en almacenar información de una serie de hilos en la zona de kernel, que pueden ser llamados **hilos de kernel**. De esta forma la tabla de hilos en el kernel tiene un tamaño limitado.

El coste de esta implementación está en que los hilos de usuario tienen que dividirse en grupos. Los pertenecientes a cada grupo se pelean por una entrada en la tabla de hilos del kernel. De esta forma se consigue el manejo de los hilos en el kernel con una tabla de hilos pequeña a cambio de la penalización por la pelea entre los hilos.



2.5.5. Hilos emergentes

Los hilos emergentes se crean dentro de un proceso cuando llega un mensaje a través de la red, proveniente de otro proceso que se está ejecutando en otra máquina. La función de este hilo es leer la información del mensaje y una vez hecho esto, se elimina.



2.5.6. Hilos en Linux

En Linux los hilos se gestionan en el espacio del kernel. La función **clone(funcion, pila_ptr, banderas_compart, arg)** sirve para crear procesos (tiene un fork interno) o hilos. Lo que crea depende de lo que se envié en el argumento de las banderas.

Se pueden crear procesos con cierto grado de compartición activando algunas banderas.

Bandera	Significado al estar activado	Significado al estar desactivado
CLONE_VM	Crea un nuevo hilo	Crea un proceso
CLONE_FS	Comparte umask, directorios raíz y de trabajo	No los comparte
CLONE_FILES	Comparte los descriptores de archivos	Copia los descriptores de archivos
CLONE_SIGHAND	Comparte la tabla de manejadores de señales	Copa la tabla
CLONE_PID	El nuevo hilo obtiene el PID anterior	El nuevo hilo obtiene su propio PID
CLONE_PARENT	El nuevo hilo tiene el mismo padre que el hilo que hizo la llamada	El padre del nuevo hilo es el que hizo la llamada

Según se usen las banderas se pueden crear procesos o hilos dentro del proceso, siendo el sistema el encargado de controlar las comparticiones.

2.6. Planificación de Procesos

La **planificación** consiste en seleccionar qué proceso es el que va a adquirir la CPU para continuar con su ejecución. Para ello existen diversos algoritmos para la toma de decisiones.

Para las planificaciones es importante saber qué tipo de sistema es. Si el sistema es un **PC**, pensado para un único usuario trabajando en él, tiene pocos procesos activos por lo que la planificación no es muy importante (normalmente solo hay un proceso principal y el resto son secundarios). En el caso de que el sistema sea un **servidor**, se esperan varios usuarios trabajando al mismo tiempo, lo que implica más procesos luchando por la CPU. En este caso la planificación es algo muy importante. El objetivo de la CPU en ambos casos es conseguir un uso eficiente.

El planificador solo trabaja con los procesos que están en estado listo y siempre habrá como mínimo uno en el sistema, que es el proceso idle.

2.6.1. Cambio de contexto

Un **cambio de contexto** o conmutación de procesos suceden cuando se le saca la CPU a un proceso para dársela a otro. Este cambio entre procesos requiere varias tareas:

1. Saltar al modo kernel
2. Tomar el estado del proceso (toda su información, incluidos todos sus registros) que se va a expropiar para que cuando retome el uso de la CPU se restaure en el mismo punto en el que la perdió. Todo esto se guarda en la tabla de procesos o en una posición de memoria cuya dirección se encuentra en la tabla de procesos. Este paso lleva mucho tiempo.
3. Seleccionar un nuevo proceso en función del algoritmo utilizado.
4. Se carga toda la información necesaria sobre el proceso que ha conseguido la CPU, tal cual estaba antes de haber perdido la CPU por última vez.

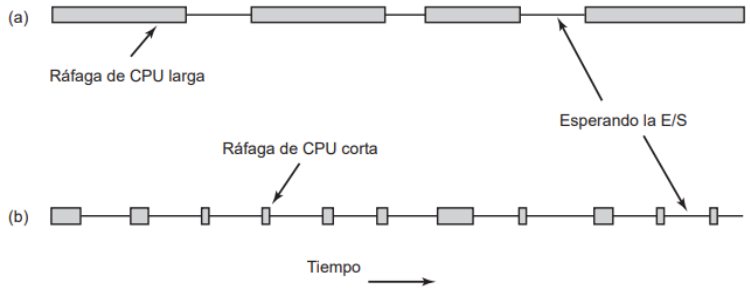
Con la conmutación de procesos es posible que desde que un proceso pierde la CPU hasta que la recupera, debido a los algoritmos de uso de la caché, toda la información que usaba el proceso

que estaba en la caché ya no este. Esto produce que cuando el proceso recupere la CPU sucedan muchos fallos caché.

Si hay demasiadas conmutaciones entre procesos, la CPU perderá demasiado tiempo realizando los cambios y los procesos no avanzarán mucho, desaprovechando la CPU.

2.6.2. Tipos de procesos y planificación

Normalmente suele haber procesos **limitados a cálculo** (a) que necesitan ráfagas de CPU largas y tienen muy pocas esperas debido a operaciones de E/S y procesos **limitados a E/S** (b) que usan muy poca CPU y tienen muchas esperas por E/S.



Por norma es recomendable darle antes la CPU a un proceso como (b) porque se sabe que va a liberar la CPU pronto y (a) siempre va a estar disponible para seguir avanzando, parándose cada vez que se necesita ejecutar el proceso (b). En el caso de priorizar el proceso (a) antes que (b), como este tiene muy pocas paradas, el tiempo de ejecución de (b) aumentará considerablemente.

Un factor clave en la planificación es la duración del **quantum** o de la **ráfaga de CPU**. Consiste en que cada cierto tiempo se lanza una interrupción desde el reloj para hacer un cambio de contexto, para que le planificador decida si hay algún proceso que tiene mayor prioridad que el que tiene la CPU actualmente y se la conceda a él.

El planificador debe cumplir las siguientes condiciones dependiendo del tipo de sistema que lo utiliza:

Todos los sistemas
- Equidad - Otorgar a cada proceso una parte justa de la CPU
- Aplicación de políticas - Verificar que se lleven a cabo las políticas establecidas
- Balance - Mantener ocupadas todas las partes del sistema

Sistemas de procesamiento por lotes
- Rendimiento - Maximizar el número de trabajos por hora
- Tiempo de retorno - Minimizar el tiempo entre la entrega y la terminación
- Utilización de la CPU - Mantener ocupada la CPU todo el tiempo

Sistemas interactivos
- Tiempo de respuesta - Responder a las peticiones con rapidez
- Proporcionalidad - Cumplir las expectativas de los usuarios

Sistemas de tiempo real
- Cumplir con los plazos - Evitar perder datos
- Predictibilidad - Evitar la degradación de la calidad en los sistemas multimedia

2.6.3. Cuando activar el planificador

Es necesario utilizar el planificador cuando:

- Se crea un nuevo proceso (ejecución de un **fork**)

- Un proceso termina su ejecución (ejecución de un **exit**)
- Bloqueo de un proceso
- Interrupción de E/S
- Interrupción de reloj (se acaba el quantum). En este caso hay dos tipos de algoritmos:
    - **No apropiativo:** no se quita el uso de la CPU hasta que el proceso no se bloquea o termina su ejecución. Son sistemas que no están regidos por el quantum.
    - **Apropiativo:** solo se permite ejecutar durante un tiempo fijado a un proceso antes de cambiarlo, es decir, solo le da la CPU hasta que se termine el quantum.

2.6.4. Planificación en sistemas de procesamiento por lotes

Estos sistemas se caracterizan por la baja interacción con el usuario, ya que solo lanzan los procesos y se van metiendo en una cola de ejecución. Los algoritmos suelen ser no apropiativos, un proceso se planifica y hasta que termine o se bloquee no suelta la CPU (no le afectan los quantums).

El algoritmo más utilizado es poner a todos los procesos en una **cola FIFO**, aunque suele ser muy ineficiente ya que considera a todos los procesos como iguales, lo que puede verse afectado por procesos con muchas operaciones de E/S.

Otra opción es lanzar a ejecución primero los procesos que tienen **menor tiempo de ejecución** o un trabajo más corto, lo que implica conocer los tiempos de ejecución de antemano, cosa que es bastante difícil. Con este algoritmo se mejora y aumenta la productividad o throughput, aunque solo es óptimo en términos de tiempo de respuesta si todos los trabajos están disponibles al mismo tiempo.

El **tiempo promedio** de respuesta es **(4a + 3b + 2c + d)/4** = T para el caso de 4 procesos, siendo a el primero en ejecutarse y d el último. Si se quiere minimizar el tiempo se debe elegir el menor a posible y que d sea el mayor de los valores debido al peso que tiene cada uno de ellos.

Como es posible que los procesos no lleguen todos al mismo tiempo, se planteó un algoritmo que ejecuta **primero el proceso de menor tiempo restante**. De esta forma cuando llega un nuevo proceso se compara su tiempo con el que ya se está ejecutando. Si el que acaba de llegar necesita menos tiempo para terminar se suspende al proceso actual y se le da la CPU al nuevo proceso. Este algoritmo es por lo tanto **apropiativo** y da un buen servicio a los procesos cortos.

2.6.5. Planificación en sistemas interactivos

Estos sistemas son de propósito general e interactúan con el usuario como los PCs. Todas las soluciones son **apropiativas**, implicando el bloqueo de procesos. Debido a esto los cambios de contexto se producen cuando un proceso quiere la CPU o cuando se pasa un quantum (una interrupción de reloj que se lanza cada x tiempo e implica un cambio de contexto cuando se recibe).

Algunos de los algoritmos de estos sistemas se basan en la planificación por **turno circular** o round-robin. Esto consiste en poner los procesos en una cola circular e ir pasando al siguiente cuando un proceso pierde su turno de CPU.

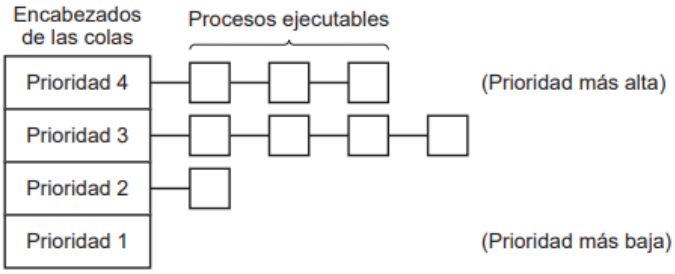
En estos sistemas es muy importante elegir la **duración del quantum**. Si los quantums son muy cortos se pierde mucho tiempo debido a los cambios de contexto y si son muy largos los procesos pueden tardar mucho en volver a tener la CPU, aumentando así los tiempos de espera. Normalmente los quantums se suelen poner de entre 20 ms a 50 ms.

Los procesos pueden tener una **prioridad** asignada para obtener mayor beneficio en el planificador de forma dinámica. Hay que evitar que los procesos con alta prioridad estén siempre

en la CPU, por lo que en cada interrupción de reloj (una vez consumido todo el quantum) se reduce la prioridad al proceso que tenía la CPU y se busca al proceso que tenga la prioridad mayor para darle la CPU.

Como se quiere que los procesos con muchas operaciones de E/S tengan mayor prioridad se pueden asignar las prioridades de forma **1/f**, dónde f es la fracción del último quantum utilizada por el proceso.

Como las prioridades suelen tener pocos números, los sistemas operativos suelen tener una cola para cada una de las prioridades y dentro de cada prioridad se utiliza el turno circular para elegir qué proceso de esa cola se ejecuta.



Otra opción es seleccionar al **proceso más corto**, de esta forma se elige al proceso estimado con menor tiempo de ejecución. La estimación de los tiempos de ejecución se hace en base a ejecuciones anteriores tal que **T = a * T0 + (1-a)*T1**, con T0 el tiempo estimado y T1 el tiempo de la última ejecución con $0 <= a <= 1$.

Otros sistemas operativos, normalmente con varios usuarios simultáneos, intentan que la **planificación esté garantizada** entre ellos. Si hay n usuarios, cada uno recibe 1/n de tiempo de CPU.

La **planificación por sorteo** es similar a la de prioridades, pero más “suave”. Cada uno va recibiendo boletos y se hace un sorteo cada vez que se tiene que dar la CPU a alguno. El que tenga más boletos tiene más posibilidades de obtener la CPU, pero un proceso con un solo boleto puede tocarle la lotería y conseguir la CPU con ese único boleto. Es posible que entre procesos se regalen boletos cuando uno o varios están esperando a que otro haga algo, de esta forma le dan los que tienen para que ese tenga más posibilidades de adquirir la CPU.

La **planificación por partes equitativas** consiste en asignar a cada usuario un valor proporcional de CPU que puede depender del número de procesos que tenga activos el usuario.

2.7. Planificación de Hilos

Si los hilos se gestionan a nivel de usuario entonces el sistema operativo no conoce los hilos y cada uno de los procesos tiene un planificador de hilos local. De esta forma puede haber planificadores específicos para cada proceso concreto.

Sin embargo, cuando los hilos son gestionados a nivel de núcleo, este selecciona un hilo sin importar el proceso al que pertenece (aunque puede tenerlo en cuenta). Hay que tener en cuenta que la conmutación entre hilos de procesos diferentes es más lenta que cuando ambos hilos son del mismo proceso, por lo que el planificador también lo puede tener en cuenta.

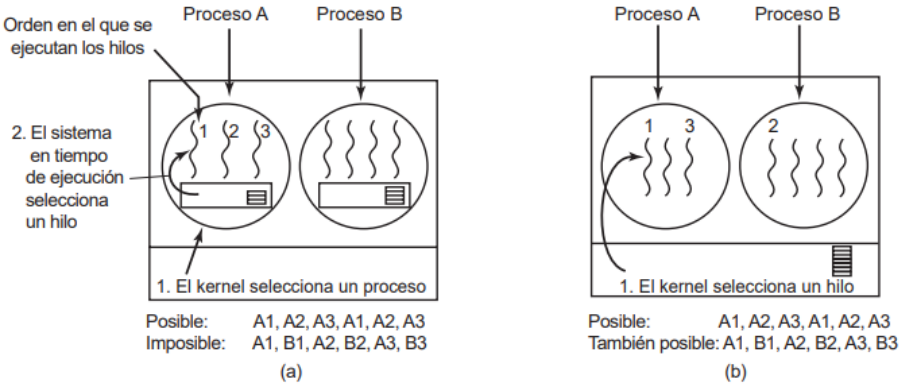


Figura 2-43. (a) Posible planificación de hilos a nivel usuario con un quántum de 50 mseg para cada proceso e hilos que se ejecutan durante 5 mseg por cada ráfaga de la CPU. (b) Posible planificación de hilos a nivel kernel con las mismas características que (a).

Tema 3

Administración de Memoria

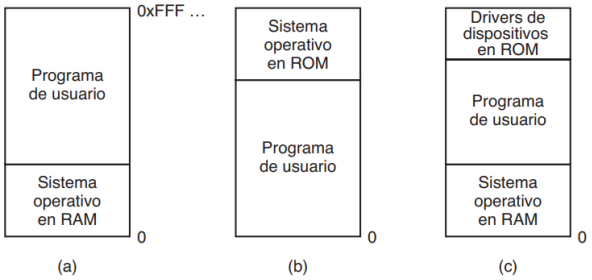
3.1. La memoria y el Sistema Operativo

Cuanto mayor sea la capacidad de la memoria, menor es su velocidad. La gestión del movimiento de datos entre la memoria principal (RAM) y la memoria secundaria (disco duro) es responsabilidad del sistema operativo.

El sistema operativo tiene que decidir cómo repartir el uso de la RAM, cómo asignar memoria a los procesos y como mover información entre la memoria principal y el disco duro, además de liberar la memoria principal cuando se termina un proceso, entre otras operaciones.

3.2. Ejecución de múltiples programas sin abstracción de memoria

Todas las direcciones de los programas son **físicas**. Desde que se inicia el sistema, el kernel se sitúa en la RAM, imagen (a), y todo el espacio que queda de usuario es donde se van a colocar los programas. Otra alternativa posible es colocar el kernel en las últimas posiciones de la memoria, dejando así las direcciones más pequeñas para los datos (b). También se pueden mezclar ambas opciones (c) dejando el código en el centro, aunque esta distribución implica algunos problemas.



Todas estas opciones son muy difíciles de controlar cuando hay varios programas o procesos. Una opción es dejar que solo haya un proceso en la memoria y cada vez que se cambie de proceso se copie este en el disco antes de traer el nuevo a la RAM. Esta opción es demasiado costosa.

Otra posibilidad es dividir la memoria en **bloques** de igual tamaño y permitir que varios procesos estén en diferentes bloques consecutivos, de forma que están completos en la memoria y así hasta que no que queda sitio. Cada bloque tiene una llave para llevar la constancia de que proceso tiene. Cuando ya no queda espacio en la memoria se debe quitar algún de los procesos. Es algo mejor que la anterior, pero sigue siendo muy mala.

Con este método es necesario hacer **reubicación estática**, es decir, actualizar las referencias a memoria al cargar el proceso en la memoria principal. Esto se debe hacer porque los procesos tienen las instrucciones pensadas para trabajar de la dirección 0 a X, pero hay que adaptarlas debido a que los bloques que ocupa el proceso pueden estar en otras direcciones. Por ejemplo, un proceso tiene en su dirección 0 una instrucción de salto a la dirección 28, si el proceso se carga en un bloque que inicia en la dirección 16384, el salto debe hacerse ahora a la dirección 16412. Esta operación de reubicación es demasiado costosa para el sistema.

La **unidad direccionable** es la cantidad de memoria a la que apunta una posición de memoria concreta. En el MIPS una posición de memoria apunta a un byte, por eso las instrucciones están almacenadas ocupando posiciones que están distanciadas 4 unidades.

3.3. Abstracción de memoria: El espacio de direcciones

La **abstracción** sirve para que el programador no tenga que pensar en las direcciones físicas de las variables y/o saltos, utilizando direcciones de la 0 a la X. Para ello hace falta un mecanismo que las traduzca a direcciones físicas.

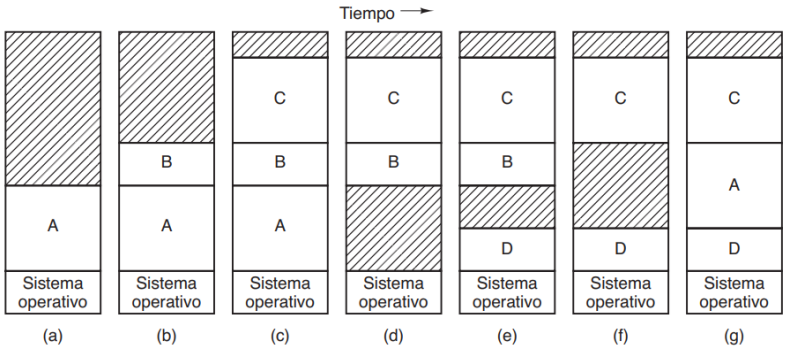
El **espacio de direcciones** es algo abstracto y equivale al conjunto de todas las direcciones de un proceso, que no tienen por qué corresponder con las direcciones físicas. Cada proceso tiene su propio espacio de direcciones y son independientes.

Al igual que cuando no hay abstracción es necesario hacer **reubicación**. Para ello se guarda en un registro, llamado **registro base**, la dirección de memoria donde empieza el programa. De esta forma cuando se quiere hacer un salto, por ejemplo, es necesario sumar el registro base más la dirección de salto. También es necesario utilizar un **registro límite** que define la longitud del programa y que se utiliza para comprobar que después de hacer la suma de una dirección más el registro base no se intente apuntar a una dirección de memoria que no corresponde al proceso.

3.3.1. Intercambio

Según se van cargando procesos, la memoria principal se va llenando (a, b, c) y, cuando esta se llena, es necesario desalojar al menos uno de los procesos que está en ella para sustituirlo por otro que la necesita (d, e). De esta forma pueden cargarse varios procesos en la memoria y cada uno de ellos puede ir en zonas diferentes (el proceso A se colocó en 2 sitios diferentes a lo largo del tiempo) utilizando la **reubicación dinámica**.

Con este método se van generando huecos en la memoria (e) en los que puede que no quepa ningún proceso llegando así a la **fragmentación** de la memoria. Para eliminar la fragmentación de la memoria se puede hacerse una **compactación** de forma que se juntan todos los huecos libres en uno solo, aunque esta operación es muy costosa.

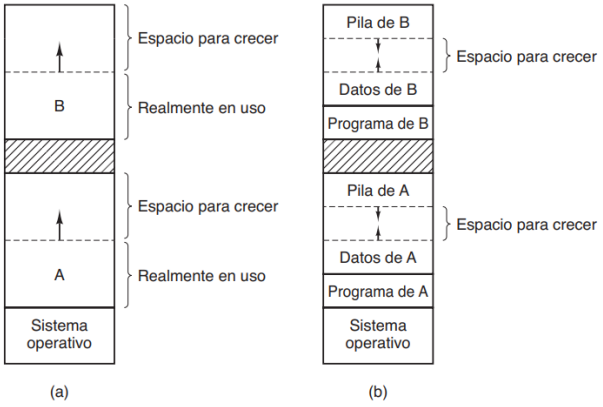


Con esta solución surge el problema de saber cuánta memoria se debe asignar a cada proceso, porque durante su ejecución estos pueden necesitar ocupar más espacio, debido al stack y a la reserva dinámica (heap). Algo que a priori no se conoce. Cuando un proceso crece se puede realizar una de las siguientes opciones:

- Ocupar memoria adyacente
- Mover el proceso a otra zona de memoria
- Intercambiar procesos para crear un hueco (pasar un proceso de la RAM al disco y traer el nuevo a la RAM)

- Suspender el proceso (no hay espacio suficiente para que crezca)

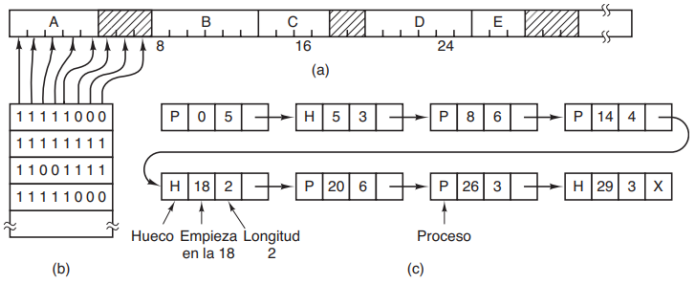
Es recomendable que cuando se carga o mueve un proceso en memoria se le asigne algo de memoria extra para el caso hipotético de que crezca (a). En (b) se deja una zona donde compiten el stack y la zona dinámica para crecer. En este caso el problema está cuando ambas partes chocan en esa zona y ya no pueden crecer más.



3.3.2. Administración de memoria libre

La información acerca del estado de la memoria, es decir, si un bloque está libre u ocupado, se puede almacenar en mapas de bits o en listas enlazadas. Ambas se utilizan con bloques de memoria de tamaño constante y fijo. Se sitúan en el kernel para que el sistema operativo tenga conocimiento del estado de la memoria y para que cuando tenga que introducir un nuevo proceso en ella sepa dónde puede ir o qué procesos sacar para que este quepa. En ambos casos, la búsqueda de un espacio libre es lenta.

En un **mapa de bits** basta con asignar un bit a cada bloque de memoria de forma que se pone a 1 si este está ocupado y a 0 si está libre (b). Su tamaño es siempre constante ya que hay un bit por cada bloque.



La alternativa es tener una **lista enlazada** que vaya uniendo procesos y huecos (c). En la imagen se representa en cada posición de la lista si es un proceso (P) o un hueco (H), la posición en la que inicia y su longitud, además de un puntero para el siguiente bloque. Su tamaño es variable en función de lo fragmentada que esté la memoria.

Cuando se utilizan listas ligadas existen diversos algoritmos para la asignación de memoria:

- **Primer ajuste:** se recorre la lista hasta que se encuentra un hueco de tamaño igual o mayor al del proceso que se va a introducir. Es una opción bastante rápida.
- **Siguiente ajuste:** la exploración empieza desde dónde se encontró el hueco anterior. Tiene un rendimiento algo peor que el del primer ajuste.

- **Mejor ajuste:** se recorre toda la lista y se elige el hueco más pequeño en el que cabe el nuevo proceso. Es una opción lenta y termina produciendo huecos en memoria muy pequeños e inutilizables.
- **Peor ajuste:** se recorre toda la lista y se elige le hueco más grande disponible. Es una opción muy poco eficiente.
- **Listas separadas de procesos y de huecos:** cuando se debe introducir un proceso solo se mira la lista de huecos y si esta está ordenada por el tamaño de los huecos. Con esto se pueden llegar a acelerar alguno de los algoritmos anteriores como el del mejor ajuste.
- **Ajuste rápido:** listas separadas para los tamaños más habitualmente solicitados, que permite una búsqueda muy rápida, pero una actualización lenta.

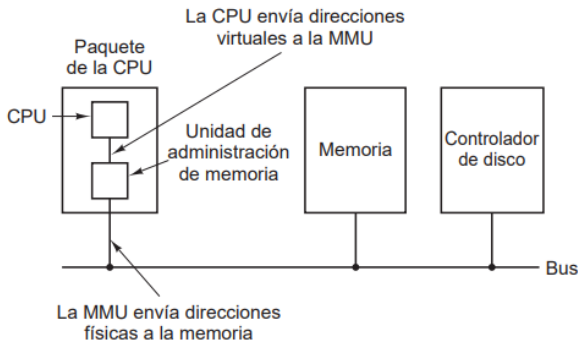
3.4. Memoria Virtual

Para esta parte de la materia el espacio de direcciones de cada proceso se llamará **memoria virtual** o espacio de direcciones virtual.

Con esta solución el espacio de direcciones se divide en bloques de igual tamaño llamados **páginas**. Las páginas deben tener un tamaño de potencia de 2 y este tamaño es común para todos los procesos del sistema. No todas las páginas de un proceso tienen que estar en memoria principal al mismo tiempo (también está dividida en páginas) para poder ejecutarse. La memoria principal en este caso no tiene los bloques de un mismo proceso juntos, es decir, es **totalmente asociativa** (cualquier página puede ir en cualquier posición) y tiene diferentes bloques de varios procesos activos al mismo tiempo en ella.

Un **fallo de página** en la memoria principal sucede cuando un proceso necesita un dato de una de sus páginas que no se encuentra en la RAM y debe ir al disco duro a por ella. Su coste es tan grande que cuando sucede el proceso que necesitaba esa página debe bloquearse.

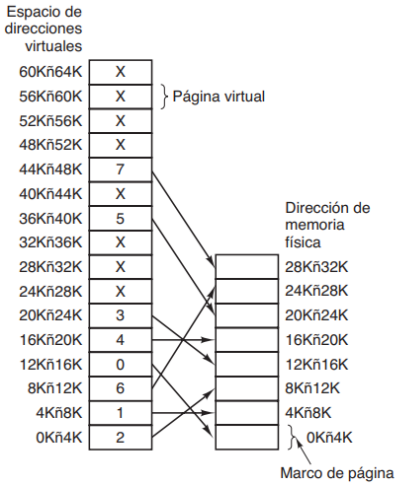
Las **direcciones virtuales** de cada proceso son las direcciones de su espacio virtual. De esta forma cada dirección virtual se corresponde con una dirección física de la RAM. Para ello hace falta un mecanismo hardware llamado **MMU** (Memory Management Unit) que pasa las direcciones virtuales a direcciones físicas.



3.4.1. Tabla de páginas

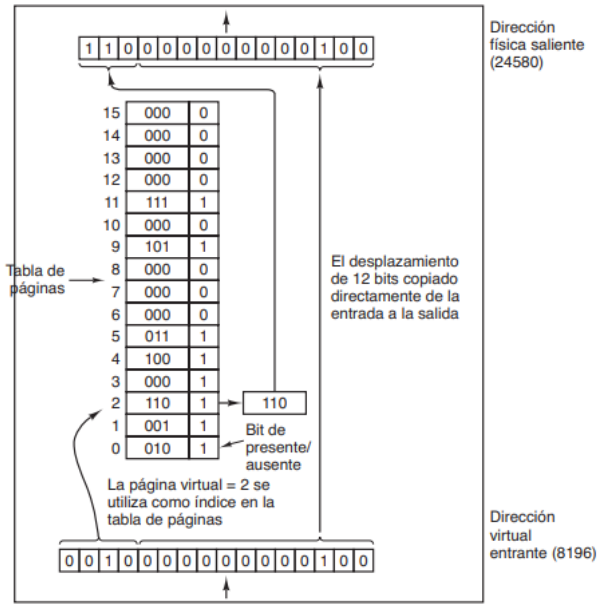
A la izquierda se muestra el espacio virtual de un proceso, que va desde la dirección 0 hasta la 62KB, dividido en páginas de tamaño de 4KB. Es lo que se llama **tabla de páginas** del proceso. Estas tablas se almacenan en la zona de kernel y hay una por cada proceso.

La de la derecha es la memoria RAM con su división en páginas también. Los números que tienen dentro las páginas del proceso representan la página en la que se encuentran en la RAM.



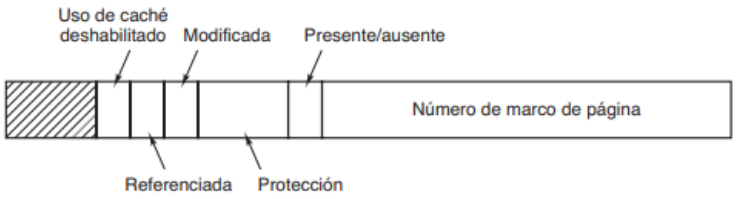
(la traducción del libro tiene una ñ donde debería haber un guion xd)

Un ejemplo del trabajo de la MMU puede ser la instrucción MOV REG, 0. En el espacio de direcciones virtuales del proceso esta instrucción apunta a la primera página, pero en la RAM se encuentra en la tercera página y en lugar de apuntar a un 0 apunta a 8K. Entonces la MMU traduce el número de página, pero el desplazamiento dentro de esta sigue siendo el mismo. Además de calcular la dirección física, la adapta al tamaño de estas porque es probable que para apuntar algo en el espacio de direcciones virtual se necesiten más bits que para las direcciones físicas.



Todas las entradas de la tabla de páginas tienen el **número del marco de página** con el número de bits que necesite (dependiente del tamaño de la memoria RAM) y el bit de residencia o **bit de presente/ausente**, que indica si la página se encuentra en la RAM o no. Esto es muy importante a la hora de hacer las traducciones entre direcciones. Se coloca un bit para saber si el sistema accede a alguna posición de esa página (**bit referenciado**), utilizado por alguno de los algoritmos de reemplazo, al igual que el **bit de modificado**, que se pone a 1 cuando uno de los datos dentro de la página ha sido modificado, sirviendo también para saber si hay que actualizarla en el disco cuando se saca de la RAM al disco. El **bit de uso de caché deshabilitado** se utiliza para impedir que ninguna de las líneas de la página se coloque en la caché por temas de coherencia caché. Por

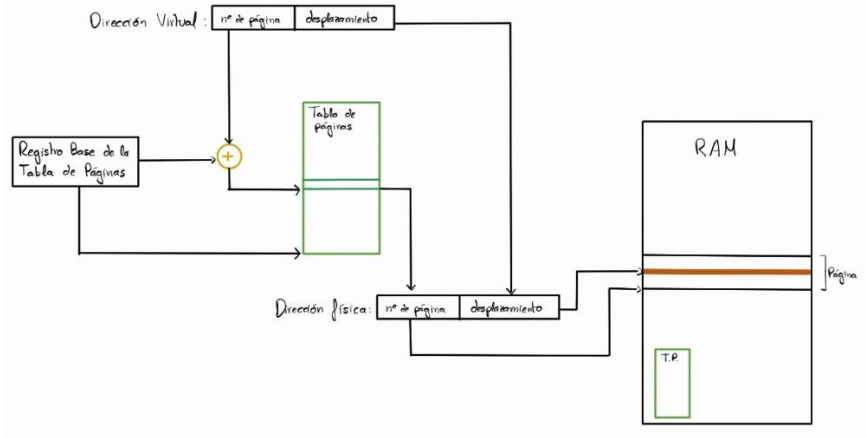
último, están los **bits de protección** que indican si hay que proteger los datos de la página frente a lectura y/o escrituras.



3.4.2. Aceleración de la paginación

El espacio que ocupan las tablas de páginas puede llegar a ser muy grande cuando se trabaja en un espacio de direcciones grande. La traducción a pesar de ser rápida implica varios accesos a la memoria principal (es dónde se encuentra la tabla de páginas) por instrucción, que provoca un problema de rendimiento.

En la CPU hay un **registro base de la tabla de páginas** que almacena dónde empieza la tabla de páginas del proceso que se está ejecutando en el momento. Cuando hay un cambio de contexto, el nuevo proceso también tiene que actualizar este registro para que la CPU sepa dónde empieza su tabla de páginas en la memoria principal. Con este mecanismo solo se puede acceder a posiciones de dicho proceso y no hay problemas por intentar acceder al espacio físico de otros.



3.4.2.1. TLB

El **búfer de traducción adelantada** o **TLB** es una caché **asociativa** situada en el procesador y usada para el proceso de traducción, que tiene una zona de directorio y otra de almacenamiento. Recibe como entrada un número de página virtual y como salida un número de página físico. En la zona de directorio se guardan aquellas páginas que están residentes en memoria principal y en la zona de almacenamiento el número de páginas físico al que corresponden.

El uso de la TLB está pensado para evitar los accesos a la memoria principal y acelerar el proceso de traducción de las direcciones. Se sitúa dentro de la MMU, que también está dentro la de CPU y tiene muy pocas entradas. En la siguiente tabla (imagen de abajo) hay un ejemplo del contenido que puede tener la TLB.

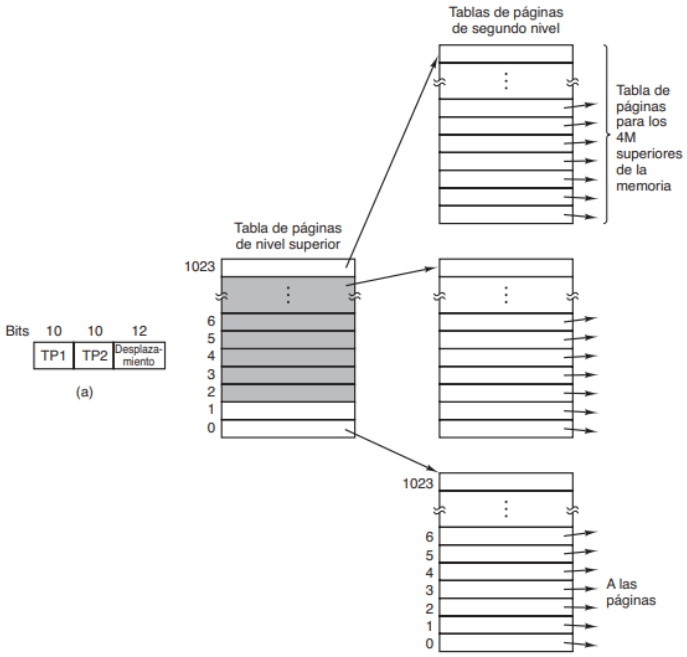
Cuando se intenta traducir una dirección de una página que no se encuentra en la TLB se dice que se ha producido un **fallo de TLB**. Esto no quiere decir que la página no se encuentre en la memoria principal, solo que no está en la TLB. Ahora hay que pasar por la tabla de páginas para hacer la traducción y acceder a la página.

Cuando esto ocurre, hay que traer la nueva página también a la TLB por el tema de la localidad.

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

3.4.2.2. Tabla de páginas multinivel

La solución para evitar que la tabla de páginas sea muy grande, ralentizando el proceso de traducción, es utilizar tablas de página de varios niveles, que normalmente son 2. En memoria principal siempre se guarda la tabla de primer nivel en la que se tiene información sobre las tablas de segundo nivel, que pueden estar o no en la memoria principal.



Ahora las direcciones virtuales deben tener unos bits para acceder a la tabla de primer nivel y otros para la del segundo. Este mecanismo ahorra mucha memoria, ya que las tablas de páginas pueden estar en el disco, pero ralentiza mucho la traducción debido a los accesos necesarios para buscar la página.

3.4.2.3. Ejercicio de examen de tablas de páginas multinivel

Se tiene la dirección virtual de 32 bits que en decimal es 0x00403004. Por los tamaños que tiene la máquina se sabe que los últimos 12 bits de la dirección corresponden al desplazamiento dentro la página, eso se sabe porque las páginas ocupan 2^12 posiciones de memoria.

El tamaño del segundo campo se calcula en función del tamaño de las tablas de segundo nivel, que en este caso son 2^10, de ahí a que se necesiten 10 bits de la dirección.

En este ejemplo, coinciden los bits de acceso de la tabla de primer nivel, no tiene por qué ser así.

Para traducir la dirección virtual a dirección física se toma el primer campo y con ella se accede a la tabla de primer nivel, que en este ejemplo es a la entrada 1 (no es a la primera porque empieza contando en 0). De ahí se sabe dónde está la tabla de segundo nivel y se accede a ella. Se posiciona en ella con los bits del segundo campo. De esa tabla se toma la página a la que se quiere acceder, y con los bits de desplazamiento se llega al dato buscado.

Puede producirse un fallo cuando se está en la tabla de primer nivel y la tabla de segundo nivel no está en la memoria principal. Lo mismo pasa cuando se está en la de segundo nivel y la página tampoco está en la RAM. De esta forma pueden ser necesarios 2 accesos al disco para leer el dato.

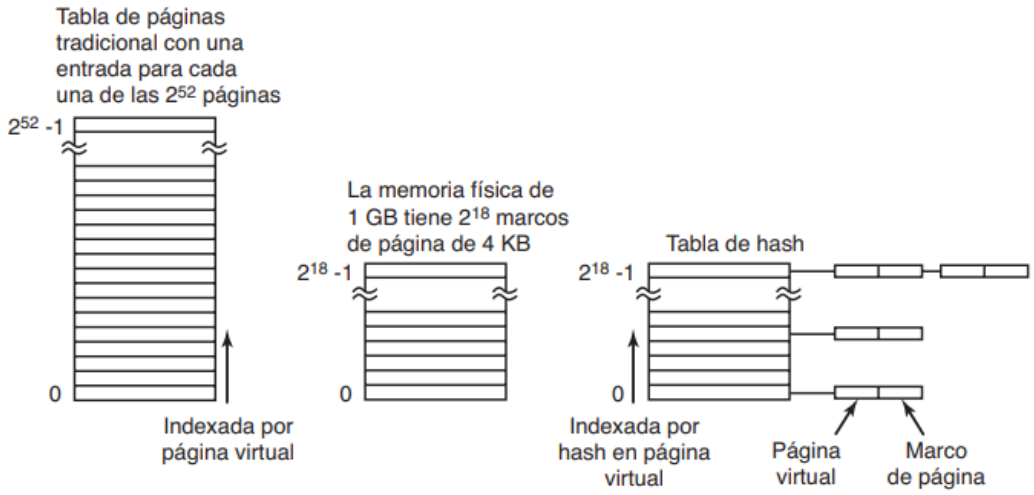
En un sistema de 64 bits y páginas de 4KB, se tienen 2^64 posiciones direccionables en el espacio virtual entre 2^12, que es el tamaño de las páginas, dando así un total de 2^52 páginas, lo que hace que cada proceso tenga una tabla de páginas con 2^52 entradas.

3.4.2.4. Tabla de páginas invertida

En lugar de tener la tabla de páginas ya vista, con tantas entradas como páginas tenga el proceso, se tiene **una tabla de páginas invertida**. Esta tiene tantas entradas como páginas quepan en la memoria principal, que en el ejemplo son 2^18, ya que la memoria es de 2^30 bits y las páginas son de 2^12 bits.

En estas entradas, para cada página física almacena la página virtual que la contiene y el proceso al que pertenece. Para estas tablas, cuando es necesario hacer la traducción hay que mirar todas las entradas para ver si lo que hay en ella pertenece al proceso que se está ejecutando y comprobar si el número de página virtual coincide con el que se quiere traducir. Si no se encuentra en ella, se produce un fallo de página.

Para evitar tener que recorrer toda la tabla se suele usar una tabla indexada por hash. También se puede trabajar con la TLB para acelerar los accesos.



3.5. Algoritmos de Reemplazo de páginas

Hay que reemplazar una página en la memoria principal cuando el programa en ejecución solicita una página que no se encuentra en ella, es decir, cuando se produce un fallo de página y la memoria principal está llena. Los fallos de página implican necesariamente un cambio de contexto porque el proceso se bloquea esperando a que se cargue la página antes de pasar al estado listo.

Si la página seleccionada para desalojarla tiene el bit de modificado a 1, es necesario reescribir toda la página en el disco duro antes de sobrescribirla con la nueva página.

Hay que tener en cuenta que la memoria principal puede dejar dar dos opciones. La primera, que un proceso tenga el número de páginas que quiera en ella, es decir, utiliza una **gestión global.** O la segunda, solo permitir un número concreto de páginas para que todos los procesos tengan la misma parte de memoria o con un tope, lo que es una **gestión local**. En ambos casos se utilizan los mismos algoritmos, pero en diferentes candidatos.

Todos los algoritmos son rutinas de kernel, por lo que son rápidas y sus respuestas siempre son las más adecuadas.

3.5.1. Óptimo

Se reemplaza la página que se vaya a referenciar más tarde. Esto es imposible de saber, por lo que este algoritmo no es aplicable si no se ha ejecutado antes ese proceso. De esta forma se pospone el fallo de página lo máximo posible.

Es utilizado para compararlo con otros algoritmos siendo este el mejor de todos.

3.5.2. No Usadas Recientemente (NRU)

Cada página tiene 2 bits de estado, R y M, que se van modificando a lo largo del tiempo en la tabla de páginas. El bit R se pone a uno cuando se rehúsa una página y el bit M cuando esta se modifica. De esta forma existen 4 posibles estados por página (del 00 al 11). Cuando se tiene que desalojar una página se busca la que tenga el estado con el menor valor. En el caso de que haya varias con el mismo estado se elige una de forma aleatoria.

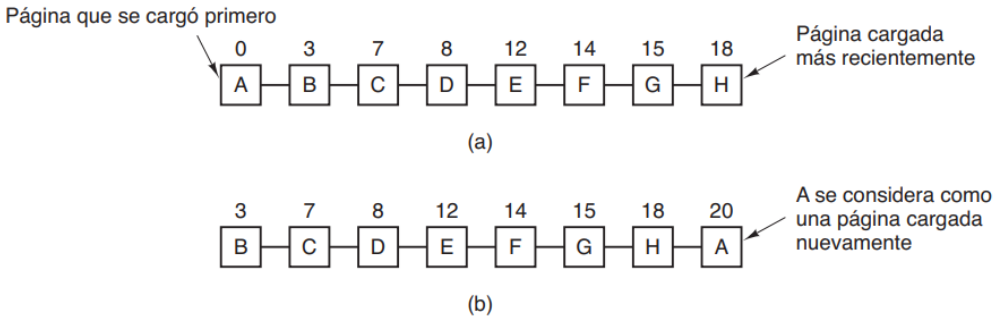
Cada cierto tiempo (por ejemplo, en cada cambio de contexto), se pone el bit R a 0 porque es posible que algunas páginas solo se utilicen una vez y no se vuelvan a usar más.

3.5.3. FIFO

Esta estrategia borra la página que lleva más tiempo en la memoria sin importar si se modificó ni ningún otro aspecto.

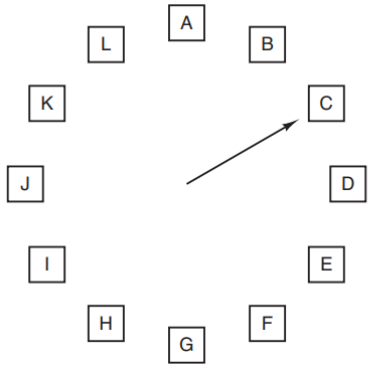
3.5.4. Segunda Oportunidad

Se inspecciona el bit R de la página que lleva más tiempo en la memoria principal. Si tiene R = 0, se desaloja y si R = 1, se pasa R a 0 y se mira la siguiente página más vieja. Se sigue el mismo criterio hasta que se encuentra una página con R = 0 o se recorren todas las páginas y se vuelve a la primera página mirada y esta es la que se elimina.



3.5.5. Reloj

Es idéntico al algoritmo de la segunda oportunidad, pero en lugar de utilizar una cola se utiliza un puntero que va recorriendo las páginas de forma circular.



Cuando ocurre un fallo de página, la página a la que apunta la manecilla se inspecciona. La acción a realizar depende del bit R:
R = 0: Desaloja la página
R = 1: Desactiva R y avanza la manecilla

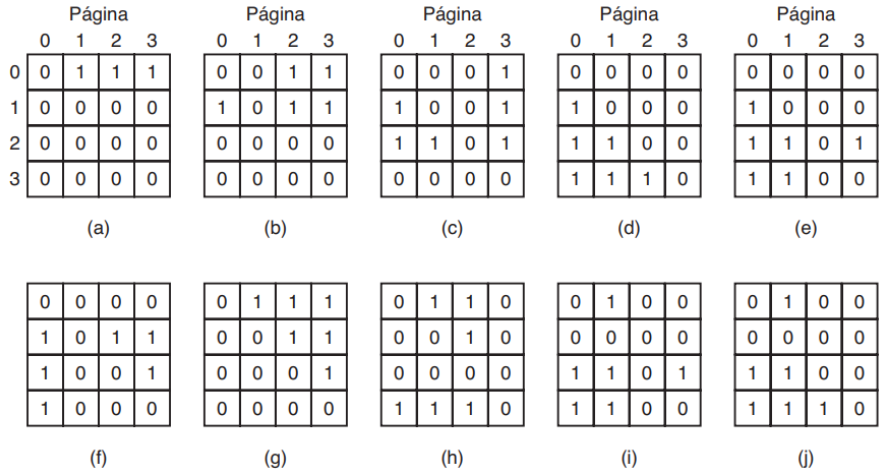
3.5.6. Menos Recientemente Usada (LRU)

Como dice su nombre, se evita eliminar páginas que han sido usadas recientemente, buscando aprovechar el principio de localidad temporal. Se utiliza una lista de páginas ordenadas según su uso, lo que implica actualizar la lista en cada referencia a memoria y es costosa de implementar.

Se puede utilizar un contador hardware que se incremente en cada instrucción y después de cada referencia a memoria se almacena el valor del contador en la entrada de la tabla de páginas. De esta forma cuando hay un fallo de página se elige la página con el contador más bajo.

En el caso de aplicarlo en la TLB, se construye una matriz de X bits con una fila y columna por cada una de las páginas que hay en ella. Cada vez que se hace una referencia a la página se pone toda su fila a 1 y después toda su columna a 0. De esta forma al final se reemplaza la página con 0 en toda su fila y todo 1 en su columna.

En el ejemplo de abajo se hacen las siguientes referencias a páginas: 0 1 2 3 2 1 0 3 2 3



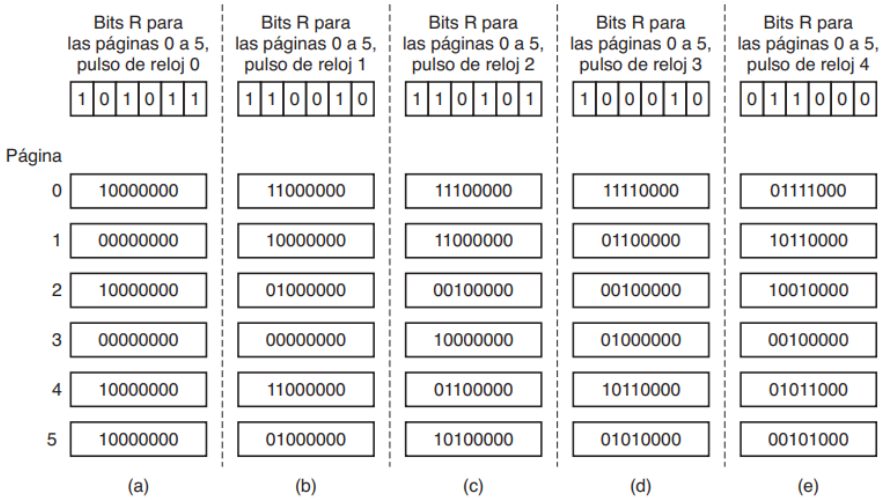
3.5.7. No Usada Frecuentemente (NFU)

Es menos estricto que el algoritmo de menos usada recientemente. Para ello emplea un contador asociado a cada página, que se inicia a 0, y en cada interrupción de reloj el sistema operativo explora las páginas y pone el bit R a 1 o a 0. De esta forma el contador lleva la cuenta aproximada de la frecuencia de referencia de cada página y se reemplaza la página que tenga el menor contador.

Como el contador solo aumenta, las páginas que acaban de llegar a la memoria tienen muchas posibilidades de ser elegidas para irse que las llevan más tiempo, aunque esas hagan mucho que no se usan.

3.5.8. Envejecimiento (NFU Modificado)

En esta estrategia se sigue utilizando el contador que se actualiza en cada interrupción de reloj, pero este se desplaza un bit a la derecha antes de agregar el nuevo R a la izquierda. De esta forma se tienen en cuenta las que más se usen recientemente y no en toda su historia.

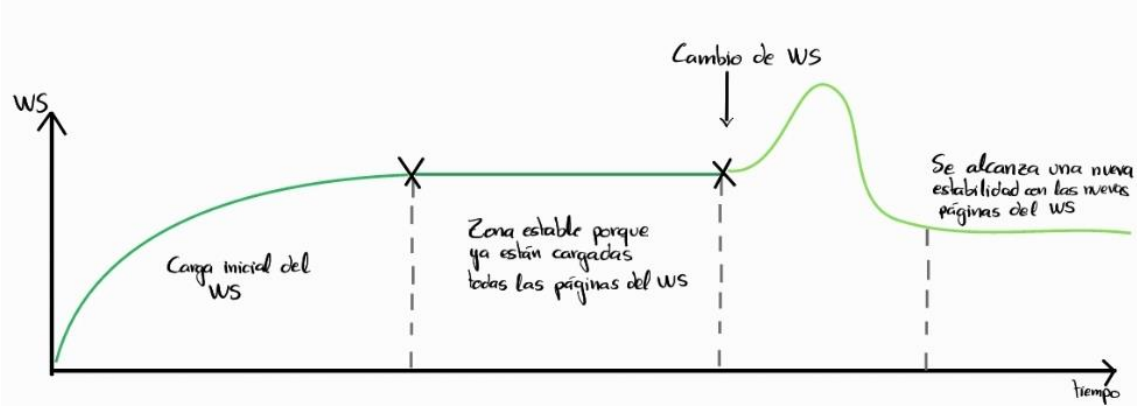


3.5.9. Conjunto de Trabajo (WS)

Suele ser el algoritmo más utilizado. El **conjunto de trabajo** es el conjunto de páginas que utiliza el proceso en un momento dado. Las páginas se cargan bajo demanda, es decir, solo se cargan cuando se solicitan datos de ellas. Se puede hacer precarga, que consiste ir al disco por un fallo de página y aprovechar para traer otra que puede que ser utilizada en el futuro, suele ser la que está después.

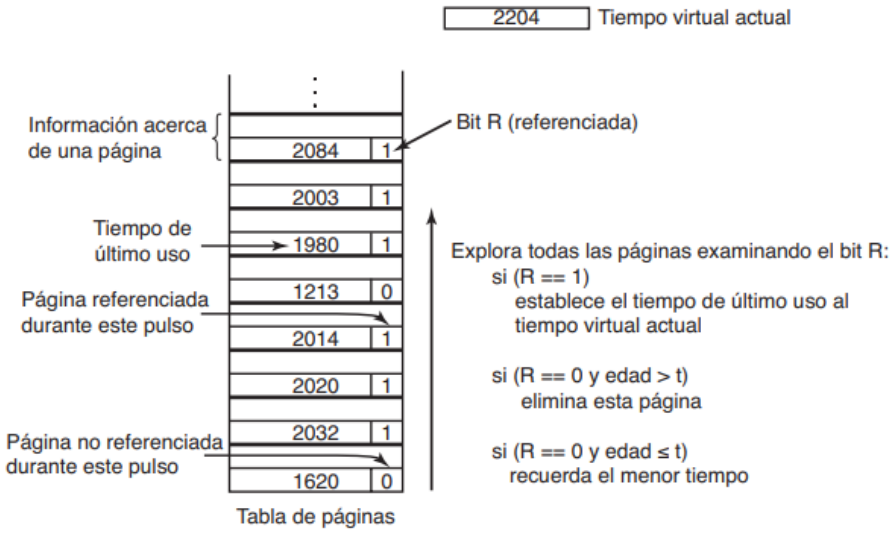
Este algoritmo intenta mantener en memoria principal el grupo de páginas que el proceso ha estado utilizando recientemente. La **sobrepaginación** es una situación en la cual un proceso produce muchos fallos de memoria en poco tiempo haciendo que su ejecución sea muy lenta, por lo que es importante evitarla, como pasa en el pico de la gráfica de abajo.

Cuando se empieza a ejecutar un proceso se producen muchos fallos de página y se dice que se ha cargado el conjunto de trabajo de ese proceso cuando se dejan de producir tantos fallos. Cuando un proceso alcanza su conjunto de trabajo, pero pierde su CPU, otros procesos le quitan sus páginas de memoria, por lo que cuando este proceso vuelva tiene que volver a cargarlas. Para evitar bloquear al proceso mientras recupera su WS (conjunto de trabajo) lo que se hace es una **prepaginación** que consiste en cargarlo todo antes de poner a ejecutar el proceso, pero para ello es necesario conocerlo de antemano, aunque este va cambiando con el tiempo.



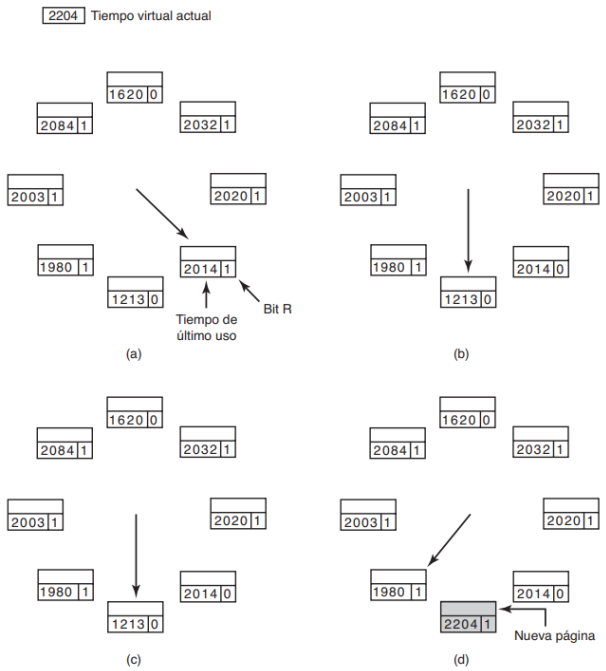
Cuando hay un fallo, se eliminan de la memoria principal **todas** las páginas que no están en el WS y se trae la nueva. Para eso se tiene un valor ***k*** que representa la duración del WS y se busca que páginas tienen un ***t*** (tiempo que lleva sin usarse) que supera a ***k***. Esto es demasiado costoso porque hay que actualizar la tabla de páginas cada vez que se produce un fallo de página.

En su lugar se utiliza un contador que tiene las páginas usadas durante los últimos X segundos de tiempo de ejecución. Por lo tanto, se utiliza un **tiempo virtual**, que no va al ritmo del reloj.



El problema de esta opción es que hay que recorrer toda la tabla y actualizarla en cada fallo de página hasta que se localiza una página para desalojar.

Otra implementación posible es utilizar un puntero circular, **WS Clock**, que va recorriendo cíclicamente la tabla de páginas. Funciona igual que el algoritmo del reloj. En esta situación solo elimina una página.



Si hay una página sucia/modificada, no se elimina de inmediato porque eso implicaría bloquear al proceso para copiar en el disco la página, por lo que se planifica su escritura en el disco y se sigue avanzando en el reloj. De esta forma se agrupan varias páginas modificadas y se copian todas después al mismo tiempo.

3.6. Cuestiones de Diseño de los sistemas de paginación

3.6.1. Políticas de asignación local contra las de asignación global

Una cuestión importante asociada con la elección de página a sustituir cuando hay un fallo de página es cómo se debe asignar la memoria entre los procesos ejecutables en competencia.

Si se puede eliminar cualquier página de la memoria principal se dice que esta tiene **asignación global** mientras que si solo se pueden sustituir páginas que correspondan al proceso que provocó el fallo de página es **asignación local**. En general funciona mejor la global ya que es más difícil que se produzca sobrepaginación (ocurre cuando un proceso produce muchos fallos de memoria en poco tiempo).

Cuando se utiliza un algoritmo global se puede dar, al inicio de cada proceso, un cierto número de páginas proporcional a su tamaño, aunque la asignación se actualice dinámicamente a medida que se va ejecutando. Para ello se puede utilizar el algoritmo **PFF** (Frecuencia de Fallo de Página) que indica cuando se debe aumentar o disminuir la asignación de páginas a un proceso en función del número de fallos de página que tenga.

En algunos de los algoritmos vistos solo tiene sentido utilizar estrategias locales como en los del conjunto de trabajo ya que trabajan en el contexto de un único proceso.

3.6.2. Control de Carga

A pesar de los algoritmos de reemplazo y la asignación global, un sistema puede sobrepaginarse de tal forma que todos los procesos necesitan más memoria y ninguno puede cederla (calculado por el PFF).

La única solución a este problema es mandar algunos de los procesos al disco duro, aunque si el número de procesos en la memoria principal es demasiado bajo puede provocar la inactividad de la CPU.

3.6.3. Tamaño de Página

El tamaño de página es un parámetro que a menudo el sistema operativo puede elegir. No hay un tamaño de página óptimo en general debido a que los factores que lo determinan son competitivos.

En promedio, los datos suelen dejar la última página que ocupan a medio llenar, dejando inutilizable esa parte, provocando la **fragmentación interna**, lo que hace preferible un tamaño de página más pequeño. Sin embargo, tener páginas pequeñas implica que los programas necesiten muchas más páginas, haciendo que la tabla de páginas sea mucho más grande.

El coste del movimiento de una página grande es similar al coste de la búsqueda de una página de menor tamaño, resultando ambas con tiempos de transferencia muy similares a pesar de las diferencias de tamaño.

En algunas máquinas, la tabla de páginas se debe cargar en registros de hardware cada vez que la CPU cambia de un proceso a otro. En estas máquinas, tener un tamaño pequeño de página

significa que el tiempo requerido para cargar sus registros aumenta a medida que se hace más pequeña la página.

Suponiendo que el tamaño promedio de un proceso es de ***s*** bytes, el tamaño de página es de ***p*** bytes y cada entrada de página requiere ***e*** bytes, se puede calcular la **sobrecarga**. Si el número de páginas necesarias por proceso es ***s/p***, ocupan ***se/p*** bytes en la tabla de páginas y la memoria desperdiciada en la última página del proceso debido a la fragmentación interna es ***p/2***, se llega a la siguiente fórmula:

$$sobrecarga = \frac{s * e}{p} + \frac{p}{2}$$

El primer término (tamaño de la tabla de páginas) es grande cuando el tamaño de página es pequeño, mientras que el segundo término (fragmentación interna) es grande cuando el tamaño de página es grande. El valor óptimo debe estar entre esos dos.

Si se saca la primera derivada con respecto a p y se iguala a 0 se llega a: $-\frac{se}{p^2} + \frac{1}{2} = 0$ y de esta se puede derivar a la fórmula que proporcione el **tamaño de página óptimo: $\boldsymbol{p = \sqrt{2se}}$.**

3.6.4. Espacios separados de instrucciones y de datos

La mayor parte de las computadoras tienen un solo espacio de direcciones que contiene tanto programas como datos y, si es lo bastante grande, funciona bien. El problema está cuando este espacio es pequeño.

Una solución a esto puede ser tener espacios de direcciones separados para las instrucciones y los datos. Ambos espacios se pueden paginar de manera independiente y cada uno tiene su tabla de páginas, con su propia asignación de páginas virtuales a marcos de páginas físicas.

3.6.5. Páginas compartidas

En un sistema de multiprogramación grande, es común que varios usuarios ejecuten el mismo programa a la vez, por lo que es muy eficiente compartir páginas entre ellos, aunque no todas las páginas pueden ser compartidas.

Cuando se utiliza un espacio de instrucciones separado del de datos se puede compartir el espacio de instrucciones. Esto puede ocasionar problemas cuando el primer proceso llega a un punto donde el planificador decide eliminar el espacio de direcciones que aún estaba usando el segundo proceso, haciendo así que el segundo provoque un gran número de fallos de página para volver a cargar esas páginas.

Compartir datos es más complicado, aunque en UNIX se suele hacer después de crear un proceso hijo. Lo que se suele hacer es dar a cada uno de estos procesos su propia tabla de páginas y hacer que ambos apunten a las mismas, aunque solo en modo lectura. Tan pronto como uno de los procesos intente modificar una de las páginas se produce un trap al sistema operativo y se hace una copia de la página para que cada proceso tenga una copia de esta, ahora con permisos de lectura y escritura.

3.6.6. Bibliotecas Compartidas

Las bibliotecas compartidas o **DLL** son archivos de solo lectura que proporcionan funciones **externas indefinidas** que los procesos utilizan. Dependiendo del sistema se cargan cuando se carga el programa o cuando las funciones en ellas se llaman por primera vez. El punto es que si un programa ya ha cargado una biblioteca, no es necesario que otra la vuelva a cargar, aunque hay

que tener en cuenta que las bibliotecas se van copiando por páginas en la RAM y no se suelen copiar enteras.

Se pueden compilar las bibliotecas compartidas con una bandera de compilador especial, para indicar al compilador que no debe producir instrucciones que utilicen direcciones absolutas en ellas. De esta forma se evitan problemas si se colocan en diferentes posiciones en los espacios de direcciones virtuales de los procesos. El código que utiliza sólo desplazamientos relativos se conoce como **código independiente de la posición**.

3.6.7. Archivos Asociados

Los archivos asociados a memoria proporcionan un modelo alternativo para la E/S. En vez de realizar lecturas y escrituras, el archivo se puede procesar como un gran array de caracteres en la memoria.

Si dos o más procesos se asocian al mismo archivo y al mismo tiempo, se pueden comunicar a través de la memoria compartida, ya que las escrituras de uno son visibles al otro. Las bibliotecas compartidas pueden usar este mecanismo.

3.6.8. Política de Limpieza

La paginación funciona mejor cuando hay muchos marcos de página libres que se pueden reclamar cuando ocurre un fallo de página. Para asegurar una provisión abundante de marcos de página libres, muchos sistemas de paginación tienen un proceso en segundo plano, conocido como **demonio de paginación**, que está inactivo la mayor parte del tiempo, pero se despierta de forma periódica para inspeccionar el estado de la memoria. Si hay pocos marcos de página libres, el demonio empieza a seleccionar páginas para desalojarlas mediante cierto algoritmo de reemplazo.

En caso de que una de las páginas desalojadas se necesite otra vez antes de que se sobrescriba su marco, puede reclamarse si se elimina de la reserva de marcos de página libres. Al mantener una provisión de marcos de página se obtiene un mejor rendimiento que al utilizar toda la memoria y después tratar de encontrar un marco cuando se necesite.

Una manera de implementar esta política de limpieza es mediante un reloj con dos manecillas. La manecilla principal es controlada por el demonio de paginación y cuando apunta a una página sucia, esa se escribe de vuelta en el disco y la manecilla principal avanza. La manecilla secundaria se utiliza para reemplazar páginas, como en el algoritmo del reloj estándar, solo que ahora, la probabilidad de que la manecilla secundaria llegue a una página limpia es mucho mayor debido al demonio de paginación.

3.7. Implementación de los sistemas de paginación

3.7.1. Manejo de fallos de página

Cuando ocurre un fallo de página de desarrolla la siguiente secuencia:

1. El hardware hace un **trap al kernel**, guardando el contador del programa en la pila, además de cierta información acerca del estado de la instrucción actual en registros especiales de la CPU.
2. Se inicia una **rutina** en código ensamblador para guardar los registros generales y demás información volátil, para evitar que el sistema operativo la destruya. Esta rutina llama al sistema operativo como un procedimiento.

3. El **sistema operativo** descubre el fallo de página y trata de descubrir qué página virtual se necesita.
4. Una vez se conoce la dirección virtual que produjo el fallo, el sistema comprueba si esta **dirección es válida** y si la protección es consistente con el acceso. De no ser así, el proceso recibe una señal o es eliminado. Si la dirección es válida y no ha ocurrido un fallo de página, el sistema comprueba si hay un **marco de página disponible**. Si no hay marcos disponibles, se ejecuta el algoritmo de reemplazo.
5. Si el marco de página seleccionado está **sucio**, la página se planifica para la transferencia al disco y se realiza un cambio de contexto, suspendiendo el proceso fallido y dejando que se ejecute otro hasta que se haya completado la transferencia al disco. En cualquier caso, el marco se marca como ocupado para evitar que sea utilizado con otro propósito.
6. Cuando el marco está **limpio**, el sistema operativo busca la dirección de disco donde se encuentra la página necesaria y planifica una **operación de disco** para llevarla a memoria. Mientras se está cargando la página, el proceso fallido sigue suspendido y se ejecuta otro proceso de usuario.
7. Cuando la interrupción de disco indica que la página ha llegado, **las tablas de páginas se actualizan** para reflejar su posición y el marco se marca como en estado normal.
8. La instrucción fallida se respalda al estado que tenía cuando empezó y se **reestablece el contador del programa** para apuntar a dicha instrucción.
9. El proceso fallido se planifica y **el sistema operativo regresa a la rutina** que lo llamó.
10. Esta rutina recarga los registros y demás información, **regresando al espacio de usuario** para continuar la ejecución como si nada.

3.7.2. Respaldo de instrucción

Cuando un programa hace referencia a una página que no está en memoria, la instrucción que produjo el fallo se detiene parcialmente y ocurre un trap al sistema operativo. Una vez que el sistema operativo obtiene la página necesaria, debe reiniciar la instrucción que produjo el trap.

Con frecuencia es imposible que el sistema operativo determine sin ambigüedad dónde empezó la instrucción. En general, en las CPUs hay un **registro interno oculto**, en el que se copia el contador de programa justo antes de ejecutar cada instrucción y se pueden tener otros registros que indiquen que registros se han modificado y cuánto. Con esta información el sistema operativo puede deshacer sin ningún problema todos los efectos de la instrucción fallida.

3.7.3. Bloqueo de páginas en memoria

En el caso de que un proceso emita una llamada al sistema para leer algún archivo o dispositivo y colocarlo en el buffer dentro de su espacio de direcciones, mientras espera a que se complete la E/S, el proceso se suspende y permite que otro proceso se ejecute. Si este nuevo proceso tiene un fallo de página existe una pequeña posibilidad de que la página que contiene el buffer de E/S sea seleccionada para irse de la memoria. Una solución a este problema es **bloquear** o **fijar** las páginas involucradas en operaciones de E/S en memoria, de manera que no se eliminen. Otra opción es enviar todas las operaciones de E/S a buffers del kernel y después copiar los datos a las páginas de usuario.

3.6.4. Almacén de respaldo

La forma más simple para asignar espacio de página en el disco es tener una **partición de intercambio** especial en el disco o en un disco separado del sistema operativo (para balancear la carga de E/S). Se utilizan números de bloque relativos al inicio de la partición.

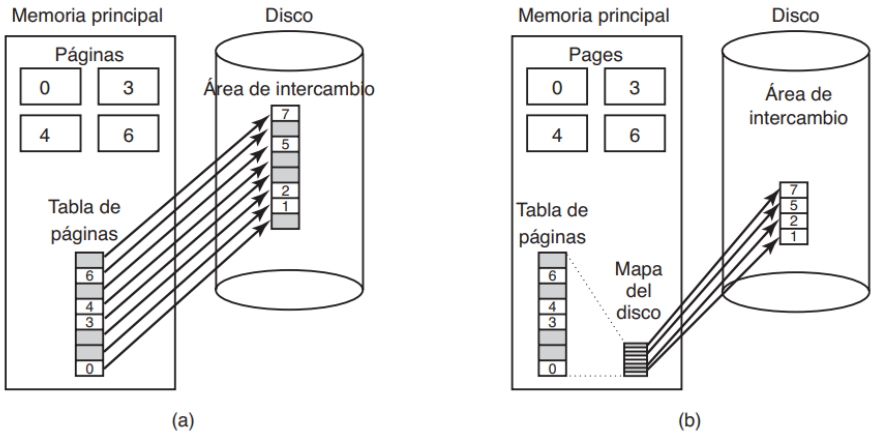
Cuando se inicia el sistema, esta partición de intercambio está vacía y se representa en memoria como una sola entrada que proporciona su origen y tamaño. En el esquema más simple, cuando

se inicia el primer proceso, se reserva un trozo del área de la partición del tamaño del primer proceso y se reduce el área restante por esa cantidad. A medida que se inician nuevos procesos, se les asigna trozos de la partición de intercambio con un tamaño equivalente al de sus imágenes de núcleo. Cuando terminan se libera su espacio en el disco. La partición de intercambio se administra como una **lista de trozos libres**.

En la tabla de procesos está la dirección de disco que tiene cada proceso asociada. De esta manera el cálculo de la dirección en la que se va a escribir una página es simple, de tal forma que solo se suma el desplazamiento de dentro de la página dentro del espacio de direcciones virtual al inicio del área de intercambio. Sin embargo, antes de que un proceso pueda empezar, se debe inicializar el área de intercambio. Una forma de hacerlo es copiar toda la imagen del proceso al área de intercambio, de manera que se pueda traer y colocar en la memoria según sea necesario. La otra es cargar todo el proceso en memoria y dejar que se **pagine hacia afuera** según se requiera.

Los procesos pueden incrementar su tamaño antes de empezar, por lo que puede ser mejor reservar áreas de intercambio separadas para el texto, datos y pila, permitiendo que cada una de estas áreas conste en más de un trozo en el disco.

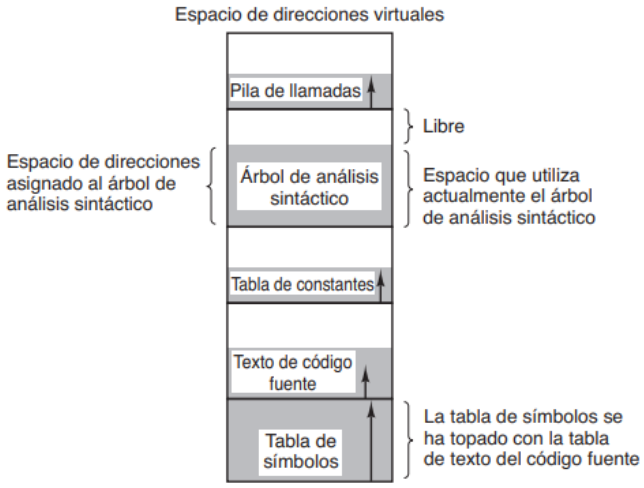
Otra opción es no asignar nada por adelantado y asignar espacio en el disco para cada página cuando esta se intercambie hacia fuera de la memoria y desasignarlo cuando se vuelva a intercambiar hacia la memoria. Lo malo de esto es que se necesita una dirección de disco en la memoria para llevar la cuenta de cada página en este, es decir, se debe hacer una tabla por cada proceso que indique en dónde se encuentra cada página en el disco.



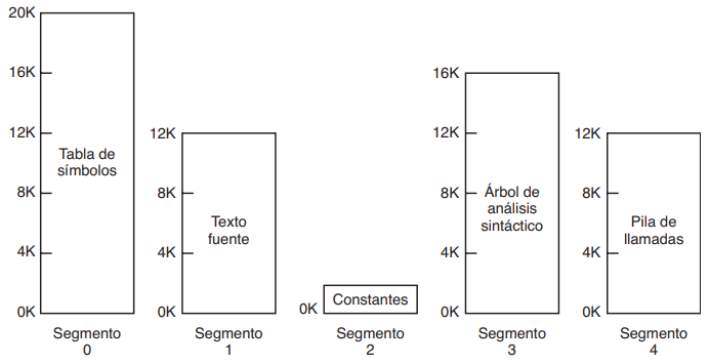
No siempre es posible tener una partición de intercambio fija, porque puede no haber particiones de disco disponibles. En este caso se pueden utilizar uno o más archivos previamente asignados dentro del sistema de archivos normal, como puede ser con los archivos ejecutables.

3.8. Segmentación

La **segmentación** se utiliza en lugar de las páginas y de esta forma se asignan segmentos a partes concretas del código y cada uno tiene todo el espacio virtual para él. Hay tantos espacios virtuales asignados a cada proceso como segmentos tenga.



Cada segmento puede tener un tamaño diferente y cada uno de ellos tiene sentido para el programa de forma que cada uno puede tener permisos diferentes y los segmentos se ubican enteros en la memoria principal, de forma que puede haber un segmento de todo código. Pero la memoria RAM es mucho más difícil de manejar porque ya no está dividida en páginas.

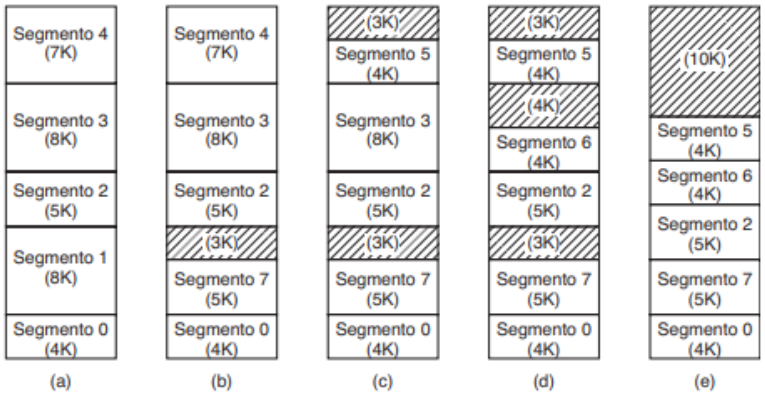


Dentro del segmento todo el espacio es direccionable y son independientes entre sí. De esta forma es más fácil la protección de segmentos y la compartición de ellos entre procesos, aunque no son comunes.

Las direcciones virtuales en este tipo de sistema tienen dos componentes, el número de segmento y el desplazamiento dentro de este. No van en la misma instrucción, lo que implica el uso de dos registros diferentes.

Consideración	Paginación	Segmentación
¿Necesita el programador estar consciente de que se está utilizando esta técnica?	No	Sí
¿Cuántos espacios de direcciones lineales hay?	1	Muchos
¿Puede el espacio de direcciones total exceder al tamaño de la memoria física?	Sí	Sí
¿Pueden los procedimientos y los datos diferenciarse y protegerse por separado?	No	Sí
¿Pueden las tablas cuyo tamaño fluctúa acomodarse con facilidad?	No	Sí
¿Se facilita la compartición de procedimientos entre usuarios?	No	Sí
¿Por qué se inventó esta técnica?	Para obtener un gran espacio de direcciones lineal sin tener que comprar más memoria física	Para permitir a los programas y datos dividirse en espacios de direcciones lógicamente independientes, ayudando a la compartición y la protección

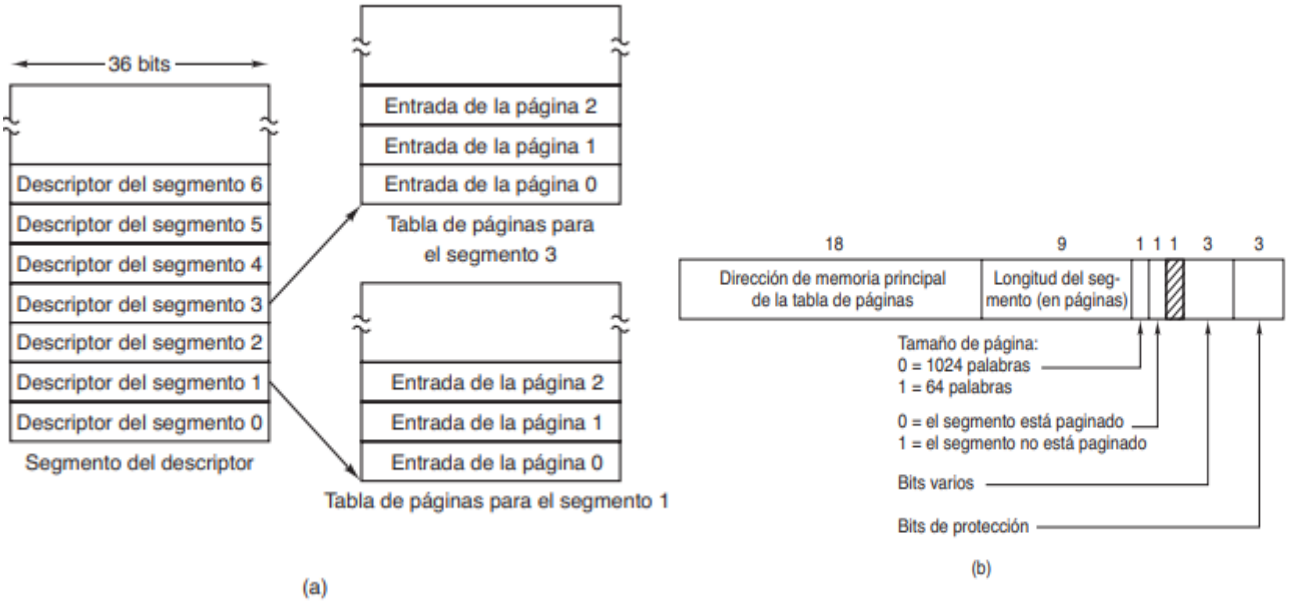
En la siguiente imagen se muestra un sistema segmentado a lo largo del tiempo que enseña los diferentes tamaños de los segmentos en la memoria principal. En estos sistemas se puede dar **fragmentación externa** (d) que deja muchos huecos muy pequeños inutilizables, de ahí el uso de la compactación (e) que es una operación muy costosa.



3.8.1. Segmentación con fragmentación

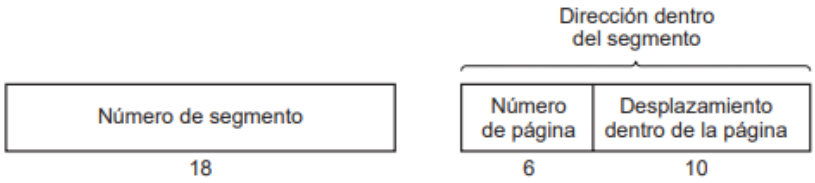
La idea es que los segmentos sigan siendo de tamaño variable pero que los tamaños sean múltiplos de bloques de tamaño fijo (páginas), de forma que los segmentos están formados por varias páginas.

De esta forma se necesita una **tabla de segmentos** por proceso situada en el kernel (a), en la que cada una de las entradas de esa tabla apunta a una posición de memoria donde está la tabla de páginas de ese segmento. Si cualquier página del segmento está en memoria, se considera que tanto el segmento como su tabla de páginas están en memoria. Es por ello por lo que un **fallo de segmento** se produce cuando la tabla de páginas del segmento no está en memoria.

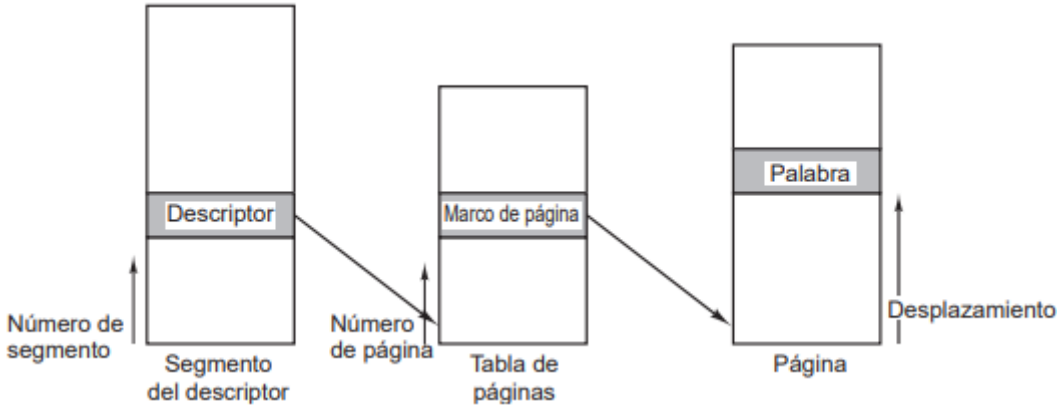


En cada entrada de la tabla de segmentos está la dirección de dónde empieza su tabla de páginas, el tamaño de dicho segmento en páginas y el resto son los mismos que los vistos para las páginas.

En la **traducción** en un sistema segmentado con paginación, las direcciones tienen los siguientes campos:



Con el número de segmento solo se accede a la tabla de segmentos y se comprueba si su tabla de páginas está en memoria. En caso de ser así, el segundo campo utiliza la primera parte de este segundo campo para posicionarse dentro de la tabla de páginas. A través de ella se obtiene la página sobre la que se va a desplazar lo que indique la segunda parte de dicho campo.



Tema 4

El Sistema de Archivos

En este tema se trata otra de las funciones abordadas por el SO: la **abstracción de sistemas de archivos**. Un **archivo** es una abstracción de los dispositivos físicos de almacenamiento, una unidad lógica de información creada por los procesos.

La parte del sistema operativo que trata con los archivos es el **sistema de archivos**.

El almacenamiento de la información en el disco duro es bastante complejo, el usuario no tiene que lidiar con esto.

4.1 Consideraciones Generales

El usuario necesita almacenar la información a largo plazo en archivos, por lo que se buscan mecanismos **no volátiles**, llamados dispositivos de almacenamiento secundario (ya que se supone que no se necesitan acceder constantemente). Gracias a ellos, tenemos garantías de que el contenido sea **persistente**, es decir, que no necesita energía para mantener los datos (cuando se apaga el ordenador, por ejemplo) y que por lo tanto no se pierde esa información. Un ejemplo sería el del disco duro.

4.1.1 Características

Los dispositivos de almacenamiento secundario necesitan almacenar mucha información, por lo que son dispositivos de alta capacidad. Son gestionados por un componente software (el driver correspondiente) y por un componente hardware (la controladora).

En el disco duro hay una zona dedicada al almacenamiento de todos los archivos y otra distinta que sirve de soporte a la ejecución de procesos (llamada zona de swapping o de intercambio). Dentro de la última encontramos, por ejemplo, las páginas que no están residentes en memoria principal.

El acceso a cualquier archivo o fichero (acciones como crear, modificar o borrar ficheros) se tiene que realizar mediante procesos. Si se quiere crear una carpeta en el entorno de ventanas, se debe crear un proceso para esa tarea. No se gestionan archivos si no es por medio de procesos. Varios procesos pueden trabajar a la vez con el mismo archivo, pero hay que tener cuidado de que no vaya incoherencias (las famosas carreras críticas).

Los procesos son programados con **llamadas al sistema para controlar estos ficheros** (las que se muestran más abajo). Esto implica un salto a alguna rutina del kernel del SO, dónde se ejecutará la acción.

Para trabajar con archivos en la línea de comandos o en el entorno de ventanas, lo que sucede es que se realiza un fork. Este crea y ejecuta un proceso para completar una operación específica (que invoque a alguna llamada al sistema, por ejemplo).

Algunas de las llamadas al sistema para archivos más comunes son:

- **Create**(“archivo.txt”, OUTPUT_MODE): Se crea el archivo sin datos. El propósito es anunciar su llegada y establecer ciertos atributos.
- **Open**(“archivo.txt”, O_RDONLY): Se abre el archivo (antes de usarlo). El sistema lleva los atributos y la lista de direcciones de disco a memoria principal.
- **Read**(variable_Archivo, buffer, BUFFER_SIZE): Se leen datos del archivo. Se deben especificar cuántos datos se necesitan y un buffer destinatario.
- **Write**(variable_Archivo, buffer, escritura): se escriben los datos en el archivo en la posición actual (si esta está en el medio del archivo, se sobreescriben).
- **Close**(variable_Archivo): Se cierra archivo (liberar memoria en la tabla interna)
- **Delete**( ): Se elimina el archivo para liberar espacio
- **Append**( ): Una forma restringida de write, agrega datos al final del archivo.
- **Seek**( ): Reposiciona el apuntador del archivo a una posición específica.
- **Get attributes**( ): Leen los atributos de un archivo.
- **Set attributes**( ): El usuario establece o crea atributos
- **Rename**( ): El usuario cambia el nombre existente de un archivo

4.1.2 Disco físico y disco lógico

La parte que gestiona los archivos en el SO se llama sistema de archivos. El SO tiene que hacer una abstracción entre la visión que ofrece de los archivos al usuario (como se estructuran, denominan, abren, protegen, implementan o administran) y la implementación a bajo nivel.

Un disco físico (hardware) es un dispositivo de almacenamiento permanente. Está organizado en modo bloque, por lo que la información se almacena en bloques de un tamaño fijo. El tamaño de bloque puede variar y muchas veces coincide con el tamaño de las páginas (o las páginas son del tamaño de un par de bloques). Cada bloque tiene un numero entero que lo identifica.

El disco lógico es una abstracción del SO del disco físico. En él, la visión de todos los bloques es una secuencia de bloques a la que se puede acceder de manera aleatoria (no consecutiva). El driver de gestión de los archivos es capaz de traducir los números de bloque lógico a números del bloque físico.

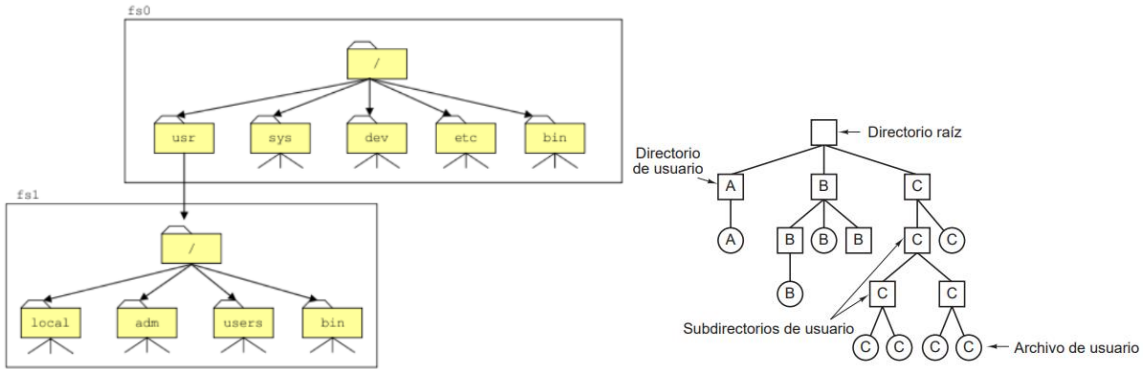
Se puede tener un disco duro físico dividido en particiones y determinar que cada una de esas es un disco lógico (desde el punto de vista lógico, tendríamos tantos discos como particiones). Un ejemplo de esto es tener en un mismo ordenador un SO Linux y otro Windows, y que en el momento de arranque se decida a cuál acceder.

Se pueden tener varios discos físicos y unirlos en un solo disco lógico.

4.1.3 Montaje de Sistemas de Archivos

Los archivos se estructuran en **árbol**. Arriba del todo se encuentra el directorio raíz (/), con varios directorios hijos (por ejemplo /usr) que a su vez pueden tener más hijos. Las hojas son los ficheros. Los nodos intermedios son directorios (que solo tienen como cometido estructurar el árbol).

Se pueden crear vínculos entre dos directorios o ficheros, permitiendo tener un fichero en dos sitios diferentes (por ejemplo, tenemos un fichero en un directorio y lo copiamos al acceso directo), pero esto causaría perder esta estructura de árbol.



Puede haber varios discos lógicos (fs1 y fs0) y unificarlos en una sola estructura de archivos. Para esto se utiliza la operación **montar un disco** (**mount(dispositivo, dir, flags)**). En la imagen se ve como fs1 es parte de fs0 (la estructura de fs1 está en la de fs0). La operación contraria sería desmontar un disco (**unmount(dispositivo)**).

Con el comando /etc/fstab se pueden ver todas las particiones y los montajes de ficheros, así como sus nombres.

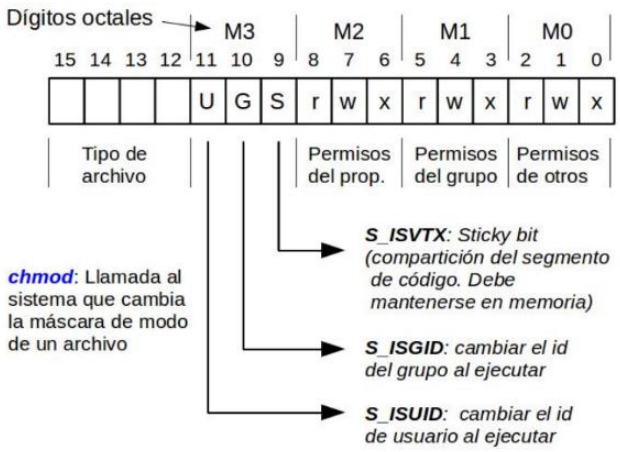
Para hacer referencia al directorio de trabajo se usa el símbolo '.'. Cuando nos movemos en la estructura jerárquica utilizando esta nomenclatura, lo que estamos haciendo es movernos dentro de un proceso (si es un comando interno se ejecuta en el propio Shell y si no se crea un proceso nuevo, esto cobra más importancia en SOII).

Todos los procesos tienen un directorio de trabajo asociado, que es donde están los ficheros con los que trabaja. Dentro de ese proceso, el nombre del directorio de trabajo puede ser el absoluto (el que se muestra usando el comando /etc/fstab) o una referencia relativa a este (./lo que sea). Usando '../' estamos haciendo referencia al directorio padre.

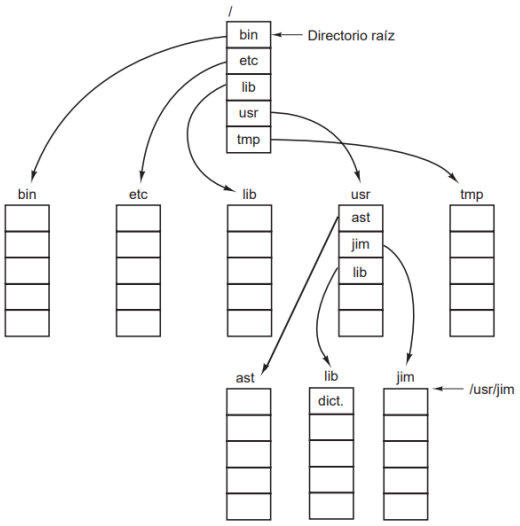
4.1.4 Los archivos desde el Punto de Vista del Usuario

Cuando creamos un archivo, este tiene asociados varios permisos que hacen referencia a qué tipo de accesos permitimos al fichero por parte de los distintos tipos usuarios (ya que hablamos de un sistema multiusuario). Se pueden conceder permisos de **lectura** (r), **escritura** (w) y **ejecución** (x). Todo esto tiene la funcionalidad de proporcionar una mayor seguridad y control.

Cada archivo tiene una máscara de 16 bits (llamada la **máscara de modo**), de los cuales 9 son para los permisos: 3 bits (M2) para los permisos del propietario (si el archivo es mío, pues para mi), 3 (M1) del grupo (un sistema UNIX puede organizar a sus usuarios en grupos) y 3 (M0) de otros (resto de usuarios ajenos a mi grupo). El único que puede cambiar estos permisos es el propietario, usando el comando o llamada al sistema **chmod**.



Los 3 bits de M3 (el 9, 10 y 11) hacen referencia a lo que se puede hacer con un fichero ejecutable (los generados al compilar el código y no son legibles por un editor de texto, ya que están en binario). En otras palabras, que tipo de permisos se pueden asumir o utilizar cuando ese fichero ejecutable pasa a convertirse en un proceso. Los bits 11 y 10 corresponden al **S_ISUID** (para usuario) y **S_ISGID** (para grupos). Esto indican si se permite cambiar el identificativo del usuario o grupo que va a ejecutar el proceso (en otras palabras, coger un ejecutable que no sea mío, y cuando lo ejecuto cambiar ese identificativo, "haciéndonos pasar por el propietario"). El último (el bit 9) es el **sticky bit**, para directorios. Permite que el usuario que ejecute el código pueda entrar en los ficheros del directorio.



Un directorio (como el de la imagen) es un fichero dónde se almacena una lista los nombres de todos sus hijos (da igual que sean ficheros o directorios). El SO tiene información de dónde físicamente están cada uno de esos ficheros, para acceder a ellos en disco duro.

3.1. Identificadores de Usuario y Grupo

Cuando se lanza a ejecutar un proceso, se hace invocando el nombre del fichero ejecutable (con sus permisos asociados). En este fichero hay información sobre el código, constantes, variables, zonas de memoria del stack y heap.

Cuando se lanza a ejecutar un proceso se pueden ver dos identificadores. Por un lado, el **usuario real (uid)**, que corresponde a quién lanza el proceso. Por otro lado, el **usuario efectivo (euid)** es

el que tiene los permisos para acceder a los ficheros que utilice este proceso, en otras palabras, el propietario de los ficheros y sus accesos.

De forma general, cuando ejecutamos un proceso (y por lo tanto nos convertimos en usuarios reales), solo se pueden abrir y trabajar con aquellos ficheros a los que tenga acceso al usuario efectivo. El usuario efectivo suele coincidir con el real (aunque no tiene por qué).

Lo mismo ocurre para grupos. Un **grupo real (gid)** es al que pertenezco yo como ejecutor del proceso. Un **grupo efectivo (egid)**, es al que pertenezco en cuanto a los accesos a los archivos que necesite ese proceso.

4.2.1 ¿Qué ocurre cuando el usuario real y el usuario efectivo no coinciden?

El usuario efectivo se establece en base de los que valga el bit S_ISUID de la máscara de modo del archivo activo. El bit S_SIGID es el equivalente, pero para grupos.

Por ejemplo, tenemos dos usuarios (U1 y U2), y un programa (fichero) llamado P que pertenece a U2 pero que quiere ejecutar U1. En este caso, el dueño de P (U2) puede establecer a través de este bit (activándolo o desactivándolo) si quien ejecute el programa puede cambiar el usuario efectivo o no. Si está activado, U1 podrá cambiarlo con éxito.

Tenemos las siguientes funciones:

- getuid / getgid: obtiene el usuario o grupo real
- geteuid / geteuid: obtiene el usuario o grupo efectivo
- setuid(par): cambia el nuevo usuario efectivo
- setgid(par): cambia el nuevo grupo efectivo

Normalmente intenta cambiar el usuario efectivo (euid) porque se quiere que el programa que se ha lanzado trabaje con mis ficheros.

Dada la función y el ejemplo de: **salida = setuid(par)**, siendo par el numero entero identificativo del usuario, hay dos posibles salidas: 0, si hubo éxito (a partir de ese momento si se hace geteuid se obtiene el valor de par) o -1 si hubo algún error.

Al ejecutar esa operación, pueden ocurrir distintas situaciones que desencadenan una salida o la otra:

- Si el usuario efectivo del proceso es un **superusuario** (en Linux hay un usuario que es root, que se salta todos los permisos de acceso de los ficheros, aunque rwx no este activado). Por tanto, independientemente de lo que valga el bit S_ISUID, como tiene acceso a todos los archivos, el resultado de la operación será 0 (se realiza con éxito).
- Si el usuario del proceso no es un superusuario (es un **usuario normal**). A veces se le permitirá y otras no. Cuando par sea yo (es decir, yo quiero acceder a mis archivos), la respuesta será afirmativa (aunque el bit S_ISUID este desactivado). Sin embargo, si se quiere poner en el argumento otro usuario distinto al mío (impar, por ejemplo), solo se permitirá si el bit está activado por el dueño del programa.

```
#include <fcntl.h>

main(){
int x, y, fd1,fd2;

x=getuid(); y=geteuid();
printf("\nUID=%d, EUID=%d\n", x, y);

fd1=open("fichero1.txt", O_RDONLY);
fd2=open("fichero2.txt", O_RDONLY);
printf("fd1=%d, fd2=%d\n", fd1, fd2);

setuid(x);
printf("UID=%d, EUID=%d\n",getuid(),geteuid());

fd1=open("fichero1.txt", O_RDONLY);
fd2=open("fichero2.txt", O_RDONLY);
printf("fd1=%d, fd2=%d\n", fd1, fd2);

setuid(y);
printf("UID=%d, EUID=%d\n",getuid(),geteuid());}
```

Tenemos dos usuarios (U1 y U2), siendo U1 el dueño del programa a ejecutar. Además, hay dos ficheros (fd1 y fd2). Uno pertenece al U1 (usuario 1) y otro al U2. Pueden darse varios casos:

1. El U1 ejecuta el programa. Tanto el usuario real como el efectivo coinciden, así que setuid siempre se cumple con éxito (los valores de uid y euid continúan como estaban). En ambos casos se puede acceder a fd1 (el suyo), pero no a fd2 porque no es suyo (de ahí que fd2 = -1).
2. El U2 ejecuta el programa (que al ser de U1 será este el usuario efectivo). Como el usuario efectivo es U1, solo se podrá acceder a la información guardada en fd1. No obstante, cuando se cambia el usuario efectivo a U2, ahora se podrán acceder a los datos del fd2 (pero ya no a los de fd1).
3. El U2 ejecuta el programa, pero el bit que permite el cambio de usuario efectivo (S_ISUID) está desactivado. Asi que, U2 solo podrá acceder a fd2 y no a fd1 en ningún momento. Cuando se quiera ejecutar el setuid este no tendrá éxito. Esto funciona como medida de seguridad para que usuarios distintos de U1 accedan a sus ficheros.

3.2. Implementación del Sistema de Archivos

Recordatorio: El sistema de archivos es la parte del SO que consigue hacer la abstracción del sistema de archivos en una estructura muy sencilla en forma de árbol.

Recordatorio x2: Un disco duro es la estructura de almacenamiento secundario más habitual. Está organizado en bloques llamados sectores de disco.

Hay dos problemas que debe resolver. Por un lado, cómo almacenar archivos y directorios dentro del disco duro (es decir la forma de organizar el contenido de un fichero en los distintos sectores). Por otro, cómo administrar el espacio en disco, en particular los sectores vacíos. Todo esto debe resolverse de forma eficiente y fiable.

A continuación, se presentan las distintas opciones de distribución de los bloques de un archivo.

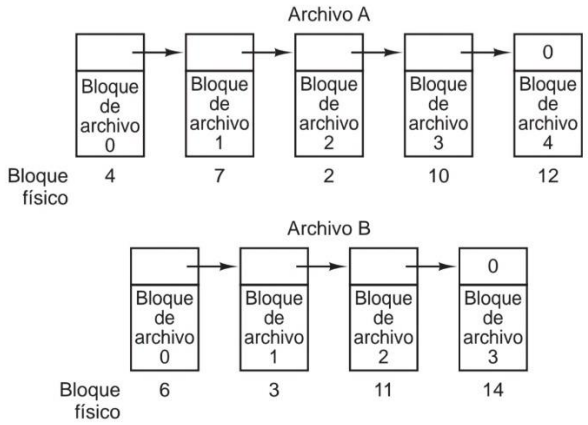
4.3.1 Lista enlazada

Una posible solución es la **lista enlazada**. Esta solución es muy mala y con muchos inconvenientes.

Tenemos un bloque (el bloque 4 para el archivo A, por ejemplo) dónde empieza a almacenarse el contenido del fichero. Si el tamaño de ese bloque no es suficiente para guardar toda la información, se necesitará un segundo para seguir almacenando. En la lista enlazada se guarda un puntero dentro del propio bloque (en la primera palabra) al siguiente bloque que contiene el siguiente trozo de archivo. En caso de ser el final se pone null.

La elección de en qué bloque se almacenará el siguiente trozo de archivo es arbitraria (no tienen por qué ser sectores consecutivos en disco).

Un archivo está caracterizado por el primero de los bloques (el bloque 6 para el archivo B, por ejemplo), porque a partir de ahí se siguen los punteros para ir recorriendo el archivo hasta el final, pasando de sector a sector dentro del disco duro.



Ventaja: no hay fragmentación externa, ya que el disco duro está organizado en sectores (por lo que solo puede haber sectores completamente vacíos útiles para almacenar), por lo que es fácil la reutilización de bloques (ya que no deben ser consecutivos).

Inconvenientes:

Sí hay **segmentación interna**. Seguramente ocurrirá en el último sector del archivo, ya que una parte del espacio disponible quedará inutilizado porque este no es ocupado por completo (salvo que el tamaño del archivo coincida con un múltiplo del tamaño de los bloques).

Acceso muy lento. Para leer un dato de un archivo, hay que ir recorriendo todo ese archivo (todos los bloques) hasta encontrar el dato. El disco duro tiene mucho trabajo y la lectura ahí es muy lenta. Para conseguir llegar al bloque n, se debe pasar por todos los anteriores (n-1).

La cantidad de almacenamiento por bloque no es potencia de 2. Los discos duros se fabrican de tal manera que los sectores sean potencias enteras de 2, pero al guardar en ellos el puntero al siguiente sector, el almacenamiento disponible restante y disponible para datos ya no será potencia de 2.

Para leer un bloque de datos (una página del espacio virtual) hay que acceder a dos bloques de disco (o sectores). Consecuencia directa de lo anterior. Las páginas tienen tamaño potencia entera de 2, pero si un sector ahora ya no es potencia de 2 no tiene suficiente tamaño y para leer una página necesito ir a dos sectores.

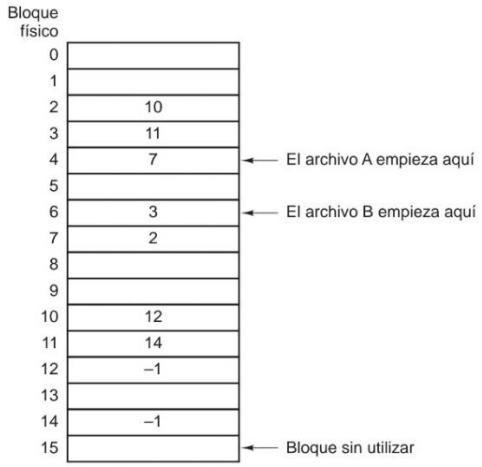
4.3.2 Tablas FAT

Otra posible solución son las tablas **FAT** (File Allocation Table). Una solución más adecuada que la anterior. La utiliza Windows, por ejemplo.

Consiste en construir una **tabla** parecida a la de la imagen (esa es de tamaño muy reducido), con **tantas entradas como bloques o sectores haya de disco**. Aquí se almacena la lista de sectores para cada archivo. Con esto se consigue almacenar la lista de punteros **en memoria física** (en la RAM, en la zona de kernel del sistema operativo) y no en disco duro.

Por ejemplo, un archivo (A) se identifica con el sector donde empieza (el 4). En la entrada 4 se pone un valor (7) para saber dónde sigue el archivo. De la 7 pasamos a la 2, luego a la 10 y a la 12. En la última hay un -1, señalando el final del archivo. Lo mismo para el fichero B.

Si solo ocupa un sector, ya en la primera entrada correspondiente se pondría un -1.



Ventajas: ahora los bloques de disco duro sí ocupan potencias enteras de 2 (porque están completamente dedicadas a datos). Soluciona el problema de las páginas. Además, el acceso aleatorio es más rápido, ya que toda esta lista está en memoria y no en disco.

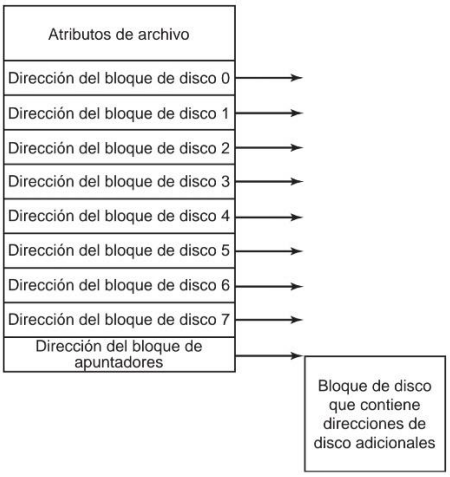
Desventajas: mala escalabilidad. La tabla puede ser muy grande (depende del número de sectores de disco, que suele ser muy elevado) y ocupar mucho espacio en memoria principal.

4.3.3 Nodos-i

Última solución, habitual en todos los sistemas UNIX.

Un **nodo-i** es una estructura de información asociada a los ficheros (cada fichero tiene su propio nodo-i). Se almacenan **en memoria principal**, en la zona de kernel. En esa estructura hay (tal y como se ve en la imagen): información sobre el fichero (permisos, en qué directorio está, …) y una lista de bloques en disco dónde está ese fichero por orden (desde el primero hasta el último, en el ejemplo habría 8 entradas).

Son de **tamaño fijo**, es decir, tienen un número fijo de apuntadores, (en este caso 8 bloques). Si el archivo necesita más hay un puntero a memoria que lleva a otro sitio dónde sigue la lista. La mayor parte de los ficheros cabrán en el espacio para el nodo.



Ventajas: la escalabilidad. Hay tantos nodos-i como archivos (tamaño por lo general menor que en la solución anterior). Como la cantidad de archivos suele ser bastante menor a la cantidad de sectores, el espacio de almacenamiento es menor. El nodo-i está en memoria solo cuando el archivo está abierto.

Pregunta posible para el examen: dada una tabla FAT, hacer el equivalente con nodos-i (o al revés).

3.3. Administración y Optimización del Sistema de Archivos

Varios aspectos de la administración del espacio de disco.

El primero: cómo se almacena un archivo en disco.

Lo habitual es dividirlo en bloques (no tienen por qué ser contiguos). Pero también pueden almacenarse en bytes consecutivos en disco, asumiendo que un archivo empieza en una determinada posición del disco y se va almacenando en sectores consecutivos desde el principio hasta el final. Esto tiene muchos inconvenientes y es poco eficiente. Tiene alguna ventaja: el acceso a archivos enteros es más rápido (lee el fichero desde el principio hasta el final, no tiene que ir saltando a posiciones aleatorias dentro del disco). Ningún SO actual usa esta segunda opción.

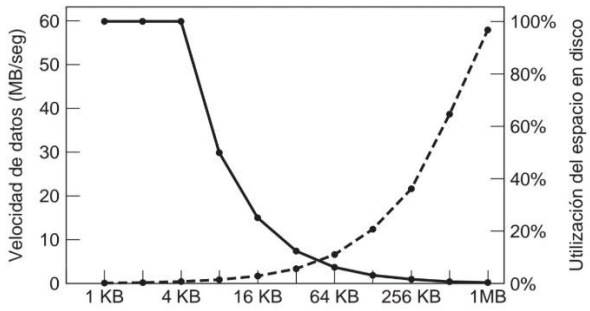
Segundo: decidir el tamaño de los bloques.

En muchos sistemas, el tamaño de bloque coincide con los sectores (no tiene porqué, se puede definir que el tamaño de bloque para los archivos sea múltiplo del tamaño de los sectores).

Si decidimos usar bloques grandes hay 1 problema. Cuánto más grandes sean, más se desperdicia el espacio (el último bloque de los archivos es raro que se complete, por lo que el trozo de bloque desaprovechado puede ser muy grande).

Si los bloques son muy pequeños se desperdicia menos espacio, pero el acceso es más lento. Hay que saltar a más bloques para el mismo tamaño de archivo. Los saltos entre bloques implican mayor número de movimientos (más búsquedas y rotaciones).

Hay que buscar una solución de compromiso entre estas dos, un balanceo en el tamaño del bloque para minimizar los inconvenientes y maximizar cada una de las ventajas. Con bloques pequeños el tiempo de acceso se hace más lento, y en grandes el desaprovechamiento de disco es mayor.



En la imagen se muestra un caso particular, que ilustra el compromiso entre velocidad y eficiencia (es decir, aprovechamiento de disco). Todos los ficheros ocupan lo mismo, 4KB cada uno. En el eje X se representa el tamaño de bloque. La línea continua hace referencia a la escala del grado de utilización de espacio de disco (aquí se tiene en cuenta que el último bloque suele quedar desaprovechado). La línea discontinua muestra la velocidad (MB/s), que crece de manera exponencial.

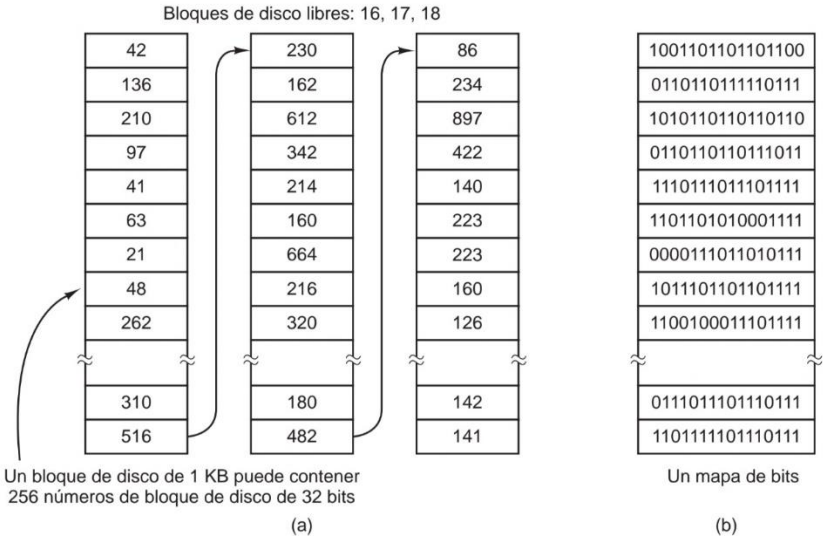
Si el tamaño de bloque es 1KB, para almacenar un archivo necesito 4 bloques y, en este caso, no se desaprovecha nada (4KB es múltiplo de 1KB). Tenemos un 100% de aprovechamiento de disco duro. Si ahora el tamaño bloque es 8KB, el aprovechamiento es del 50% y si subimos a 16KB es del 25%. Ya en el otro extremo, con 1MB casi el 99% se desaprovecha. Habría que buscar un tamaño que sea lo más equitativo posible.

4.4.1 Registro de Bloques Libres

Se deben conocer los sectores que están libres y se pueden utilizar en cada momento. Hay dos formas.

La **lista enlazada** (a): se forma una lista dónde se van almacenando los sectores que están libres. Estas tablas tienen un tamaño fijo, seguramente este coincide con el tamaño de las páginas. Si hay más espacios libres que los que caben en una de una tabla, un puntero al final de esta conduce a otra tabla que sigue la secuencia. Los sectores no están ordenados, son arbitrarios.

El **mapa de bits** (b): es un vector de bits organizado en palabras, que almacena 0 y 1, indicando si un sector está libre u ocupado. Por ejemplo, en la imagen (dando por hecho que el 1 señala que el sector está ocupado, cosa que no tiene por qué ser así, esto depende de cada sistema) están ocupados los sectores 0, 3, 4, 6, … y los 1, 2, 5, … libres.



No está claro cuál de las dos soluciones es la mejor, se usa la que menos memoria ocupe dependiendo de la situación concreta. Tanto una opción como la otra se almacenan en **memoria principal**. Mientras la lista enlazada será tan grande como el número de sectores libres que haya (cambiando a través del tiempo), el mapa de bits necesitará un bit por sector (dado un disco concreto, su **tamaño** es una **constante** invariable).

En casos particulares: si el disco está lleno, la lista enlazada necesita menos bloques. Por el contrario, si está vacío, será el mapa de bits el mejor candidato.

Alternativa a la lista encadenada: aprovecharse de que haya **bloques libres** en el disco **consecutivos**. Lo que se hace en ese caso es decir dónde empieza esa secuencia de bloques libres y cuántos son, todo en una sola entrada. Ahora cada entrada almacena dos datos, así que solo será mejor que la lista enlazada cuándo haya muchas secuencias de sectores libres consecutivos.

Tema 5

Entrada / Salida

La **entrada / salida** es otra gran abstracción del SO. Estas operaciones para un programador son muy sencillas y se basan o bien en librerías de funciones en alto nivel o en syscall (o traps) en nivel ensamblador. No dejan de ser **llamadas al sistema**, cuyo código se encuentra en el kernel del sistema operativo.

5.1 Hardware de Entrada / Salida

Las tres acciones más importantes que debe hacer el SO con relación a los dispositivos de E/S son las siguientes:

1. **Emitir comandos**: a través de syscalls o llamadas al sistema para realizar las operaciones de E/S. Para ejecutar estas llamadas al sistema se va a un planificador y en función del tipo de syscall, salta a una rutina concreta (código asociado a esa operación concreta) para ejecutarla. Esos códigos asociados a esas llamadas al sistema para realizar operaciones E/S para muchos dispositivos se llaman **drivers** del dispositivo y son códigos específicos. Convierten ese código de propósito general (printf) en código concreto sobre el dispositivo sobre el que se actuará.
2. **Captar o manejar las interrupciones** asociadas a operaciones de E/S.
3. **Manejar errores**: ver qué pasa cuando hay un error y como lo maneja.

La abstracción que ofrece el sistema operativo al usuario es una interfaz simple entre los dispositivos y el sistema, tan simple que es igual para todos los dispositivos (se programan todos los dispositivos igual, independientemente de sus características). Son vistos por el programador de la misma manera (a pesar de ser muy diferentes).

5.1.1. Tipos de dispositivos

- **Dispositivos de bloque:** aquellos en los que la información que entra o sale, respecto al procesador, está organizada en bloques de tamaño fijo, como es el disco duro, y se leen bloques enteros del dispositivo. Cada bloque tiene su propia dirección, se leen y escriben de forma independiente del resto y se puede hacer la transferencia de uno o más bloques consecutivos.
- **Dispositivos de carácter:** se basan en una especie de streaming. Se lee la información de E/S carácter a carácter (que suele ser la unidad mínima, un byte). Estos dispositivos pueden ser ratones, impresoras o similares. No son direccionables y no tienen operaciones de búsqueda.

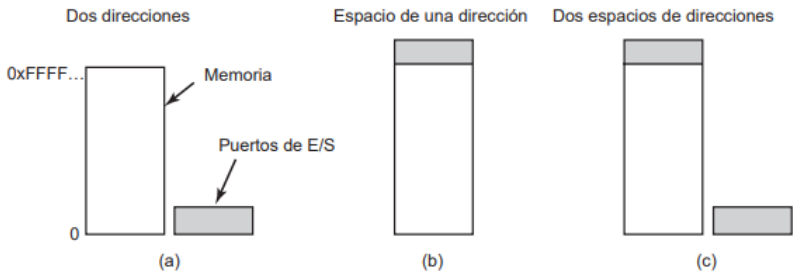
Hay dispositivos de entrada/salida que no se clasifican en ninguno de estos, como el reloj del sistema, que genera una señal cuadrada con una frecuencia. Al reloj no se le envían ni devuelve datos, lo único que hace es marcar la frecuencia con la cual van actuando los diferentes elementos hardware del sistema. El reloj emite interrupciones (interrupciones de reloj, cuando acaba un quantum de uso de un proceso en la CPU, se recibe una interrupción de reloj). El reloj es gestionado por el sistema operativo como si fuera un dispositivo de E/S más.

5.2. Comunicación Controladora – CPU

La controladora tiene registros que se utilizan para comunicarse con la CPU. De esta forma, al escribir en los registros el sistema operativo envía órdenes y cuando los lee puede conocer el

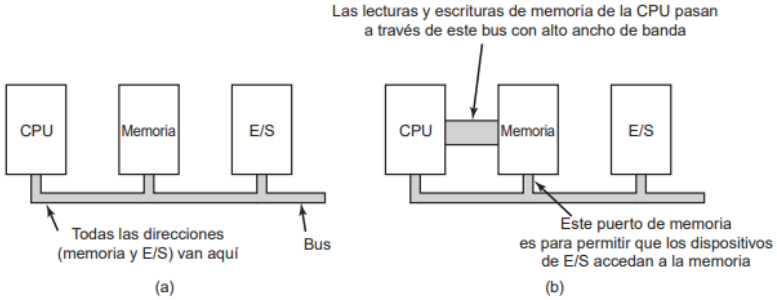
estado del dispositivo al que corresponde dicho registro o el valor de un dato solicitado, aunque esos registros suelen ser buffers.

Hay dos tipos de **formas de comunicar** a la CPU con los registros de control y buffers de datos:



- **Puertos de E/S** (a)**:** se utilizan dos direcciones diferentes de forma que hay una para la memoria del proceso y otra para los puertos de E/S, que son los registros y buffers de las controladoras.
  Las operaciones de lectura y escritura en los puertos no se pueden hacer con operaciones como *lw* y *sw*, porque son instrucciones de acceso a memoria. Es por eso por lo que hay que utilizar **instrucciones especiales** para estas operaciones, como IN y OUT, que mueven información entre registros de la CPU y registros/puertos de la controladora.
  En esta forma de comunicación, la CPU debe colocar a través del bus los valores adecuados en los registros para que la controladora sepa que operación hacer y en qué dirección hacerla, además de un bit adicional para especificar si esa operación es para puertos o para la memoria física.

- **E/S con asignación de memoria** (b)**:** en el espacio virtual del proceso se reservan posiciones para que sirvan de direccionamiento de los registros de las controladoras. Son direcciones físicas dentro del espacio virtual del proceso y en esas direcciones no hay memoria asignada.
  Para las operaciones de lectura y escritura se utilizan las instrucciones *lw* y *sw*, ya que forman parte del espacio de direcciones del proceso. De esta forma la CPU lanza la dirección a través del bus y ya se sabe a qué dispositivo va enfocada.
  Cuando se utiliza un bus extra entra la CPU y la RAM, en la CPU se deben diferenciar instrucciones que van a la memoria de las que van a los dispositivos, para saber en qué bus mandarlas.



Si llega a caché un registro de control pueden surgir problemas porque al estar en caché, las siguientes comprobaciones que se pueden hacer sobre el puerto se leen de la caché sin preguntar al dispositivo y en caso de haber cambios en él, el sistema operativo no lo notará. Es por ello por lo que se hace la **deshabilitación selectiva de cache** por páginas, hacer los datos no-cacheables.

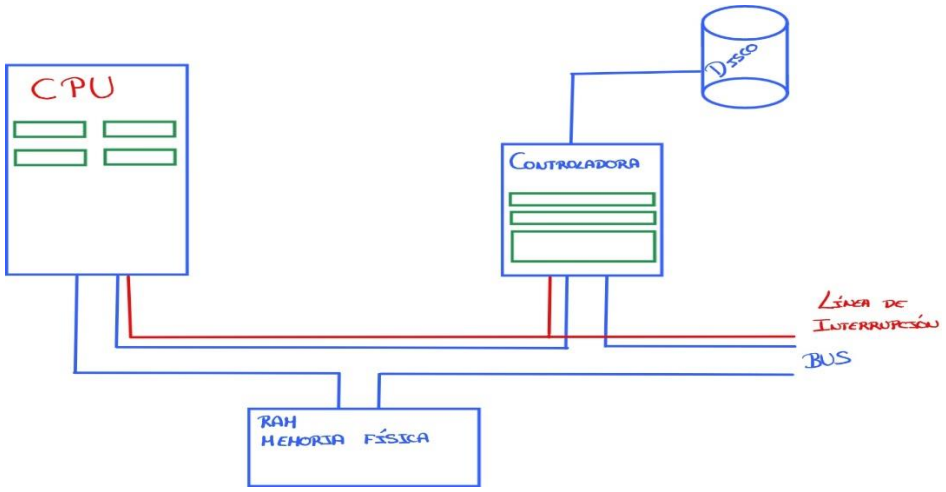
- **Esquema híbrido:** permiten la mezcla de las dos opciones anteriores (c), de forma que dividen las direcciones de los buffers en el espacio virtual del proceso y las de los registros en la otra parte.

5.2.1. Ejemplo de controladora del disco duro

El procesador o CPU se conecta a través del bus con las diferentes controladoras de los dispositivos, que en este ejemplo solo es la controladora del disco duro. Las controladoras son hardware electrónico en el que hay un registro y un buffer.

La línea en rojo es la línea de interrupciones del bus y en verde están los diferentes registros del sistema.

Las operaciones de E/S se basan en mover información entre los registros de la CPU y los registros de la controladora. Cada controladora puede tener diferentes buffers y registros (no todas los mismos, porque tienen diferentes usos).



La controladora tiene **registros para comunicarse con la CPU**. El sistema operativo, para realizar una operación de E/S, debe escribir en esos registros la información concreta a transmitir. Por ejemplo, en la controladora del disco duro puede haber un registro en el que se especifique, con un bit, el tipo de operación que se quiere hacer (leer o escribir) y ese valor es escrito por la CPU en él.

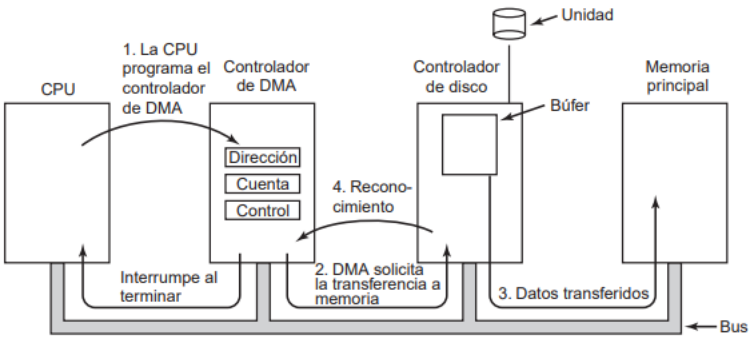
Los **registros de estado** indican que está pasando con el disco duro: si está encendido, apagado, preparado para una nueva solicitud, procesando una previa, …. La CPU lee el valor de ese registro y obra en consecuencia.

Si en el disco duro la operación es de entrada (como puede ser la lectura de un sector) debe haber un registro en la controladora para especificar el sector involucrado en la operación. Puede haber otro registro para indicar si se va a operar en un sector o en varios consecutivos.

La controladora tiene registros para bits muy concretos como los mencionados y en algunas controladoras, como la del disco, hay un buffer donde guardar lo que viene del disco y que está esperando a ser leído.

5.2.2. Acceso Directo a Memoria (DMA)

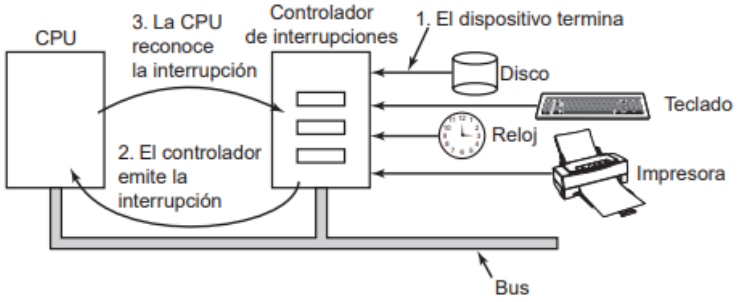
Un **DMA** libera a la CPU de muchas labores poco sofisticadas de las operaciones de E/S. La DMA es una controladora con sus propios registros, cuyo trabajo consiste en realizar el trabajo simple de la gestión de la E/S sin involucrar a la CPU en ello.



La CPU le dice lo que quiere hacer a la DMA, colocando la información en los registros de la DMA, diciendo que controladora o dispositivo es el involucrado, qué operación se va a hacer, cuántos datos se van a mover y en qué posición de memoria principal se va a utilizar para dicha operación.

La controladora es la encargada de acceder al puerto y esperar a que este realice la operación indicada. Cuando esta está lista, se avisa a la DMA y esta se encarga de transferir los datos del buffer de la controladora a la memoria principal. Es posible que esto lo tenga que hacer varias veces debido al tamaño de los datos obtenidos de la operación de E/S y al tamaño del bus.

Una vez completada la operación, la DMA se encarga de notificárselo a la CPU con una interrupción, que la CPU debe captarla y manejarla con el manejador de interrupciones y el vector de estas (2.4.1).



La CPU es más rápida que la DMA, pero de esta forma se permite que la CPU se dedique a otras cosas mientras se realiza la operación de E/S. Normalmente el bus da problemas porque al haber uno solo, ahora la CPU y la DMA compiten por el uso del bus.

Actualmente las DMAs son capaces de gestionar varias transferencias al mismo tiempo y pueden operar con el bus en dos formas:

- **Modo palabra:** compite con la CPU por el acceso al bus, robando ciclos de bus a la CPU para hacer las transferencias de datos que necesita.
- **Modo bloque:** la DMA adquiere el bus, emite la transferencia y libera el bus, como en una ráfaga. Es mucho más eficiente que el robo de ciclo, pero puede bloquear a la CPU.

La mayoría de las DMA utilizan direcciones físicas, ahorrando el trabajo de las traducciones. Los sistemas cuya CPU tiene poco trabajo normalmente no utilizan la DMA para evitar que la CPU esté largos periodos sin trabajar y es más rápida que la DMA.

5.3. Software de E/S

Hay dos tipos de transferencias de E/S:

- **Síncronas** que implican un bloqueo del proceso involucrado.
- **Asíncronas**, que son controladas con interrupciones. El sistema operativo hace que este tipo de operaciones parezcan de bloqueo para los usuarios.

La mayoría suelen ser asíncronas de forma que la CPU inicia la transferencia ya tiende a otro proceso hasta que llega la interrupción.

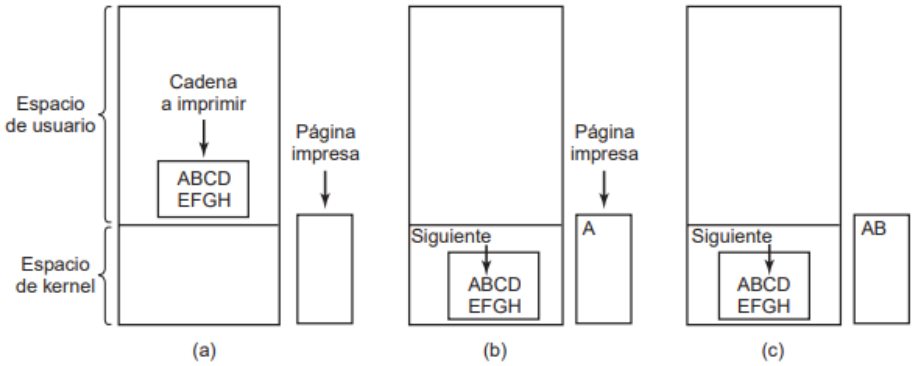
En todas se utiliza un buffer de almacenamiento temporal, en la zona de kernel, hasta que se llevan los datos a su destino final, que puede llegar a conocerse solo después de haber examinado lo que tiene el buffer.

Hay tres formas distintas de llevar a cabo las E/S

- **E/S Programada:** la CPU es la encargada de realizar toda la operación y no se utilizan interrupciones, de forma que se ocupa la CPU para toda la operación y no se bloquea al proceso que la causó. El código y la foto de abajo muestran un ejemplo del funcionamiento de una llamada al sistema para imprimir una serie de datos.

```
copiar_del_usuario(bufer, p, cuenta);                /* p es el búfer del kernel */
for (i=0; i<cuenta; i++) {                           /* itera en cada carácter */
        while (*reg_estado_impresora != READY);      /* itera hasta que esté lista */
        *registro_datos_impresora = p[i];            /* imprime un carácter */
}
regresar_al_usuario();
```

  Con este código se copia carácter a carácter al registro de datos de la controladora y después de cada uno, el sistema operativo comprueba si la impresora está lista, controlando el registro de estado.



- **E/S Controlada por interrupciones:** la CPU realiza la transferencia y se planifica algún otro proceso. De esta forma, cuando el dispositivo termina genera una interrupción que obliga a detener al proceso actual para que sea atendida por la CPU.
  En (a) está el código que se ejecuta e la CPU al momento en que se hace una llamada al sistema para imprimir y en (b) como se procede con las interrupciones del dispositivo. Con este código se recibe una interrupción por carácter enviado.

```
copiar_del_usuario(bufer, p, cuenta);
habilitar_interrupciones();
while (*reg_estado_impresora != READY);
*registro_datos_impresora = p[0];
planificador();
```

(a)

```
if (cuenta==0) {
  desbloquear_usuario();
} else {
  *registro_datos_impresora = p[i];
  cuenta=cuenta – 1;
  i = i + 1;
}
reconocer_interrupcion();
regresar_de_interrupcion();
```

(b)

Este método implica muchas interrupciones con los consecuentes cambios de contexto

- **E/S mediante DMA:** el DMA realiza la operación de E/S sin molestar a la CPU. Es una E/S programada pero ahora es la controladora DMA quien hace el trabajo y no la CPU. A la izquierda se muestra el código que debe ejecutar la CPU para enviar la petición y pasa a ejecutar otro proceso. En la derecha, cuando acaba la DMA lanza una única interrupción a la CPU y está la debe reconocer, desbloqueando al proceso que se había bloqueado.

```
copiar_del_usuario(bufer, p, cuenta);
establecer_controlador_DMA();
planificador();
```

```
reconocer_interrupcion();
desbloquear_usuario();
regresar_de_interrupcion();
```