TEMA2-FSoftware.pdf

Airin86

Fundamentos del Software

1º Doble Grado en Ingeniería Informática y Matemáticas

Facultad de Ciencias
Universidad de Granada

Fundamentos del Software Irene Pérez Martínez

TEMA 2: Introducción a los Sistemas Operativos (SO)

¿Pero qué es un Sistema Operativo? ¡No vamos a empezar a hablar de ellos sin antes saber qué son!

Un SO es el software principal o conjunto de programas de un sistema informático que gestiona los recursos de hardware y provee servicios a los programas de aplicación de software, ejecutándose en modo privilegiado respecto de los restantes.

¿Y hablando en español? Pues son varios programas que se ven como una entidad, que se encargan de gestionar los recursos que el hardware del computador tiene. Es decir, hacer cálculos aritméticos (cuentecillas), manejar los datos que tenemos almacenados en el ordenador o móvil, hacer que se encienda el ordenador cuando le damos al botón de on (de esto se encarga una parte concreta del SO: la BIOS), que si mandamos algo a imprimir se haga "solo"... Todo lo que podemos hacer con el ordenador en general es porque tenemos al duendecillo SO que se encarga de mandar el mensaje, de planificar las actividades que queremos que se realicen y hacer que se ejecuten los procesos (que no son otra cosa que órdenes que le damos al ordenador, pero escritas como datos y código).

Así pues, vamos a ver a lo largo del tema el papel del SO (aunque al principio veremos un par de casos donde no está actuando), y todas las herramientas que tiene para poder llevar a cabo sus tareas. Al final es como una fábrica; el SO es el capataz, el hardware son las herramientas que tiene y los trabajadores son los distintos elementos que iremos viendo que ayudan al SO (dispositivos de E/S, el PCB, etc).

1. Componentes de un Sistema Operativo multiprogramado

1.1 Sistemas multiprogramados y de tiempo compartido

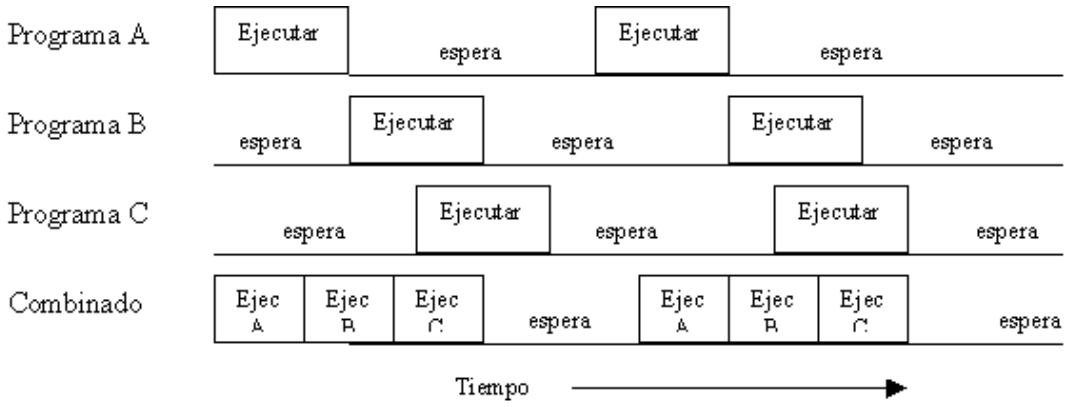
Antes de ver en qué consisten los sistemas multiprogramados vamos a tratar otros sistemas más sencillos. Esto nos dará una idea general y nos irá mostrando por qué son tan importantes los multiprogramados además de por qué se usan.

Procesamiento en serie

En este método **el programador interactúa directamente con la máquina**, no existe S.O. La consecuencia es que todo programa debe ser organizado y mandado a ejecución por parte del programador. ¿Qué problema tiene esto? Pues que el tiempo de uso de la CPU es bajo comparándolo con el de planificación: el programador dedicaría todo el tiempo a estar organizando a mano cómo quiere realizar la ejecución en vez de ejecutar (¡**perdemos muchísimo tiempo organizando**!).

Sistemas por lotes (Sistemas Batch)

Este sistema se basa en agrupar trabajos similares, por lo que reduce el tiempo de planificación (¡estamos resolviendo el problema que planteaba el procesamiento en serie!). Básicamente hace **grupos de trabajos** para realizarlos en un orden, intentando reducir el tiempo total de ejecución lo máximo posible. Hay falta de interacción entre el usuario y el computador mientras se ejecuta su trabajo (programa + datos + ordenes de control para el sistema).



Como se ve en la imagen, para evitar perder tiempo ejecutando los programas, se combinan de manera que se aproveche el tiempo que otros programas están usando dispositivos de E/S.

El principal problema es que la **CPU está ociosa durante las E/S** (la solución: Spooling, superpone la E/S de un trabajo al cómputo de otros). A continuación un ejemplo gráfico de lo que se está explicando.

Tipos de Sistemas Operativos

Podemos definir el Sistema Operativo de diversas maneras en función de la característica que miremos; veamos algunas de ellas:

- Por el número de procesos que ejecuta a la vez
  - **S.O. monoprogramado**: solo ejecuta un proceso por vez, nunca más de uno.
  - **S.O. multiprogramado**: permite que se ejecute más de un proceso *simutáneamente* cuyos datos e instrucciones se encuentran en memoria principal. Esto en alguno casos puede suponer una mejora de rendimiento muy grande, ya que podemos reducir el tiempo de ejecución incluso de forma exponencial. Aunque parece el sistema perfecto hay veces en las que no podremos mejorar el tiempo de ejecución: si el tiempo de uso de dispositivos de E/S es muy alto hay poco que el sistema pueda hacer.

        - **S.O. de tiempo compartido**: un tipo de S.O. multiprogramado donde se realiza un reparto de tiempo del procesador en pequeños trozos de tal forma que todos los procesos pueden avanzar adecuadamente. Especialmente diseñado para sistemas interactivos.
- Por el número de usuarios a los que atiende
    - **S.O. monousuario**: proporciona servicios a un único usuario.
    - **S.O. multiusuario**: proporciona servicios a varios usuarios *simultáneamente*.
- Por el número de procesadores que tiene el sistema de computación que gestiona
    - **S.O. monoprocesador**: gestiona un sistema de computación de un único procesador.
    - **S.O. multiprocesador**: gestiona un sistema de computación de varios procesadores.

```
    ALGUNAS PREGUNTAS
-¿Un S.O. multiprogramado es un S.O. de tiempo compartido? ¿y al contrario?
    Ya de la definición sabemos que un tiempo compartido es un multiprogramado.
En cambio, un     multiprogramado no tiene por qué ser de tiempo compartido
porque no tiene por qué             repartir el tiempo equitativamente o para
permitir un avance adecuado.

-¿Un S.O. de tiempo compartido tiene que ser multiusuario? ¿y monousuario?
    Pienso que podría ser tanto monousuario como multiusuario, aunque puede que
sea más costoso     si es de este último.

-¿Un S.O. monoprocesador tiene que ser monousuario? ¿y multiusuario?
    Claramente puede ser monousuario. Tenemos que saber que en multiusuario
pueden compartir el     procesador, por lo que aunque haya solo uno se puede
atender a varios usuarios.

-¿Un S.O. multiprocesador tiene que ser monousuario? ¿y multiusuario?
    Si con un solo procesador funciona en ambos casos (eso pienso), ¿por qué no
iba a funcionar     con varios?

ATENCIÓN: Las respuestas no han sido verificadas y por tanto no tienen por qué
ser correctas
```

1.2 Concepto de proceso

Un proceso está formado por un **programa ejecutable** y **datos** que necesita el SO para ejecutar el programa. Cuando decimos proceso nos podemos referir a:

1. Un programa en ejecución.
2. Una instancia de un programa ejecutándose en un ordenador.
3. La entidad que se puede asigna ro ejecutar en un procesador.
4. Una **unidad de actividad** caracterizada por un solo flujo de ejecución, un estado actual y un conjunto de recursos del sistema asociados. Según el profesor, esta es la mejor definición.

Bloque de Control de Proceso (PCB)

Para entendernos, tenemos que saber que existe una parte de la memoria (memoria restringida) donde se almacena una lista de procesos a ejecutar. Allí, sabiendo en qué dirección de memoria se almacena la información, estarán los datos que necesita el proceso, el código que se está ejecutando, los recursos utilizados... Básicamente, cada bloque de esa memoria con información asociada al proceso es el PCB de ese proceso.

Definido de manera más formal, el PCB es un registro en el cual el SO vuelca toda la información que necesita sobre un proceso para identificarlo unívocamente. Siempre que se crea un proceso (para ser ejecutado), se creará a su vez un PCB asociado (lo crea el SO). Cuando se termina de ejecutar el proceso, se elimina el bloque para liberar espacio. Solo cuando un proceso tiene un PCB creado puede ser ejecutado, o estar pendiente de ejecución. Para que se entienda: sin la información asociada que contiene el PCB, el SO no sabe nada sobre el proceso aún y por tanto no puede ejecutarlo.

El Bloque de Control de Proceso tiene una estructura concreta dividida en campos, que registran tanto los diferentes datos necesarios para la ejecución del proceso como la utilización de recursos. Los campos que forman al PCB son los siguientes:

1. **Identificador de proceso** (PID, del inglés Process IDentificator).
2. **Contexto de ejecución**: Contenido de los registros del procesador.
3. **Memoria** donde reside el programa y sus datos.
4. **Información** relacionada con recursos del sistema.
5. **Estado**: En que situación se encuentra el proceso en cada momento (modelo de estados).
6. **Otra información**

Esta implementación permite ver al proceso como una estructura de datos, y asegura la coordinación y cooperación de los procesos.

¿Cómo se lleva a cabo la implementación de un proceso?

A la hora de pasar de un proceso a otro el SO ejecuta un programa determinado que se llama **dispatcher**: módulo del SO que se encarga de realizar el intercambio de procesos en el procesador. Una vez ejecutado, se buscará el contenido del programa a ejecutar. Pero, ¿cómo llegar a la información del proceso? Muy sencillo, el PCB. De ahí se extrae la información que el SO necesita para realizar la ejecución de ese proceso que queríamos

Traza de ejecución

Una traza de ejecución es un **listado de la secuencia de las instrucciones de un programa** que realiza el procesador para un proceso. Desde el punto de vista del procesador se entremezclan las trazas de ejecución de los procesos y las trazas del código del SO . Hablando en plata: la traza es, en cada ciclo de reloj del procesador, la parte de código de un proceso que se ejecuta en ese tiempo.

```
ALGUNAS PREGUNTAS
-¿En qué situaciones se pueden entremezclar las trazas de los procesos y las
trazas del código del SO?
    Solamente cuando, tras estar ejecutando un proceso, se deba interrumpir y se
ejecuten instrucciones SO. Esto se explicará en el siguiente apartado: las trap.
Si mientras se está ejecutando un proceso se lleva a cabo una interrupción o trap
se detendría el proceso para ejecutar código del SO, y eso se mostraría en la
traza.
```

Llamadas al Sistema

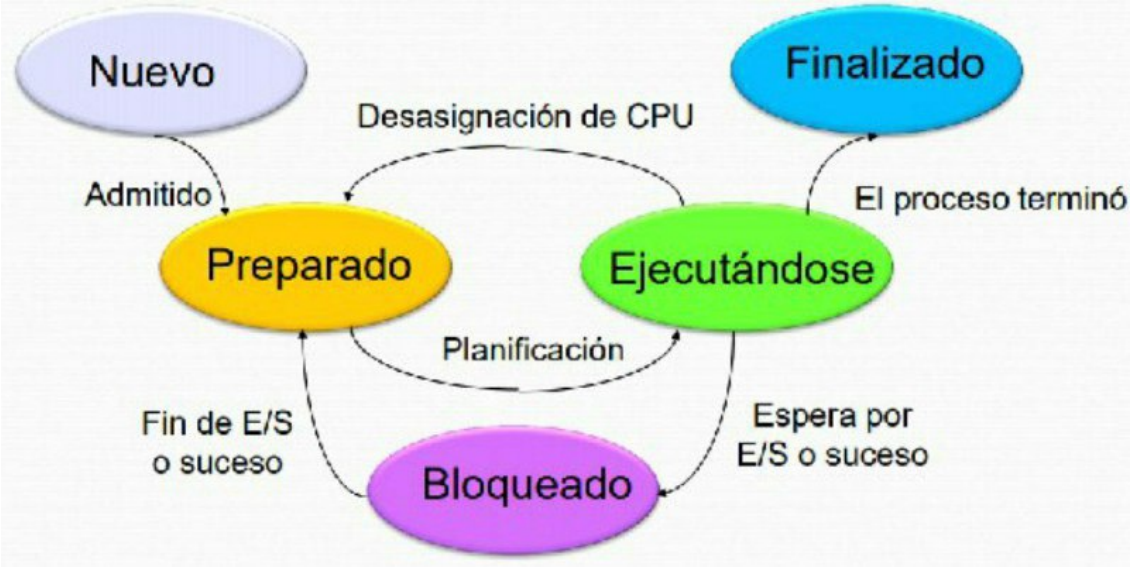
Son la forma de comunicación entre los programas de usuario y el Sistema Operativo en tiempo de ejecución. Algunos ejemplos son solicitar el uso de dispositivos de E/S, gestionar procesos o gestionar la memoria. Estas comunicaciones se implementan mediante las denominadas **trap** o interrupciones software.

1.3 Modelo de cinco estados de los procesos

Desde dentro del SO puede que sea difícil ver qué está ocurriendo realmente con los procesos, ¡necesitamos abstraernos para comprender! Con la finalidad de organizar el funcionamiento que tiene, y que nosotros entendamos mejor qué está ocurriendo, nace el modelo de cinco estados: trata de representar las **actividades** que el SO lleva a cabo **sobre los procesos**. Para ello establece cinco fases o estados en los que un proceso puede estar cuando va a ser ejecutado o está interactuando con el SO. Esos **estados de los procesos** son los siguientes:

1. **Nuevo**: el proceso no se ha ejecutado recientemente y no tenemos su información en el PCB, por lo que el SO debe cargar sus datos en la memoria restringida
2. **Preparado**: está listo para ejecutarse, sus datos ya están cargados y está esperando en la cola a que llegue su turno
3. **Ejecutándose**: el SO lo ha puesta a ejecutar
4. **Bloqueado**: por alguna razón no se puede seguir ejecutándose, y en vez de mandarlo a la cola de preparados lo manda a la de bloqueados (para distinguir cuales tienen parte ya ejecutada y cuales no). Un proceso puede estar bloqueado por muchas razones: si necesita que se realice alguna actividad con dispositivos de E/S y tiene que esperar, si ha llegado otro proceso con más prioridad que él, si ha habido algún tipo de interrupción o fallo...
5. **Finalizado**: ya se ha realizado la ejecución entera del proceso o se ha abortado la ejecución del proceso (por la razón que sea)

Para mostrar visualmente las transiciones entre estados y comprender mejor el modelo está la siguiente imagen:



2. Descripción y control de procesos

2.1 Descripción del proceso

Para hacer una descripción de un proceso de manera que al sistema operativo le sea útil tenemos el Bloque de Control de Procesos que veíamos en el apartado anterior. Puede parecer algo repetitivo hablar de nuevo del PCB, pero aquí nos vamos a fijar en la parte que corresponde a la descripción del proceso.

Todo proceso debe tener en el PCB asociado estos tres ámbitos para estar bien descritos (que al final son los campos de cada bloque de un PCB):

1. **Identificadores**: necesitamos saber quién es, y para eso no nos vale únicamente un identificador del proceso. Necesitamos identificarlo a él, al proceso padre, al usuario que necesita su ejecución, etc. En sí, hacer referencia a todo elemento asociado al proceso y que nos sea de ayuda. Para que quede más claro, como cuando hablas de un amigo y dices su nombre, quienes son sus padres, de dónde es, algo así.
2. **Contexto de ejecución**: es el contenido de los registros del procesador, como decíamos antes. ¿Cuáles vamos a tener? Pues el **PC** (contador de programa, indica la posición donde está el procesador en su secuencia de instrucciones), el **PSW** (program status word, contiene información sobre el estado de un programa utilizado por el SO) o el **SP** (puntero de pila, busca mantener la pista de la posición actual de la pila de llamadas).
3. **Información para control del proceso**: los demás campos de un bloque del PCB. Básicamente la información de estado y planificación, la descripción de las regiones de memoria asignadas, los recursos asignados, los enlaces a colas de procesos y la comunicación entre procesos.

De acuerdo, esto es recordatorio de lo que el PCB almacena de cada proceso, pero ¿cómo inicializamos el PCB para que el proceso pueda ser ejecutado? Pues simplemente tenemos que rellenar los campos, como era de esperar. ¿Qué hacemos? Pues asignamos el **identificador del proceso**, asignamos un **nuevo PCB**, le damos **memoria** para guardar el programa asociado e **inicializamos** el PCB.

2.2 Control de procesos

Vale, ya hemos aprendido cómo describir un proceso y crear su PCB asociado para introducirlo en la cadena de ejecución del SO, pero no sabemos absolutamente nada del control que se lleva a cabo sobre los procesos. Imagina que de pronto un proceso está mal y empieza a cambiar datos del computador, ¡puede borrar datos importantes, cambiar nuestras contraseñas, o quién sabe!

¿Qué mecanismos tiene el SO para controlar que los procesos no la líen? Muy sencillo, **dos modos de ejecución**. En función de cómo se va a ejecutar puede o no acceder a una serie de datos y recursos del ordenador, de manera que no pueda modificar elementos importantes ya que tiene el acceso restringido. Veamos los dos modos de ejecución:

- **Modo usuario**: El programa (de usuario) que se ejecuta en este modo sólo se tiene acceso a un subconjunto de los registros del procesador, a un subconjunto del repertorio de instrucciones máquina y a un área de la memoria.
- **Modo núcleo (kernel o supervisor o sistema)**: El programa (SO) que se ejecuta en este modo tiene acceso a todos los recursos de la máquina, tanto software como hardware. Aquí no tiene acceso restringido y por eso el SO debe tener cuidado a la hora de permitir este tipo de modo.

Solo se cambia de modo usuario a modo kernel por tres razones: cuando hay una **interrupción**, cuando hay una **excepción** o si hay una **llamada al sistema**. En estos casos el propio hardware cambia automáticamente a modo kernel, para que el SO ejecute la rutina establecida dependiendo del evento y, una vez terminada esta, que el hardware vuelve a modo usuario.

Las interrupciones y excepciones las vimos en el tema anterior, pero vamos a ver una breve descripción de las llamadas al sistema:

- **Llamadas al sistema**: es el mecanismo usado por una aplicación para solicitar un servicio al SO. El software de aplicación está diseñado para realizar un grupo de funciones, tareas o actividades coordinadas para el beneficio del usuario, por lo que habrá veces en las que necesite acceso a zonas restringidas. En esos casos debe hacer una llamada al sistema para pedir permiso al SO.

Estas tres situaciones pueden causar dos resultados diferentes: que se pare momentáneamente la ejecución del proceso (cambio de modo) o que deje de ejecutarse y entre otro proceso a ejecutarse (cambio de contexto). Para cada caso contamos con dos operaciones importantes que el SO lleva a cabo y que permiten la actuación sobre los procesos de manera segura.

- **Operación de cambio de modo**: Se ejecuta una rutina del SO en el contexto del proceso que se encuentra en estado “Ejecutándose”. Puede realizarse tras una interrupción, una excepción o una llamada al sistema. Sus pasos:
    - El hardware automáticamente salva como mínimo el PC y PSW y cambia el bit de modo a modo kernel.
    - Determinar automáticamente la rutina del SO que debe ejecutarse y cargar el PC con su dirección de comienzo.
    - Ejecutar la rutina. Posiblemente la rutina comience salvando el resto de registros del procesador y termine restaurando en el procesador la información de registros previamente salvada.
    - Volver de la rutina del SO al proceso que se estaba ejecutando. El hardware automáticamente restaura en el procesador la información del PC y PSW previamente salvada

- **Operación de cambio de contexto**: Un proceso en estado “Ejecutándose” cambia a otro estado y un proceso en estado “Preparado” pasa a estado “Ejecutándose”. Puede realizarse tras una interrupción, una excepción o una llamada al sistema. Sus pasos:
    - Salvar los registros del procesador en el PCB del proceso que actualmente está en estado “Ejecutándose”.
    - Actualizar el campo estado del proceso al nuevo estado al que pasa e insertar el PCB en la cola correspondiente.
    - Seleccionar un nuevo proceso del conjunto de los que se encuentran en estado “Preparado” (Scheduler o Planificador de CPU).
    - Actualizar el estado del proceso seleccionado a “Ejecutándose” y sacarlo de la cola de preparados.
    - Cargar los registros del procesador con la información de los registros almacenada en el PCB del proceso seleccionado

```
PREGUNTA:
¿Si se produce una interrupción, una excepción o una llamada al sistema, se
produce necesariamente un cambio de modo? ¿y un cambio de contexto?
    No tiene por qué, pienso yo. Dependiendo del caso puede ocurrir una cosa o la
otra. Si no es necesario frenar la ejecución del proceso actual no se llevará un
cambio de contexto, y si es necesario parar el proceso actual pues no hace falta
el cambio de modo.

    (Repito que esto no está verificado, es solo lo que yo diría)
```

3. Hebras (hilos)

El concepto de **proceso** (tarea) tiene dos características diferenciadas que permiten al SO, por una parte, controlar la asignación de los recursos necesarios para la ejecución de programas y, por otra, la ejecución del programa asociado al proceso de forma intercalada con otros programas. El concepto de proceso (tarea) y **hebras** asociadas se basa en separar estas dos características: la tarea se encarga de soportar todos los recursos necesarios (incluida la memoria) mientras que cada una de las hebras permite la ejecución del programa de forma “independiente” del resto de hebras.

Así pues, un hebra será una secuencia de tareas encadenadas muy pequeña que puede ser ejecutada por un sistema operativo. Se pueden ejecutar de una en una o a la vez, si comparten los mismos recursos (en ese caso al conjunto de hebras se le llamará proceso). Puede ser algo lioso de primeras porque el concepto de proceso y de hebra están muy ligados, pero para entendernos la hebra será la mínima subdivisión de un proceso. Al proceso lo llamamos tarea, pero una gran tarea; esta tarea se puede subdividir en muchas subtareas: esas subtareas son las hebras.

Por poner un ejemplo visual, imagina realizar una división. La división es el proceso, mientras que cada uno de los pasos que tenemos que ir dando (multiplicar, restar, decidir el número a añadir en el cociente...) son las hebras.

Igual que podemos clarificar el proceso de ejecución de los procesos mediante el modelo de los cinco estados, se pueden aplicar las mismas reglas a las hebras, teniendo así un **modelo de cinco estados para hebras** con los mismos estados: nuevo, preparado, ejecutándose, bloqueado y finalizado.

Ahora bien, ¿es útil dividir los procesos en hebras? Veamos sus ventajas:

- Menor tiempo de creación de una hebra en un proceso ya creado que la creación de un nuevo proceso. Es decir, crear un proceso trae cargar en el PCB mucha información, además de tener que reservar memoria. Si solo queremos cargar una hebra necesitamos menos espacio (se necesitan muchos menos datos) y si ya se ha ejecutado alguna hebra del mismo proceso gran parte de los recursos que necesitamos están ya activados.
- Menor tiempo de finalización de una hebra que de un proceso, no solo porque la hebra realiza menos tarea que el proceso, sino también porque es más fácil gestionar un grupo de hebras que verlas todas como una sola entidad, es decir, verlas como proceso.
- Menor tiempo de cambio de contexto (hebra) entre hebras pertenecientes al mismo proceso, ya que no hay que cambiar datos globales o recursos.
- Facilitan la comunicación entre hebras pertenecientes al mismo proceso (recursos compartidos, variables globales...).
- Permiten aprovechar las técnicas de programación concurrente y el multiprocesamiento simétrico. Podemos mejorar el rendimiento de la ejecución si lo vemos como tareas separadas en hebras que como proceso, ya que reducimos los tiempos muertos ejecutando otras hebras cuando las demás están ocupadas en otros dispositivos.

4. Gestión básica de memoria

Buscamos hacer una gestión de la memoria lo más eficiente posible, es decir, hacer un uso dinámico de la memoria de manera que no queden huecos de descanso (tiempo donde no se está ejecutando nada).

Veamos algunas definiciones previas que nos serán de ayuda en este apartado:

- **Carga absoluta**: Asignar direcciones físicas (direcciones de memoria principal) al programa en tiempo de compilación. El programa no es reubicable posteriormente.
- **Reubicación**: Capacidad de cargar y ejecutar un programa en un lugar arbitrario de la memoria, lo que permite reubicar la localización del programa en memoria después.
    - **Reubicación estática**: El compilador genera direcciones lógicas (relativas) de 0 a M. La decisión de dónde ubicar el programa en memoria principal se realiza en tiempo de carga. El cargador añade la dirección base de carga a todas las referencias relativas a memoria del programa.
    - **Reubicación dinámica**: El compilador genera direcciones lógicas (relativas) de 0 a M. La traducción de direcciones lógicas a físicas se realiza en tiempo de ejecución luego el programa está cargado con referencias relativas. Requiere apoyo hardware

La memoria no es una unidad inquebrantable, sino que se divide en secciones. Esto ayuda al SO a encontrar más fácil y rápidamente los datos que necesite, y a tener una mayor organización de lo que se almacena. Podemos dividir la memoria en distintos espacios:

- **Espacio de direcciones lógico**: Conjunto de direcciones lógicas (o relativas) que utiliza un programa ejecutable.
- **Espacio de direcciones físico**. Conjunto de direcciones físicas (memoria principal) correspondientes a las direcciones lógicas del programa en un instante dado.
- **Mapa de memoria de un ordenador**: Todo el espacio de memoria direccionable por el ordenador. Normalmente depende del tamaño del bus de direcciones.
- **Mapa de memoria de un proceso**: Se almacena en una estructura de datos (que reside en memoria) donde se guarda el tamaño total del espacio de direcciones lógico y la

correspondencia entre las direcciones lógicas y las físicas.

Retomando la idea que nos introducía el apartado, hay veces en los que la memoria tiene huecos en blanco, lo que en la informática se denomina **fragmentación de la memoria**. Esto se debe a que los procesos o datos que almacena no tienen siempre el mismo tamaño; puede que una vez borrado un proceso, que no necesitase mucho espacio en memoria, ya ejecutado deje un hueco demasiado pequeño en la memoria para que nadie lo pueda usar.

¿Cómo podemos resolver este problema? Mediante técnicas que nos ayuden a trocear el espacio lógico de la memoria en unidades más pequeñas. Estos trozos que creamos en memoria no tienen por qué ubicarse consecutivamente en el espacio físico, sino que pueden estar unidas de manera lógica por la organización propia de la memoria. Para realizar este troceado tenemos dos herramientas: la **paginación** y la **segmentación**.

4.1 Paginación

El espacio de direcciones físicas de un proceso puede ser no contiguo, es decir, que las direcciones a los datos de un proceso no tienen por qué estar almacenadas en memoria una detrás de otra. La idea de la paginación es dividir la memoria en bloques físicos y lógicos para simplificar el uso.

La *memoria física* se divide en bloques de tamaño fijo, denominados **marcos de página**, que siempre viene dado en potencia de dos, de 512 B a 8 KB. Por otro lado, *el espacio lógico* de un proceso se divide conceptualmente en bloques del mismo tamaño, denominados **páginas**. Los marcos de página contendrán páginas de los procesos, y ambos tienen el mismo tamaño.

Direcciones	Número de ...	Desplazamiento
Lógicas	página => p	d
Físicas	marco => m	d

Resumiendo, debemos distinguir entre memoria física (que se corresponde con el espacio hardware que requiere para ser almacenado, con bloques denominados marcos de página) y memoria o espacio lógico (espacio software que ocupa, cantidad de datos, con bloques denominados páginas). Las direcciones lógicas, que son las que genera la CPU, se dividen en número de página (p) y desplazamiento dentro de la página (d). Las direcciones físicas se dividen en número de marco (m, marco donde está almacenada la página) y desplazamiento (d).

```
Hablando en plata para entender la idea:

Entendamos la memoria como un libro, que se divide en páginas, y entendamos los
procesos o información a almacenar como los capítulos. Si lo vemos de forma
lógica, cada capítulo tiene una página donde comienza, que será p, y ocupa un
número de páginas, que será d, todo el desplazamiento que podemos hacer sin
acabar en una página que no sea del capítulo. Nunca se mezclan dos capítulos en
una página, sino que cuando empezamos uno nuevo también empezamos una nueva
página. ¿Qué puede ocurrir? Que un capítulo acabe a mitad de la hoja y haya
espacio que no se use, pero preferimos eso a mezclar los capítulos.
```

Contamos con dos tablas para almacenar y obtener la información de dirección en memoria de cada proceso. Cuando la CPU genere una dirección lógica será necesario traducirla a la dirección física correspondiente; la **tabla de páginas** (existe una tabla de páginas por proceso) mantiene información necesaria para realizar dicha traducción (Dirección física = p * tamaño_página + Desplazamiento). Por otro lado tenemos la **tabla de marcos de página**, usada por el SO, que contiene información sobre cada marco de página.

Así, cada entrada de la tabla de páginas se corresponde con una página del proceso, y contendrá el número de marco en el que está almacenada la página si está en MP (memoria principal) y el modo de acceso autorizado a la página (bits de protección). Esta tabla se mantiene en MP, para llegar hasta ella tenemos el registro base de la tabla de páginas (RBTP) que se suele almacenar en el PCB de cada proceso (es decir, el PCB tiene la dirección a las entradas de la tabla de páginas, para que el SO puede encontrar toda la información del proceso fácilmente). Como conclusión, para acceder a un dato debemos hacer dos accesos a memoria: uno a la tabla de páginas y, una vez sabemos dónde están las cosas almacenadas, otro a memoria.

Hasta ahora podía parecer buena idea lo de la paginación, pero si cuesta tiempo y esfuerzo acceder a memoria, imagina hacerlo doble por cada dato que necesitemos. ¡Una locura! ¿Qué podemos hacer para evitar ese doble acceso? Usar el TLB.

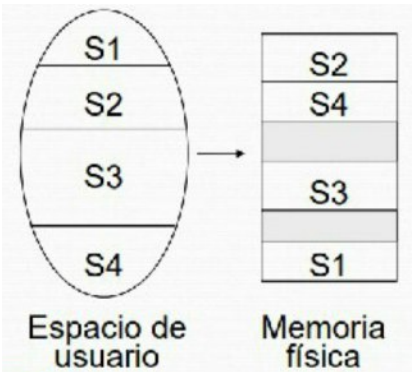
Búfer de Traducción Adelantada (TLB)

El problema de los dos accesos a memoria se resuelve con una caché hardware de consulta rápida denominada búfer de traducción adelantada o TLB (Translation Look-aside Buffer). El TLB se implementa como un conjunto de registros asociativos que permiten una búsqueda en paralelo, de forma que para traducir una dirección:

1. Si existe ya en el registro asociativo, obtenemos el marco.
2. Si no, la buscamos en la tabla de páginas y se actualiza el TLB con esta nueva entrada.

4.2 Segmentación

La segmentación propone una alternativa a la paginación: quiere un esquema de organización de memoria que soporte mejor la visión de memoria del usuario: un programa es una colección de unidades lógicas -segmentos-, como por ejemplo procedimientos, funciones, pila, tabla de símbolos, matrices, etc. Quiere que la idea de memoria que tiene el usuario en su espacio se mantenga también en la memoria física.



En este método también se usa una tabla, la **tabla de segmentos**, que aplica direcciones bidimensionales definidas por el usuario en direcciones físicas de una dimensión. Cada entrada de la tabla tiene los siguientes elementos (aparte de presencia, modificación y protección):

1. **Base**: dirección física donde reside el inicio del segmento en memoria.
2. **Tamaño**: longitud del segmento.

La tabla de segmentos se mantiene en memoria principal. El Registro Base de la Tabla de Segmentos (RBTS) apunta, en este caso, a la tabla de segmentos (suele almacenarse en el PCB del proceso). El Registro Longitud de la Tabla de Segmentos (STLR) indica el número de segmentos del proceso; el nº de segmento s, generado en una dirección lógica, es legal si s < STLR (suele almacenarse en el PCB del proceso).