
# SISTEMAS OPERATIVOS
# GESTIÓN DE MEMORIA

Pedro de Miguel Anasagasti

# ÍNDICE




# SERVIDOR DE MEMORIA

# INTRODUCCIÓN

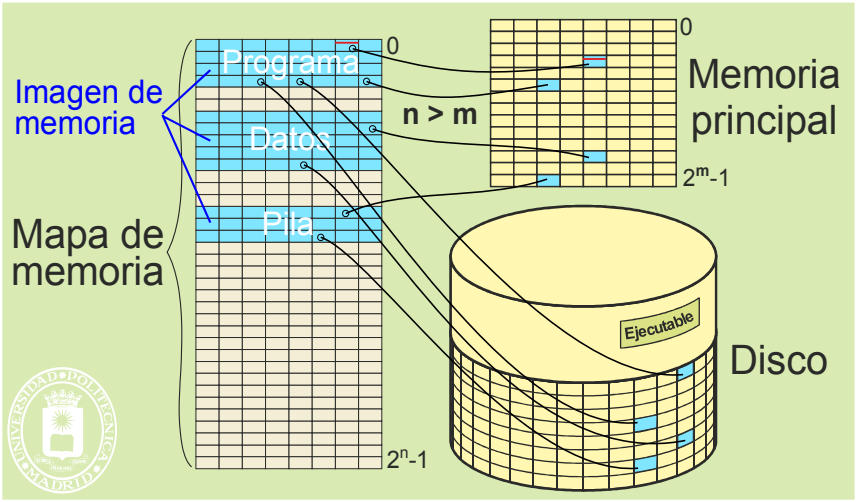


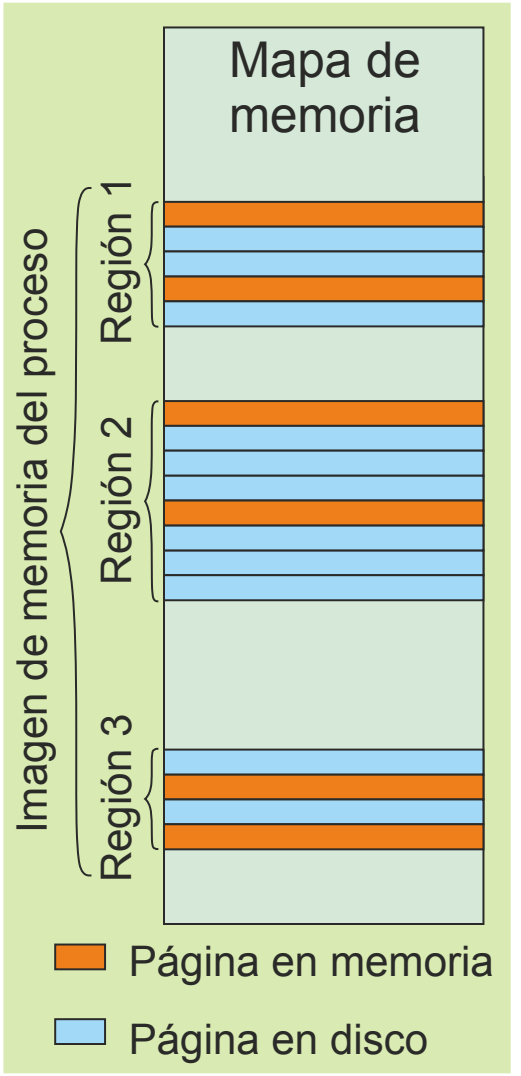
**El gestor de memoria de un SO de propósito general tiene dos facetas complementarias.**

- **Servidor de memoria para los procesos.**
- **Soporte a la memoria virtual.**

**Los procesos no entienden del soporte físico del mapa de memoria, ya sea este soporte memoria principal o memoria virtual, solamente entienden de direcciones dentro del mapa de memoria del procesador.**

**Cuando el gestor de memoria asigna un marco de página a un proceso, no significa que el proceso vea más memoria. Simplemente establece un soporte físico más rápido a una zona de la memoria del proceso.**



# REQUISITOS DE LA GESTIÓN DE MEMORIA



## El S.O. multiplexa los recursos entre los procesos

- Cada proceso cree que tiene una máquina para él solo.
  - Gestión de procesos: Reparto de procesador.
  - Gestión de memoria: Reparto de memoria.

## Funciones del servidor de memoria

- Crear la imagen de los procesos a partir de los ficheros ejecutables.
  - Ofreciendo a cada proceso los recursos de memoria necesarios, dando soporte a las regiones necesarias.
  - Proporcionando grandes espacios de memoria a los procesos.
- Proporcionar protección entre procesos. Aislar los procesos.
  - Pero permitir que los procesos compartan memoria de forma controlada.
- Controlar los recursos:
  - Direcciones de los mapas de memoria ocupadas y libres.
  - Direcciones de memoria principal y de intercambio ocupadas y libres.
  - Recuperar los recursos de memoria liberados por los procesos.
- Tratar los errores de acceso a memoria: detectados por el HW.
- Optimizar las prestaciones del sistema.


# REGIONES DE MEMORIA DEL PROCESO

# MEMORIA DEL PROCESO I



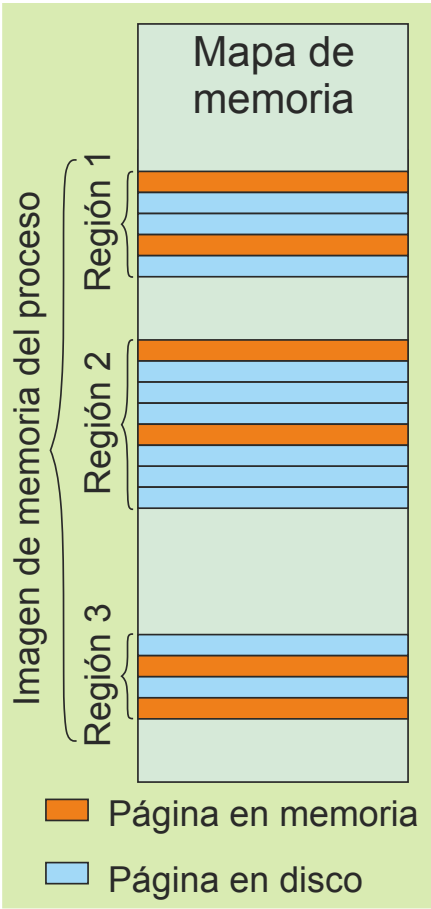
Las necesidades de memoria de un proceso se resuelven a dos niveles:

- Por el SO: Visión global o macroscópica consistente en regiones.
- Por las bibliotecas del lenguaje usado para el programa: Visión de detalle o microscópica consistente en objetos dentro de las regiones.

Los lenguajes de programación suelen permitir la asignación dinámica de objetos de memoria, lo que puede obligar a modificar las regiones.

El SO tiene un visión macroscópica de la memoria de un proceso, consistente en la imagen de memoria del proceso y formado por las regiones, que son grandes trozos de memoria contigua.

- El proceso tiene unas pocas regiones (p.e. código, datos y pila).
- En un sistema con memoria real (sin memoria virtual) todas las regiones están en una misma zona contigua de memoria principal.
  - Se deja un espacio de memoria principal para el crecimiento dinámico de las regiones (desaprovechamiento de recursos).
- En memoria virtual las regiones están separadas, alineadas a página y están formadas por un número entero de páginas.
  - Al crear o crecer es cuando se requiere asignar soporte físico.
  - Los espacios entre regiones no tienen soporte físico.



# MEMORIA DEL PROCESO II



**El programa tiene una visión microscópica de la memoria, consistente en variables y estructuras de datos.**

- **Crear una variable dinámica que quepa en la correspondiente región no implica al SO (lo resuelven las bibliotecas del lenguaje).**
- **Crear una variable dinámica que no quepa en la correspondiente región implica activar al SO para que aumente dicha región, o cree una nueva. De ello se encargan las bibliotecas del lenguaje, liberando al programador de esta tarea.**
- **Las bibliotecas del lenguaje utilizado en el desarrollo del programa gestionan el espacio disponible en la región de datos dinámicos.**
  - **Solamente llaman al SO cuando tienen que variar el tamaño de la región, o crear una nueva región.**

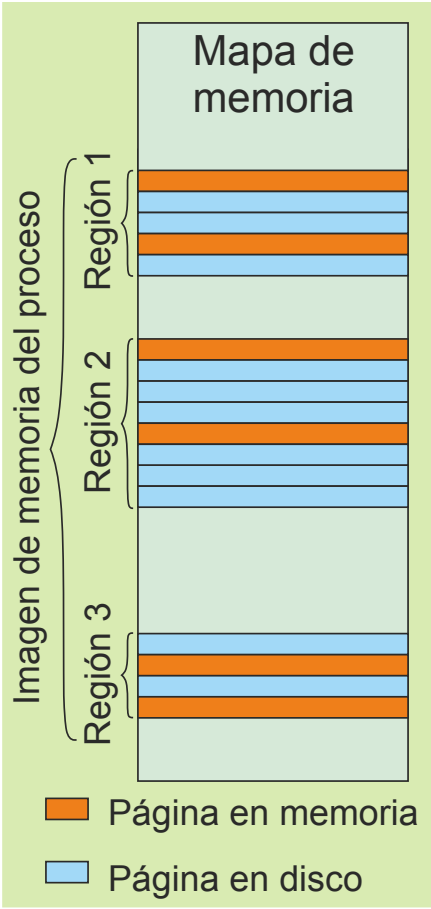
# IMAGEN DE MEMORIA



**Imagen de memoria: conjunto de regiones (o segmentos) de memoria asignados a un proceso.**

## Características de una región

- Es una zona contigua de direcciones de memoria definida por:
  - Una dirección de comienzo
  - y un tamaño.
- Fuente: lugar donde se almacena el valor inicial.
- Puede ser compartida o privada.
- Niveles de protección típicos: RWX.
- Puede tener tamaño fijo o variable.



# REGIONES DEL PROCESO



Las regiones más relevantes de la imagen de memoria del proceso son:

- **Código** (texto): Contiene el código máquina del programa.
- **Datos**, que se organiza en:
  - Datos con valor inicial: Variables globales inicializadas.
  - Datos sin valor inicial: Variables globales no inicializadas.
  - Datos creados dinámicamente o heap.
- **Pila**: soporta los registros de activación de los procedimientos.

La estructuración en regiones depende del diseño del SO.

- Puede haber una sola región que englobe datos con valor inicial, datos sin valor inicial y datos creados dinámicamente.
- O puede haber regiones separadas para distintos tipos de datos.

En un sistema con memoria virtual las regiones se alinean a página, ocupando un número entero de páginas.

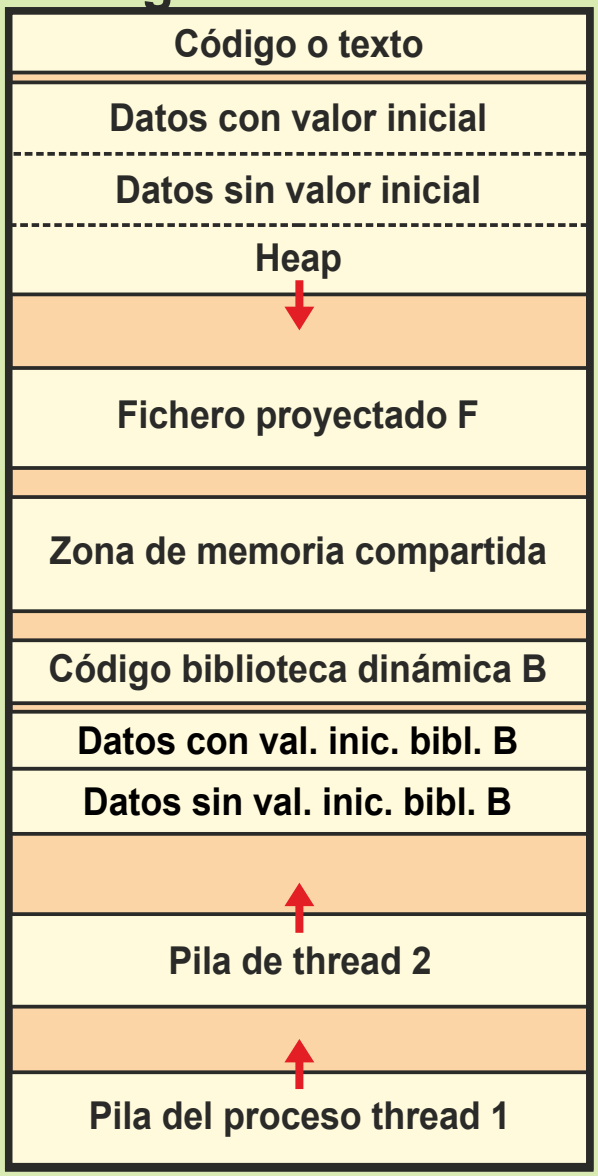
# OPERACIONES SOBRE REGIONES



- Creación de región
  - Al crear el mapa inicial o por una solicitud posterior
  - En sistemas con memoria virtual no se asignan marcos de memoria principal, se asigna swap o rellenar con 0
- Liberación de región
  - Al terminar el proceso o por solicitud posterior
  - Se recuperan los recursos (swap y marcos)
- Cambio de tamaño de región
  - Del heap o de la pila
  - En sistemas con memoria virtual no se asigna memoria principal
- Duplicado de región
  - Operación requerida por el servicio FORK de POSIX

# MAPA DE MEMORIA DE UN PROCESO HIPOTÉTICO






| | | Tamaño | Fuente |
|---|---|---|---|
| Compartido | R–X | Fijo | Ejecutable |
| Privado | RW– | Fijo | Ejecutable |
| Privado | RW– | Fijo | ← 0 |
| Privado | RW– | Variable | ← 0 |
| Com./Priv. | ??– | Variable | Fichero |
| Compartido | ??– | Variable | ← 0 |
| Compartido | R–X | Fijo | Biblioteca |
| Privado | RW– | Fijo | Biblioteca |
| Privado | RW– | Fijo | ← 0 |
| Privado | RW– | Variable | ← 0 |
| Privado | RW– | Variable | ← 0 (pila inicial) |

# CARACTERÍSTICAS DE LAS REGIONES




## Características típicas de las regiones creadas en el arranque del proceso

**Región de código.**

- **Compartida, RX, tamaño fijo, fuente: el fichero ejecutable.**

**Región de datos con valor inicial.**

- **Privada, RW, tamaño fijo, fuente: el fichero ejecutable.**

**Región de datos sin valor inicial.**

- **Privada, RW, tamaño fijo, fuente: rellenar con 0.**

**Región de pila.**

- **Privada, RW, tamaño variable, fuente: rellenar con 0.**
- **Pila inicial (creada al arrancar el programa):**
  - **Variables de entorno.**
  - **Argumentos del programa.**

**(En la sección de protección se tratará el porqué de rellenar con 0).**

# Otras regiones de la imagen creadas durante ejecución




**Región de Heap.**

- Soporte de memoria dinámica gestionada por el lenguaje (p.e. malloc en C).
- Persistencia controlada por el programador.
- Privada, RW, tamaño variable, fuente: rellenar con 0.

**Memoria compartida.**

- Región asociada a la zona de memoria compartida.
- Compartida, tamaño variable, fuente: rellenar con 0.
- Protección especificada en proyección.

**Fichero proyectado.**

- Región asociada a cada fichero proyectado.
- Compartida o privada, tamaño variable, fuente: el fichero proyectado.
- Protección especificada en proyección.

**Pilas de threads.**

- Cada pila de thread corresponde con una región.
- Mismas características que pila del proceso.

**Biblioteca dinámica.**

- Regiones asociadas al código y datos de cada biblioteca dinámica.

# CREACIÓN DE REGIONES



**El gestor de memoria crea las regiones de memoria cuando:**

- **se crea un proceso (fork),**
- **se cambia el programa del proceso (exec),**
- **o se solicita una nueva región.**

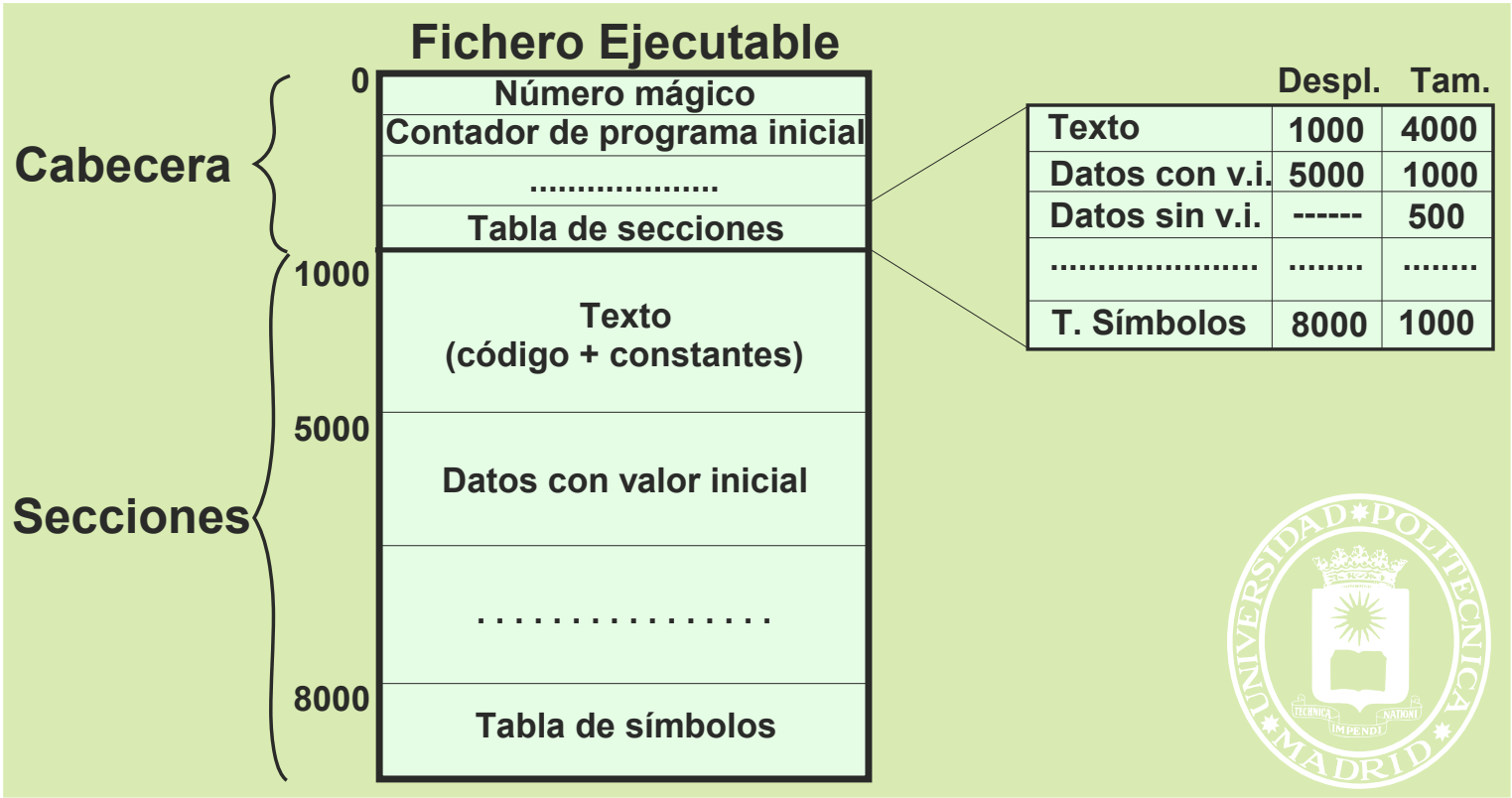

# FICHERO EJECUTABLE

## El fichero ejecutable contiene toda la información para crear el proceso.

**Existen distintos formatos (p.e. Executable and Linkable Format (ELF)).**

**Estructura del ejecutable: Cabecera y conjunto de secciones.**

**Cabecera:**
- **Número mágico.**
- **Registros (contador de programa inicial).**
- **Tabla de secciones.**



# SECCIONES DEL EJECUTABLE



## Secciones principales

- Sección de código (texto): Contiene código del programa. Suele incluir también las constantes del programa y cadenas de caracteres.
- Sección de datos con valor inicial: Variables globales inicializadas.

## Otras secciones

- Tabla de símbolos para depuración y montaje dinámico.
- Lista de bibliotecas dinámicas usadas.

## No existen secciones para

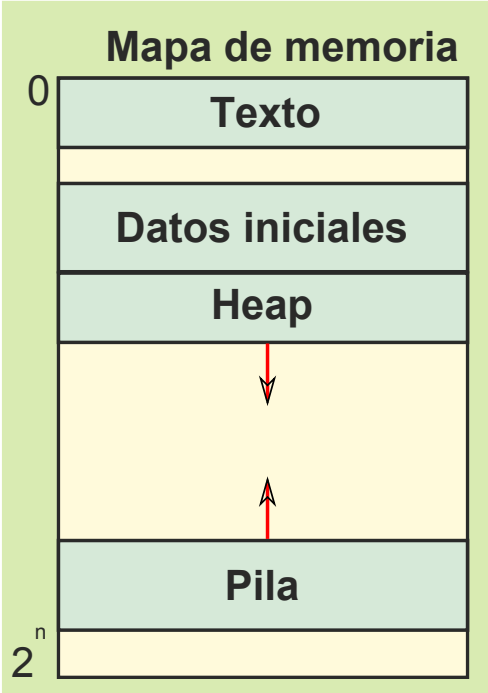
- Datos estáticos sin valor inicial: Variables globales no inicializadas.
  - Aparece en tabla de secciones pero no se almacena en el ejecutable.
- Variables locales.
- Pila.

# VARIABLES GLOBALES VERSUS LOCALES



## Variables globales

- Estáticas.
- Se crean al iniciarse el programa.
- Existen durante toda la ejecución del mismo.
- Dirección fija en memoria y en ejecutable.



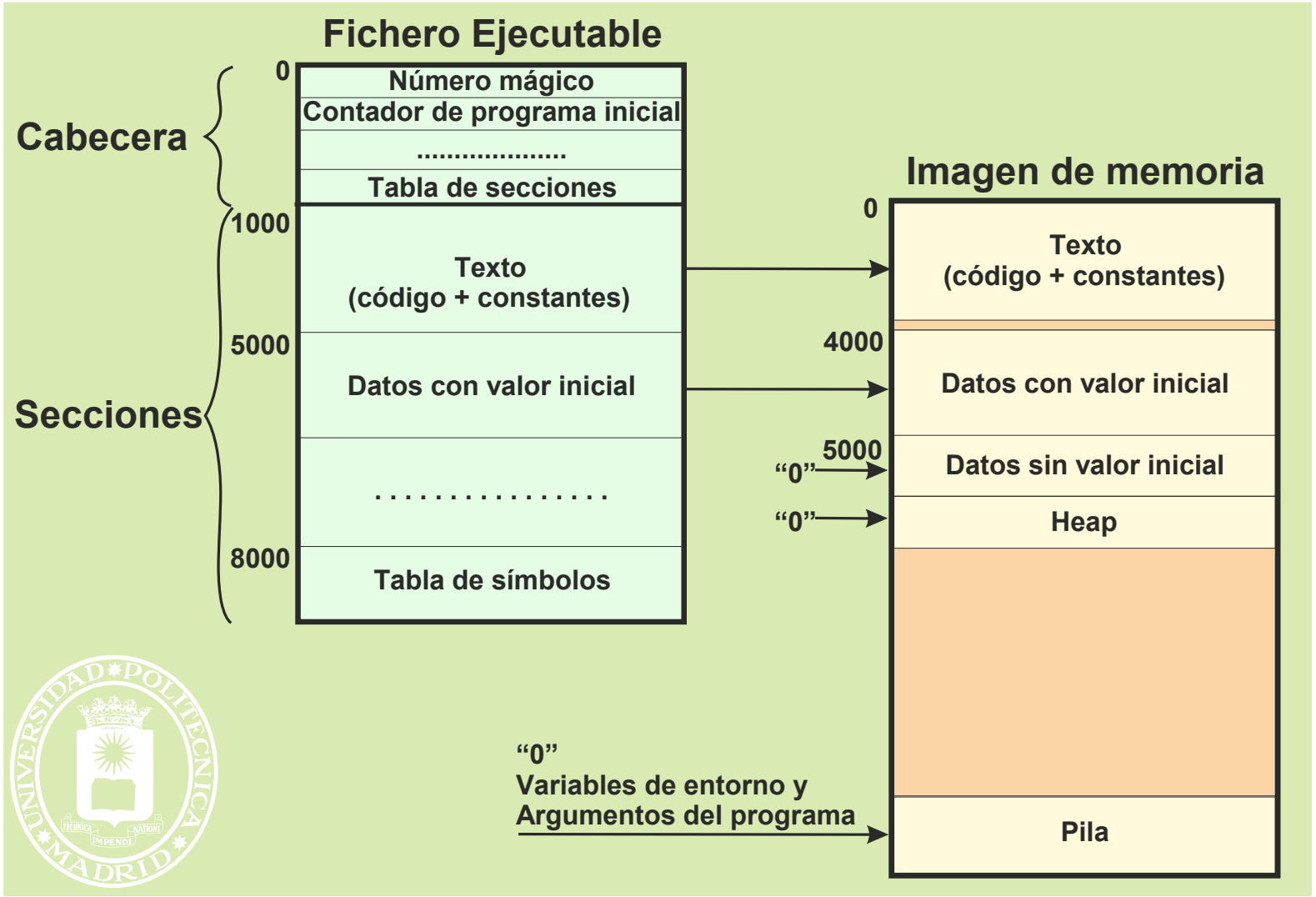

## Variables locales y parámetros

- Dinámicas. Creadas dentro del RAR (registro de activación de rutina).
- Se crean al invocar una función. Su valor inicial se establece por el propio código.
- Se destruyen al retornar (no se borra, queda como basura).
- La dirección de la variable es relativa al RAR.
- Recursividad: varias instancias de una variable, cada una en una dirección distinta.

# CREAR IMAGEN DESDE EJECUTABLE



**El gestor de memoria se encarga de crear la imagen del proceso a partir de un fichero ejecutable. En UNIX esta operación la realiza el `exec`.**




# UTILIZACIÓN DE MEMORIA EN BRUTO

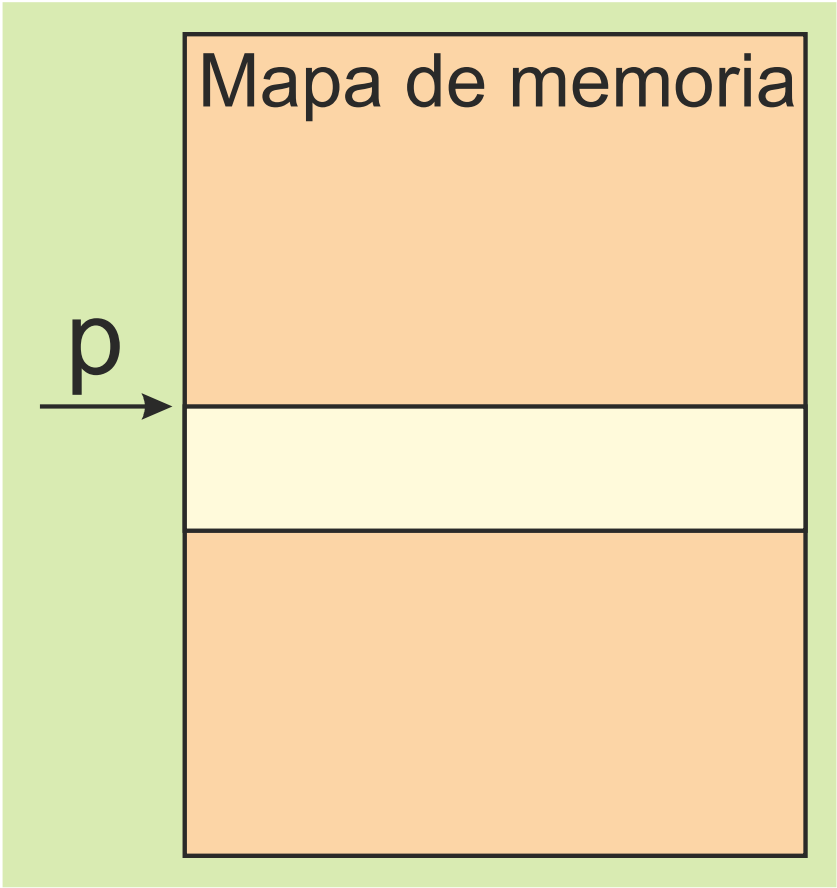
# OBTENCIÓN DE MEMORIA EN BRUTO



**Un programa puede obtener memoria en bruto mediante los siguientes procedimientos:**

- **Mediante las bibliotecas del lenguaje (malloc, new, …).**
- **Mediante el SO (se detallan más adelante).**
  - **Fichero proyectado en memoria.**
  - **Región de memoria compartida.**
  - **Nueva región de memoria.**

**En todos los casos el programa recibe un puntero, referencia o manejador a esa memoria en bruto a través del cual podrá utilizarla.**



# UTILIZACIÓN DE MEMORIA EN BRUTO

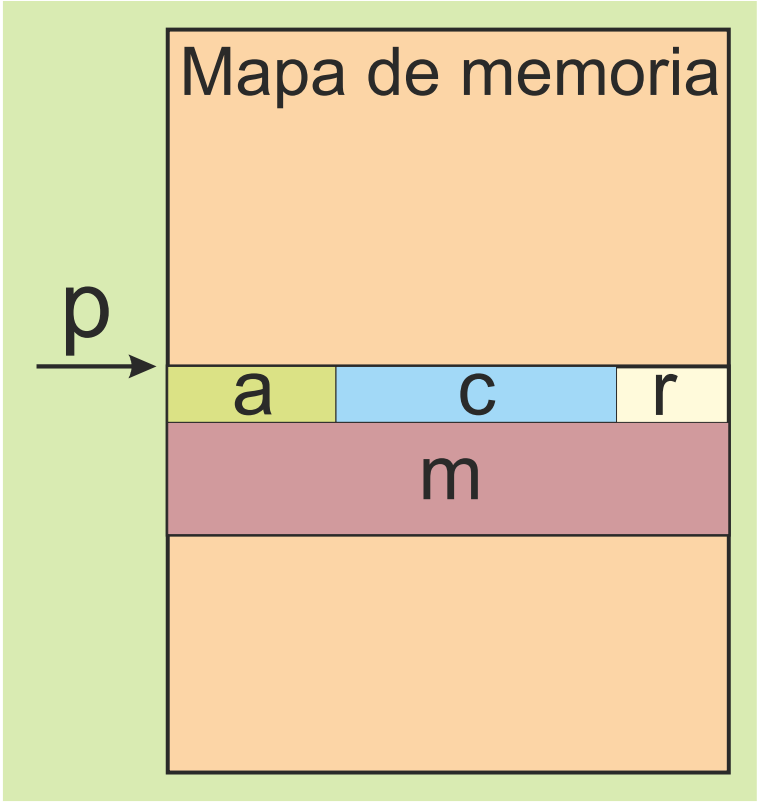


**Los lenguajes de alto nivel utilizan la memoria mediante variables no con direcciones.**

**Se proyecta sobre la zona asignada uno o varios tipos de datos: estructura, vector, array, etc.**

**Se utilizan los datos mediante la referencia o el puntero devuelto por el servicio o por la función del lenguaje.**

**Para memoria compartida hay que proyectar los mismos datos en todos los programas.**




**Los errores típicos de utilización se verán más adelante.**


# NECESIDADES DE MEMORIA DE UN PROCESO

# NECESIDADES DE MEMORIA DEL PROCESO



**Los procesos necesitan memoria para almacenar el código y los datos.**

```
int a;
int b = 5;
const float pi = 3.1416;

void f(int c) {
    int d;
    static int e = 2;
    d = 3;
    b = d + 5;
    .......
    return;
}

int main (int argc, char *argv[]) {
    char *p;
    p = malloc (1024);
    f(b);
    .......
    free (p);
    printf ("Hola mundo\n");
    return 0;
}
```

**Tipos de memoria requeridos**

1.- Código

2.- Datos declarados

- Estáticos
  - Constantes + cadenas
  - Con valor inicial
  - Sin valor inicial
- Dinámicos
  - Con valor inicial
  - Sin valor inicial

3.- Datos en bruto

- Asignación de memoria

# FICHERO EJECUTABLE

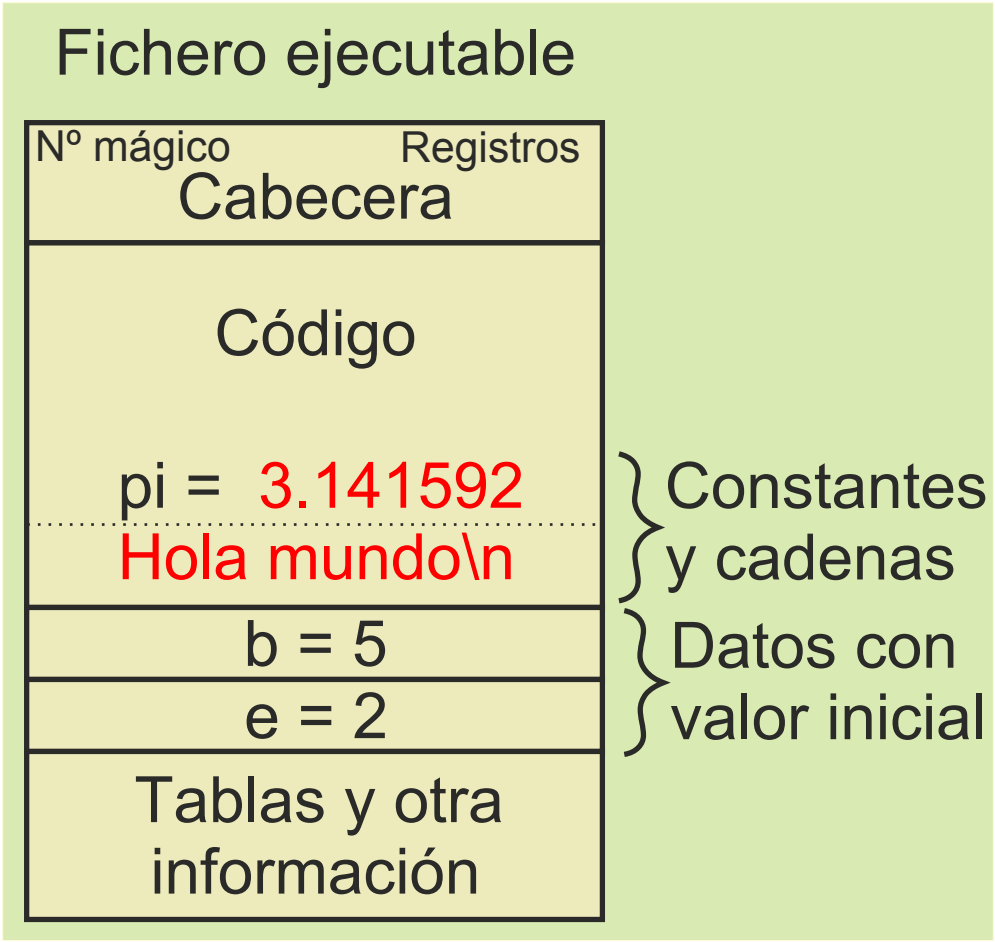


**Las constantes y las cadenas de caracteres deben ser inmutables, por lo que se suelen asociar al código.**

```
int a;
int b = 5;
const float pi = 3.1416;

void f(int c) {
      int d;
      static int e = 2;
      d = 3;
      b = d + 5;
      .....
      return;
}

int main (int argc, char *argv[]) {
      char *p;
      p = malloc (1024);
      f(b);
      .....
      free (p);
      printf ("Hola mundo\n");
      return 0;
}
```



# FICHERO EJECUTABLE

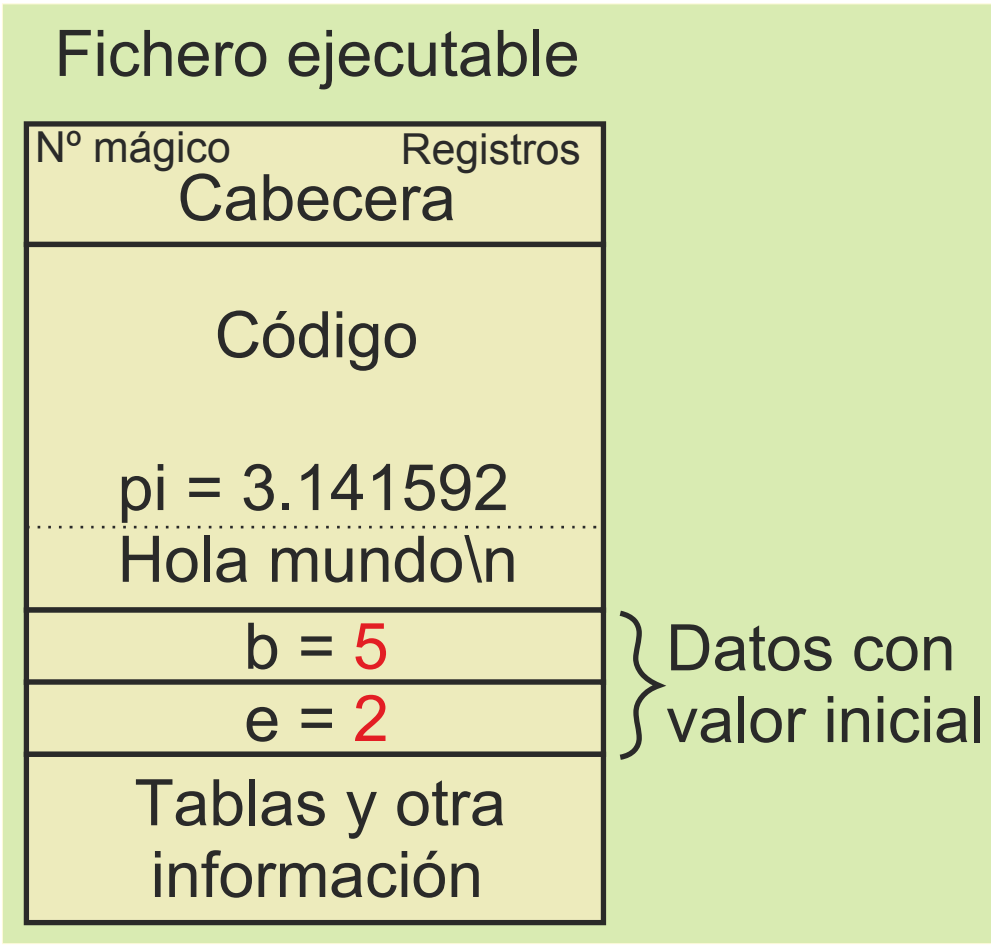


**Los datos estáticos y con valor inicial aparecen en la zona de datos del ejecutable.**

```
int a;
int b = 5;
const float pi = 3.1416;

void f(int c) {
      int d;
      static int e = 2;
      d = 3;
      b = d + 5;
      .......
      return;
}

int main (int argc, char *argv[]) {
      char *p;
      p = malloc (1024);
      f(b);
      .......
      free (p);
      printf ("Hola mundo\n");
      return 0;
}
```



# IMAGEN DE MEMORIA AL INICIO DE LA EJECUCIÓN

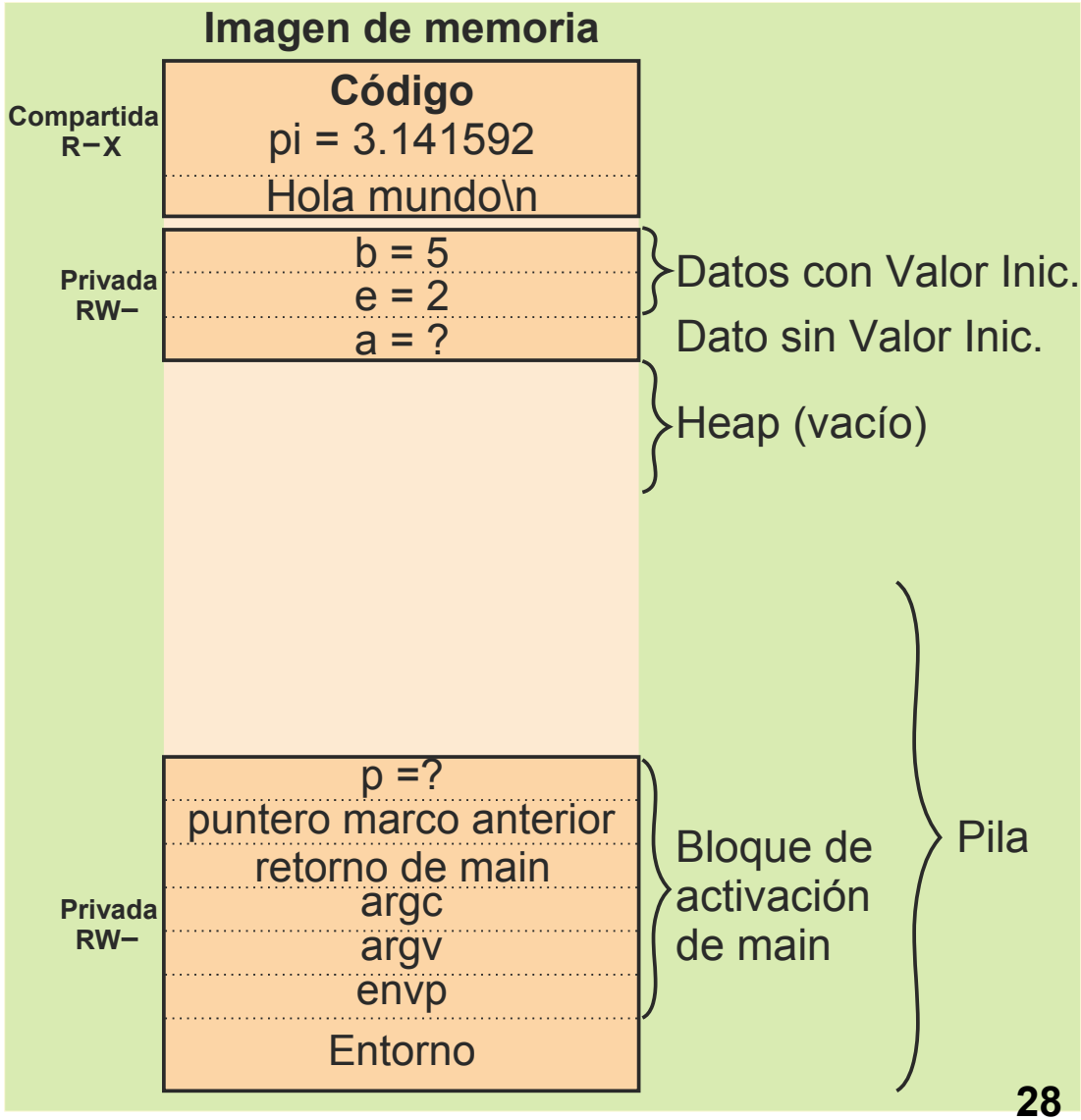


**La imagen de memoria inicial incluye el código, los datos estáticos con y sin valor inicial, y la pila inicial.**

```
int a;
int b = 5;
const float pi = 3.1416;

void f(int c) {
      int d;
      static int e = 2;
      d = 3;
      b = d + 5;
      .......
      return;
}

int main (int argc, char *argv[]) {
      char *p;
      p = malloc (1024);
      f(b);
      .......
      free (p);
      printf ("Hola mundo\n");
      return 0;
}
```



# IMAGEN DE MEMORIA AL EJECUTAR MALLOC

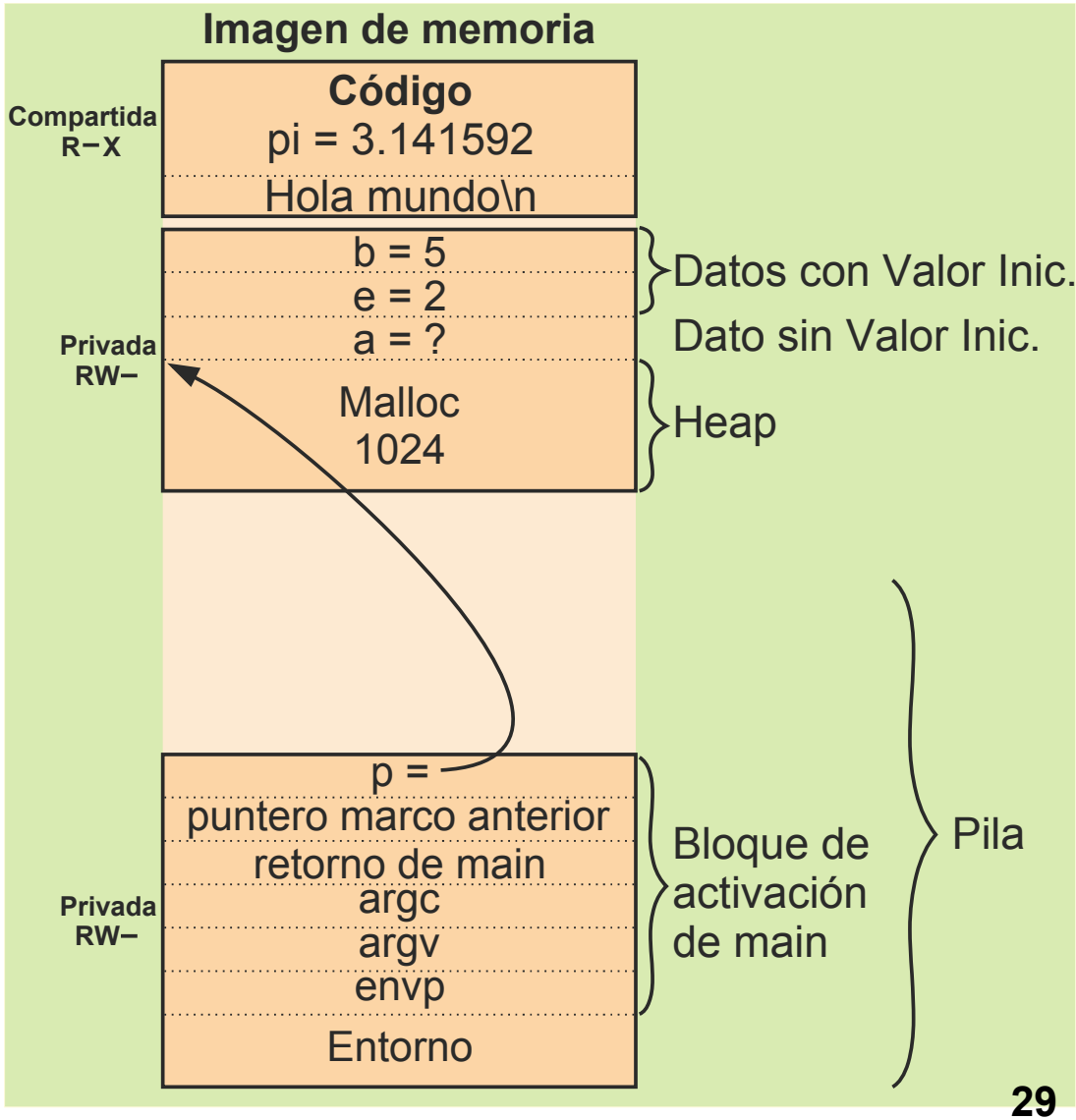


**El malloc lo resuelve la biblioteca de C. Si la región de datos no tiene libre los 1024 bytes solicitados hará una llamada al SO para aumentar la región.**

```
int a;
int b = 5;
const float pi = 3.1416;

void f(int c) {
      int d;
      static int e = 2;
      d = 3;
      b = d + 5;
      .......
      return;
}

int main (int argc, char *argv[]) {
      char *p;
      p = malloc (1024);
      f(b);
      .......
      free (p);
      printf ("Hola mundo\n");
      return 0;
}
```



# IMAGEN DE MEMORIA A LA LLAMADA A f

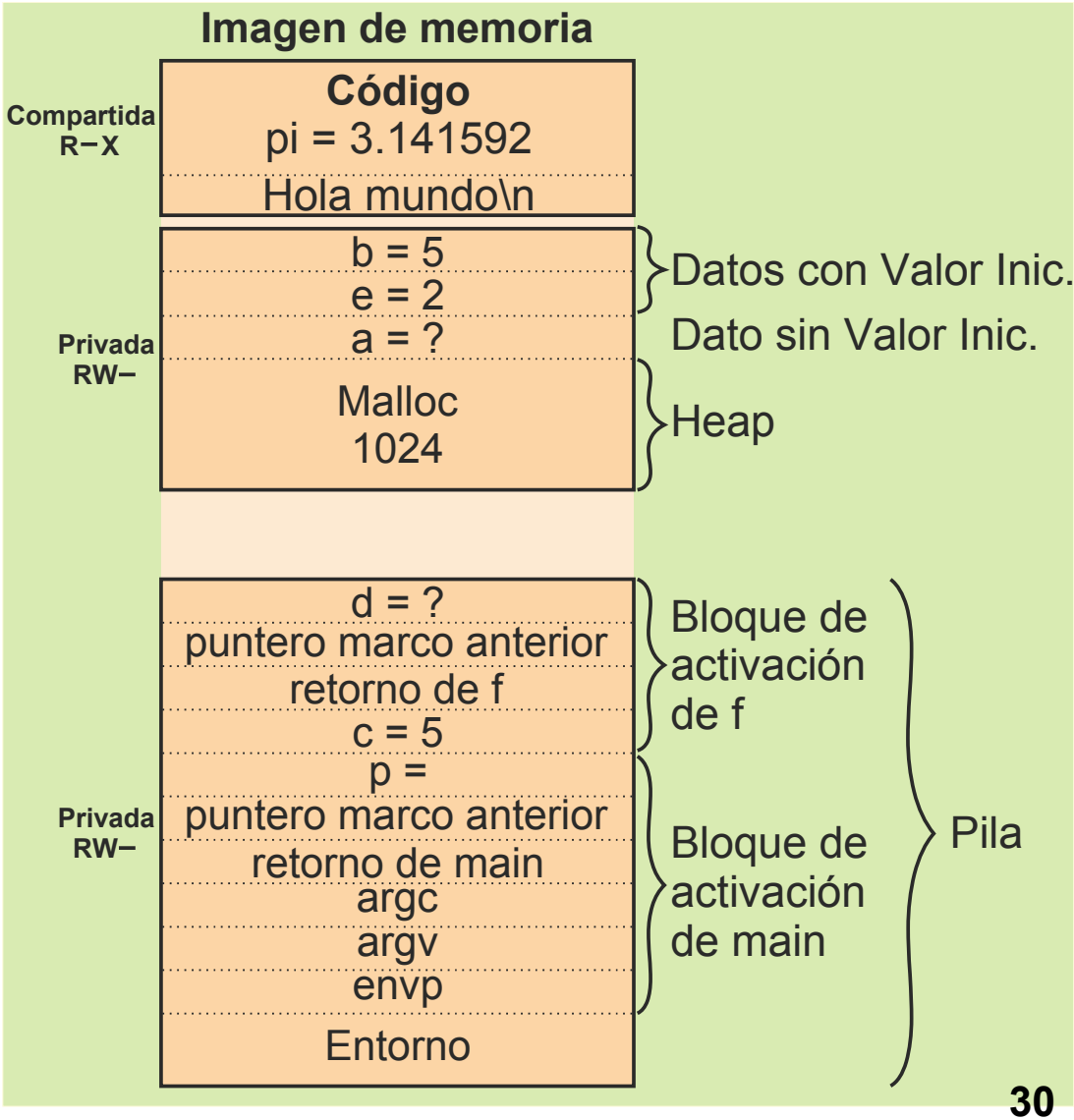


**Al realizarse la llamada el programa crea el bloque de activación con los parámetros de invocación y las variables locales de la función.**

```
int a;
int b = 5;
const float pi = 3.1416;

void f(int c) {
      int d;
      static int e = 2;
      d = 3;
      b = d + 5;
      .......
      return;
}

int main (int argc, char *argv[]) {
      char *p;
      p = malloc (1024);
      f(b);
      .......
      free (p);
      printf ("Hola mundo\n");
      return 0;
}
```

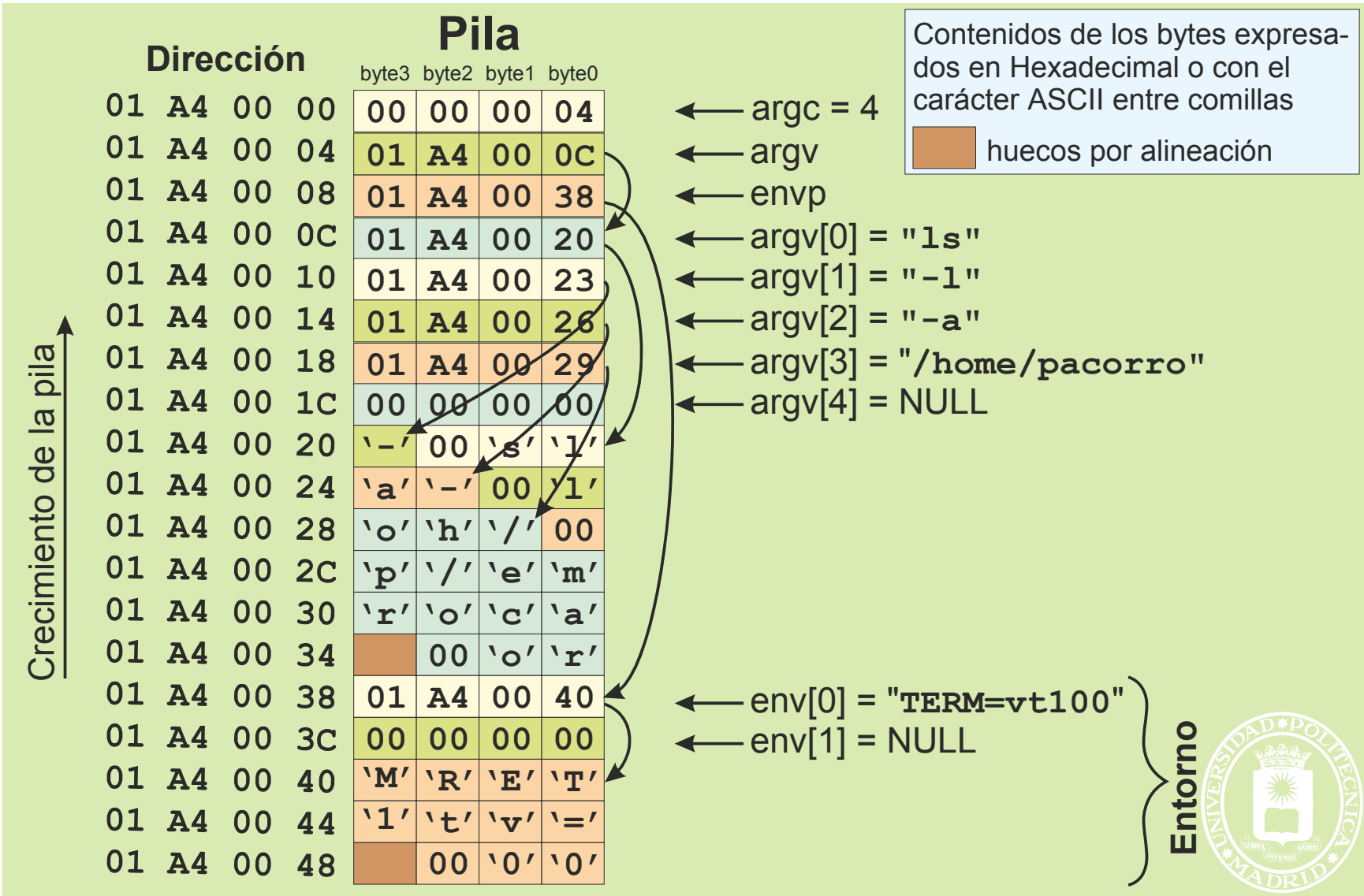


# EJEMPLO ARGUMENTOS DEL MAIN



**Invocación:** `ls -l -a /home/pacorro`

**Prototipo:** `int main(int argc, char* argv[], char* envp[]);`

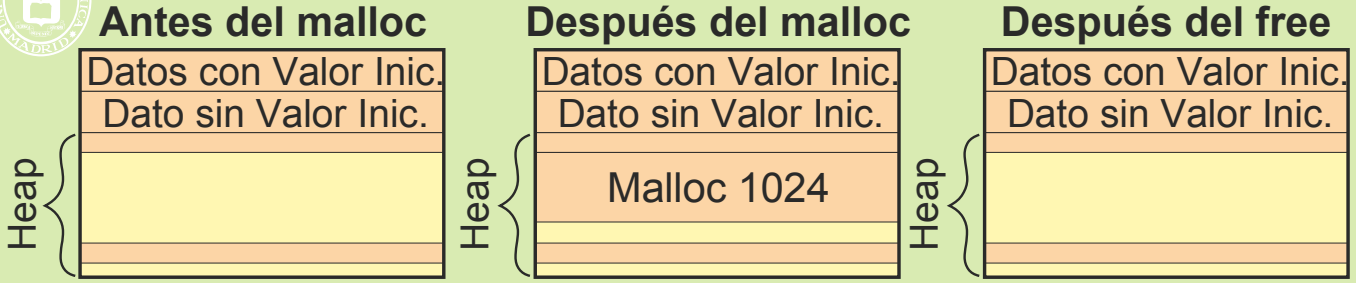


# DETALLE DEL MALLOC

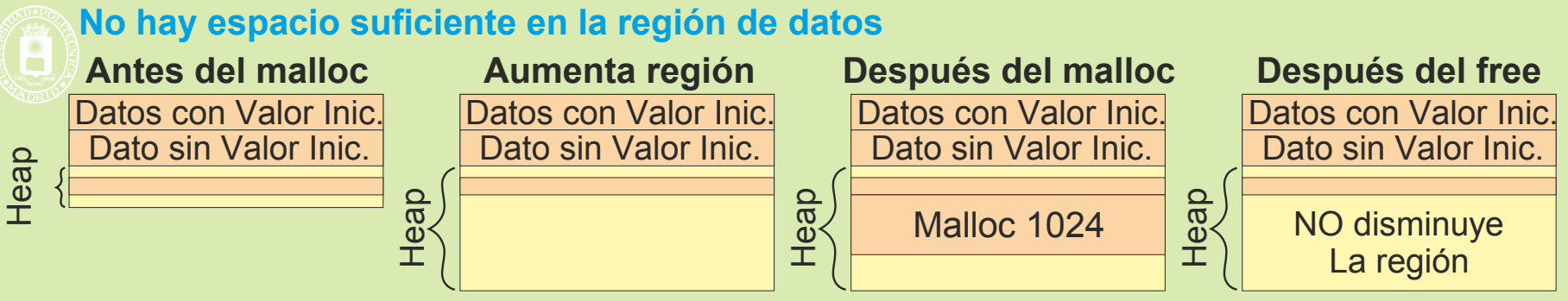


- El malloc no es un servicio del SO sino una función del lenguaje.
- Si la memoria solicitada en el malloc no cabe en la región, es necesario solicitar al SO una ampliación de la región de datos (servicio Unix: `brk`).
- La memoria obtenida sobrevive al retorno de la función donde se ubicó. Ha de liberarse explícitamente (p.e. con free() o por el recolector de basura del lenguaje). Al hacer un free no se reduce el tamaño del heap

## Hay espacio suficiente en la región de datos




## No hay espacio suficiente en la región de datos



# PARÁMETROS



- Los parámetros de un procedimiento constituyen los primeros elementos de su bloque de activación.
- El programa llamante, al ir construyendo dicho bloque de activación, mete en la pila los valores asignados a los parámetros (en orden inverso: primero el último). Una vez pasado el control al procedimiento, éste es el único que accede a dichas posiciones de la pila.
- Observamos que este mecanismo sirve para parámetros de entrada con paso por valor. Cuando se pasa la dirección de un dato, en vez del valor de un dato, se está haciendo un paso por referencia.
- Para los parámetros de salida la técnica a emplear es pasar al procedimiento, mediante el mecanismo básico anterior, una dirección de una variable o estructura del programa llamante. De esta forma, el procedimiento llamado puede acceder al dato del llamante, y modificarlo.
- El valor de la función es un parámetro de salida que se pasa por valor (por ejemplo en un registro del procesador).

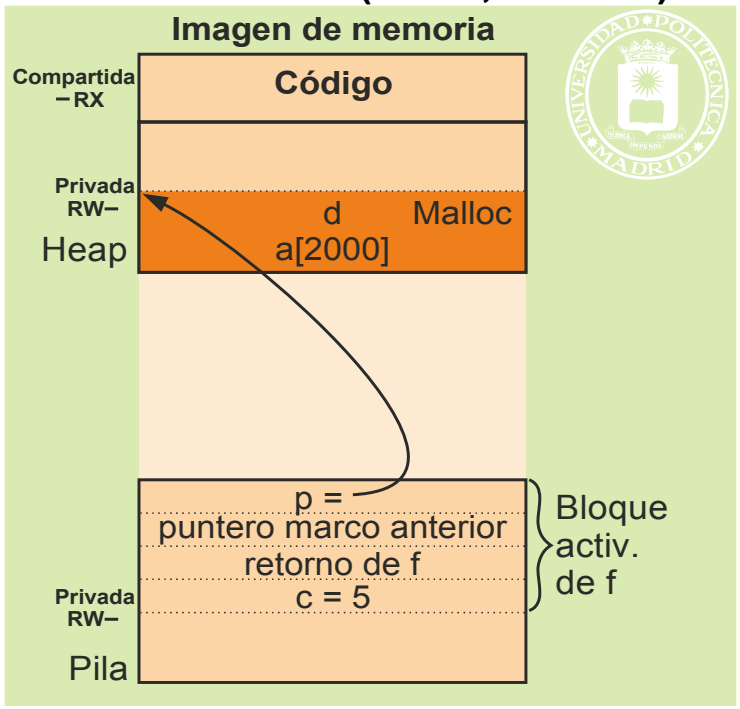
# HEAP versus BLOQUE DE ACTIVACIÓN



```
struct s {int d; char a[2000];};
struct s * fun(int a) {
        struct s *p;
        p = malloc(sizeof (struct s));
        .....
        return p;}
```

La función puede devolver p (la zona de memoria sobrevive el retorno de f)

Idóneo para funciones que crean estructuras dinámicas (listas, árboles)

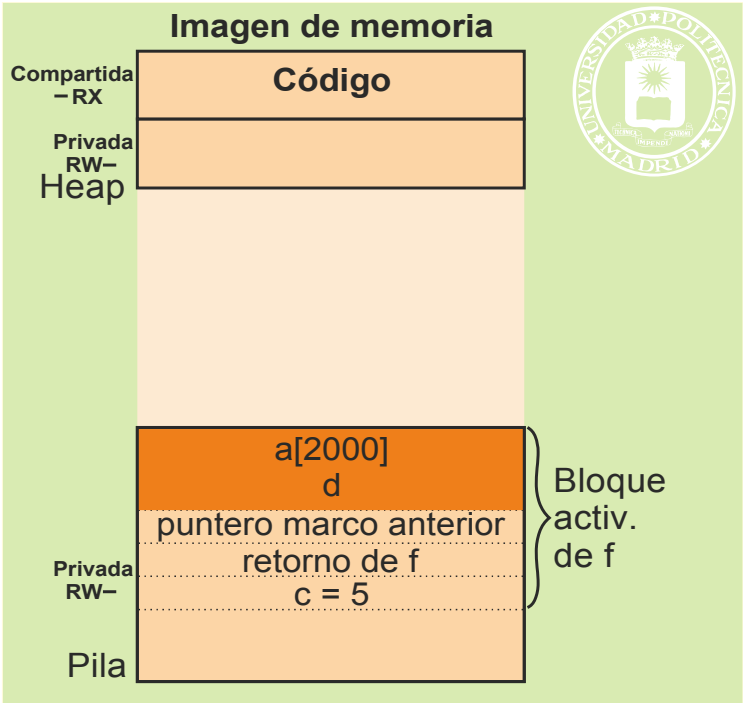



```
struct s {int d; char a[2000];};
char * fun(int a) {
        struct s p;
        .....

        // no poner return p.a;
```

Mucho menor coste computacional que el malloc o el new.

La zona de memoria se recupera en el retorno de f (no devolver dirección p.a).

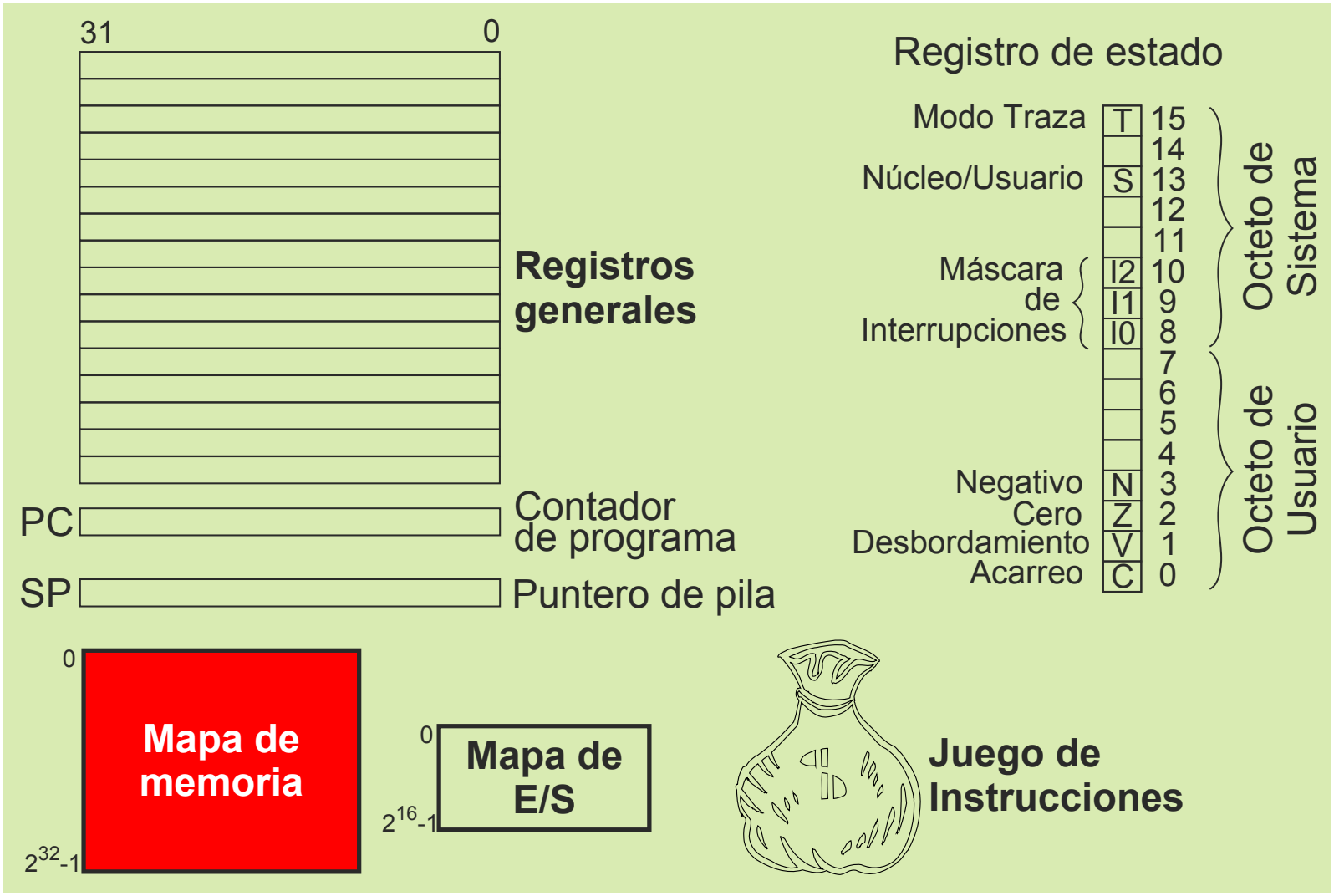



# MEMORIA VIRTUAL

# MEMORIA REAL y VIRTUAL



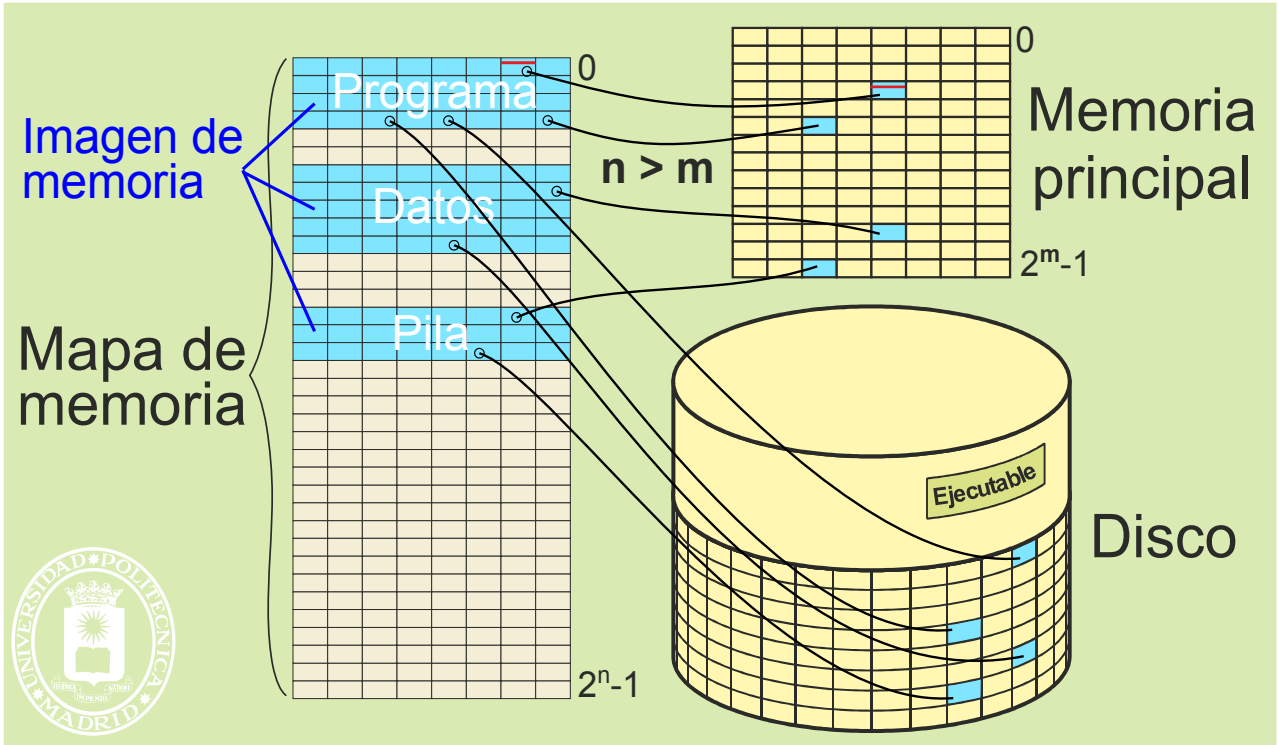
- **Memoria real: Mapa de memoria soportado sólo en memoria principal**
- **Memoria virtual: Mapa de memoria soportado conjuntamente por una parte del disco y por memoria principal.**



# PÁGINAS Y MARCOS



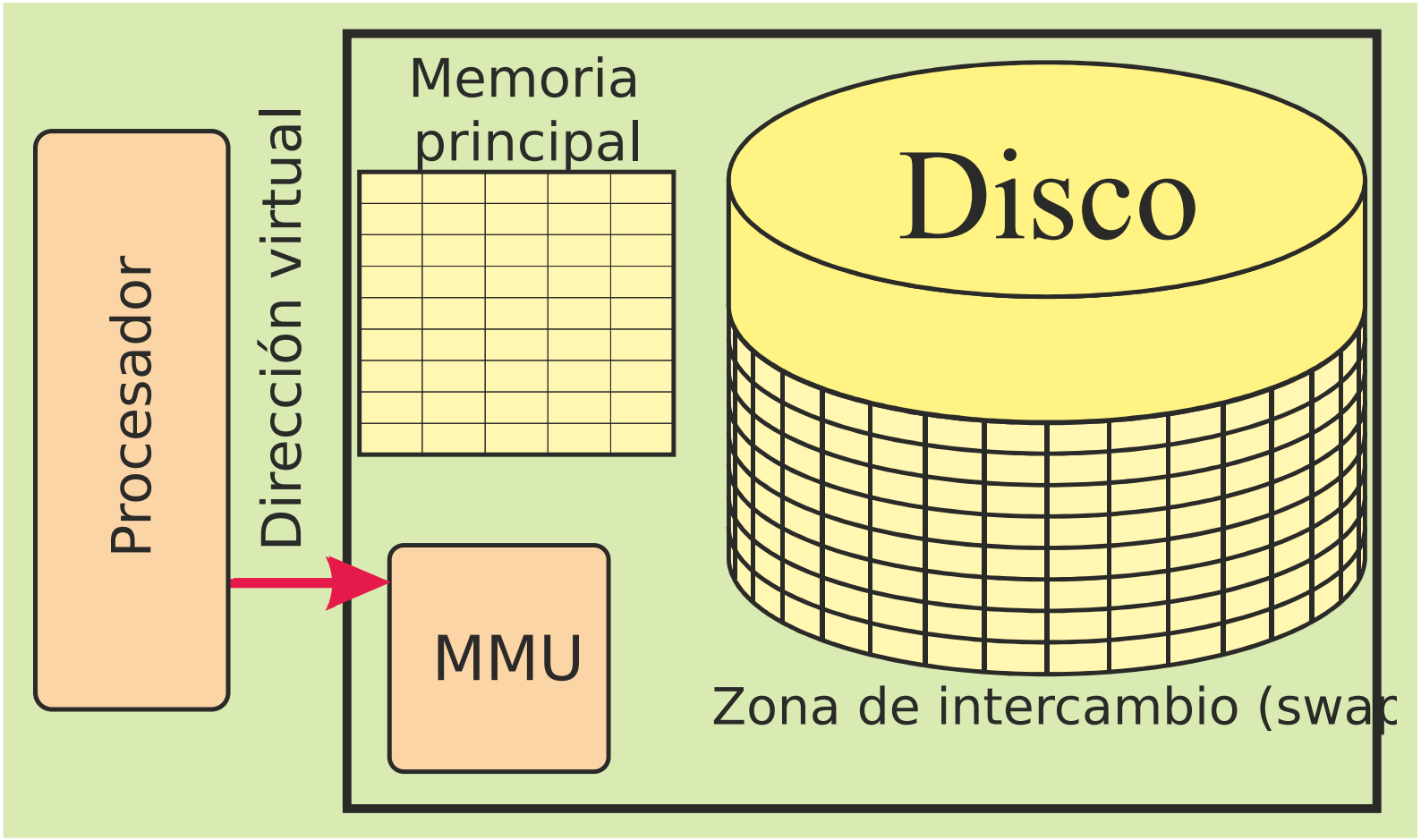
- El espacio virtual se considera organizado en páginas de tamaño fijo.
- El espacio de disco dedicado a soportar la memoria virtual se divide en **páginas**.
- La memoria principal se divide en **marcos** de página. En estos marcos se ubican las páginas.
- El SO ha de conseguir que las páginas usadas en cada momento estén ubicadas en marcos de memoria principal.
- Las páginas del mapa de memoria no asignadas, no tienen soporte físico. Ni en swap ni en marcos.



# MEMORY MANAGEMENT UNIT



- El programa máquina sólo entiende de direcciones virtuales. El procesador genera las direcciones que indica el programa, direccionando sobre su mapa de memoria, que es virtual.
- El mapa de memoria está soportado por la zona de intercambio (swap) del disco, la memoria principal y la MMU (Memory Management Unit)



# MEMORIA VIRTUAL



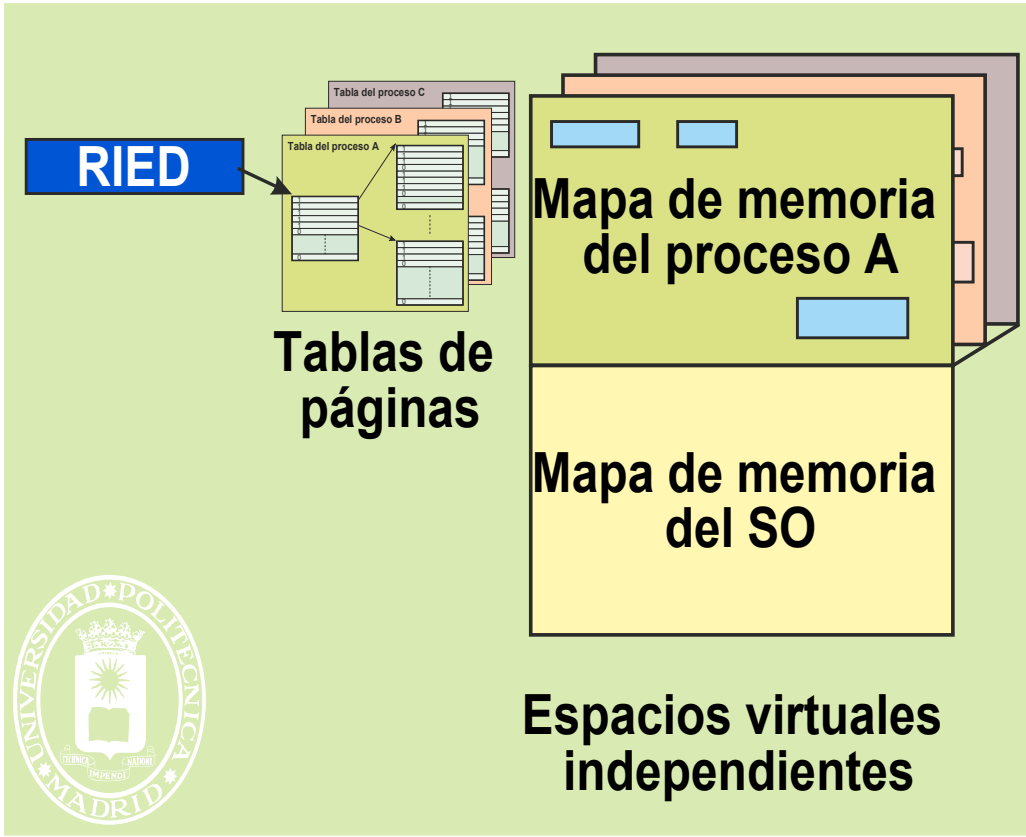
- El procesador utiliza y genera direcciones virtuales.
- Parte del mapa de memoria (virtual) está soportado en disco (swap) y parte en memoria principal.
- La MMU (Memory Management Unit) traduce las direcciones virtuales en reales.
- La traducción se basa en una estructura de información denominada tabla de páginas. Cada proceso tiene su tabla de páginas.
- La MMU produce un fallo de página (excepción) cuando la página afectada no está en memoria principal, pero sí en la tabla de páginas.
- La MMU produce una excepción de violación de memoria cuando la página afectada no está en la tabla de páginas del proceso.
- El SO trata el fallo de página, haciendo un transvase entre la memoria principal y el swap (disco).
- El SO trata la violación de memoria abortando el proceso.

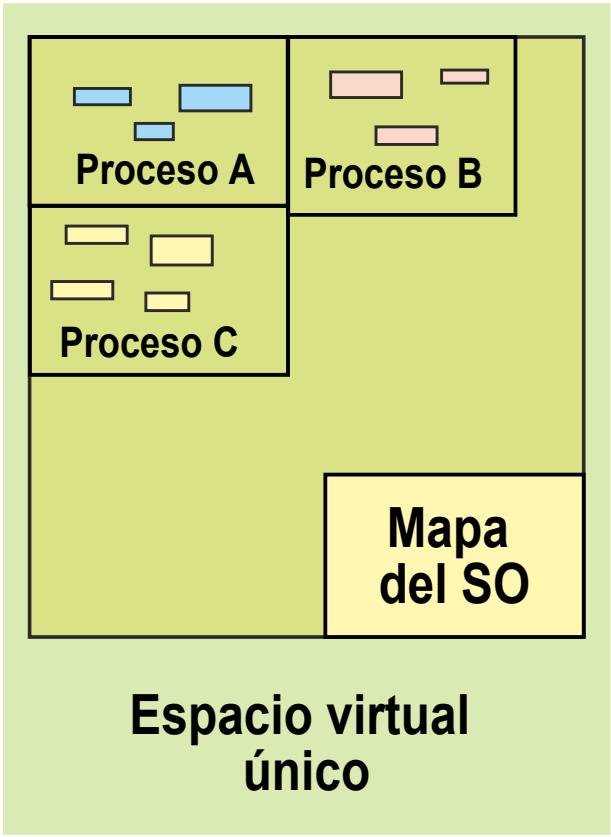
# ESPACIO VIRTUAL



**Dos alternativas:**

- **Cada proceso tiene su propio espacio virtual (máquinas de 32 bits). Esto se consigue mediante tablas de páginas independientes por proceso.**
- **Hay un único espacio virtual compartido (algunas máquinas de 64 bits).**

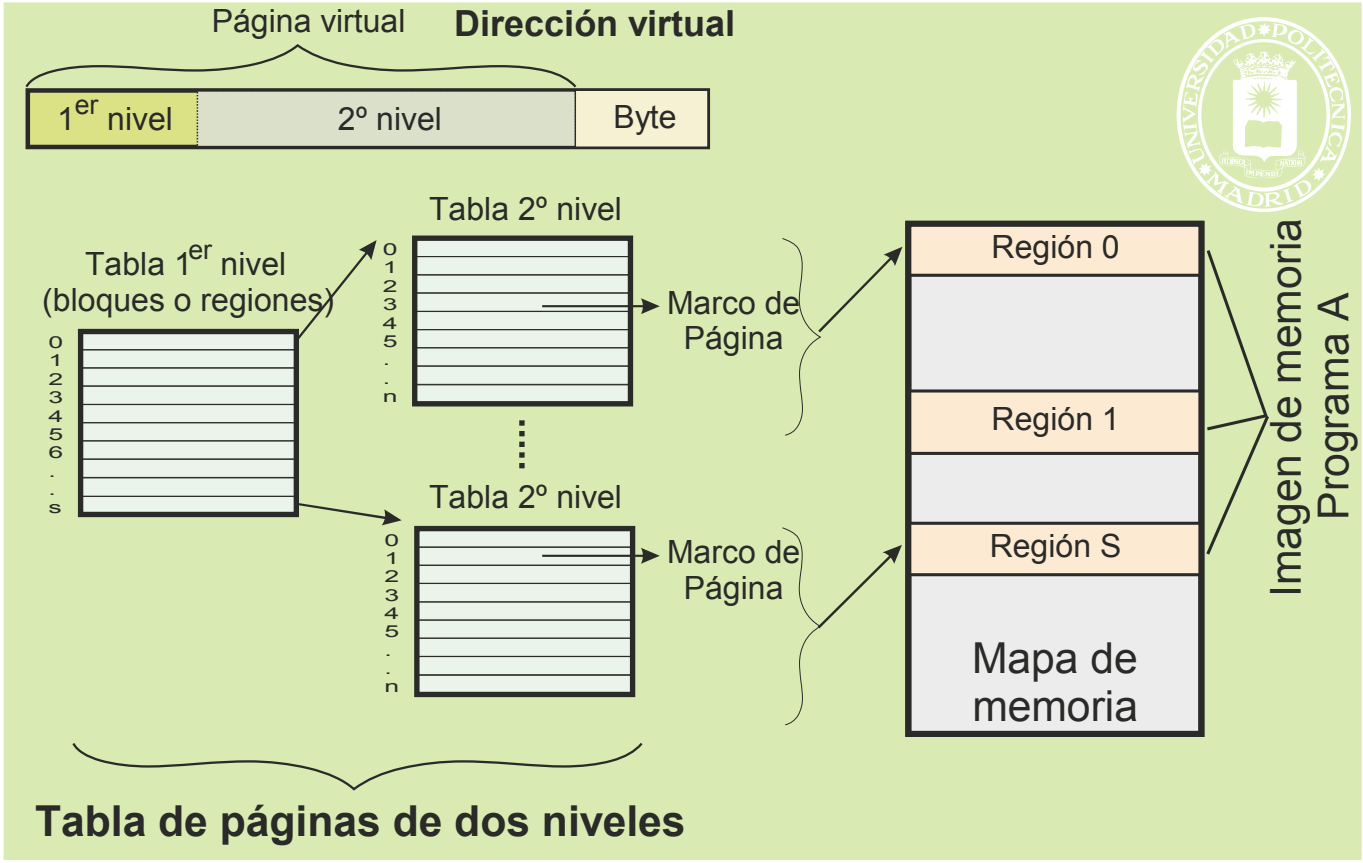


# TABLA DE PÁGINAS (ÁRBOL DE TABLAS)



- Las hojas de la tabla de páginas indican la ubicación física de la página (marco o swap) así como información de protección.
- Se almacenan en memoria principal (Mp).
- Necesita varios accesos a Mp por cada acceso de la UCP. TLB: cache especial para evitar los accesos a Mp.
- Se suelen utilizar tablas de 3 o más niveles.



# SOPORTE FÍSICO DE LA MEMORIA VIRTUAL

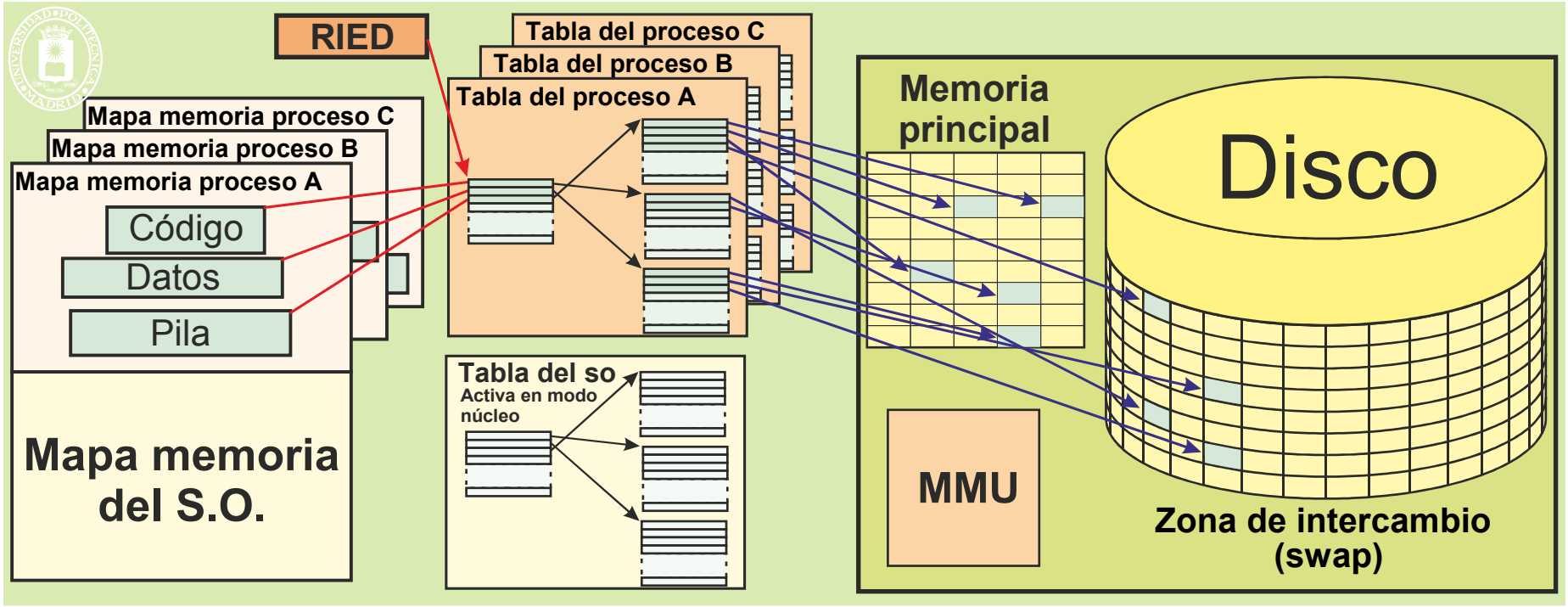


## Metainformación

- Tablas de página, almacenadas en memoria principal.

## Información

- Páginas almacenadas en la zona de intercambio del disco.
- Páginas almacenadas en marcos de página de la memoria principal.
- En algunos SO las páginas que se copian a memoria se liberan del disco, para su reutilización. En otros se mantiene la copia del disco.



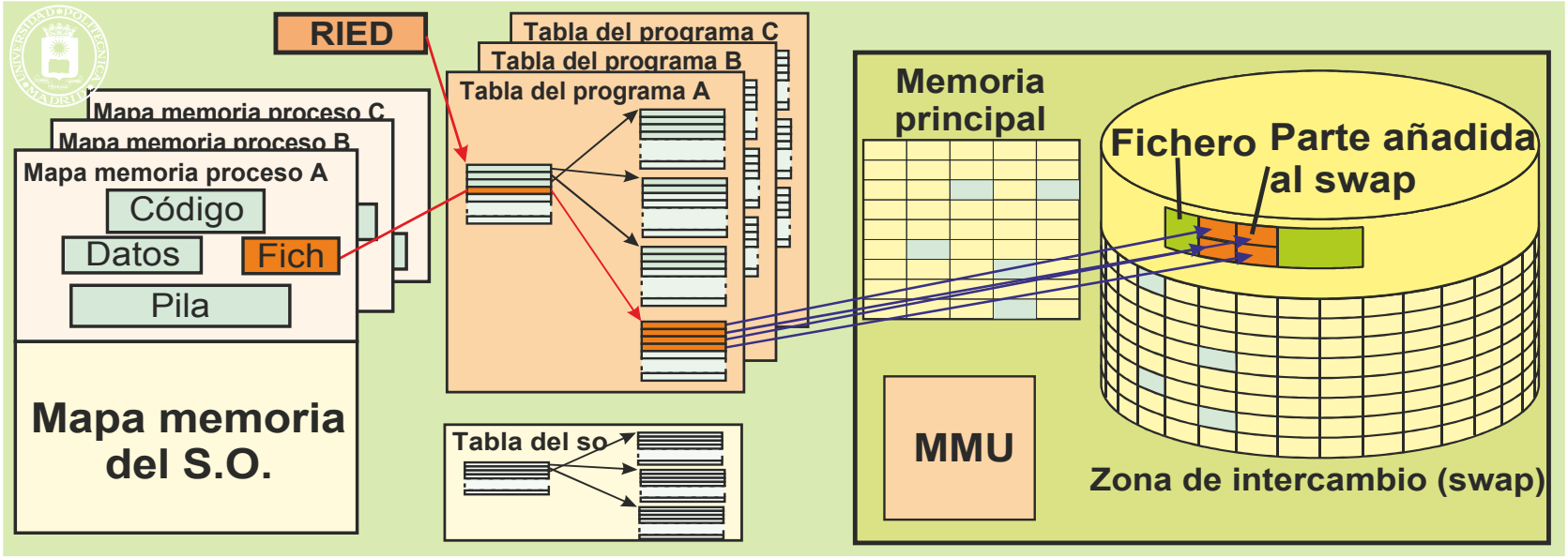

# FICHERO PROYECTADO EN MEMORIA

# FICHERO PROYECTADO EN MEMORIA



## Proyectar un fichero supone dos acciones:

- Extender la zona de intercambio con una parte de un fichero.
  - Tamaño entero de páginas, aunque se pida menos.
- Crear una región de memoria en el proceso del mismo tamaño que la parte de fichero añadida al intercambio.




## Gestión de páginas

- Al acceder el programa por primera vez a una página de esa región se realizará una migración de la página desde el fichero a un marco de página de memoria principal.
- La expulsión se hace al fichero o a swap dependiendo si la proyección es o no compartida.

# MODOS DE PROYECTAR EL FICHERO



## Proyección compartida

- Los cambios realizados sobre la región:
  - Son visibles por otros procesos que tengan proyectado el fichero.
  - Modifican el fichero en el disco.
- Las regiones de código se suelen construir proyectando la sección de código de los ficheros ejecutables en modo compartido (exec). Ello puede exigir que esta sección se ajuste a un número entero de páginas.

## Proyección privada

- Los cambios realizados sobre la región:
  - No son visibles por otros procesos que tengan proyectado el fichero.
  - Ni modifican el fichero en el disco.
- Exige realizar una copia privada de la zona de disco proyectada.
- Las regiones de datos con valor inicial se suelen construir proyectando la sección de datos con valor inicial de los ficheros ejecutables en modo privado (exec). Ello puede exigir que esta sección se ajuste a un número entero de páginas.


# MEMORIA VIRTUAL
# CONCEPTOS ADICIONALES

# FALLO DE PÁGINA



Cuando se produce un fallo de página la MMU genera una excepción que es tratada por el gestor de memoria que:

- Selecciona un marco de página.
- En caso necesario libera el marco de página, devolviendo su contenido a swap (algoritmo de remplazo).
- Lee del disco la página requerida y la transfiere al marco seleccionado.
- Actualiza la tabla de páginas con la nueva metainformación.

El gestor de memoria del SO crea y mantiene las tablas de páginas (menos los bits de referenciada y modificada que los mantiene la MMU).

# MEMORIA VIRTUAL: BUFFERING DE PÁGINAS



- **Peor caso en tratamiento de fallo de página:**
  - **2 accesos a dispositivo (que hay que resolver cuando el procesador está ocupado)**
- **Alternativa: mantener una reserva de marcos libres (buffer de páginas)**
- **Fallo de página: siempre usa marco libre (no reemplazo)**
- **Si número de marcos libres < umbral y/o procesador libre**
  - **"demonio de paginación" aplica repetidamente el algoritmo de reemplazo, seleccionado páginas a reemplazar:**
    - se marcan como no disponibles en la tabla de páginas, pero:
    - las páginas no modificadas pasan a lista de marcos libres
    - las páginas modificadas pasan a lista de marcos modificados
      - » cuando se escriban a disco pasan a lista de libres
      - » pueden escribirse en tandas (mejor rendimiento)
- **Si se referencia una página mientras está en estas listas:**
  - **fallo de página la recupera directamente de la lista (no hay E/S)**
  - **puede arreglar el comportamiento de algoritmos de reemplazo "malos"**

# MEMORIA VIRTUAL: COPIA POR ASIGNACIÓN



- El SO asigna un marco relleno con la información en vez de copiar la información al proceso.
- Útil para E/S.
- Se asignan páginas enteras.

# MEMORIA VIRTUAL: HIPERPAGINACIÓN (THRASHING)



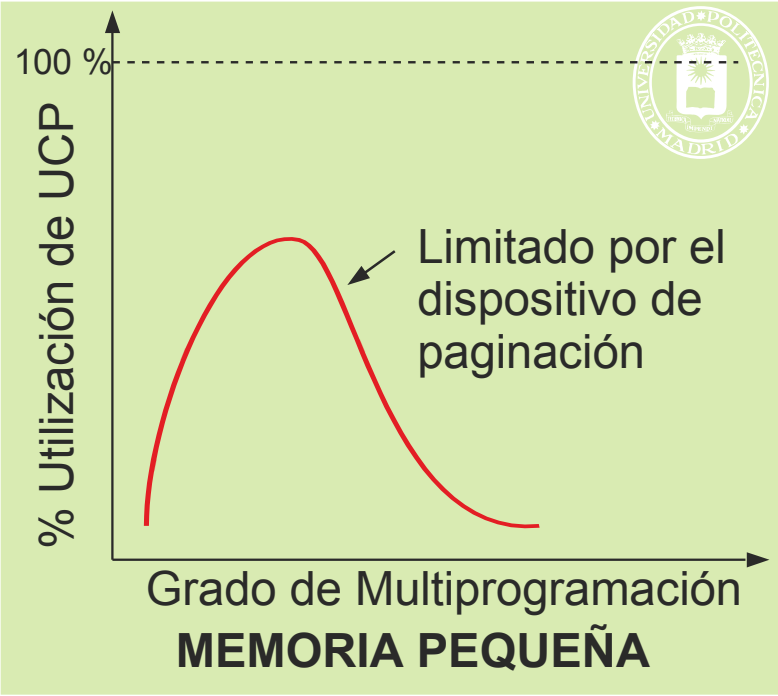
**Conjunto de trabajo**: páginas que emplea el proceso en un pequeño intervalo

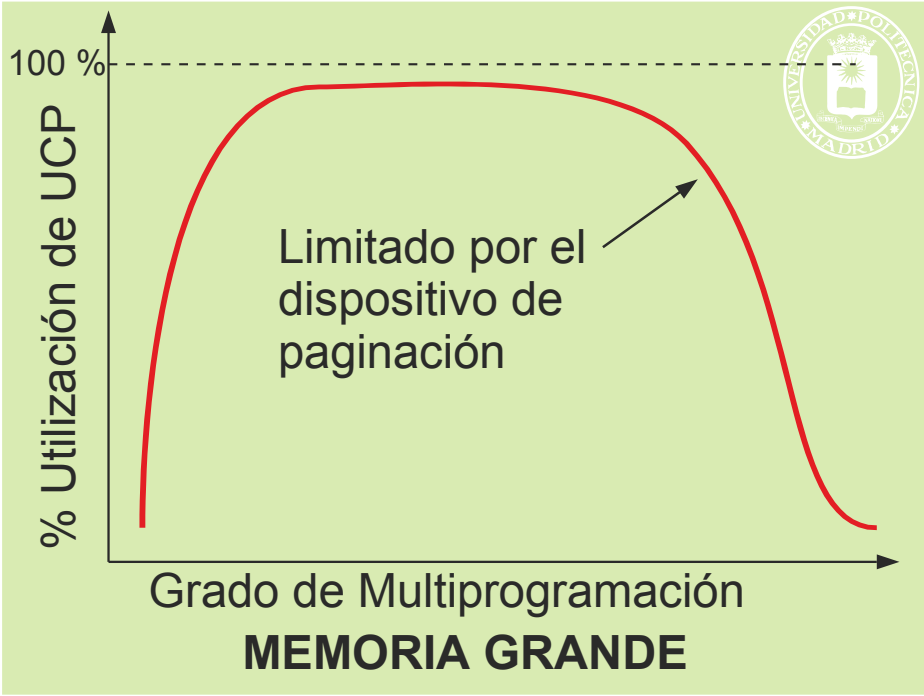
**Conjunto residente**: páginas que tiene el proceso en marcos de Mp

**Grado de multiprogramación**: número de procesos en memoria.

- Al aumentar el grado de multiprogramación:
  - hay menos marcos de página para cada proceso
  - aumentan los fallos de página.

**Hiperpaginación**: Tasa excesiva de fallos de página



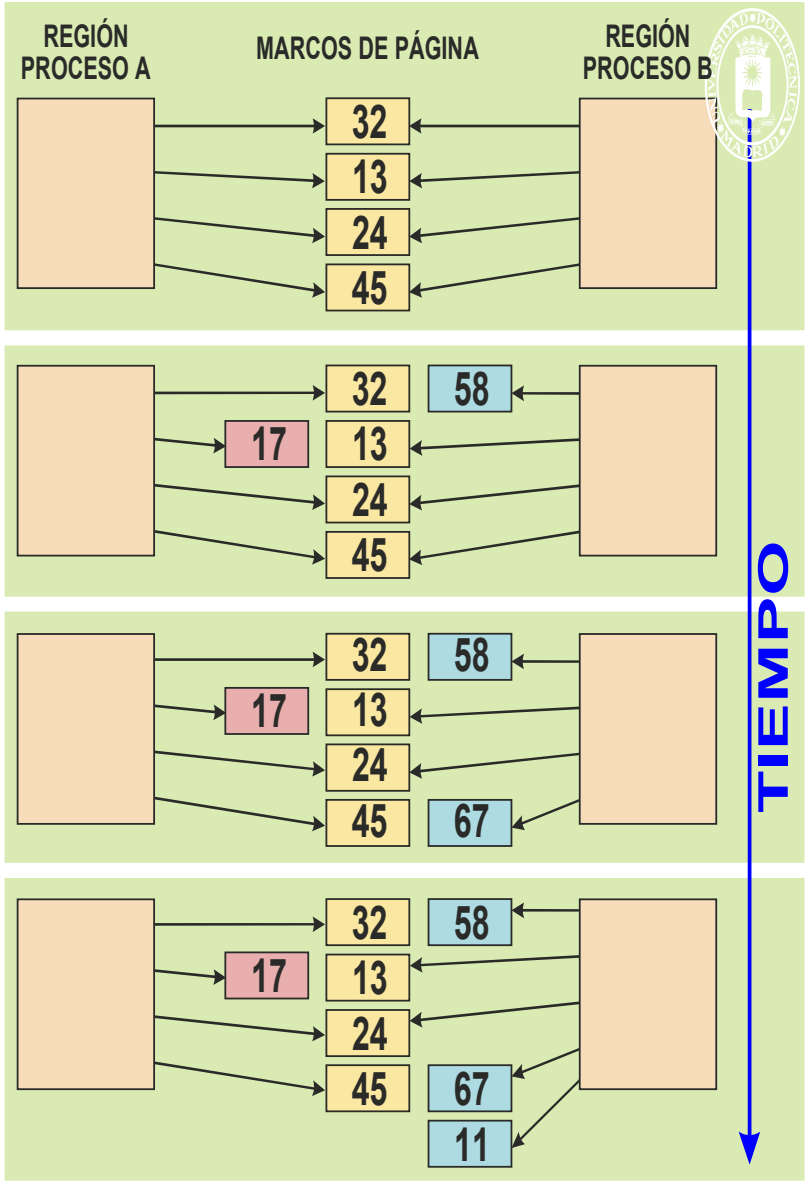
## Hiperpaginación

- **Tasa excesiva de fallos de página de un proceso o en el sistema**
- **Con asignación fija: Hiperpaginación en $P_i$**
  - **Si conjunto residente de $P_i$ < conjunto de trabajo $P_i$**
- **Con asignación variable: Hiperpaginación en el sistema**
  - **Si nº marcos disponibles < Σ conjuntos de trabajo de todos**
  - **Grado de utilización de UCP cae drásticamente**
  - **Procesos están casi siempre en colas de dispositivo de paginación**
  - **Solución (similar al swapping): Control de carga**
    - disminuir el grado de multiprogramación
    - suspender 1 o más procesos liberando sus páginas residentes
  - **Problema: ¿Cómo detectar esta situación?**

# MEMORIA VIRTUAL: DUPLICADO DE UNA REGIÓN



- Copia de región de un proceso en el mapa de otro
  - Operación costosa: se debe copiar contenido
- Optimización: copy-on-write (COW)
  - Se comparte una página mientras no se modifique
  - Si un proceso la modifica se crea una copia para él
  - "Duplicado por demanda"
- COW es útil para el fork y para los datos con valor inicial
- Control de páginas en COW
  - Contador por página



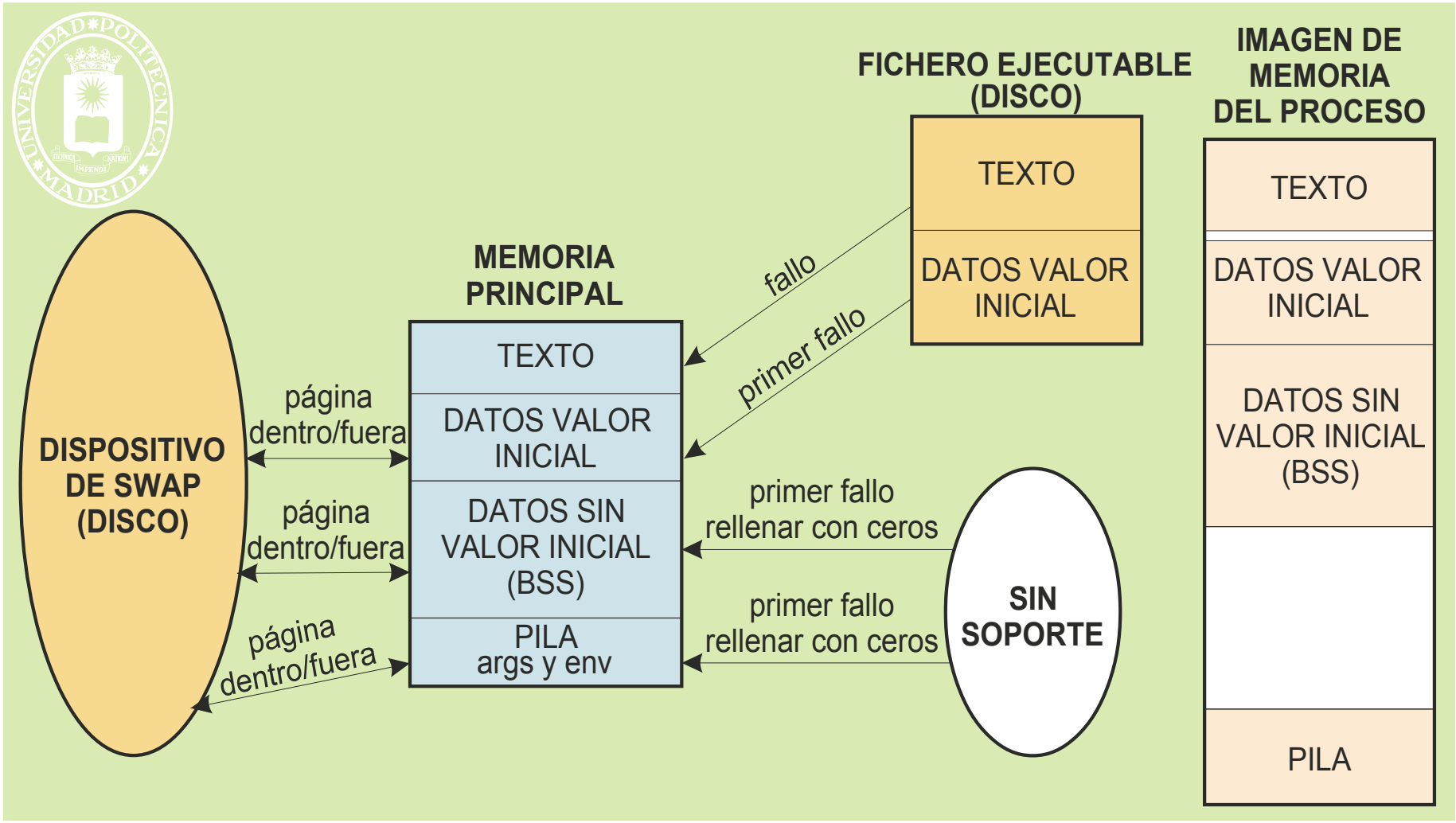
# MEMORIA VIRTUAL: COPY-ON-WRITE



- **Implementación de COW**
  - **Se comparten páginas de regiones duplicadas pero:**
    - se marcan de sólo lectura y con bit de COW
    - primera escritura –› Fallo de protección –› copia privada
  - **Puede haber varios procesos con misma región duplicada**
    - Existe un contador de uso por página
    - Cada vez que se crea copia privada se decrementa el contador
    - Si llega a 1, se desactiva COW ya que no hay duplicados
- **FORK con COW**
  - **Se comparten todas las regiones**
  - **Las regiones privadas se marcan como COW en padre e hijo**
- **Resultado de la optimización del FORK:**
  - **En vez de duplicar espacio de memoria sólo se duplica la tabla de páginas**

# MEMORIA VIRTUAL: EXEC





# SWAP



- **Preasignación de swap: cada página virtual asignada tiene un espacio fijo en el swap.**
- **Sin preasignación de swap: una página virtual asignada puede estar en:**
  - swap
  - un marco de memoria
  - un fichero proyectado
  - a rellenar a 0
  - **En reemplazo hay que copiar la página a swap (menos si es de la región de código o de un fichero proyectado compartido).**
- **Solamente se dedica swap a las regiones asignadas a los procesos.**
  - **Tamaño típico de swap es del orden del doble de la memoria principal disponible.**
- **Swap reside normalmente en el disco. En:**
  - **una partición**
  - **o un fichero**
- **En algunos casos el swap reside en una memoria RAM.**


# PROTECCIÓN DE MEMORIA

**Monoprogramación: Hay que proteger al S.O.**

**Multiprogramación: Proteger al S.O. y a los procesos entre sí.**

## Es necesario validar todas las direcciones que genere el programa.

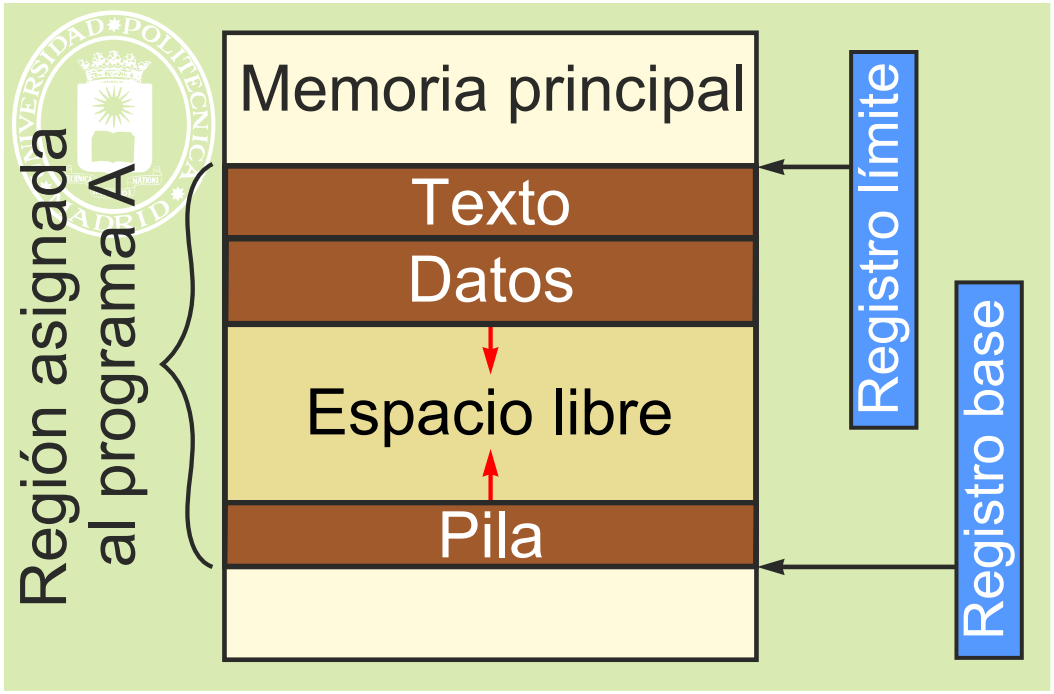
- **La detección o supervisión debe realizarla el hardware del procesador.**
  - **El HW comprueba que la dirección y el tipo de acceso son correctos.**
- **El tratamiento de la infracción lo hace el SO (p.e. aumenta la pila o envía una señal al proceso que normalmente lo mata).**

## Rellenar con 0

- **Un posible problema de seguridad aparece cuando se le suministra a un proceso un soporte físico de memoria (ya sea una página en el espacio de swap o un marco de página) y éste no tiene valor inicial.**
- **Si no se rellena a 0 por el SO el proceso podrá leer el contenido que dejó en ese soporte físico el proceso que lo utilizó anteriormente, pudiendo recuperar información valiosa del mismo.**

## Máquinas con memoria real (sin memoria virtual)

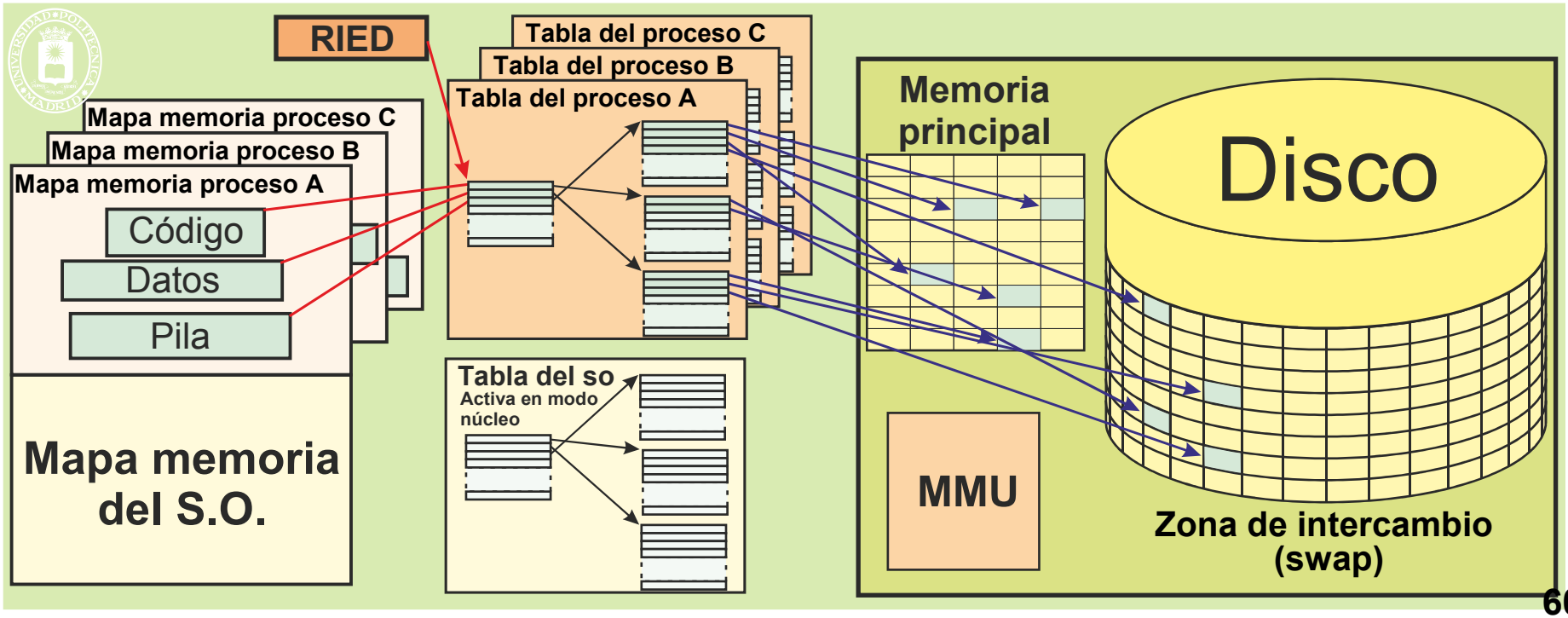
- La memoria principal tiene asociado unos registros valla que limitan la región que puede acceder el programa.
- Si un programa intenta acceder a una posición que esté fuera de su región, el HW de los registros valla genera una **excepción** de violación de memoria.
- El SO trata la excepción. Se entera de que el programa trata de acceder a una región no válida y toma las acciones correctoras oportunas.



# PROTECCIÓN DE MEMORIA EN MÁQUINA VIRTUAL



- Espacios de memoria virtual independientes.
- La MMU utiliza la tabla del programa en ejecución. Sólo permite accesos a las páginas de la tabla de ese programa.
- La tabla de páginas tiene permisos rwx de cada región y de cada página.
- Si el programa intenta salirse de la región asignada la MMU produce una excepción de violación de memoria.
- En modo usuario sólo está activa la tabla de páginas apuntada por RIED. En modo núcleo está además activa la del SO.



# ERRORES DE ACCESO A MEMORIA



## Errores y avisos detectados por el HW y tratados por el SO.

- Error: Acceso a una dirección fuera de las regiones asignadas.
  - Acción del SO: Enviar una señal al proceso, que suele matarlo.
- Error: Acceso a una dirección correcta pero con un tipo de acceso no permitido (en algunos casos es un aviso).
  - Acción del SO: Enviar una señal al proceso, que suele matarlo.
- Aviso: Desbordamiento de pila.
  - Se pone la última página como de lectura. Si se intenta escribir es que queda poca pila.
  - Acción del SO: Incrementar la región de pila. Si no es posible el incremento, enviar una señal al proceso para matarlo.
- Aviso: Fallo de página.
  - Acción del SO: Realizar la migración de la página fallida y actualizar la tabla de páginas afectada.
- Aviso: COW.
  - Las páginas están como de lectura.
  - Acción del SO: Desdoblar la página.


# MEMORIA COMPARTIDA

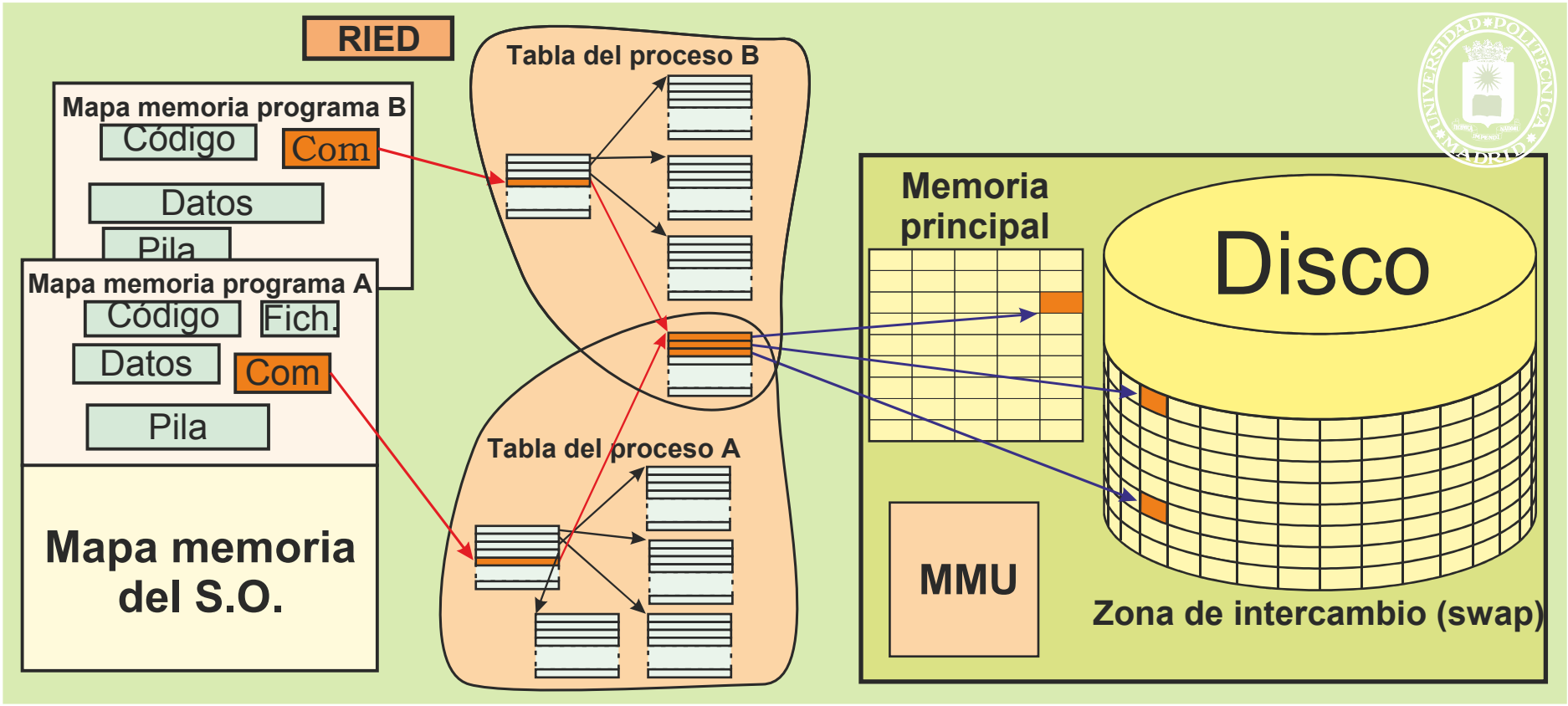
# COMPARTIR MEMORIA



- Se rompe el concepto de independencia de los procesos.
- Se permite que unas regiones –del mismo tamaño– de dos o más procesos utilicen el mismo soporte físico y, por tanto, compartan la misma información.
- Se hace bajo control del S.O. y se exige que los procesos lo soliciten expresamente.
- El código de los procesos y librerías dinámicas se comparte por defecto.
- Beneficios:
  - Procesos que ejecutan el mismo programa comparten el código.
  - Mecanismo muy rápido de comunicación entre procesos.
- Utilización
  - Los procesos han de ponerse de acuerdo para escribir y leer de esa memoria común de forma ordenada (según se tratará en el tema de comunicación y sincronización entre procesos).

# COMPARTIR MEMORIA

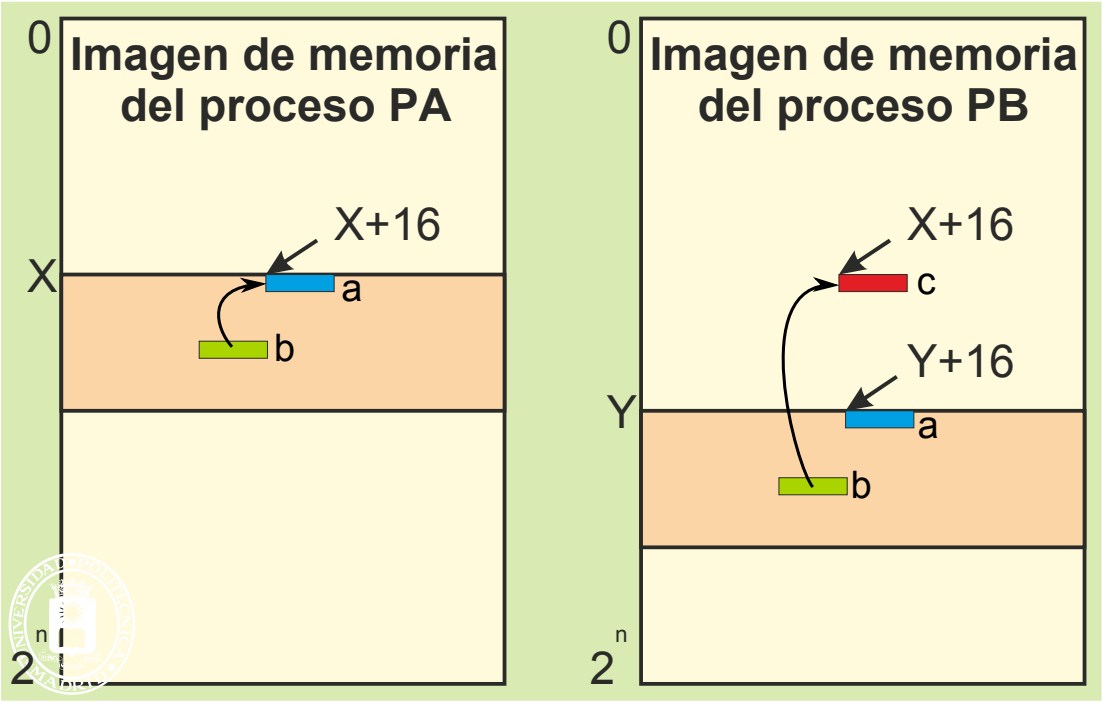




# DIRECCIONES EN MEMORIA COMPARTIDA



- La región compartida se encuentra en la dirección X en el caso del proceso PA y en la dirección Y en el caso del proceso PB.
- La misma palabra se accede en PA con la dirección X+16 y PB con la dirección Y+16.
- Supongamos que PA crea un puntero para manejar esa variable y almacena el puntero en memoria compartida, para que lo pueda utilizar PB.
- Si PB utiliza ese puntero intentará acceder a una dirección incorrecta que puede estar fuera de la región compartida.




Problema de autorreferencias


# BIBLIOTECAS DINÁMICAS

# LLAMADA A PROCEDIMIENTO



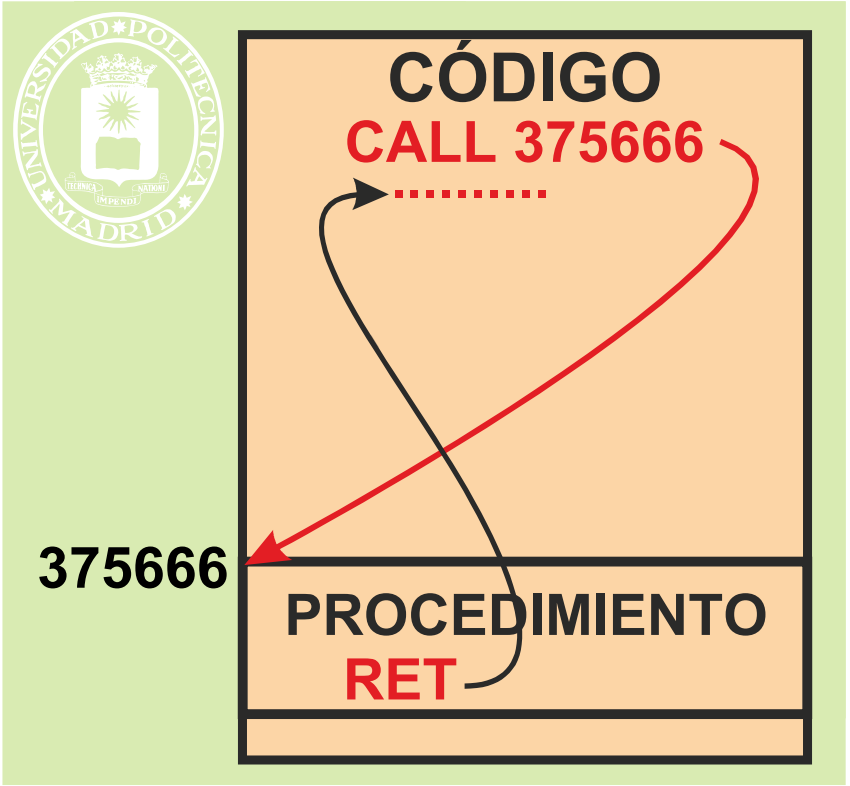
La llamada a procedimiento supone una combinación de instrucciones de máquina **call** y **ret**.

La instrucción de call ha de tener la dirección de memoria donde comienza el procedimiento.

Llamaremos **punto de enlace** a esa dirección de comienzo del procedimiento.

**Biblioteca**: colección de módulos objeto relacionados.

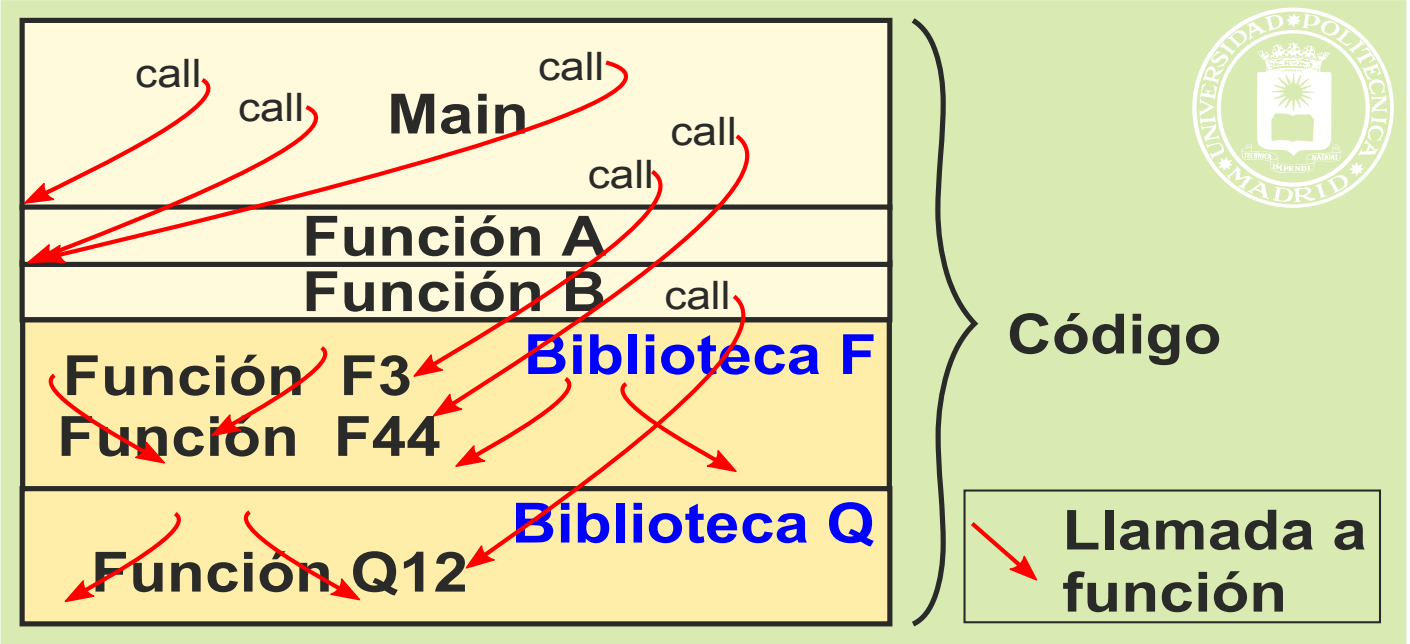
Bibliotecas del sistema o creadas por el usuario.



# BIBLIOTECAS ESTÁTICAS



- Montaje: enlaza los módulos objeto con copias de módulos de las bibliotecas.
- Ejecutable autocontenido.
- Desventajas del montaje estático
  - Ejecutables grandes.
  - Código de función de biblioteca repetido en muchos ejecutables.
  - Múltiples copias en memoria del código de función de biblioteca.
  - La actualización de la biblioteca implica volver a montar cada programa que las usa.



## Carga y montaje de la biblioteca en tiempo de ejecución

### Ventajas del montaje dinámico

- Menor tamaño de los ejecutables.
  - Código de rutinas de biblioteca sólo en el fichero de biblioteca.
- Los procesos pueden compartir el código de la biblioteca (región compartida en modo read only).
- Actualización automática de las bibliotecas: Uso de versiones.

### Inconvenientes

- Bibliotecas del mismo nombre incompatibles (versiones incompatibles).
- Eliminación de bibliotecas obsoletas.
- Sobrecarga en tiempo de ejecución.

### Alternativas

- Montaje en tiempo de carga en memoria.
- Montaje al invocar el procedimiento.
- Montaje explícito.

## Montaje en tiempo de carga en memoria

Al iniciar la ejecución del programa el SO completa el ejecutable incluyendo todas las bibliotecas dinámicas y resolviendo los enlaces.

## Montaje al invocar el procedimiento

Solamente cuando el proceso trata de ejecutar un procedimiento por primera vez se incluye la biblioteca correspondiente y se resuelve su enlace.

Produce menos sobrecarga que la solución anterior. El ejecutable contiene:

- Nombre de la biblioteca así como los puntos de enlace de las rutinas de la biblioteca.
- Rutina encargada de cargar y montar la biblioteca en tiempo de ejecución al realizarse la primera referencia a un procedimiento de la misma.

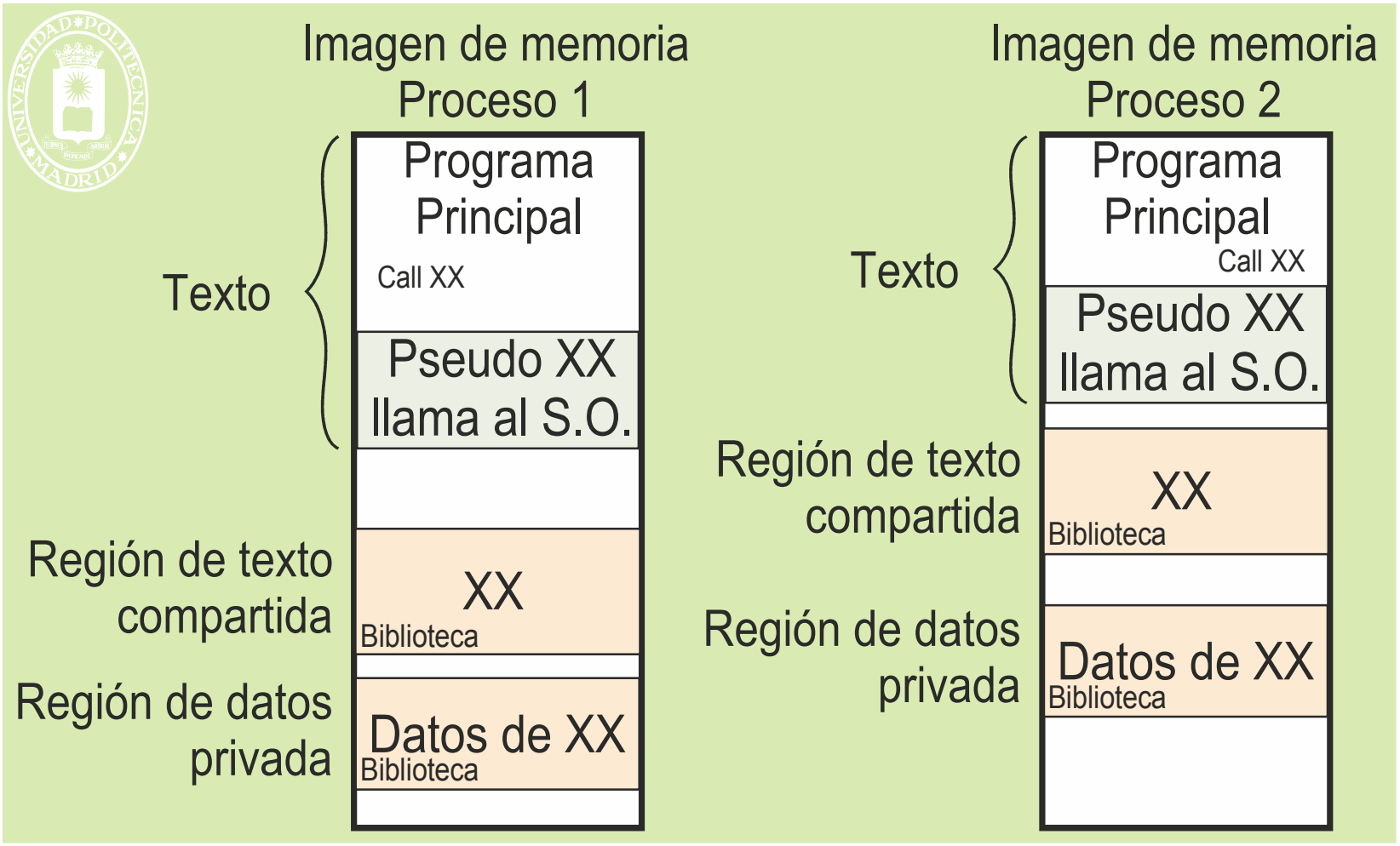
## Montaje explícito

- Es más flexible que anterior puesto que se puede seleccionar la biblioteca en tiempo de ejecución.
- El ejecutable ha de preguntar a la biblioteca sobre los puntos de enlace de sus funciones.
- Es más frágil que los anteriores, puesto que el compilador no ayuda a verificar los prototipos de las llamadas.

# REGIONES PARA UNA BIBLIOTECA DINÁMICA



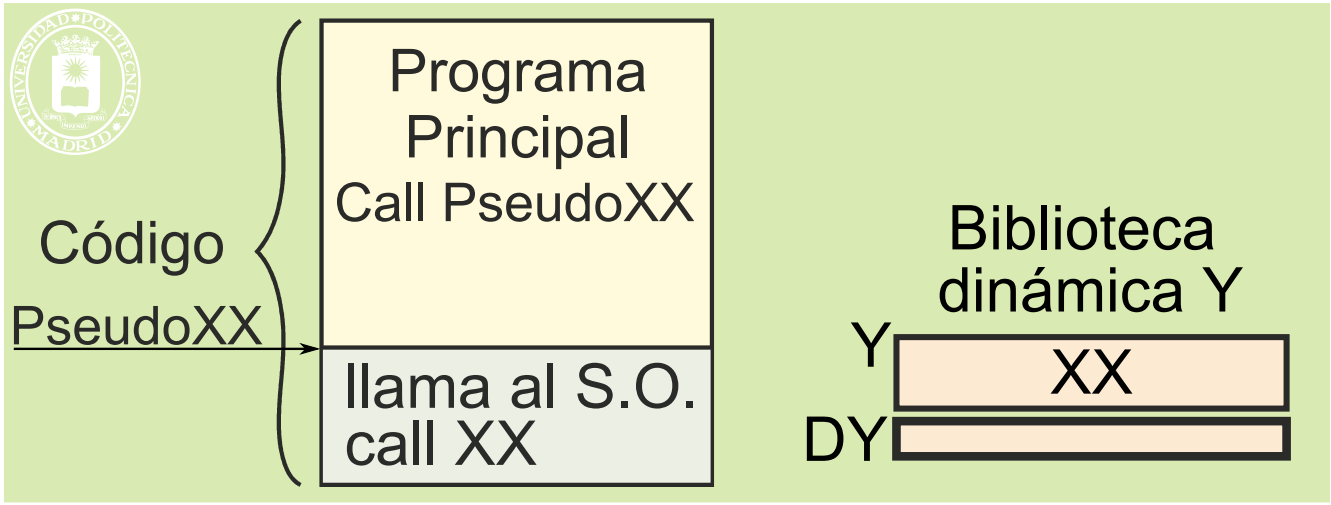
- La región de código de la biblioteca dinámica puede ser compartida.
- La región de datos de la biblioteca dinámica ha de ser privada a cada proceso que utiliza la biblioteca.



# MONTAJE AL INVOCAR EL PROCEDIMIENTO



- El programa principal llama a la pseudo función XX.
- Esta función contacta con el SO y le pide la biblioteca Y que contiene la función XX.
- El SO operativo comprueba si Y ya está cargada en memoria por algún proceso.
  - En caso negativo la carga.
  - En caso positivo comparte esa región de memoria con este proceso.
- El SO crea la región DY de datos necesaria para la biblioteca Y.
- El SO devuelve a la Pseudo XX los punteros con las posiciones de Y y DY.
- La función Pseudo XX llama a la función XX, sumando al puntero Y la posición relativa de la función XX (punto de enlace de XX).



# EJEMPLO DE COMPILACIÓN I



mi_cos.c:

```
#include <stdio.h>
#include <math.h>
double coseno(double v) {
   return cos(v);
}
int main() {
   double a, b = 2.223;
   a = coseno(b);
   printf("coseno de %lf:%.f\n",b,a);
   return 0;
}
```

```
# gcc mi_cos.c -o cos_dyn -lm
```

Compilación estándar (dinámica)

```
# nm cos_dyn
....
          U cos
08048450  T coseno
08048474  T main
          U printf
....
```

Mandato que devuelve los símbolos del ejecutable

[U (undefined)]

[T (texto)]

```
# ldd cos_dyn
    libm.so.6 => /lib/libm.so.6 (0x4001b000)
    libc.so.6 => /lib/libc.so.6 (0x4003c000)
```

Mandato que devuelve las bibliotecas dinámicas

# EJEMPLO DE COMPILACIÓN II



```
# gcc -static mi_cos.c -o cos_sta -lm
```
Compilación estática

```
# nm cos_sta
```
Mandato que devuelve los símbolos del ejecutable

```
....
08048240 T cos
080481c0 T coseno
080481e4 T main
080485e0 T printf
....
```

```
# ldd cos_sta
    not a dynamic executable
```
Mandato que devuelve las bibliotecas dinámicas

Si se hace un size de cada uno de los ejecutables se obtiene:

**Dinámico:**

```
# size cos_dyn
      text    data     bss       dec      hex       filename
      1096     280      24      1400      578       cos-dyn
```

**Estático:**

```
# size cos_sta
      text    data     bss       dec      hex       filename
    360013   10724    3976    374713   5b7b9       cos-sta
```

data: región de datos con valor inicial, bss: región de datos sin valor inicial


# SERVICIOS UNIX DE GESTIÓN DE MEMORIA

# SERVICIOS UNIX DE GESTIÓN DE MEMORIA



## Aumento del tamaño de una región

- `int brk( void *addr);`
  - Coloca , si es posible, el fin del heap en la dirección addr.
  - Supone que aumenta o disminuye el heap del segmento de datos.
  - Lo utiliza la biblioteca del lenguaje que maneja memoria dinámica.
- La región de pila crece automáticamente según la necesidad.

## Creación y destrucción de regiones

- En UNIX se realiza mediante las primitivas para compartir memoria y para proyectar ficheros en memoria.

## Proyección de ficheros (`mmap` y `munmap`)

## Uso de memoria compartida

- `mmap`: Si se proyecta el mismo fichero en dos o más procesos se está compartiendo memoria, por lo que el servicio mmap se utiliza para compartir memoria.
- Existen otros servicios según la versión de UNIX.

## Montaje explícito de biblioteca dinámica (`dlopen`)

## Bloquear páginas en memoria (`mlock`)

# PROYECCIÓN DE FICHEROS



**Establecer una proyección entre el espacio de direcciones de un proceso y un descriptor de fichero u objeto de memoria compartida.**

- `void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);`
- ***addr*** **dirección dónde se quiere la región. En general NULL**
- ***len*** **especifica el número de bytes a proyectar. Se proyecta un número entero de páginas, por lo que la proyección puede ser mayor que len.**
- ***prot*** **el tipo de acceso (lectura, escritura o ejecución).**
  - **PROT_READ, PROT_WRITE, PROT_EXEC o PROT_NONE.**
- ***flags*** **especifica información sobre el manejo de los datos proyectados (compartidos, privado, etc.).**
  - **MAP_SHARED, MAP_PRIVATE, MAP_ANONYMOUS, ...**
- ***fildes*** **representa el descriptor de fichero (o descriptor del objeto de memoria a proyectar, si no hay sistema de ficheros).**
- ***off*** **desplazamiento dentro del fichero a partir del cual se realiza la proyección. Debe ser múltiplo de página.**
  - `sysconf(_SC_PAGESIZE)` **devuelve el tamaño de página.**

**Desproyectar** parte del espacio de direcciones de un proceso.

- `void munmap(void *addr, size_t len);`
- *addr* dirección inicio de la desproyección. Debe ser múltiplo de página.
- *len* tamaño de la desproyección. Se desproyecta un número entero de páginas , por lo que la desproyección puede ser mayor que len.

**Compartir memoria**

- Varios procesos pueden proyectar el mismo fichero con MAP_SHARED.
  - Los procesos deben conocer el nombre del fichero.
- Varios procesos pueden heredar la región del fichero proyectado con MAP_SHARED.
- Utilizando el *flag* MAP_ANONYMOUS se crea una región sin tener un fichero de base.
  - La región no tiene nombre, por lo que solamente se puede compartir si se hereda y está definida con MAP_SHARED.
  - `fildes` y `off` se ignoran (conviene poner `fildes` = -1).

# ACCESO A FICHERO CON PROYECCIÓN



**Ejemplo: ¿Cuántas veces aparece un determinado carácter en un fichero?**

```
int main(int  argc, char **argv) {
    char *p, *org; struct stat bstat;
    ..................................
    caracter = *argv[1];            //Carácter a buscar
    fd=open(argv[2], O_RDONLY));    //Abre fichero
    fstat(fd, &bstat);              //Averigua longitud del fichero

    /* Se proyecta el fichero */
    org=mmap(NULL, bstat.st_size, PROT_READ, MAP_PRIVATE, fd,0);
    close(fd);                      //Cierra el descriptor del fichero

    p = org;
    contador = 0;
    for (i=0; i < bstat.st_size; i++, p++) //Bucle de acceso
        if (*p == caracter) contador++;

    munmap(org, bstat.st_size);             //Se elimina la proyección

    printf("%d\n", contador);
    return 0;
}
```

# COPIA DE FICHERO CON PROYECCIÓN



```
int main(int  argc, char **argv) {
 char *p, *q, *org, *dst ; struct stat bstat;
    ..........................................
    fdo=open(argv[1], O_RDONLY); //Abre el fichero origen
    fdd=open(argv[2], O_CREAT|O_TRUNC|O_RDWR, 0640); //Crea destino
    fstat(fdo, &bstat);     //Averigua la longitud del fichero origen
    ftruncate(fdd, bstat.st_size); //Longitud destino = origen

    /* Se proyectan en memoria ambos ficheros */
    org=mmap(NULL, bstat.st_size, PROT_READ, MAP_PRIVATE, fdo, 0);
    dst=mmap(NULL, bstat.st_size, PROT_WRITE, MAP_SHARED, fdd, 0);

    close(fdo); //Se cierran los descriptores de fichero
    close(fdd);

    p=org;
    q=dst;
    for (i=0; i<bstat.st_size; i++)   //Bucle de copia
        *q++= *p++;

    /* Se eliminan las proyecciones */
    munmap(org, bstat.st_size);
    munmap(dst, bstat.st_size);
    return 0;
}
```

# MONTAJE EXPLÍCITO DE BIBLIOTECA DINÁMICA



## Montar biblioteca

- `void *dlopen (const char *filename, int flag);`
- Monta la biblioteca cuyo nombre se especifica en la cadena `filename`.
- El parámetro `flag` puede tener los valores RTLD_NOW o RTLD_LAZY.
- Devuelve un manejador.

## Obtener el punto de enlace de un procedimiento

- `void *dlsym(void *handle, char *symbol);`
- Devuelve la dirección de enlace o NULL (que puede no significar error).

## Cerrar biblioteca

- `int dlclose(void *handle);`
- La biblioteca solamente se elimina de memoria cuando no queda ningún proceso usándola (se ejecuta el último `dlclose`).
- Devuelve 0 en caso de éxito y otro valor si error.

## Obtener errores

- `const char *dlerror(void);`
- Devuelve NULL o una cadena con el código de error y borra dicho código de error.

# MONTAJE EXPLÍCITO DE BIBLIOTECA DINÁMICA



```
int main(int argc, char **argv) {
    void *handle;
    double (*cosine)(double);  //cosine es un puntero a función
    char *error;

    handle = dlopen ("/lib/libm.so", RTLD_LAZY);
    if (!handle) {
        fprintf (stderr, "%s\n", dlerror());
        exit(1);
    }

    cosine = dlsym(handle, "cos");    //Se asigna valor al puntero
    if ((error = dlerror()) != NULL)  {
        fprintf (stderr, "%s\n", error);
        exit(1);
    }

    printf ("%f\n", (*cosine)(2.0)); //Se obtiene el coseno de 2.0
    dlclose(handle);
    return 0;
}
```

# BLOQUEAR PÁGINAS EN MARCOS DE MEMORIA PRINCIPAL



**Interesante cuando no se puede tolerar el retardo producido por fallos de página, por ejemplo, en sistemas de tiempo real.**

```
int mlock(void *addr, int len);
```

- **Bloquea todas las páginas que contengan posiciones contenidas entre `addr` y `addr + len`.**
- **Está reservado al superusuario**

```
int munlock(void *addr, int len);
```

- **Desbloquea las páginas que contengan parte del espacio de direcciones comprendido entre `addr` y `addr + len`.**