

# T3-Memoria



# Índice

---

- Conceptos relacionados con la gestión de memoria
- Servicios básicos para la gestión de memoria
  - Carga de programas en memoria
  - Memoria dinámica
  - Soporte HW a la gestión de memoria
    - ▶ A la asignación de memoria
    - ▶ A la traducción de direcciones
- Servicios para la optimización del uso de memoria física
  - COW
  - Memoria virtual
  - Prefetch
- Linux sobre Pentium

---

Memoria física vs. Memoria lógica

Espacio de direcciones de un proceso

Asignación de direcciones a un proceso

Tareas del Sistema operativo en la gestión de memoria

Soporte del hardware a la gestión de memoria

# CONCEPTOS

# Memoria física vs. Memoria lógica

---

- CPU sólo puede acceder directamente a memoria y registros
  - Instrucciones y datos deben cargarse en memoria para poder referenciarse
  - Carga: reservar memoria, escribir en ella el programa y pasar la ejecución al punto de entrada del programa
- Tipos de direcciones:
  - Referencia emitida por la CPU: @ lógica
  - Posición ocupada en memoria: @ física
  - No tienen por qué coincidir si el **SO** y el **HW** ofrecen soporte para la **traducción**
    - ▶ Los sistemas de propósito general actuales lo ofrecen

# Espacio de @ de un proceso

---

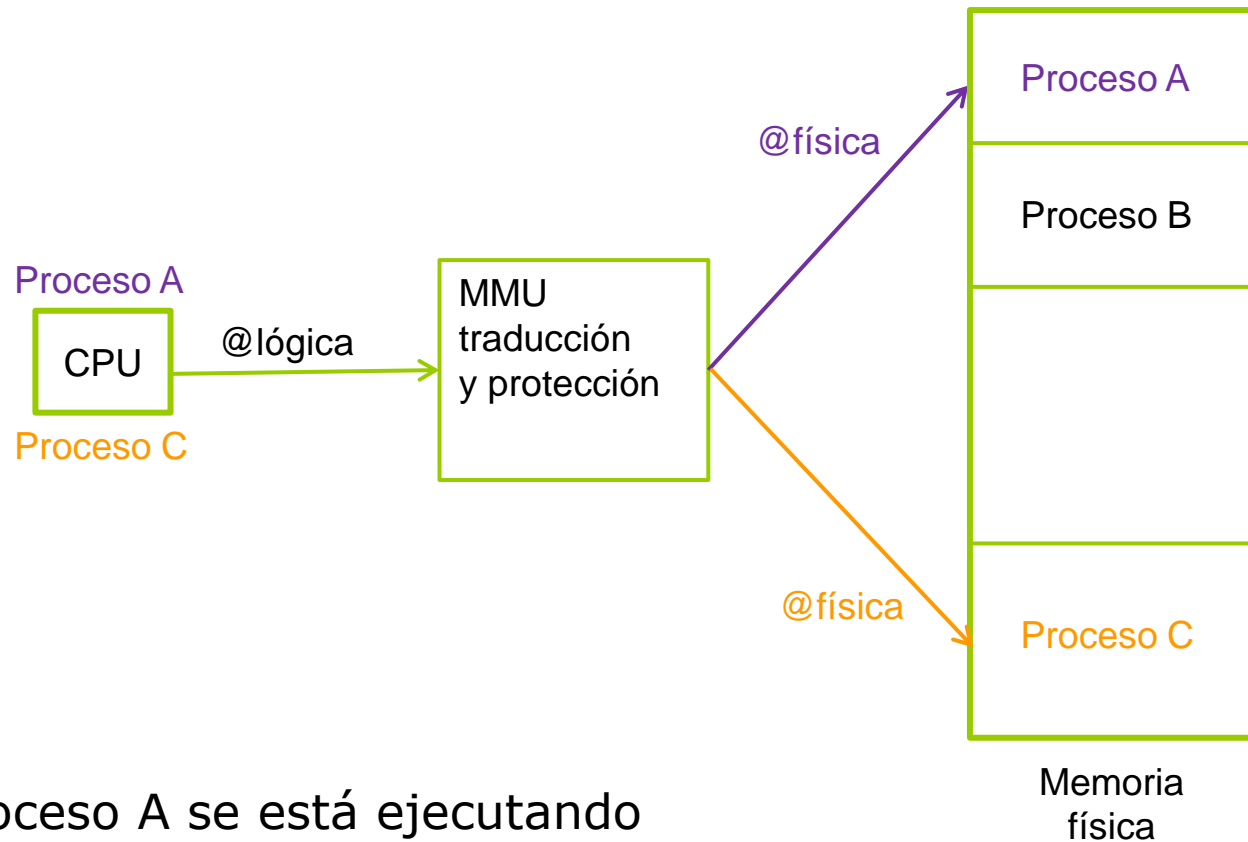
- Espacio de **direcciones del procesador**
  - Conjunto de @ que el procesador puede emitir, (depende del bus de direcciones)
- Espacio de **direcciones lógicas de un proceso**
  - Conjunto de @ lógicas que un proceso puede referenciar (que el kernel decide que son válidas para ese proceso)
- Espacio de **direcciones físicas de un proceso**
  - Conjunto de @ físicas asociadas al espacio de direcciones lógicas del proceso (decidido también por el kernel)
- Correspondencia entre @ lógicas y @ físicas
  - Fija: Espacio de @ lógicas == Espacio de @ físicas
  - Traducción:
    - ▶ Al cargar el programa en memoria: el kernel decide donde poner el proceso y se traducen las direcciones al copiarlas a memoria
    - ▶ **Al ejecutar: se traduce cada dirección que se genera**
      - **Colaboración entre HW y SO**
        - » HW ofrece el mecanismo de traducción
          - » Memory Management Unit (MMU)
        - » **El Kernel lo configura**

# Sistemas multiprogramados

---

- Sistemas multiprogramados
  - Varios programas cargados en memoria física simultáneamente
  - Facilita la ejecución concurrente y simplifican el cambio de contexto
    - ▶ **1 proceso en la CPU pero N procesos en memoria física**
    - ▶ Al hacer cambio de contexto no es necesario cargar de nuevo en memoria el proceso que ocupa la cpu
  - SO debe **garantizar protección** de la memoria física
    - ▶ Cada proceso sólo debe acceder a la memoria física que tiene asignada
    - ▶ **Colaboración entre SO y HW**
      - MMU ofrece el mecanismo para detectar accesos ilegales
      - SO configura la MMU
  - El kernel debe modificar la MMU para reflejar cualquier cambio:
    - ▶ Al hacer cambio de contexto el SO debe actualizar la MMU con la información del nuevo proceso
    - ▶ Si se aumenta el espacio de direcciones
    - ▶ etc

# Sistemas multiprogramados



- 1-Proceso A se está ejecutando
- 2-Cambio de contexto a C

# Asignación de @ a un programa

---

- **Hay otras alternativas pero... en los sistemas actuales la asignación de @ a instrucciones y datos se realiza en tiempo de ejecución**
  - @ físicas != @ lógicas → requiere traducción en tiempo de ejecución
  - Procesos pueden cambiar de posición en memoria sin modificar su espacio lógico de @
    - ▶ Ejemplo: Paginación (Visto en EC)



# Soporte HW: MMU

---

- MMU(Memory Management Unit). Componente HW que ofrece la traducción de direcciones y la protección del acceso a memoria. Como mínimo ofrece **soporte a la traducción y a la protección** pero puede ser necesario para otras tareas de gestión
- **SO es el responsable de configurar la MMU con los valores de la traducción de direcciones correspondientes al proceso en ejecución**
  - Qué @ lógicas son válidas y con qué @ físicas se corresponden
  - Asegura que cada proceso sólo tiene asociadas sus @ físicas
- **Soporte HW a la traducción y a la protección entre procesos**
  - MMU recibe @ lógica y usa sus estructuras de datos para traducirla a la @ física correspondiente
    - ▶ Si la @ lógica no está marcada como válida o no tiene una @ física asociada genera una excepción para avisar al SO
- **SO gestiona la excepción en función del caso**
  - Por ejemplo, si la @ lógica no es válida puede eliminar al proceso (SIGSEGV)

# Soporte HW: Traducción

---

- Cuando el SO tiene que modificar la traducción de direcciones???
- Al asignar memoria
  - Inicialización al **asignar nueva memoria. (en la mutación, execlp)**
  - **Cambios en el espacio de direcciones:** aumenta/disminuye. Al pedir/liberar memoria dinámica.
- En el **cambio de contexto**
  - Para el proceso que abandona la CPU: si aún no ha acabado la ejecución almacenar en las estructuras de datos del proceso (PCB) la información necesaria para reconfigurar la MMU cuando vuelva a ocupar la CPU
  - Para el proceso que pasa a ocupar la CPU: configurar la MMU

# Soporte HW : Protección

---

- Se realiza en los mismos casos que la asignación
- También permite implementar protección contra accesos/tipos de accesos no deseados
  - Direcciones lógicas inválidas
  - Direcciones lógicas válidas con acceso incorrecto (escribir en zona de lectura)
  - Direcciones lógicas válidas y acceso “incorrecto” pero que el SO ha marcado como incorrecto para implementar alguna optimización
    - Por ejemplo COW que veremos más adelante
- En cualquier caso → excepción capturada por la CPU y gestión por parte del SO
  - El kernel siempre tiene la información correcta sobre el espacio de direcciones, por lo que pueden comprobar si es realmente un fallo o no

# Tareas del SO en la gestión de memoria

---

- Carga de programas en memoria
- Reservar/Liberar memoria dinámicamente (mediante llamadas a sistema)
- Ofrecer compartición de memoria entre procesos
  - ▶ Con COW habrá compartición de forma transparente a los procesos en zonas de solo lectura
  - ▶ Existe compartición explícita de memoria (mediante llamadas a sistema) pero no lo trabajaremos este curso
- Servicios para la optimización del uso de memoria
  - COW
  - Memoria virtual
  - Prefetch

---

Carga de un programa

Memoria dinámica

Asignación de memoria

Compartición de memoria entre procesos

# SERVICIOS BÁSICOS DEL SO

# Servicios básicos: carga de programas

---

- El ejecutable debe estar en memoria para ser ejecutado, pero los ejecutables están en “disco”
- SO debe
  1. Leer e Interpretar el ejecutable (los ejecutables tienen un formato)
  2. Preparar el esquema del proceso en memoria lógica y **asignar memoria física**
    1. **Inicializar estructuras de datos** del proceso
      1. Descripción del espacio lógico
        1. Qué @ lógicas son válidas
        2. Qué tipo de acceso es válido
      2. Información necesaria para configurar la MMU cada vez que el proceso pasa a ocupar la CPU
    2. **Inicializar MMU**
  3. **Leer secciones del programa** del disco y escribir memoria
  4. Cargar registro **program counter** con la dirección de la instrucción definida en el ejecutable como **punto de entrada**

# Servicios básicos: carga de programas

---

- Optimizaciones aplicadas a la carga de programas
  - Carga bajo demanda
  - Librerías compartidas y enlace dinámico
- En Linux se provoca cuando un proceso muta (**exec**)

# Carga: formato del ejecutable

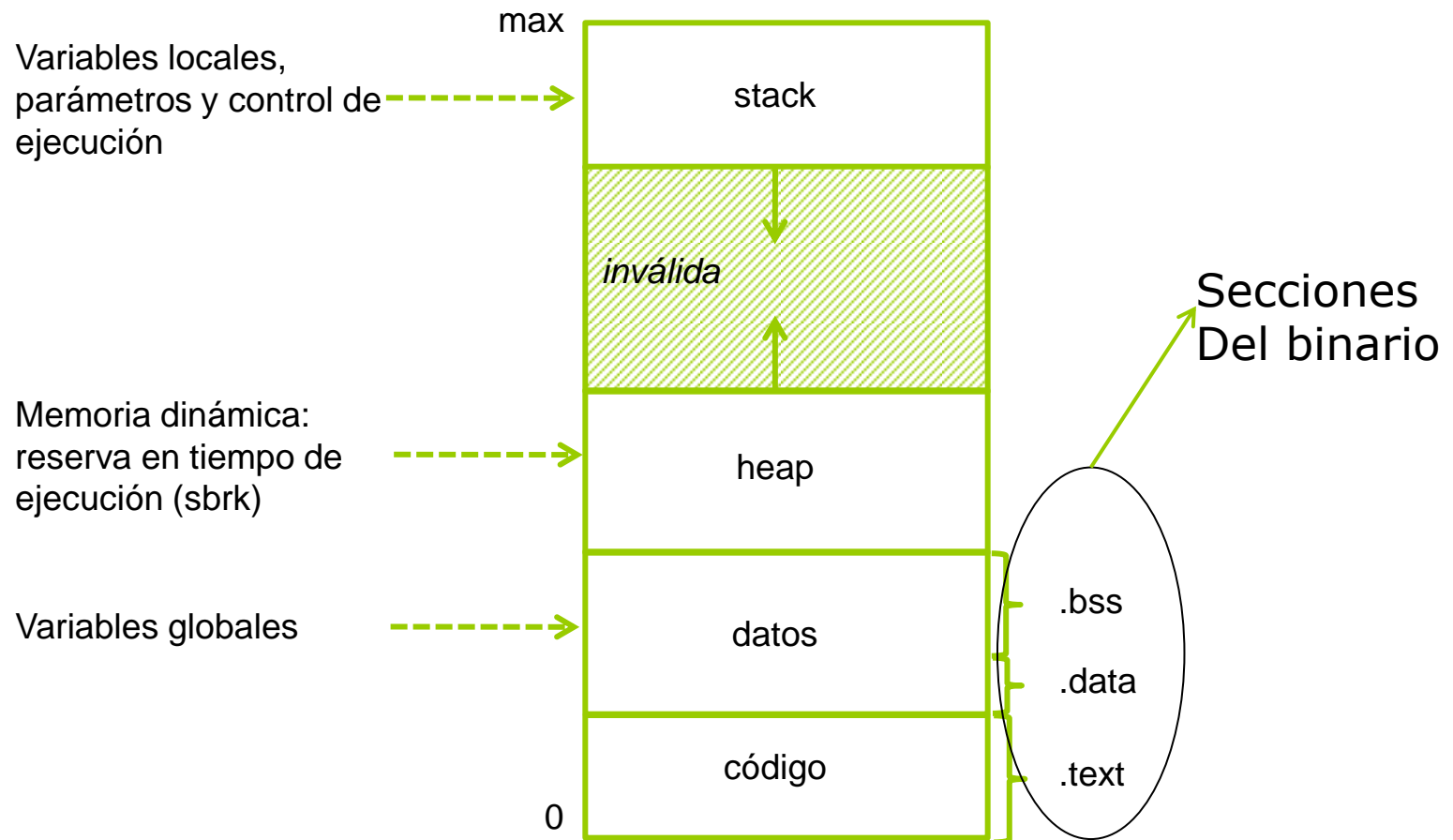
- PASO 1: Interpretar el formato del ejecutable en disco
  - Si la traducción se hace en tiempo de ejecución...¿que tipo de direcciones contienen los binarios? Lógicas o Físicas
- Cabecera del ejecutable define las secciones: tipo, tamaño y posición dentro del binario (podéis probar `objdump -h programa`)
- Existen diferentes formatos de ejecutable
  - ▶ ELF (*Executable and Linkable Format*): es el más extendido en sistemas POSIX

Algunas secciones por defecto de un ejecutable ELF	
<b>.text</b>	<b>código</b>
<b>.data</b>	<b>Datos globales inicializados</b>
<b>.bss</b>	<b>datos globales sin valor inicial</b>
.debug	información de debug
.comment	información de control
.dynamic	información para enlace dinámico
.init	código de inicialización del proceso (contiene la @ de la 1ª instrucción)

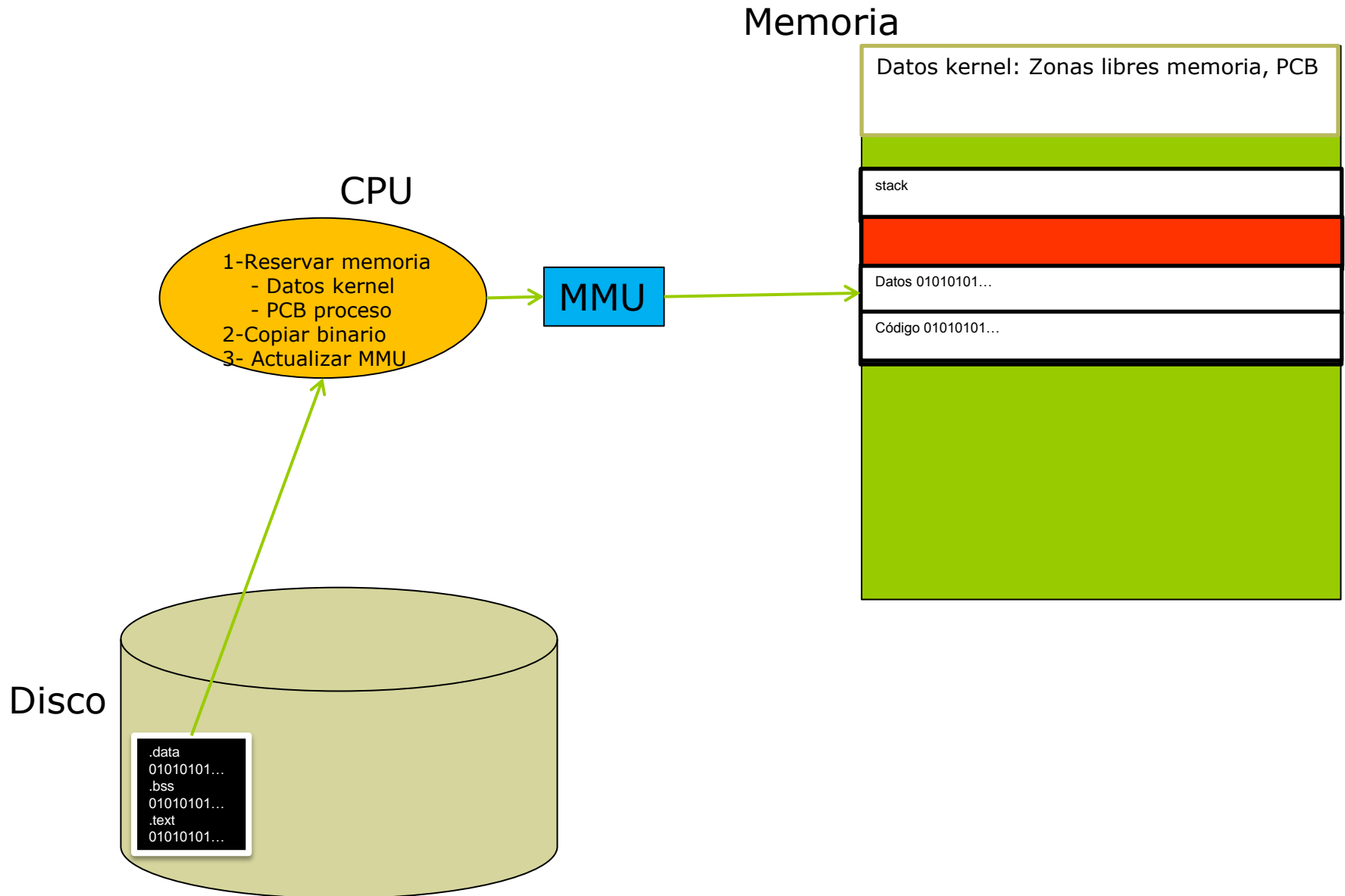


# Carga: Esquema del proceso en memoria

- PASO 2: Preparar el esquema del proceso en memoria lógica
  - Esquema habitual



# Carga ejecutables



# Carga: Optimización carga bajo demanda

---

- Optimizaciones: **carga bajo demanda**
  - **Una rutina no se carga hasta que se llama**
  - Se aprovecha mejor la memoria ya que no se cargan funciones que no se llaman nunca (por ejemplo, rutinas de gestión de errores)
  - Se acelera el proceso de carga (aunque se puede notar durante la ejecución)
  - Hace falta un mecanismo que detecte si las rutinas no están cargadas. Por ejemplo:
    - ▶ SO:
      - Registra en sus estructuras de datos que esa zona de memoria es válida y de dónde leer su contenido
      - En la MMU no le asocia una traducción
    - ▶ Cuando el proceso accede a la @, la MMU genera una excepción para avisar al SO de un acceso a una @ que no sabe traducir
    - ▶ SO comprueba en sus estructuras que el acceso es válido, provoca la carga y reanuda la ejecución de la instrucción que ha provocado la excepción

# Carga: Optimización de librerías compartidas

---

- Los binarios (en disco) no contienen el código de las librerías dinámicas, solo un enlace  
→ Ahorra mucho espacio en disco
  - ▶ **Se retrasa el enlace hasta el momento de ejecución**
  - ▶ Pensad en cuantos programas utilizan la libC, cuanto espacio necesitamos si cada uno tiene una copia (idéntica) de la librería
- Los procesos (en memoria) pueden compartir la zona en memoria que contiene el código (que es sólo de lectura) de las librerías comunes → Ahorra espacio en memoria
- Facilita la actualización de los programas para que usen las nuevas versiones de las librerías de sistema
  - No hace falta recompilar, al ejecutar el programa se enlazará con la nueva versión
- Mecanismo
  - Binario contiene el código de una rutina de enlace (*stub*), es un tipo de rutina que hace de puente a la que contiene el código realmente
    - ▶ Comprueba si algún proceso ya ha cargado la rutina de la librería compartida y la carga si no es así
    - ▶ Substituye la llamada a sí misma por la llamada a la rutina de la librería compartida

# Servicio: Reservar/Liberar memoria dinámica

---

- Hay variables cuyo tamaño depende de parámetros de la ejecución
  - Fijar el tamaño en tiempo de compilación no es adecuado
    - ▶ O se desaprovecha memoria o se tiene error de ejecución por no haber reservado suficiente
- Los SO ofrecen llamadas a sistema para reservar nuevas regiones de memoria en tiempo de ejecución: **memoria dinámica**
  - Se almacena en la **zona heap** del espacio lógico de @
- Implementación
  - Puede retrasar el momento de asignar @ físicas hasta que se intente escribir en la región
    - ▶ Se asigna temporalmente una zona inicializada con 0 para resolver lecturas. El interfaz puede definir que la región está inicializada con 0 o no
  - Actualiza la MMU en función de la política de asignación de memoria que siga

# Servicio: Reservar/Liberar memoria dinámica

---

## ■ Linux sobre Pentium

- Interfaz tradicional de Unix poco amigable
  - ▶ `brk` y `sbrk` (usaremos esta)
  - ▶ Permiten modificar el límite del heap . El SO no tiene conciencia de que variables hay ubicadas en que zonas, simplemente aumenta o reduce el tamaño del heap
  - ▶ Programador es responsable de controlar posición de cada variable en el heap → La gestión es compleja

```
^limite_anterior_heap(int) sbrk(tamaño_variacion_heap);
```

- `>0` aumenta el heap
- `<0` reduce el heap
- `==0` no se modifica

# Sbrk:ejemplo

```
int main(int argc, char *argv[])
{
    int num_procs = atoi(argv[1]);
    int *pids;
    pids = sbrk(num_procs * sizeof(int));
    for(i=0; i<10; i++){
        pids[i] = fork();
        if (pids[i] == 0){
            ....
        }
    }
    sbrk(-1 * num_procs * sizeof(int));
}
```

max

PILA (STACK)

HEAP

DATOS

CODIGO

0

Sencillo si tenemos una variable, que  
Pasaría si tenemos varias y queremos  
"liberar" una del medio del heap??  
NO SE PUEDE!

# Servicio: Reservar/Liberar memoria dinámica

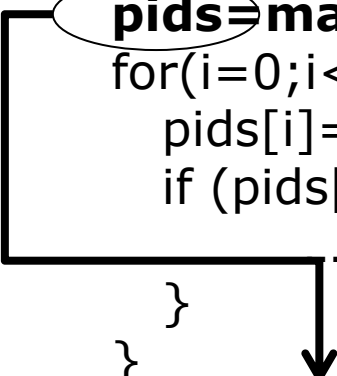
---

- La librería de C añade la gestión que vincula las direcciones con las variables.
  - Es una gestión transparente al kernel
- **Librería de C.** Pedir memoria: `malloc(tamaño_en_bytes)`
  - ▶ Si hay espacio **consecutivo** suficiente, lo marca como reservado y devuelve la dirección de inicio
  - ▶ Si no hay espacio **consecutivo** suficiente, aumenta el tamaño del heap
  - ▶ La librería de C gestiona el heap, sabe que zonas están libres y que zonas usadas. Intentar satisfacer peticiones sin recurrir al sistema
    - Al aumentar el heap, se reserva más de lo necesario con el objetivo de reducir el número de llamadas a sistema y ahorrar tiempo. La próxima petición del usuario encontrará espacio libre
- **Librería de C.** liberar memoria: `free(zona_a_liberar)`
  - ▶ Cuando el programador libera una zona se decide si simplemente pasa a formar parte de la lista de zonas libres o si es adecuado reducir el tamaño del heap
  - ▶ La librería ya sabe que tamaño tenía la zona ya que se supone que corresponde con una zona pedida anteriormente con `malloc`



# Como sería con malloc/free

```
int main(int argc, char *argv[])
{
    int num_procs = atoi(argv[1]);
    int *pids;
    pids = malloc(num_procs * sizeof(int));
    for(i=0; i<10; i++){
        pids[i] = fork();
        if (pids[i] == 0){
            ..
        }
    }
    free(pids);
}
```

A diagram consisting of a horizontal line that starts from the left side of the `pids = malloc` line and extends to the right. From the end of this line, a vertical line descends and ends in an arrow pointing to the `free(pids)` line. This visualizes the flow of memory allocation and subsequent deallocation.

A la hora de pedir es igual, pero al Liberar hemos de pasar un puntero Concreto, no un tamaño

# Memoria dinámica (IV): ejemplos

- Qué diferencias a nivel de *heap* observáis en los siguientes ejemplos?

- Ejemplo 1:

```
...  
new = sbrk(1000);  
...
```



- Ejemplo 2:

```
...  
new = malloc(1000);  
...
```



- Cambia el tamaño del *heap* en los dos casos?

# Memoria dinámica (V): ejemplos

- Qué diferencias a nivel de *heap* observáis en los siguientes ejemplos?

- Ejemplo 1:

```
...  
ptr = malloc(1000);  
...
```



- Ejemplo 2:

```
...  
for (i = 0; i < 10; i++)  
    ptr[i] = malloc(100);  
...
```



- Se reservan las mismas posiciones de memoria lógica?
  - Ejemplo1: necesitamos 1000 bytes consecutivos
  - Ejemplo2: Necesitamos 10 regiones de 100 bytes

# Memoria dinámica (VI): ejemplos

- Qué errores contienen los siguientes fragmentos de código?

- Código 1:

```
...  
for (i = 0; i < 10; i++)  
    ptr = malloc(SIZE);  
  
// uso de la memoria  
// ...  
  
for (i = 0; i < 10; i++)  
    free(ptr);  
...
```



- Código 2:

```
int *x, *ptr;  
  
...  
ptr = malloc(SIZE);  
...  
x = ptr;  
...  
free(ptr);  
  
sprintf(buffer, "...%d",  
*x);
```



- Código 1: ¿Que pasará en la segunda iteración del segundo bucle?
- Código 2: ¿Produce error siempre el acceso a “\*x”?

# Servicios básicos: asignación de memoria

---

- Se ejecuta cada vez que un proceso necesita memoria física:
  - En linux: creación (fork), mutación del ejecutable (exec)==carga, uso de memoria dinámica, implementación de alguna optimización (carga bajo demanda, memoria virtual, COW...).
- Pasos
  - **Seleccionar memoria física libre** y marcarla como ocupada en las estructuras de datos del SO
  - Actualizar MMU con el mapeo @ lógicas → @ físicas
    - ▶ Necesario para implementar la **traducción de direcciones**
- Cuando tenemos un problema de asignar una cantidad X (en este caso memoria) en una zona más grande, dependiendo de la solución aparecen problemas de **FRAGMENTACION**
  - También aparece en la gestión del disco

# Asignación: Problema fragmentación

---

- Fragmentación de memoria: memoria que está libre pero no se puede usar para un proceso
  - **Fragmentación interna:** memoria asignada a un proceso aunque no la necesita. Esta reservada pero no ocupada.
  - **Fragmentación externa:** memoria libre y no asignada pero no se puede asignar por no estar contigua. No esta reservada pero no sirve.
    - ▶ Se puede evitar compactando la memoria libre si el sistema implementa asignación de @ en tiempo de ejecución
      - Costoso en tiempo

# Servicios básicos: asignación de memoria

---

- Primera aproximación: **asignación contigua**
  - Espacio de @ físicas contiguo
    - ▶ Todo el proceso ocupa una partición que se selecciona en el momento de la carga
  - Poco flexible y dificulta aplicar optimizaciones (como carga bajo demanda)
- **Asignación no contigua**
  - Espacio de @ físicas no contiguo
  - Aumenta flexibilidad
  - Aumenta la granularidad de la gestión de memoria de un proceso
  - Aumenta complejidad del SO y de la MMU
- Basada en
  - **Paginación** (particiones fijas)
  - **Segmentación** (particiones variables)
  - Esquemas combinados
    - ▶ Por ejemplo, segmentación paginada

Visto en EC

# Asignación: Paginación

---

## ■ Esquema basado en paginación

- Espacio de @ lógicas dividido en particiones de tamaño fijo: **páginas**
- Memoria física dividida en particiones del mismo tamaño: **marcos**
- Asignación
  - ▶ Para cada página del proceso buscar un marco libre
    - Lista de marcos libres
  - ▶ Puede haber fragmentación interna
- Cuando un proceso acaba la ejecución devolver los marcos asignados a la lista de libres
- **Página: unidad de trabajo del SO**
  - ▶ Facilita la carga bajo demanda
  - ▶ Permite especificar protección a nivel de página
  - ▶ Facilita la compartición de memoria entre procesos
  - ▶ Normalmente, por temas de permisos, una página pertenece a una región de memoria (código/datos/heap/pila)



# Asignación: Paginación

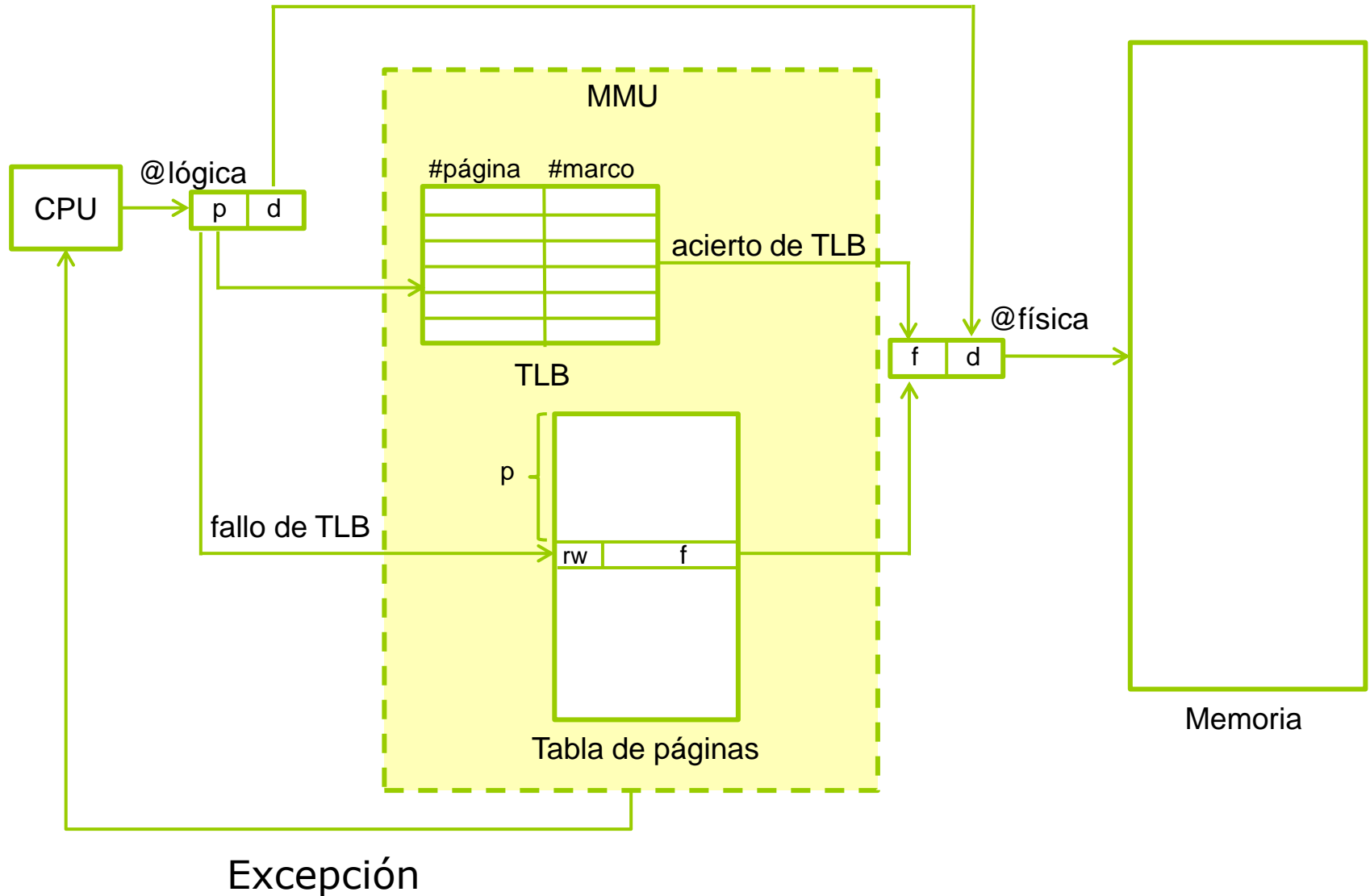
---

## ■ MMU

### ● Tabla de páginas

- ▶ Para mantener información a nivel de página: validez, permisos de acceso, marco asociado, etc....
- ▶ Una entrada para cada página
- ▶ Una tabla por proceso
- Suele guardarse en memoria y SO debe conocer la @ base de la tabla de cada proceso (por ejemplo, guardándola en el PCB)
- Procesadores actuales también disponen de TLB (*Translation Lookaside Buffer*)
  - ▶ Memoria asociativa (cache) de acceso **más rápido** en la que se almacena la información de traducción para las páginas *activas*
  - ▶ Hay que actualizar/invalidar la TLB cuando hay un cambio en la MMU
    - Gestión HW del TLB/Gestión Software (SO) del TLB
    - Muy dependiente de la arquitectura

# Asignación: Paginación



# Asignación: Paginación

- PROBLEMA: Tamaño de las tablas de página (que están guardadas en memoria)
- Tamaño de página potencia de 2
  - Tamaño muy usado 4Kb ( $2^{12}$ )
  - Influye en
    - ▶ Fragmentación interna y granularidad de gestión
    - ▶ Tamaño de la tabla de páginas
- Esquemas para reducir el espacio ocupado por las TP: TP multinivel
  - TP dividida en secciones y se añaden secciones a medida que crece el espacio lógico de direcciones

	Espacio lógico de procesador	Número de páginas	Tamaño TP
Bus de 32 bits	$2^{32}$	$2^{20}$	4MB
Bus de 64 bits	$2^{64}$	$2^{52}$	4PB

# Asignación: Segmentación

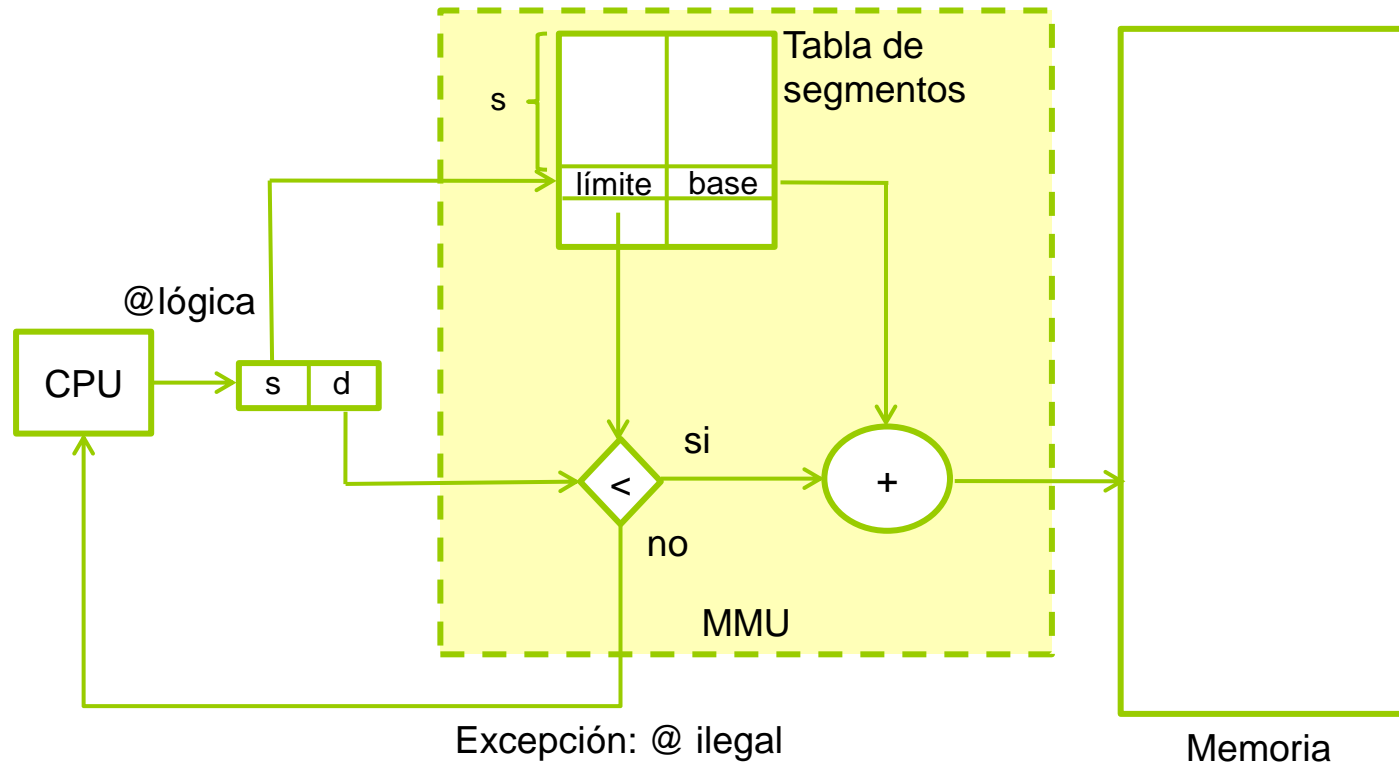
---

## ■ Esquema basado en segmentación

- Se divide el espacio lógico del proceso teniendo en cuenta el tipo de contenido (código, datos, etc)
  - ▶ Aproxima la gestión de memoria a la visión de usuario
- Espacio de @ lógicas dividido en particiones de tamaño variable (**segmentos**), **ajustado a lo que se necesita**
  - ▶ Como mínimo un segmento para el código y otro para la pila y los datos
  - ▶ Las referencias a memoria que hace el programa están formadas por un segmento y el desplazamiento dentro del segmento
- Memoria física libre contigua forma una partición disponible
- Asignación: Para cada segmento del proceso
  - Busca una partición en la que quepa el segmento
  - Posible políticas: first fit, best fit, worst fit
  - Selecciona la cantidad de memoria necesaria para el segmento y el resto continúa en la lista de particiones libres
  - ▶ Puede haber fragmentación externa
  - ▶ No todos los “trozos” libres son igual de buenos.

# Asignación: Segmentación

- MMU
  - Tabla de segmentos
    - ▶ Para cada segmento: @ base y tamaño
    - ▶ Una tabla por proceso



# Asignación: Esquema mixto

## ■ Esquemas combinados: segmentación paginada



- Espacio lógico del proceso dividido en segmentos
- Segmentos divididos en páginas
  - ▶ Tamaño de segmento múltiplo del tamaño de página
  - ▶ Unidad de trabajo del SO es la página

# Servicios básicos: compartición

---

- Compartición de memoria entre procesos
  - Se puede especificar a nivel de página o de segmento
  - Para procesos que ejecutan el mismo código no es necesario varias copias en memoria física (acceso de lectura)
    - ▶ **Librerías compartidas (implícito)**
  - Los SO proporcionan llamadas a sistema para que un proceso cree zonas de memoria en su espacio lógico que sean compartibles y para que otro proceso la pueda mapear en su espacio de memoria
    - ▶ **Memoria compartida** como mecanismo de **comunicación entre procesos (explícito)**
  - El resto de memoria es **privada** para un proceso y nadie la puede acceder

---

COW

Memoria virtual

Prefetch

# **SERVICIOS PARA LA OPTIMIZACIÓN DEL USO DE MEMORIA**



# Optimizaciones: COW (Copy on Write)

---

- Objetivo: reducir la reserva/inicialización de memoria física hasta que sea necesario
  - Si no se accede a una zona nueva → no necesitamos reservarla realmente
  - Si no modificamos una zona que es una copia → no necesitamos duplicarla
  - Ahorra tiempo y espacio de memoria
- En el fork:
  - **Retrasar el momento de la copia de código, datos, etc mientras sólo se acceda en modo lectura**
  - Se puede evitar la copia física si los procesos sólo van a usar la región para leer, por ejemplo el código
  - Se suele gestionar a nivel de página lógica: se van reservando/copiando páginas a medida que se necesita
- Se puede aplicar
  - Dentro de un proceso : al pedir memoria dinámica
  - Entre procesos (por ejemplo, fork de Linux)
  - En general siempre que se aumenta/modifica el espacio de direcciones.

# COW: Implementación

---

- **La idea es: el kernel asume que se podrá ahorrar la reserva de la memoria física, pero necesita un mecanismo para detectar que no es así y realizar la reserva si realmente SI era necesaria**
- En el momento que habría que hacer la asignación:
  - En la estructura de datos que describe el espacio lógico del proceso (en el PCB) el SO marca la región destino con los permisos de acceso reales
  - En la MMU el SO marca la región destino y la región fuente con permiso sólo de lectura
  - En la MMU el SO asocia a la región destino las direcciones físicas asociadas:
    - ▶ A las regiones del padre si era un fork (misma traducción, memoria compartida)
    - ▶ A una páginas que actuan de comodín en el caso de memoria dinámica
- Si un proceso intenta escribir en la zona nueva, la MMU genera excepción y SO la gestiona haciendo la reserva real y reiniciando el acceso

# COW: ejemplo

---

- Proceso A ocupa:
  - Código: 3 páginas, Datos 2 páginas, Pila: 1 página, Heap: 1 página
- Si proceso A ejecuta fork, justo después del fork:
  - Total memoria física:
    - ▶ Sin COW: proceso A= 7 páginas + hijo A= 7 páginas= 14 páginas
    - ▶ Con COW: proceso A= 7 páginas + hijo A=0 páginas = 7 páginas
- Al cabo de un rato...depende de lo que hagan los procesos, por ejemplo:
  - Si hijo A muta (y el nuevo espacio del hijo ocupa 10 páginas):
    - ▶ Sin COW: proceso A= 7 páginas + hijo A= 10 páginas= 17 páginas
    - ▶ Con COW: proceso A= 7 páginas + hijo A=10 páginas = 17 páginas
  - Si hijo A no muta, depende de lo que haga, pero el código al menos puede ser compartido, suponiendo que el resto no lo sea:
    - ▶ Sin COW: proceso A= 7 páginas + hijo A= 7 páginas= 14 páginas
    - ▶ Con COW: proceso A= 7 páginas + hijo A=4 páginas = 11 páginas
- **En cualquier caso hay que ver que páginas se modifican (por lo tanto no se pueden compartir) y que páginas si se pueden compartir**

# Optimizaciones: Memoria Virtual (I)

---

## ■ Memoria virtual

- Extiende la idea de la carga bajo demanda
- Además de “traer” cosas a memoria bajo demanda, permite “sacar” cosas bajo demanda
- Objetivo
  - ▶ Reducir la cantidad de memoria física asignada a un proceso en ejecución
    - **Un proceso realmente sólo necesita memoria física para la instrucción actual y los datos que esa instrucción referencia**
  - ▶ Aumentar el grado de multiprogramación
    - Cantidad de procesos en ejecución simultáneamente

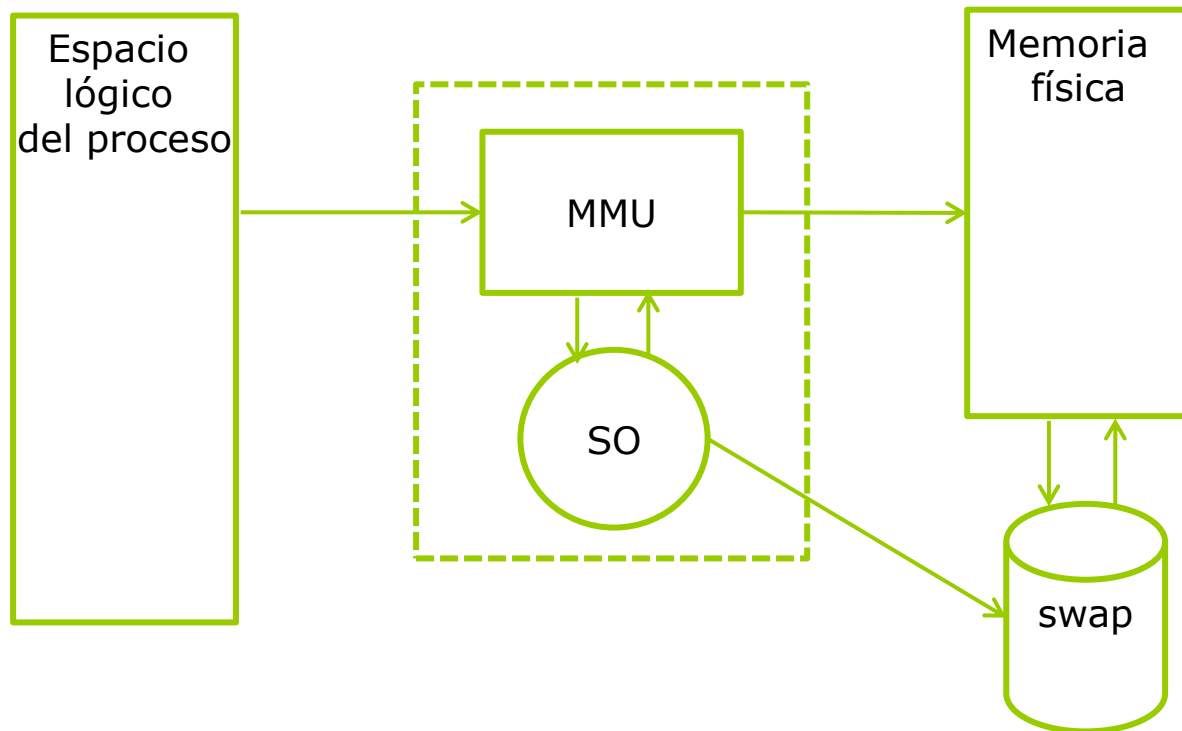
# Optimizaciones: Memoria Virtual (II)

---

- Primera aproximación: intercambio (*swapping*)
  - Idea: sólo hace falta tener en memoria el proceso activo (el que tenía la CPU asignada)
    - ▶ Si el proceso activo necesitaba más memoria física que la disponible en el sistema se puede expulsar temporalmente de memoria alguno de los otros procesos cargados (*swap out*)
    - ▶ Almacén secundario o de soporte (*backing storage*):
      - Dispositivo de almacenaje en el que se guarda el espacio lógico de los procesos a la espera de volver a ocupar la CPU
        - » Mayor capacidad que la que ofrece la memoria física
      - Típicamente una zona de disco: espacio de intercambio (*swap area*)
    - ▶ Estado de los procesos: no residentes (*swapped out*)
    - ▶ Al asignar la cpu a un proceso no residente es necesario cargarlo en memoria de nuevo antes de permitir que reanude la ejecución
      - Ralentiza la ejecución
  - Evolución de la idea
    - ▶ Evitar expulsar de memoria procesos enteros para minimizar la penalización en tiempo de la ejecución
    - ▶ Se puede aprovechar la granularidad que ofrece la paginación

# Optimizaciones: Memoria Virtual (III)

- Memoria virtual basada en paginación
  - **Espacio lógico de un proceso está distribuido entre memoria física (páginas residentes) y área de swap (páginas no residentes)**

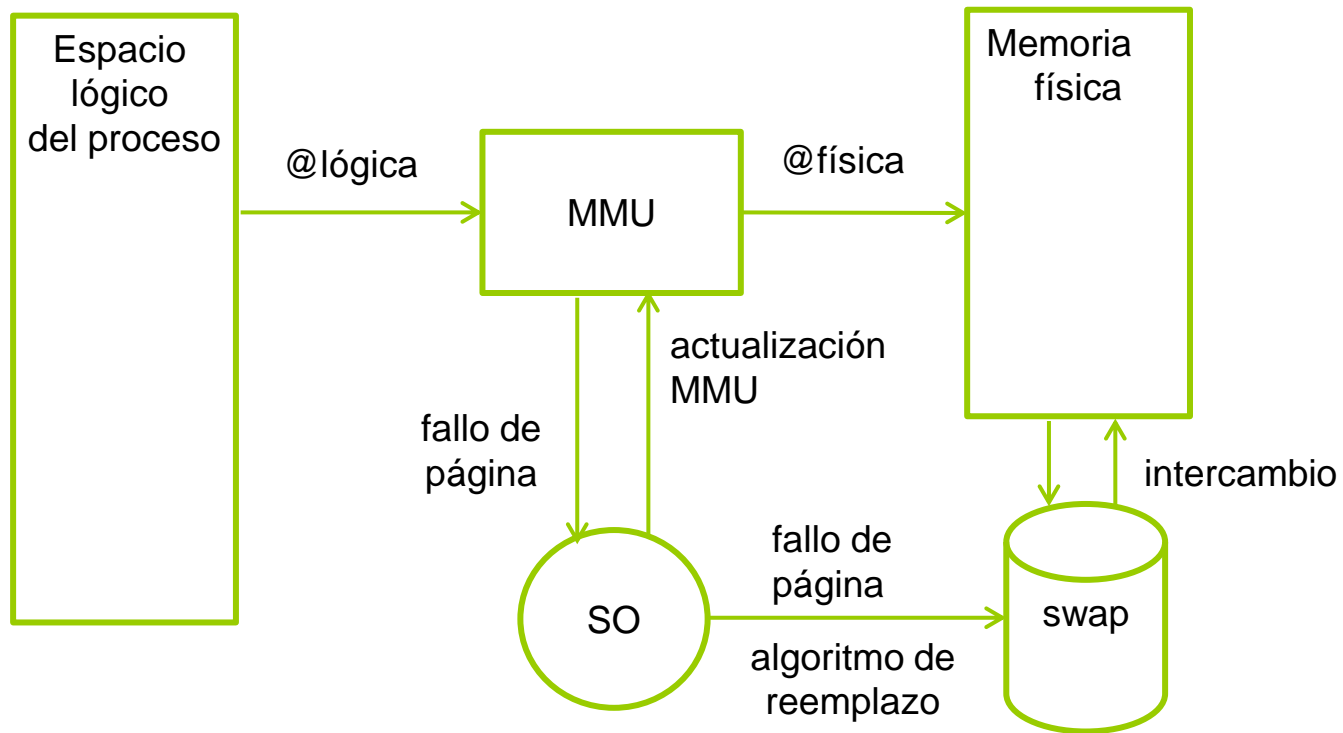


# Optimizaciones: Memoria Virtual (IV)

---

- **Reemplazo de memoria:** cuando SO necesita liberar marcos
  - Selecciona una página *víctima* y actualiza la MMU eliminando su traducción
  - Guarda su contenido en el área de swap para que se pueda recuperar
  - Asigna el marco ocupado a la página que se necesita en memoria
- Cuando se accede a una página guardada en el área de swap
  - MMU no puede hacer la traducción: genera excepción
    - ▶ **Fallo de página**
  - SO
    - ▶ Comprueba en las estructuras del proceso que el acceso es válido
    - ▶ Asigna un marco libre para la página (lanza el reemplazo de memoria si es necesario)
    - ▶ Localiza en el área de swap el contenido y lo escribe en el marco
    - ▶ Actualiza la MMU con la @física asignada

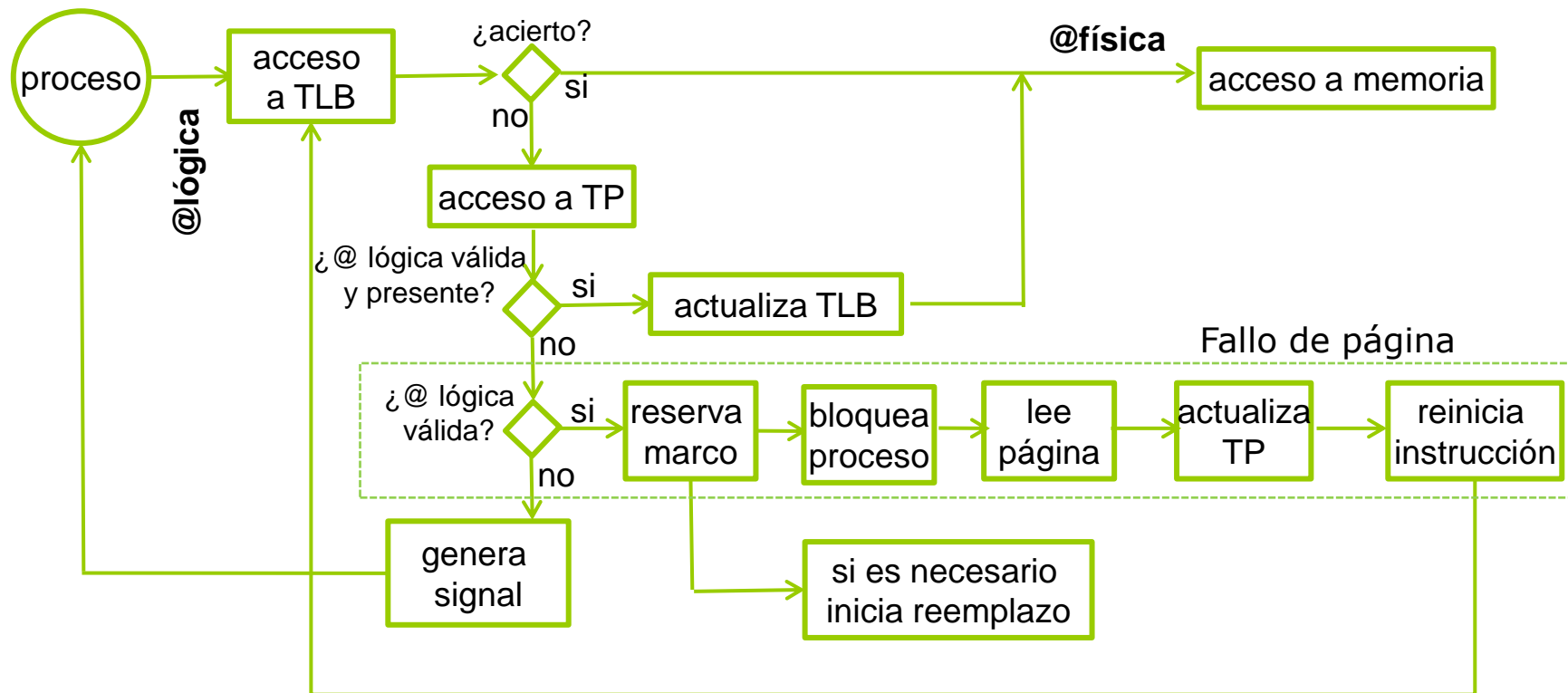
# Optimizaciones: Memoria Virtual (V)





# Optimizaciones: Memoria Virtual (VI)

## ■ Pasos en el acceso a memoria



# Optimizaciones: Memoria Virtual (VII)

---

- Efectos del uso de la memoria virtual
  - La suma de los espacios lógicos de los procesos en ejecución puede ser mayor que la cantidad de memoria física de la máquina
  - El espacio lógico de un proceso también puede ser mayor que la memoria física disponible
  - Acceder a una página no residente es más lento que acceder a una página residente
    - ▶ Excepción + carga de la página
    - ▶ Importante minimizar el número de fallos de página

# Optimizaciones: Memoria Virtual (VIII)

---

- Modificaciones en el SO
  - Añadir las estructuras de datos y los algoritmos para gestionar el área de swap
    - ▶ Asignación, liberación y acceso
  - Algoritmo de reemplazo
    - ▶ ¿Cuándo se ejecuta? ¿Cómo se seleccionan las páginas víctimas?  
¿Cuántas páginas víctimas en cada ejecución del algoritmo?
    - ▶ Objetivo: minimizar el número de fallos de página y acelerar su gestión
      - **Intentar seleccionar las víctimas entre las páginas que ya no se necesitan o que se va a tardar más tiempo en necesitar**
        - » Ejemplo: Least Recently Used (LRU) o aproximaciones
      - Intentar que siempre que se da un fallo de página haya un marco disponible
- Modificaciones en la MMU: depende de los algoritmos de gestión de memoria virtual.
  - Por ejemplo, algoritmo de reemplazo puede necesitar un bit de referencia por página

# Optimizaciones: Memoria Virtual (IX)

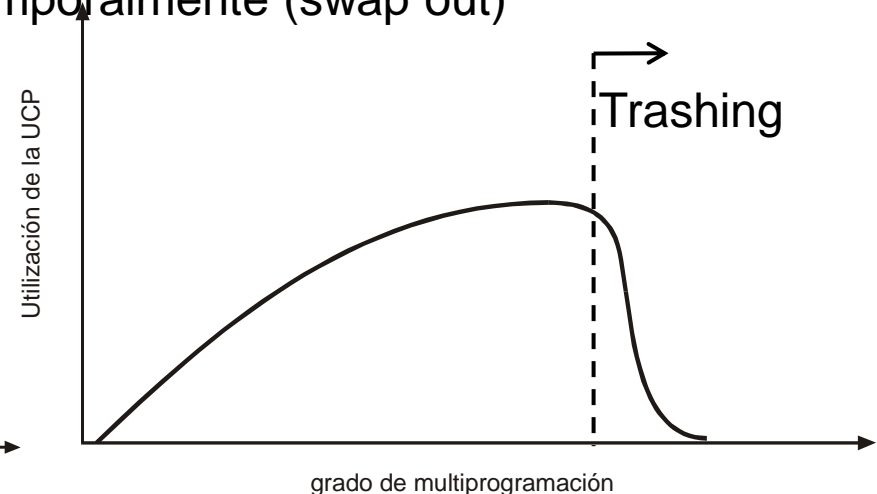
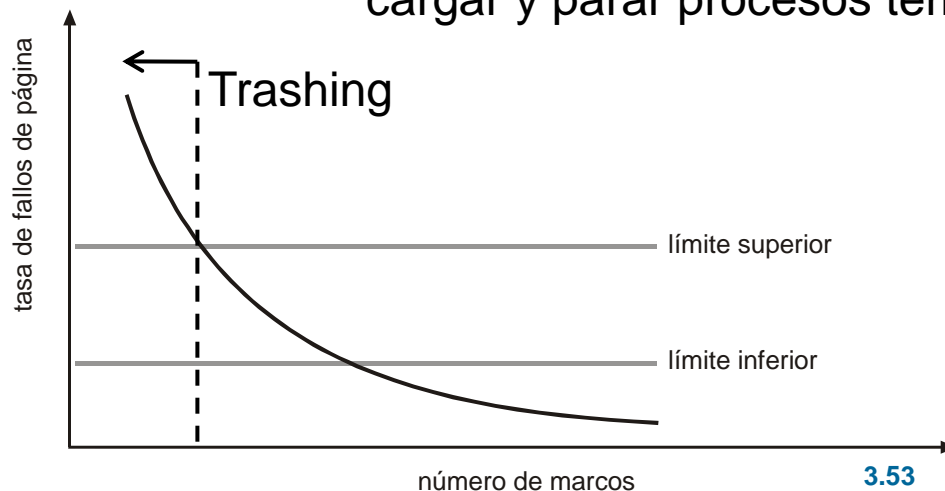
## ■ Sobrepaginación (*thrashing*)

### ● Proceso en thrashing

- ▶ **Invierte más tiempo en el intercambio de memoria que avanzando su ejecución**
- ▶ No consigue mantener simultáneamente en memoria el conjunto mínimo de páginas que necesita para avanzar

### ● Se debe a que se ha sobrecargado el sistema de memoria

- ▶ Detección: controlar tasa de fallos de página por proceso
- ▶ Tratamiento: controlar el número de procesos que se permiten cargar y parar procesos temporalmente (swap out)



# Optimizaciones: Memoria prefetch

---

- Objetivo: minimizar número de fallos de página
- Idea: anticipar qué páginas va a necesitar el proceso en el futuro inmediato y cargarlas con anticipación
- Parámetros a tener en cuenta:
  - Distancia de prefetch: con qué antelación hay que cargar las páginas
  - Número de páginas a cargar
- Algoritmos sencillos de predicción de páginas
  - Secuencial
  - Strided

# Resumen: Linux sobre Pentium

---

- Llamada a sistema **exec**: provoca la **carga** de un nuevo programa
  - Inicialización del PCB con la descripción del nuevo espacio de direcciones, asignación de memoria, ...
- Creación de procesos (**fork**):
  - Inicialización del PCB con la descripción de su espacio de direcciones (copia del padre)
  - Se utiliza COW: hijo comparte marcos con padre hasta que algún proceso los modifica
  - Creación e inicialización de la TP del nuevo proceso
    - ▶ Se guarda en su PCB la @ base de su TP
- Planificación de procesos
  - En el **cambio de contexto** se actualiza en la MMU la @ base de la TP actual y se invalida la TLB
- Llamada a sistema **exit**:
  - Elimina la TP del proceso y libera los marcos que el proceso tenía asignados (si nadie más los estaba usando)

# Resumen: Linux sobre Pentium

---

- Memoria virtual basada en segmentación paginada
  - Tabla de páginas multinivel (2 niveles)
    - ▶ Una por proceso
    - ▶ Guardadas en memoria
    - ▶ Registro de la cpu contiene la @ base de la TP del proceso actual
  - Algoritmo de reemplazo: aproximación de LRU
    - ▶ Se ejecuta cada cierto tiempo y cuando el número de marcos libres es menor que un umbral
- Implementa COW a nivel de página
- Carga bajo demanda
- Soporte para librerías compartidas
- Prefetch simple (secuencial)

# Jerarquía de almacenamiento

