Sistemas Operativos

Gestión de memoria

Fernando Pérez Costoya
fperez@fi.upm.es (Despacho 4201)

Un primer experimento

fperez@box1: ~/SII/experimentos_memoria/1_experimento

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int variable=10;

int main(int argc, char **argv) {
        printf("Proceso %d: variable (dir=%p; val=%d)\n", getpid(),
                &variable, variable++);
        printf("%d: Antes de fin; pulsa para acabar ", getpid());
        getchar();

        return 0;
}



~
~
~
~
~
~
"programa.c" 17L, 288C                                  6,0-1         Todo
```

La pregunta

Si ejecutamos varias veces el programa…

¿saldrá siempre la misma dirección para la variable?

La hipótesis

Si ejecutamos varias veces el programa…

¿saldrá siempre la misma dirección para la variable?

Hipótesis:

No, porque hay varios programas ejecutando en la máquina y el SO puede asignarle una diferente cada vez

El resultado

Ha salido siempre la misma dirección…

Conjetura:

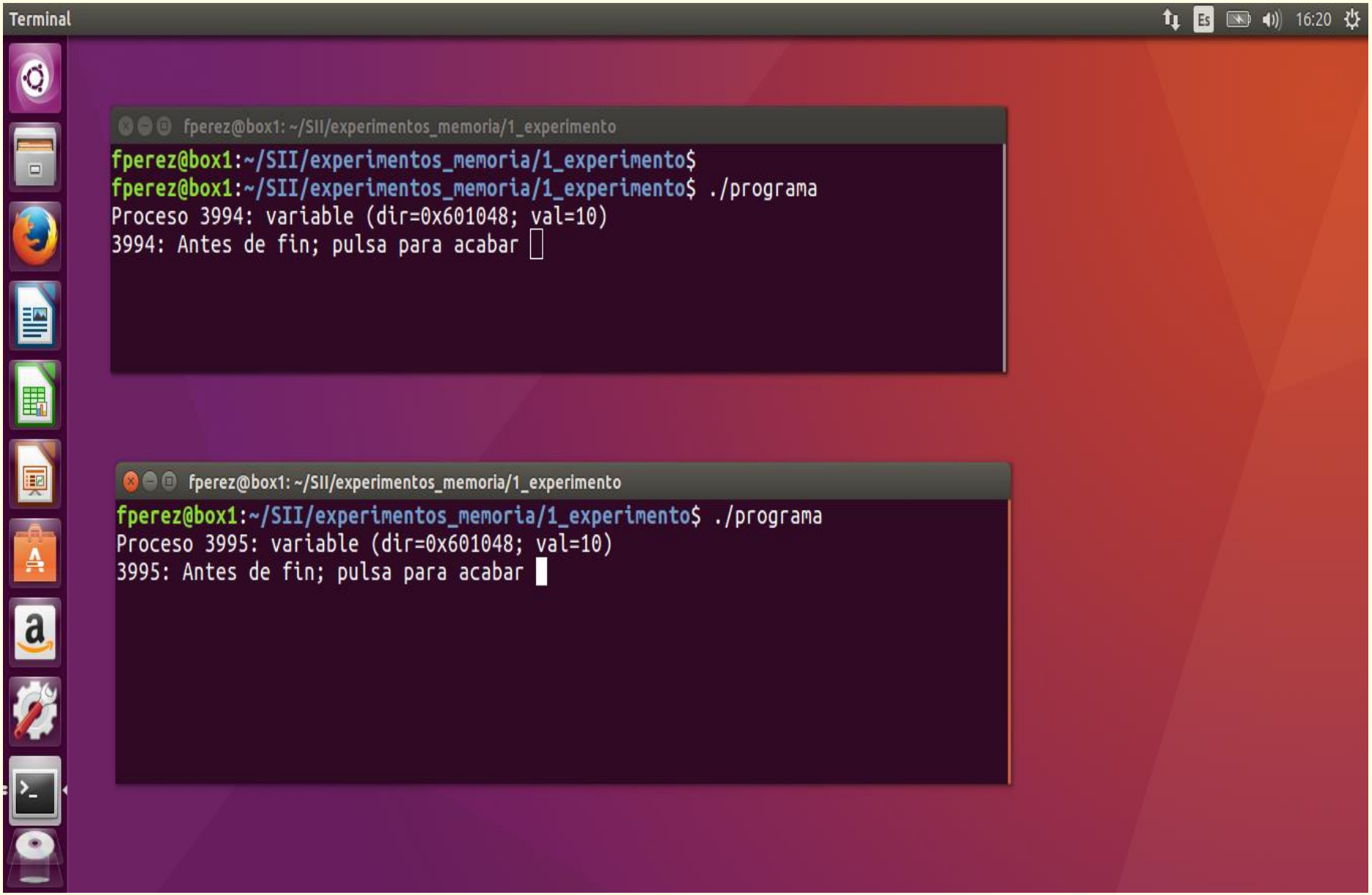
Quizás, por eficiencia, el SO intenta asignarle la misma...

Nuevo experimento

Ejecutamos el programa en 2 ventanas diferentes…

Hipótesis alternativas:

- ¿En cada ventana saldrá una dirección diferente?
- ¿O saldrá la misma dirección?
  - ¿Pero entonces comparten la variable y los cambios que hace un proceso los debe ver el otro?


Terminal
Es
16:20
fperez@box1: ~/SII/experimentos_memoria/1_experimento
fperez@box1:~/SII/experimentos_memoria/1_experimento$
fperez@box1:~/SII/experimentos_memoria/1_experimento$ ./programa
Proceso 3994: variable (dir=0x601048; val=10)
3994: Antes de fin; pulsa para acabar
fperez@box1: ~/SII/experimentos_memoria/1_experimento
fperez@box1:~/SII/experimentos_memoria/1_experimento$ ./programa
Proceso 3995: variable (dir=0x601048; val=10)
3995: Antes de fin; pulsa para acabar

Nuevo resultado

Misma dirección pero no comparten la variable

¿Y en el ejecutable?

- ¿El código máquina usará la misma dirección?

objdump –d programa

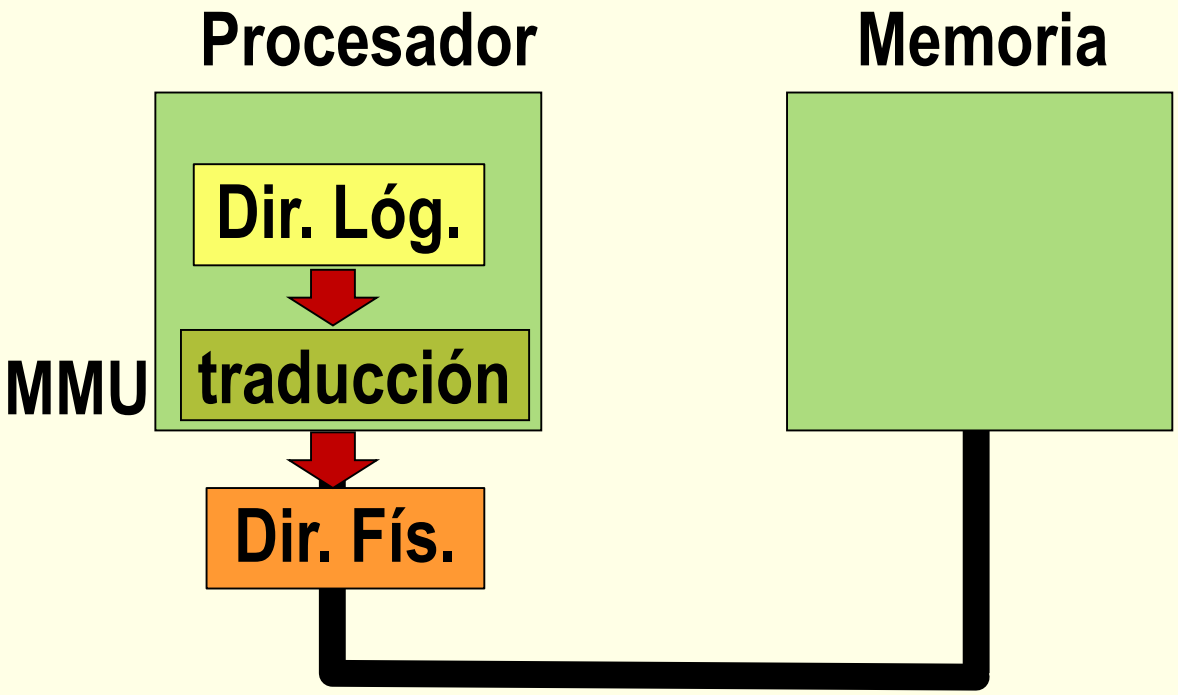
objdump –d programa

```
fperez@box1: ~/SII/experimentos_memoria/1_experimento
48 <variable>
  4005d5:       e8 96 fe ff ff          callq  400470 <getpid@plt>
  4005da:       89 d9                   mov    %ebx,%ecx
  4005dc:       ba 48 10 60 00          mov    $0x601048,%edx
  4005e1:       89 c6                   mov    %eax,%esi
  4005e3:       bf a8 06 40 00          mov    $0x4006a8,%edi
  4005e8:       b8 00 00 00 00          mov    $0x0,%eax
  4005ed:       e8 8e fe ff ff          callq  400480 <printf@plt>
  4005f2:       e8 79 fe ff ff          callq  400470 <getpid@plt>
  4005f7:       89 c6                   mov    %eax,%esi
  4005f9:       bf d0 06 40 00          mov    $0x4006d0,%edi
  4005fe:       b8 00 00 00 00          mov    $0x0,%eax
  400603:       e8 78 fe ff ff          callq  400480 <printf@plt>
  400608:       e8 93 fe ff ff          callq  4004a0 <getchar@plt>
  40060d:       b8 00 00 00 00          mov    $0x0,%eax
  400612:       48 83 c4 18             add    $0x18,%rsp
  400616:       5b                      pop    %rbx
  400617:       5d                      pop    %rbp
  400618:       c3                      retq
  400619:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)

0000000000400620 <__libc_csu_init>:
  400620:       41 57                   push   %r15
:
```

La conclusión

- Las direcciones de memoria que genera un proceso no se corresponden con las direcciones de la memoria del equipo
  - si no, los dos procesos compartirían el valor de la variable
  - Son las que aparecen en el fichero ejecutable
  - Pero tendrán que ser traducidas en tiempo de ejecución
- Esa es la "magia" del gestor de memoria del SO:
  - Establece la función de correspondencia entre cada:
    - dirección **lógica** generada por proceso → dirección **física** asignada
  - El hardware de gestión de memoria aplica esta función
    - MMU: *Memory Management Unit*
  - Cambio proceso: SO instala en MMU nueva función traducción
  - Cada proceso "ve" su propia memoria desde 0 hasta cierto valor
    - **El mapa de memoria del proceso**

Modo de operación de la MMU



¿Qué aprenderemos en este tema?

- Al finalizar este tema deberíamos ser capaces de
  - Entender las diversas necesidades de memoria de un programa en ejecución y cómo el SO las satisface
  - Conocer de manera global cómo el SO reparte la memoria entre los procesos activos en el sistema
  - Comprender el concepto de memoria virtual y su implementación
  - Aprender a usar la técnica de proyección de ficheros (*file mapping*) como una forma alternativa de acceso a los ficheros
  - Entender el modo de operación de las bibliotecas dinámicas, sus beneficios y cómo usarlas

Contenido

- **Aspectos generales de la gestión de memoria**
- Mapa de memoria de un proceso
- Gestión de la memoria del sistema
- Proyección de archivos
- Bibliotecas dinámicas

Aspectos generales de la gestión de memoria

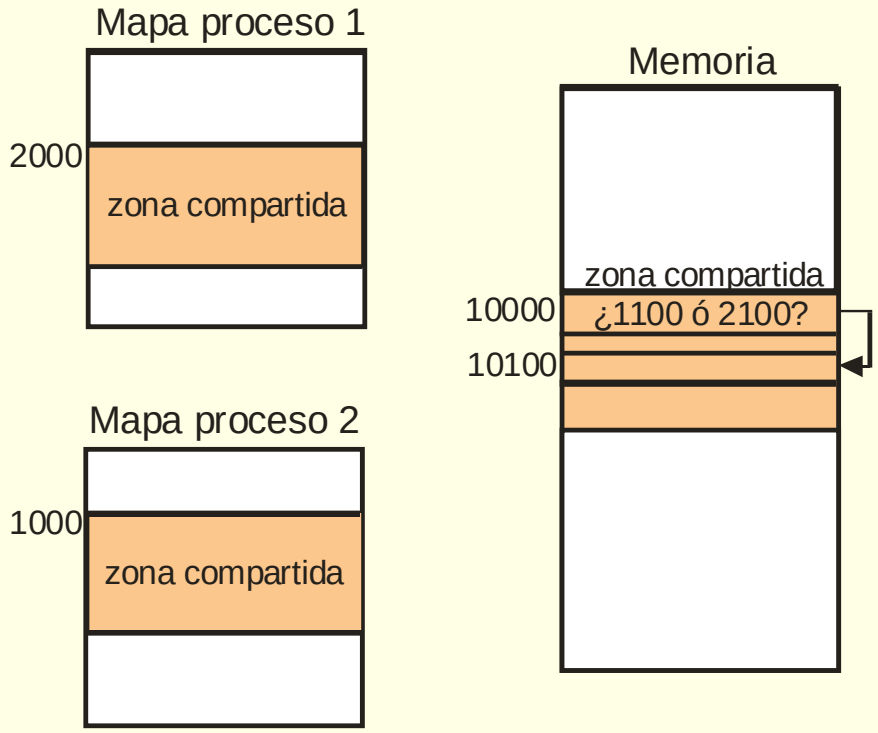
- SO multiplexa recursos entre procesos
  - Gestión de procesos: Reparto de procesador
  - Gestión de memoria: Reparto de memoria
- Gestor de memoria: elevada complejidad y dependencia del HW
- Gestión integral: SO, compilador, montador y hardware
- Ofrece un nº muy reducido de llamadas al sistema
  - Pero posee una gran complejidad
- **Objetivos del gestor de memoria:**
  - Crear espacio lógico independiente para cada proceso
    - Proceso genera **direcciones lógicas** entre 0 y N
    - Función traducción hace corresponder con **d. físicas asignadas**
  - Satisfacer necesidades de memoria de los procesos
    - El espacio lógico de un proceso no es homogéneo y estático

Creación de espacios lógicos independientes

- Reparto seguro y eficiente de la memoria entre procesos activos
  - Segura: procesos no pueden afectarse entre sí ni al SO
  - Eficiente: rápida y con buen aprovechamiento de la memoria
- Proporcionando un mapa/espacio lógico propio a cada proceso
  - Un conjunto de direcciones de 0 hasta un cierto valor
    - Linux x86-32 bits → 0–0xBFFFFFFF (3GB; 1GB restante SO)
    - Linux x86-64 bits → $0–2^{47}-1$ (128TB; 128TB restantes para SO)
    - Que puede ser mayor que la memoria física disponible
  - Función de traducción asegura *estanqueidad*
    - Hace corresponder direcciones de memoria distintas a los procesos
  - Pero permitiendo que compartan memoria de forma controlada
    - Para optimizar su uso (p.e. procesos comparten código del *bash*)
    - Como medio de comunicación eficiente

Compartimiento de memoria para comunicación

Dir. lógicas de procesos distintas corresponden a misma dir. física



Problema de las autorreferencias

Soporte de necesidades de memoria de procesos

- Satisfacer necesidades memoria del proceso durante su ejecución
- Mapa de memoria de un proceso es heterogéneo
  - Formado por conjunto de regiones con distintas características
  - Código solo se puede ejecutar; constantes solo leer,…
  - SO debería detectar inmediatamente operaciones no permitidas
    - Función de traducción debe detectar el error y avisar al SO
    - Facilita depuración del programa
- Mapa de memoria de un proceso es dinámico
  - Regiones cambian de tamaño, se crean y destruyen
    - La pila y la región de memoria dinámica (*malloc*) crecen
    - Cuando se crea un *thread* hay que habilitarle una pila,…
  - "Huecos" en el espacio lógico no deben consumir memoria
    - Función de traducción debe detectar error de acceso y avisar al SO

Contenido

- Aspectos generales de la gestión de memoria
- **Mapa de memoria de un proceso**
- Gestión de la memoria del sistema
- Proyección de archivos
- Bibliotecas dinámicas

Mapa de memoria de un proceso

- Mapa de memoria o imagen del proceso: conjunto de regiones
- Región: área contigua en espacio lógico con mismas características
  - protección: RWX
  - origen (soporte) de la región: En fichero o sin soporte (anónima)
    - Si no tiene soporte hay que rellenarla con ceros por seguridad
  - compartida o privada
    - Privada: Cada proceso tiene su propia copia de la región
    - Privada: Modificaciones no afectan al soporte original
  - dirección de comienzo y tamaño inicial
  - tamaño fijo o variable (crecimiento ascendente o descendente)
  - ubicación en el mapa lógico prefijada o libre
  - Si paginación, d. inicio y tamaño múltiplos de tamaño de página

Un segundo experimento

- ¿Cuántas regiones tendrá un proceso en Linux?
  - Código, datos, pila,…
- ¿Quizás tenga media docena de regiones?
- Aunque si el proceso es complejo, ¿1 par de docenas?
- Estos mandatos muestran las regiones del proceso (1 línea/región):
  pmap PID o *less /proc/PID/maps*
- Arranque Firefox, busque su PID y ejecute uno de esos mandatos
  - ¿Cuántas regiones aparecen en la salida estándar?

Un segundo experimento

- ¿Cuántas regiones tendrá un proceso en Linux?
  - Código, datos, pila,…
- ¿Quizás tenga media docena de regiones?
- Aunque si el proceso es complejo, ¿1 par de docenas?
- Estos mandatos muestran las regiones del proceso (1 línea/región):
  pmap PID o *less /proc/PID/maps*
- Arranque Firefox, busque su PID y ejecute uno de esos mandatos
  - ¿Cuántas regiones aparecen en la salida estándar?
- **¡Varios centenares de regiones!**
  - Un objetivo de este tema es llegar a entender por qué hay tantas
- Vamos a relacionar las entidades del programa y las regiones
  - Programa imprime direcciones de distintos tipos de variables

Un segundo experimento

fperez@box1: ~/SII/experimentos_memoria/2_experimento

```
int global_noini;               /* Variable global escalar sin valor inicial */
int global_ini=666;             /* Variable global escalar con valor inicial */
int vec_global_noini[4000];     /* Variable global vector sin valor inicial */

/* Variable global vector con valor inicial */
int vec_global_ini[]={31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

const long double constante= 3.14159265358979323 8L; /* constante número ica */
char *string="ABCDEFG"; /* literal, que también se comporta como constante */


int main(int argc, char **argv) {   /* argc es un parámetro de una función */
        int local;              /* Variable local escalar */
        int vec_local[2500];    /* Variable local de tipo vector */
        static int local_estatica; /* Variable local estática */
        int *vec_dinamico = malloc(250*sizeof(int)); /* Variable dinámica */

        printf("\nproceso %d:\n\tmain %p\n\tconstante %p\n\tstring %p\n\tglobal_ini
%p\n\tvec_global_ini(%p,%p)\n\tglobal_noini %p\n\tvec_global_noini(%p,%p)\n\targc %p
\n\tlocal %p\n\tvec_local(%p,%p)\n\tlocal_estatica %p\n\tvec_dinamico(%p,%p)\n",
                getpid(), main, &constante, string, &global_ini,
                vec_global_ini, &vec_global_ini[sizeof(vec_global_ini)/sizeof(int)],
                &global_noini, vec_global_noini, &vec_global_noini[sizeof(vec_global
_noini)/sizeof(int)],
                &argc, &local, vec_local, &vec_local[sizeof(vec_local)/sizeof(int)],
                &local_estatica,
                vec_dinamico, &vec_dinamico[250]);

        printf("%d: Pulsa para acabar ", getpid());
                                                        12,0-1        33%
```

Salida del 2º experimento

```
fperez@box1: ~/SII/experimentos_memoria/2_experimento
fperez@box1:~/SII/experimentos_memoria/2_experimento$ ./programa

proceso 4983:
        main 0x400666
        constante 0x400800
        string 0x400810
        global_ini 0x601080
        vec_global_ini(0x6010a0,0x6010d0)
        global_noini 0x601100
        vec_global_noini(0x601120,0x604fa0)
        argc 0x7ffc780a088c
        local 0x7ffc780a0894
        vec_local(0x7ffc780a08a0,0x7ffc780a2fb0)
        local_estatica 0x6010e4
        vec_dinamico(0x251d010,0x251d3f8)
4983: Pulsa para acabar
```

pmap 4983

```
fperez@box1: ~/Imágenes
4983:   ./programa
0000000000400000      4K r-x-- programa
0000000000600000      4K r---- programa
0000000000601000      4K rw--- programa
0000000000602000     12K rw---   [ anon ]
000000000251d000    132K rw---   [ anon ]
00007fb17fe57000   1792K r-x-- libc-2.23.so
00007fb180017000   2048K ----- libc-2.23.so
00007fb180217000     16K r---- libc-2.23.so
00007fb18021b000      8K rw--- libc-2.23.so
00007fb18021d000     16K rw---   [ anon ]
00007fb180221000    152K r-x-- ld-2.23.so
00007fb180428000     12K rw---   [ anon ]
00007fb180444000      8K rw---   [ anon ]
00007fb180446000      4K r---- ld-2.23.so
00007fb180447000      4K rw--- ld-2.23.so
00007fb180448000      4K rw---   [ anon ]
00007ffc78084000    132K rw---   [ pila ]
00007ffc780e7000      8K r----   [ anon ]
00007ffc780e9000      8K r-x--   [ anon ]
ffffffffff600000      4K r-x--   [ anon ]
 total             4372K
:
```

Tipos de datos

- Estáticos
  - En C variables globales o locales con *static*
- Automáticos
  - En C parámetros de funciones y variables locales sin *static*
- Dinámicos
  - En C creados mediante *malloc*

Dato estático

- Existe durante toda la vida del programa
  - Ya aparece en el mapa inicial del proceso
  - Y seguirá en el mismo hasta que termine el programa (*exit*|*exec*)
- Tiene asignada una dirección fija
- Con o sin valor inicial asignado
  - Si tiene valor inicial, almacenado en el ejecutable
- Constante o variable
  - Si constante, incluido en región sin permiso de escritura
- Pueden ser locales o globales
- Operación sobre el dato puede usar direccionamiento absoluto
  - Instrucción ensamblador usa directamente la dirección del dato

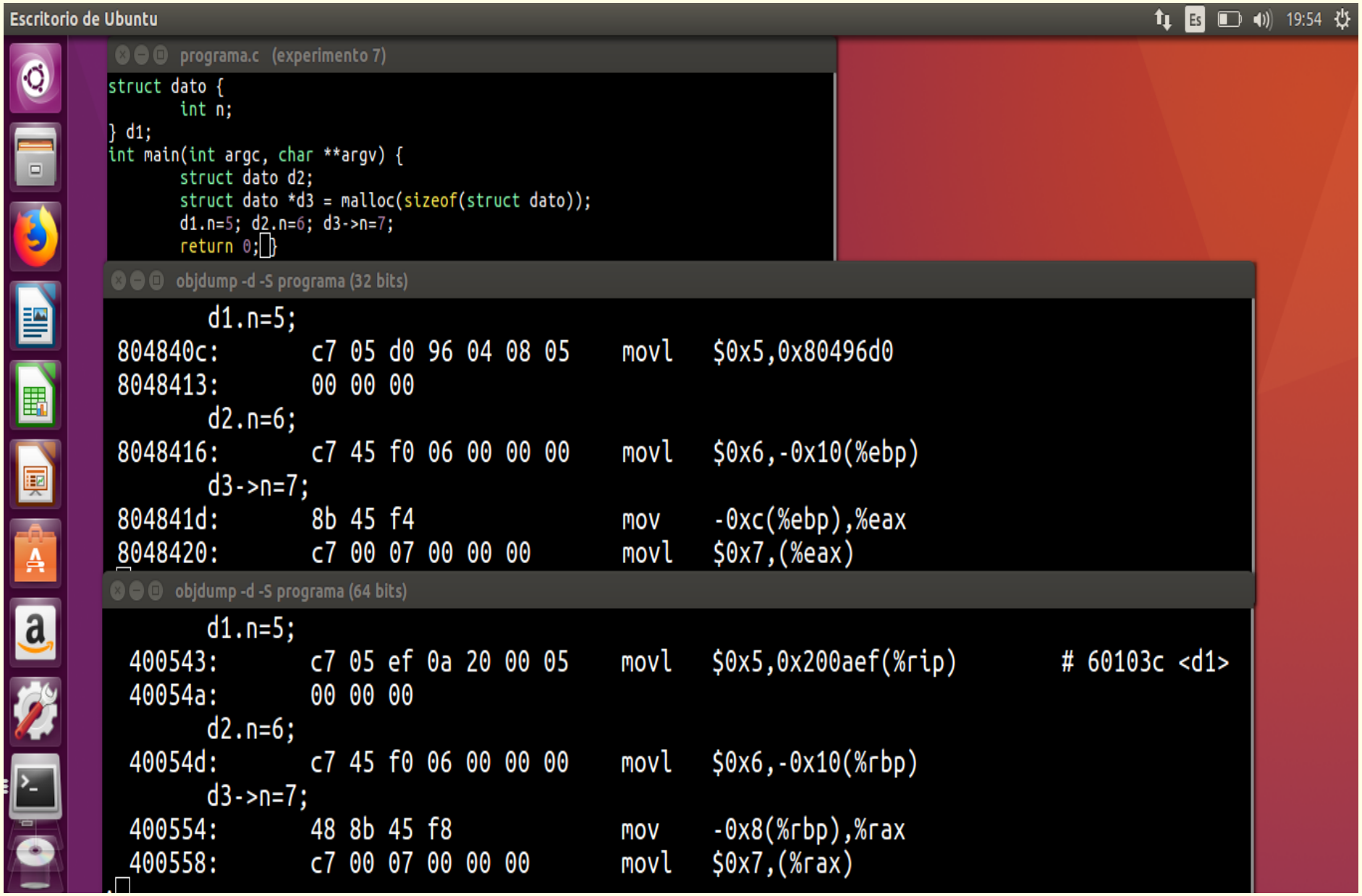
Dato automático

- Solo existe mientras dura la función a la que está vinculado
  - Se crea al llamar a la función y desaparece al terminar ésta
- No tiene dirección fija
- Puede haber múltiples instancias simultáneas
  - Función recursiva
- No incluido en mapa inicial del proceso ni en el ejecutable
- Con o sin valor inicial asignado
  - El valor inicial lo asigna el propio proceso
- Incluido en la región de pila
- Llamada a función incluye **registro de activación** en cima de pila
  - Incluye parámetros, v. locales de la función y dirección retorno
  - Al terminar la función se elimina el r. activación de cima de pila
- Operación sobre dato usa direccionamiento relativo a cima de pila

Dato dinámico

- El código del programa determina la vida del dato
- Lo crea explícitamente usando funciones del lenguaje (*malloc*)
- Destrucción del dato:
  - El programa lo libera explícitamente (*free*)
  - Automáticamente por recolección de basura (p. e. en Java)
- Almacenados en la región de *heap*
  - SO no interviene en cada *malloc* y *free*: los gestiona el lenguaje
- Operación sobre el dato requiere dos accesos a memoria
  - Primero hay que obtener en qué dirección está el dato
  - Y luego acceder al dato
- Tipo de dato con gestión menos eficiente que los dos anteriores
  - Sobrecarga de creación y destrucción del dato
    - Hay que buscar espacio en el *heap* y, al destruirlo, devolverlo
  - Sobrecarga en el acceso
- Pero tienen un modo de operación muy flexible

Experimento: acceso a distintos tipos de datos



Región de código

- Almacenada en el ejecutable
- Protección RX
- Compartida
  - También puede considerarse privada
    - Al no poder modificarse, es básicamente lo mismo
- Tamaño fijo
- Ubicación prefijada
- Suele incluir también constantes y literales

Región de datos con valor inicial

- Guarda valores iniciales de variables estáticas con valor inicial
- Almacenada en el ejecutable
- Protección RW
- Privada
  - Después de *fork*, padre e hijo tienen su propia copia
- Tamaño fijo
- Ubicación prefijada
  - Habitualmente, aparece después de región de código

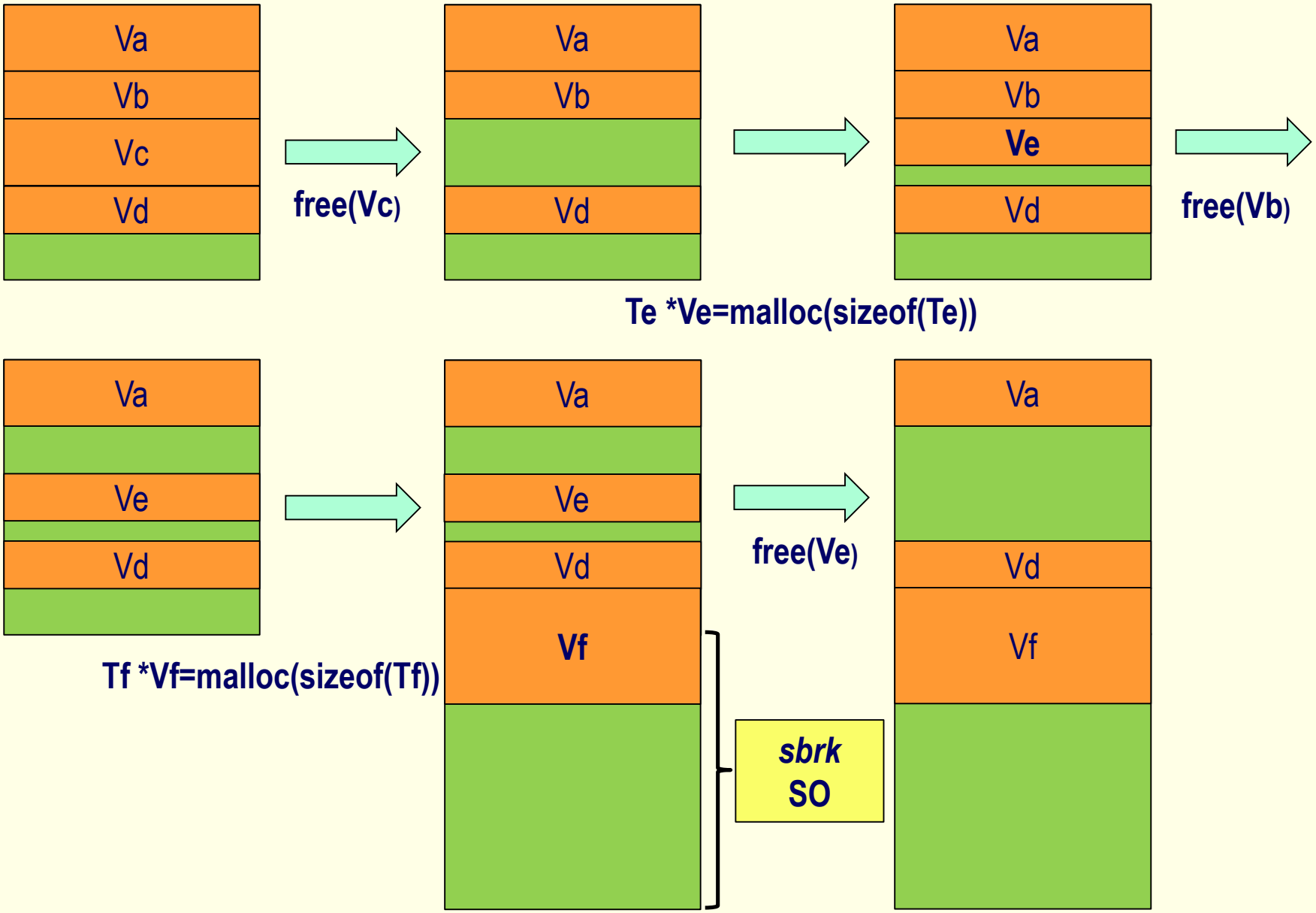
Región de datos sin valor inicial (*bss*)

- No almacenada en el ejecutable
  - no hay nada que guardar
  - pero en la cabecera del ejecutable aparece el tamaño de la misma
- Sin soporte: por seguridad se rellena con ceros
- Protección RW
- Privada
  - Después de *fork*, padre e hijo tienen su propia copia
- Tamaño fijo
- Ubicación prefijada
  - Habitualmente, justo después de región datos con valor inicial
    - Algunos datos de este tipo pueden incluirse en región datos v.inicial
    - Tamaño región múltiplo tamaño página aunque requiera - espacio

Región de *heap*

- Soporte de datos dinámicos
- No almacenada en el ejecutable ni en el mapa inicial
- Sin soporte
- Protección RW; Privada
  - Después de *fork*, padre e hijo tienen su propia copia
- Gestión interna del *heap* la realiza *runtime* del lenguaje (no el SO)
- Tamaño variable creciendo hacia direcciones ascendentes
  - No va creciendo por cada solicitud *malloc*
  - Cuando se agota, solicita a SO expansión grande mediante *sbrk*
    - Siguientes *malloc* usan expansión hasta que se agota de nuevo
- En algunos sistemas no es una región independiente
  - Se considera región de datos sin valor inicial de tamaño variable
- Ubicación libre: después de datos sin valor inicial
  - Se suele cambiar para dificultar ataques contra la seguridad

Evolución del *heap*



Región de pila

- Soporte de datos automáticos
- No almacenada en el ejecutable ni en el mapa inicial
- Sin soporte
- Contenido inicial:
  - argumentos y entorno del proceso
- Protección RW
- Privada
  - Después de *fork*, padre e hijo tienen su propia copia
- Tamaño variable creciendo hacia direcciones descendentes
  - Crece según se anidan llamadas a funciones
- Ubicación libre: hacia el final del mapa de memoria del proceso
  - Se suele cambiar para dificultar ataques contra la seguridad

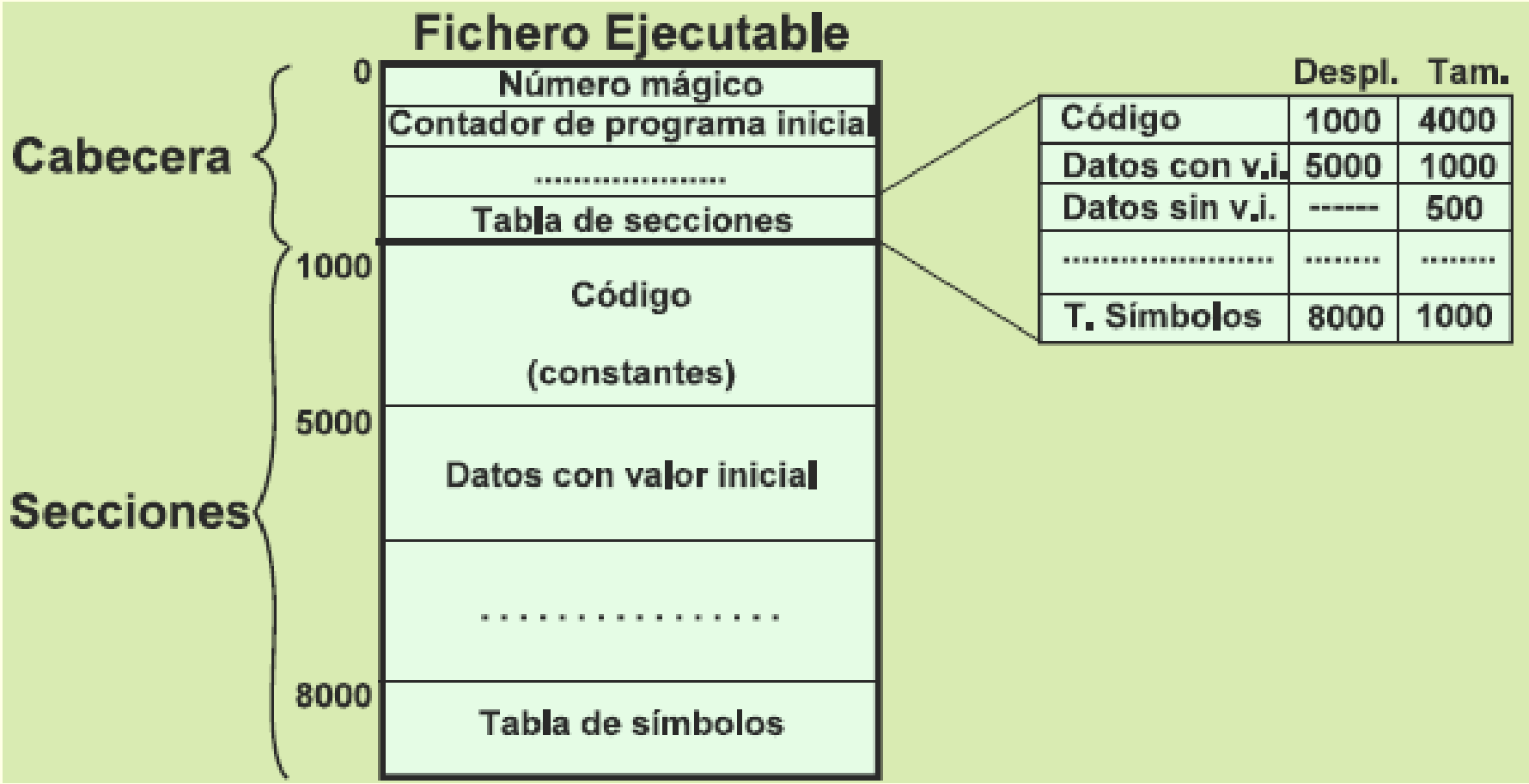
Otras regiones del mapa

- Pila de *thread*
  - Mismas características que la pila del programa
- Fichero proyectado (*mmap*)
  - Características dependen de parámetros de la petición
- Zona de memoria compartida (*shmat*, *shm_open*)
  - Compartida; demás características dependen de parámetros
  - Actualmente, muy similares a *mmap*
- Biblioteca dinámica; Conlleva regiones de:
  - Código, datos con valor inicial y sin valor inicial de la biblioteca

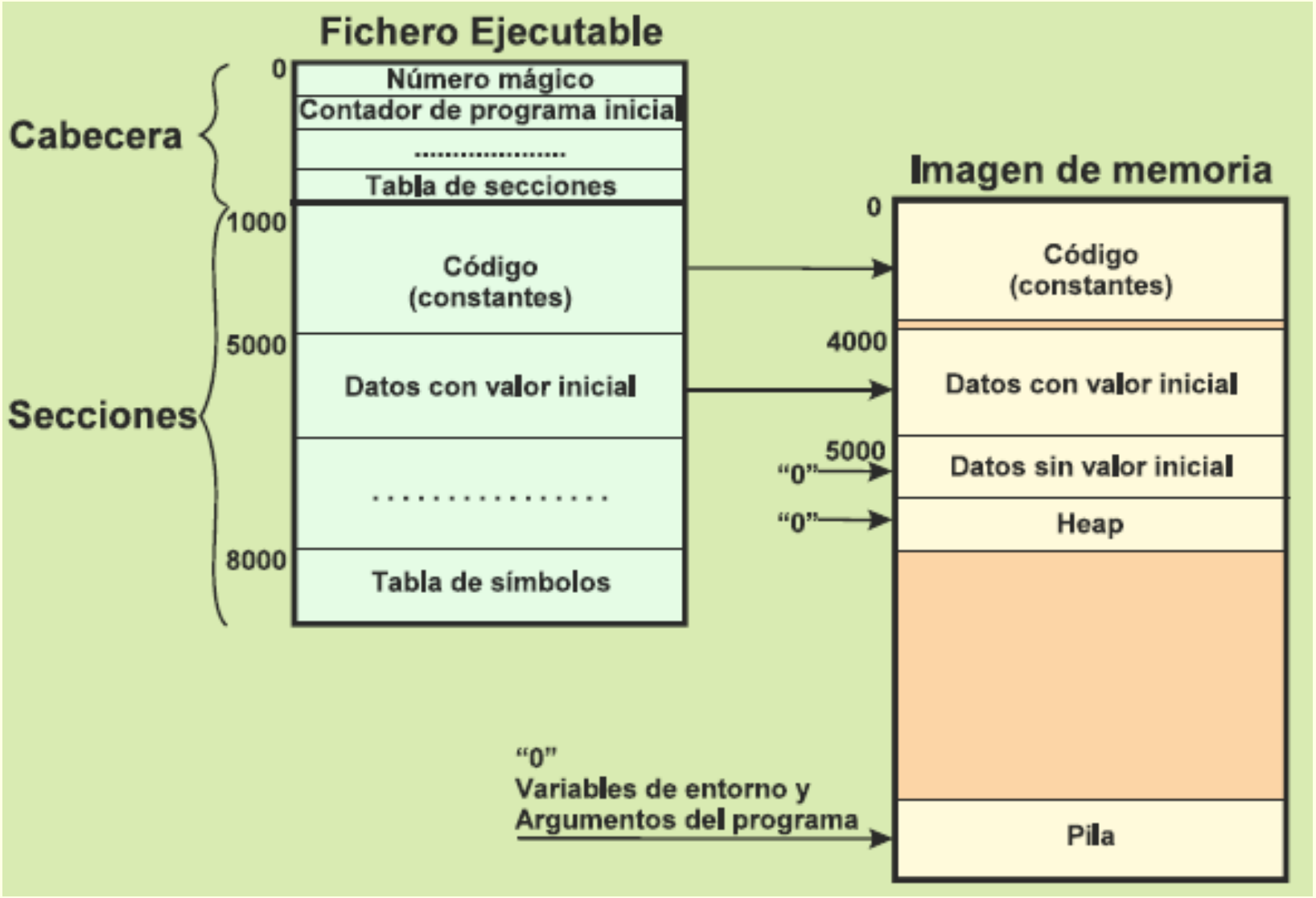
---

- Creación del mapa inicial del proceso (en UNIX, en *exec*)
  - A partir del ejecutable

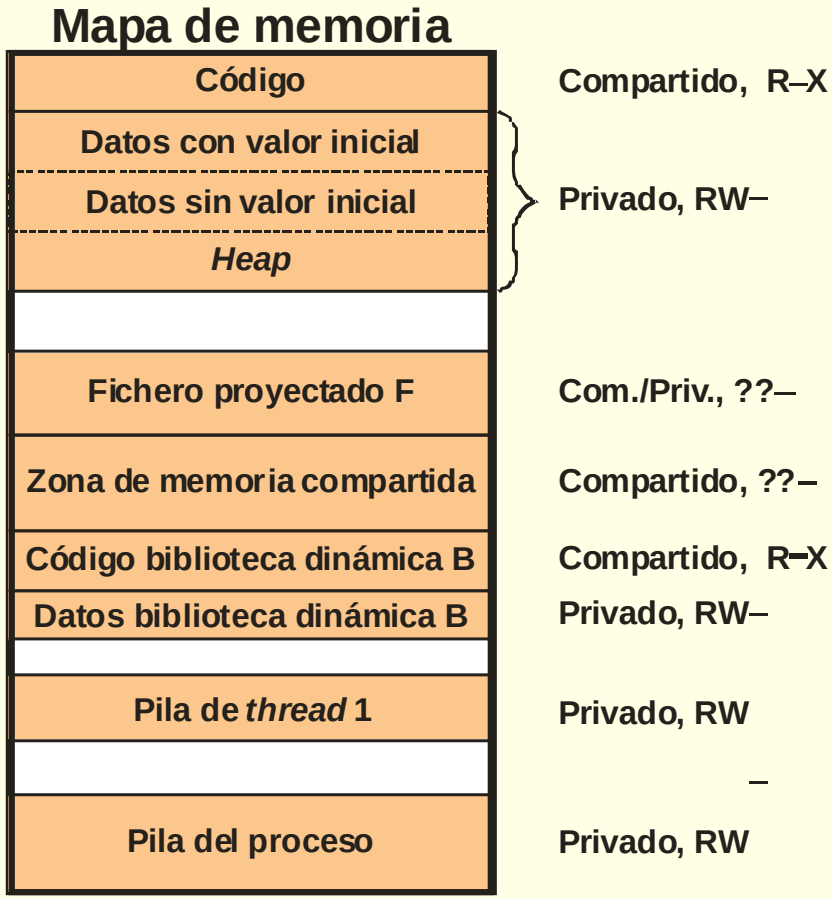
Formato del ejecutable



Crear mapa desde ejecutable



Mapa de memoria de un proceso hipotético



Regiones del mapa de memoria

Región	**Soporte**	**Protección**	**Comp/Priv**	**Tam**	**Ubicación**
Código	Fichero	RX	Compartida	Fijo	Prefijada
Dat. con v.i.	Fichero	RW	Privada	Fijo	Prefijada
Dat. sin v.i.	Sin soporte	RW	Privada	Fijo	Prefijada
Pilas	Sin soporte	RW	Privada	Var	Cualquiera
Heap	Sin soporte	RW	Privada	Var	Cualquiera
F. Proyect.	Fichero	por usuario	Comp./Priv.	Fijo	Cualquiera
M. Comp.	Sin soporte	por usuario	Compartida	Fijo	Cualquiera
Bib.dinám.	Regiones para código y datos (con y sin valor inicial)				

Mandato *size*: tamaño secciones del ejecutable

```
fperez@box1: ~/SII/experimentos_memoria
fperez@box1:~/SII/experimentos_memoria$ size 1_experimento/programa /usr/li
b/firefox/firefox
   text    data     bss     dec     hex filename
   1488     572       4    2064     810 1_experimento/programa
 151521    1728    1808  155057   25db1 /usr/lib/firefox/firefox
fperez@box1:~/SII/experimentos_memoria$
```

Contenido

- Aspectos generales de la gestión de memoria
- Mapa de memoria de un proceso
- **Gestión de la memoria del sistema**
  - Paginación
  - Memoria virtual
- Proyección de archivos
- Bibliotecas dinámicas

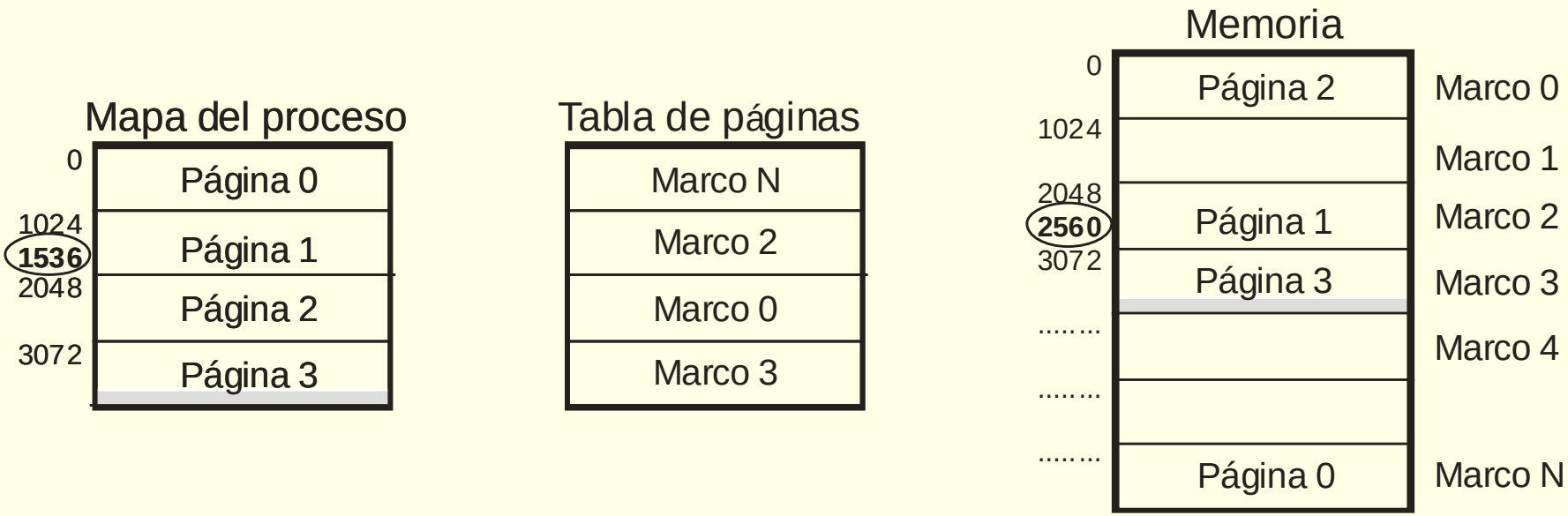
Creación de espacios lógicos independientes

- Se requiere función de traducción para cada proceso que:
  - Cree un espacio lógico de 0 hasta un cierto valor
  - Proporcione aislamiento entre los procesos
  - Pero permitiendo compartir memoria de forma controlada
  - Gestione los permisos de acceso a cada posición de memoria
    - Detectando si se producen accesos no permitidos
  - Evite que huecos en espacio lógico consuman espacio
    - Detectando si se producen accesos a direcciones en esos huecos
- Solución no factible: Tabla por proceso que haga corresponder
  - Cada dirección lógica con dirección física asociada
  - Sería enorme
- Cambio de escala: de 1 dirección a un bloque de direcciones
  - Denominado **página** (tamaño típico 4KB)

Paginación

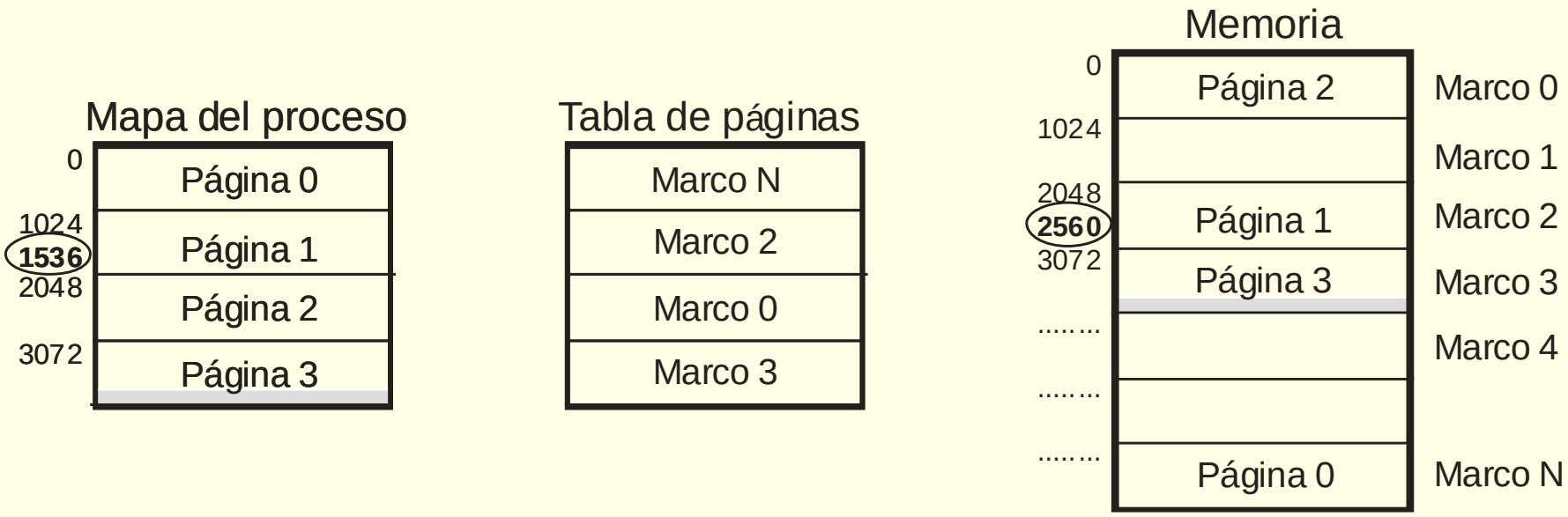
- Espacio lógico del proceso dividido en páginas
- Memoria física dividida en marcos de página
- Función de traducción por cada proceso: Tabla de páginas
  - Tablas de páginas son grandes: se almacenan en memoria
- Entrada de tabla de páginas contiene entre otros:
  - Marco asociado a esa página (nulo si corresponde a un hueco)
  - Permisos de direcciones incluidas en esa página (RWX)
  - Bit de referencia y de modificado
    - Indican si página ha sido accedida y modificada, respectivamente
- Si operación corresponde a hueco o no permitida
  - Se produce una excepción de **fallo de página** o **de protección**
- A cada región se le asigna un nº entero de páginas
- Compartir región entre procesos:
  - entradas de t. páginas de procesos apuntan a mismos marcos

Traducción con paginación (p.e. páginas 1KB)



¿Fórmula que realiza esta traducción?

Traducción con paginación (p.e. páginas 1KB)



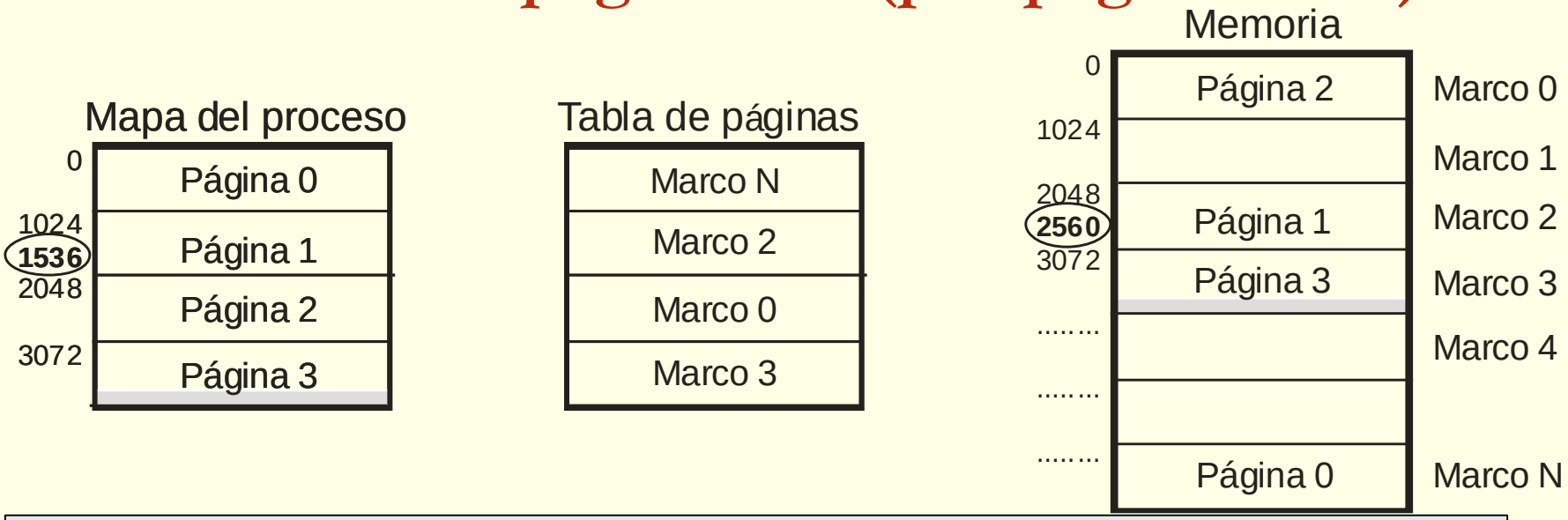
Dfís = TablaPág[DLóg/TamPág] * TamPág + DLóg%TamPág

Dfís = TablaPág[1536/1024] * 1024 + 1536%1024

Dfís = TablaPág[1] * 1024 + 512 → Dfís = 2*1024+512 → Dfís = 2560

¡Por cada acceso a memoria multiplicaciones y divisiones! ¿Eficiencia?

Traducción con paginación (p.e. páginas 1KB)



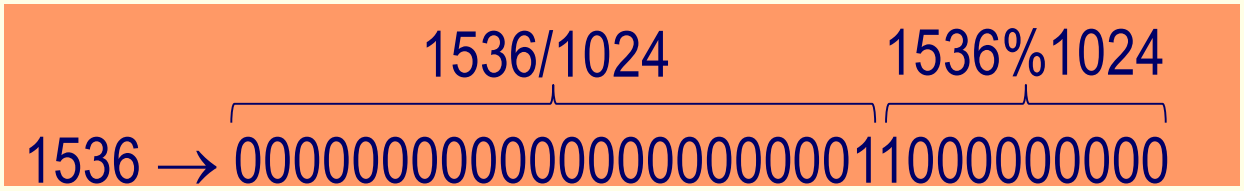
Dfís = TablaPág[DLóg/TamPág] * TamPág + DLóg%TamPág

Dfís = TablaPág[1536/1024] * 1024 + 1536%1024

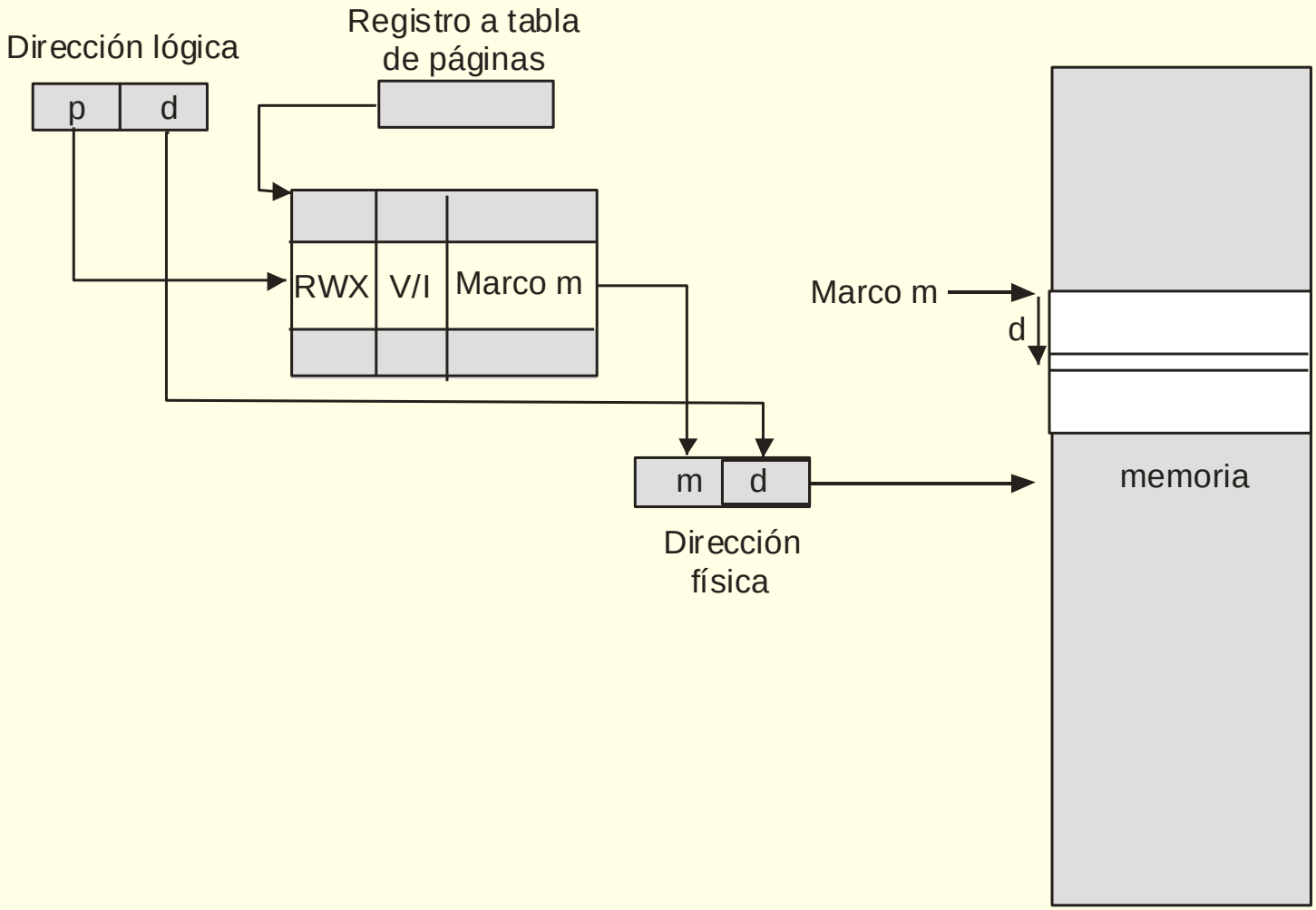
Dfís = TablaPág[1] * 1024 + 512 → Dfís = 2*1024+512 → Dfís = 2560

¡Por cada acceso a memoria multiplicaciones y divisiones! ¿Eficiencia?

Tamaño de tabla de páginas es potencia de 2: * / % son inmediatas



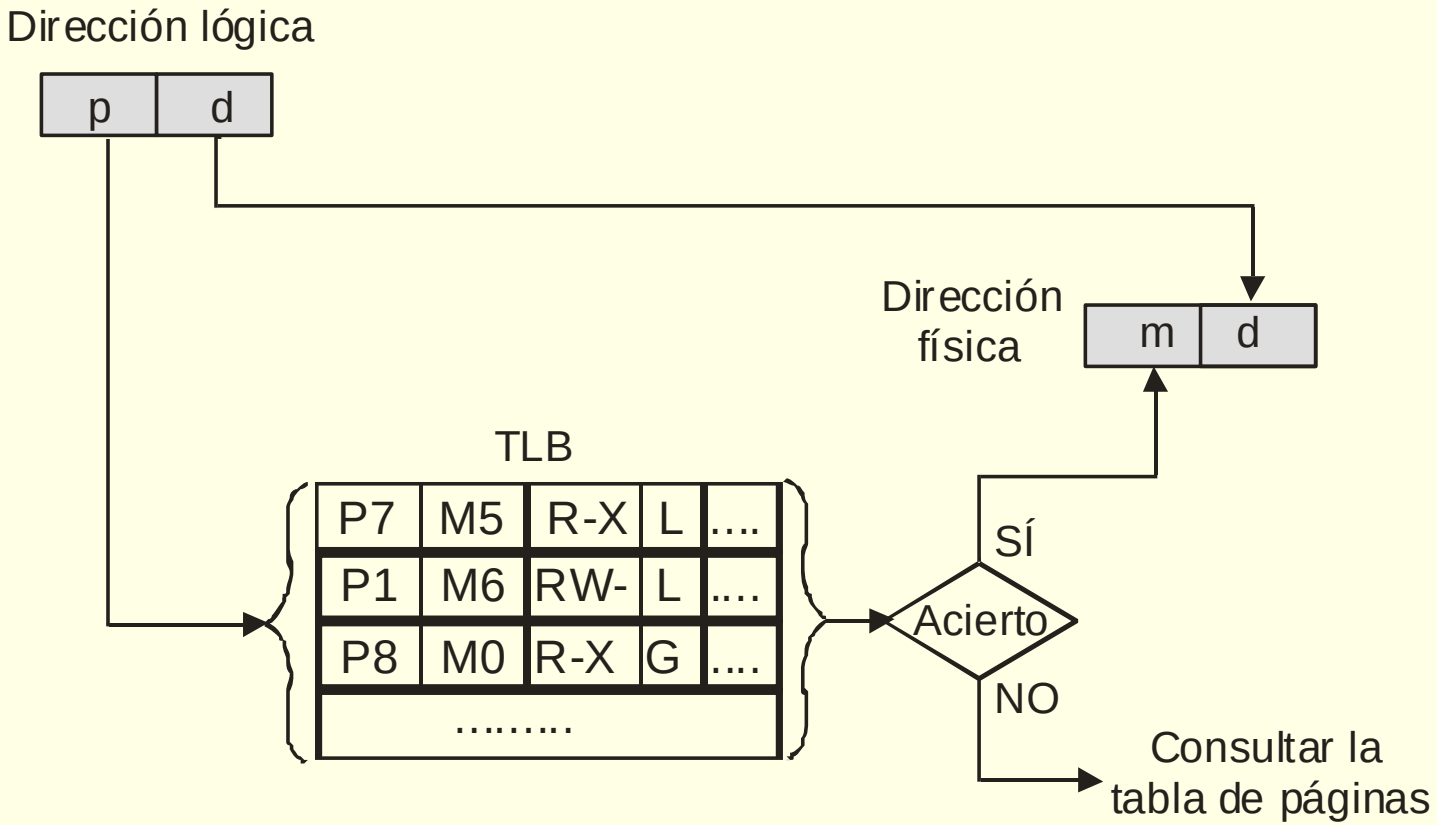
Hardware de paginación



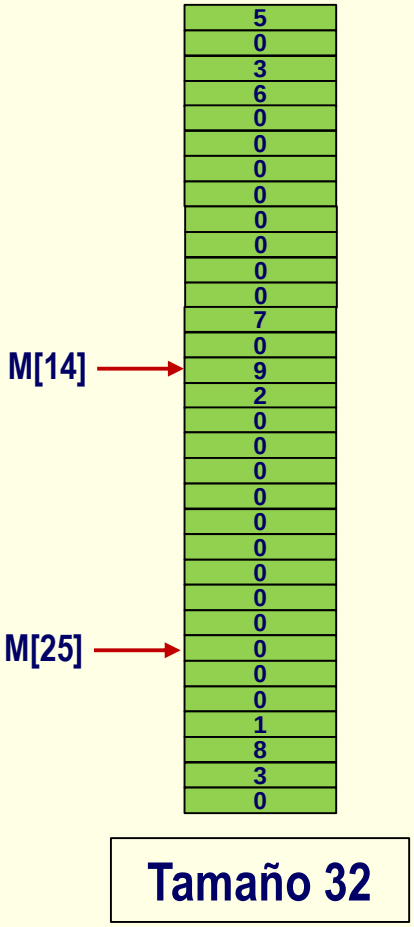
Implementación de TP

- TP se mantiene normalmente en memoria principal: 2 problemas
- Eficiencia: Cada acceso lógico requiere 2 accesos a m. principal
  - Solución: *caché* de traducciones → TLB
    - Memoria asociativa con info. sobre últimas páginas accedidas
    - Parte de la MMU: Si fallo en TLB, MMU usa la TP en memoria
    - En cambio de contexto, SO debe invalidar su contenido
      - Excepto si es entre *threads* del mismo proceso
- Gasto de almacenamiento: Tablas muy grandes
  - **Ejemplo**: páginas 4K, dir. lógica 32 bits y 4 bytes/entrada
    - Tamaño TP: 2^{20} *4 = 4MB/proceso; si 1K procesos→ ¡4G en TPs!
      - aunque un cierto proceso use solo 12MB superiores y 4MB inferiores
  - Solución: tablas multinivel (x86-64 → 4 niveles)
    - Tablas de páginas son muy grandes con muchas entradas nulas
    - Solución: Fragmentar tabla y acceder mediante tabla maestra

TLB

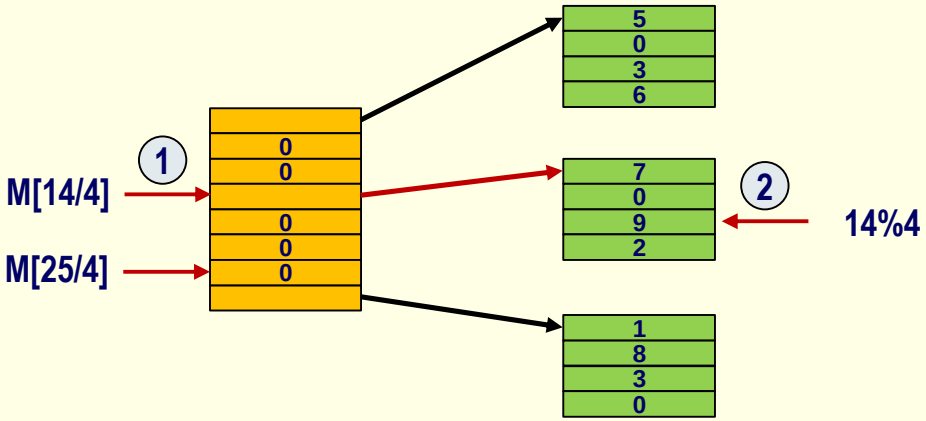

Dirección lógica
p
d
Dirección
física
m
d
TLB
P7 M5 R-X L ....
P1 M6 RW- L ....
P8 M0 R-X G ....
SÍ
Acierto
NO
Consultar la
tabla de páginas

Compactando una matriz dispersa



Tamaño 32

Pro: Menor ocupación
Con: 2 accesos



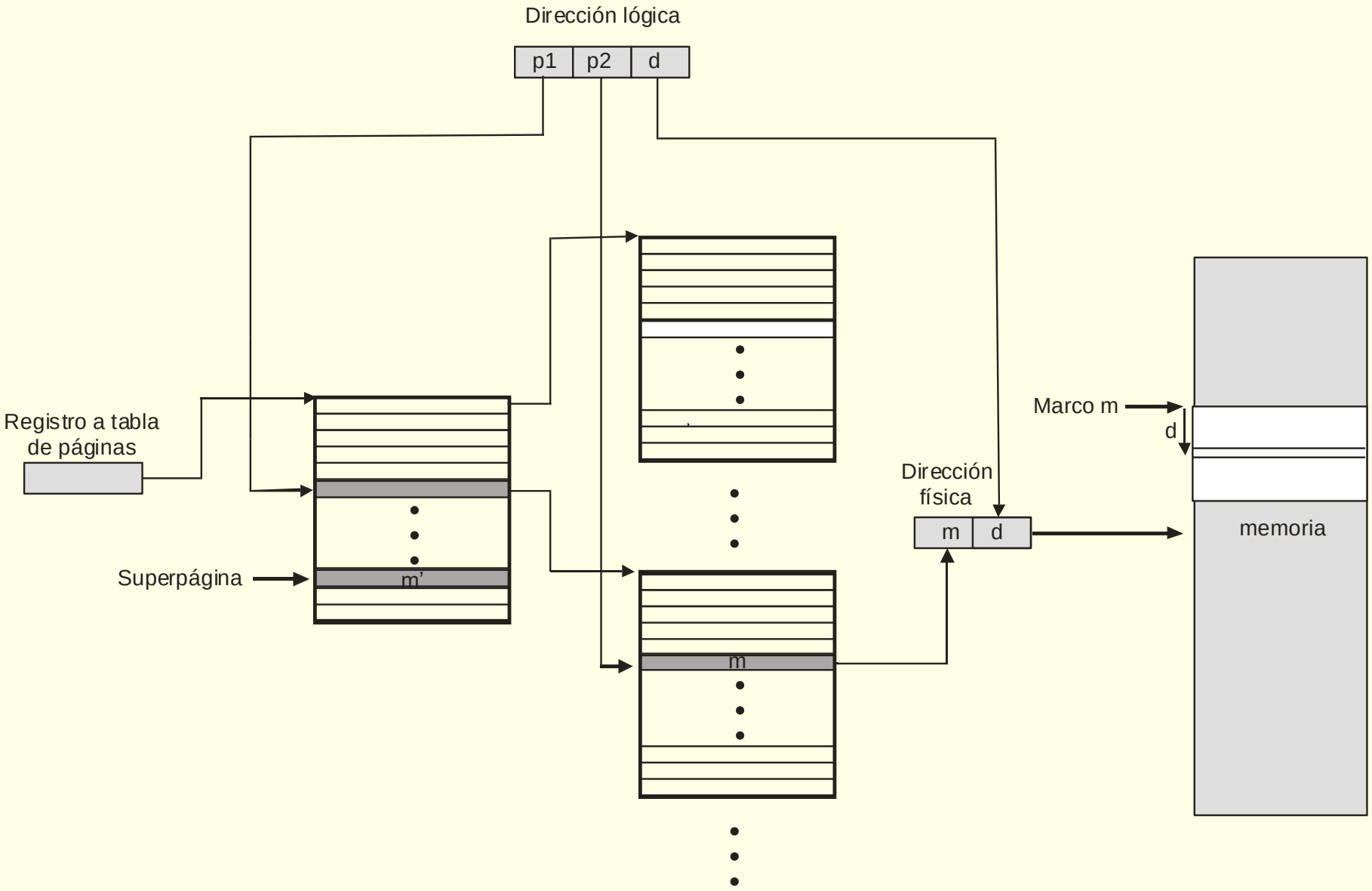
Tamaño 20

/ y % inmediatas si potencia de 2

14/4 14%4

14 → 0 1 1 1 0

Tablas multinivel con 2 niveles



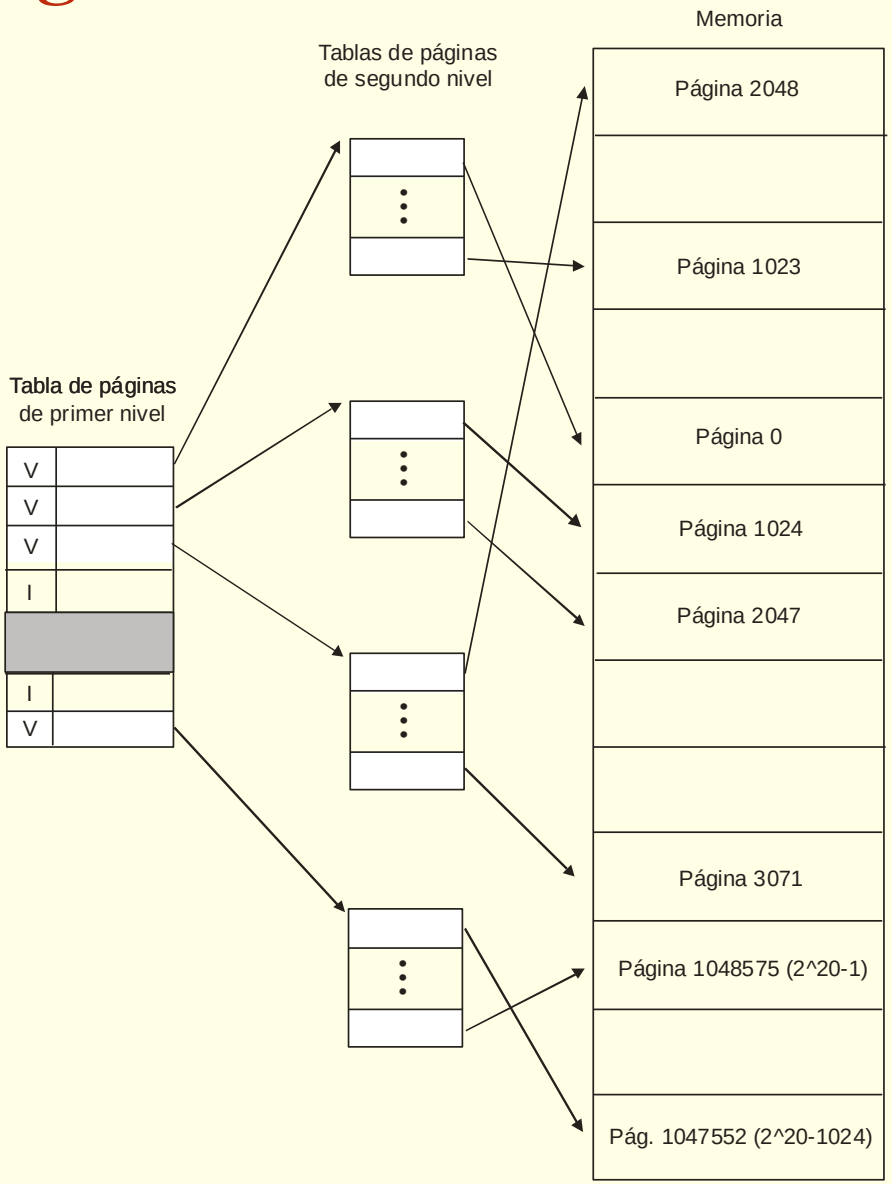
Ventajas de tablas de páginas multinivel

Si en ejemplo se usan
2 niveles con 10 bits/nivel

Tamaño total en t. de pág.:

1 TP N1 + 4 TP N2= 5 * 4KB
20KB (frente a 4MB)

si 1K procesos iguales→ 20MB
(frente a 4G)



Memoria virtual

- Aplicable ya que procesos cumplen *proximidad de referencias*
  - Si se accede a una dir. lógica, alta probabilidad de que en breve
    - se acceda a esa misma dirección o a una próxima
  - Procesos sólo usan parte de su mapa en intervalo de tiempo
- Idea: mantener en memoria sólo esa parte de procesos activos
  - Parte usada (*cjto de trabajo*) → en memoria (*cjto residente*)
- Beneficios:
  - Aumenta el grado de multiprogramación
  - Permite ejecución de programas que no quepan en memoria
- Al crear mapa, no se carga nada en memoria
  - Espera a 1er acceso a cada página → **Paginación por demanda**
- Para páginas válidas pero no residentes en memoria, :
  - Se marca como inválida en tabla de páginas pero
    - se guarda en qué bloque de disco está almacenada o a rellenar a 0
- Acceso a página no residente en memoria:
  - Excepción de fallo de página → La procesa el SO

Tratamiento de fallo de página

- Si dirección lógica no es válida (hueco)
  - SO aborta al proceso (envía señal *SIGSEGV*)
- Si es válida, se busca un marco libre
- Si no hay ningún marco libre, obtiene uno:
  - Selecciona uno ocupado por otra página
    - **algoritmo de reemplazo**
  - Si esa página se ha modificado, hay que escribirla en disco
    - Se guarda información de en qué bloque de disco se ha escrito
- Usa el marco libre para la página que causó el fallo
  - Si almacenada en disco, la lee del mismo al marco libre
  - Si rellenar a ceros, escribe ceros en el marco libre
  - Actualiza entrada tabla páginas para que referencie a ese marco
- Al finalizar tratamiento de fallo de página
  - Proceso prosigue su ejecución justo en el punto donde estaba

Algoritmos de reemplazo

- Objetivo: Minimizar la tasa de fallos de página.
  - Poca sobrecarga en tiempo y espacio y MMU estándar
- Algoritmo óptimo: Irrealizable (necesitaría conocer el futuro)
  - Selecciona página residente que tardará más en accederse
- Algoritmo FIFO: Selecciona página más tiempo residente
  - Mala idea: puede ser una página que se usa mucho
- Algoritmo LRU (*Least Recently Used*)
  - Selecciona página residente menos recientemente usada
  - Proximidad de referencias: pasado reciente $\rightarrow$ futuro próximo
  - Difícil implementación estricta
    - Precisaría una MMU específica
    - Hay aproximaciones: P.e. Algoritmo del reloj
      - Requiere solo el bit de referencia

Buffering de páginas

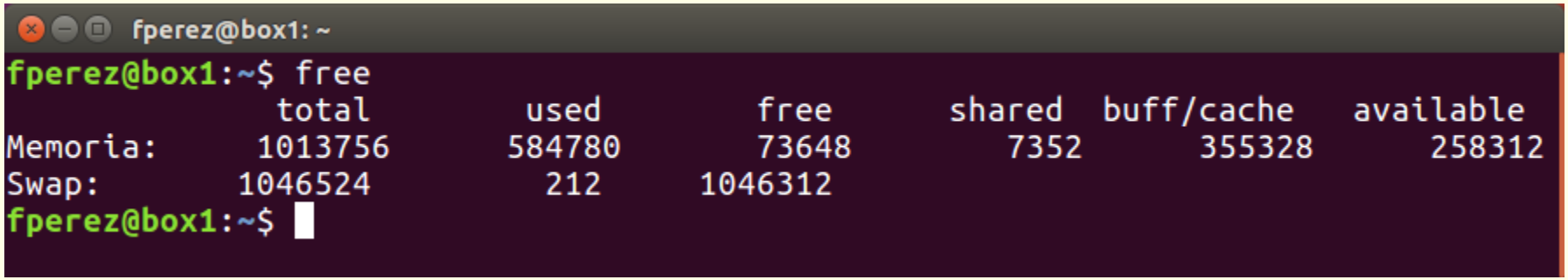
- No parece razonable esperar hasta que se agote la memoria
  - En vez de reemplazo dentro de rutina de fallo de página
  - Mejor por anticipado
- Se mantiene una reserva de marcos libres
- Fallo de página: siempre usa marco libre (no reemplazo)
- Si nº marcos libres < umbral
  - "demonio de paginación" aplica algoritmo de reemplazo
    - páginas no modificadas → lista de marcos libres
    - páginas modificadas → lista de marcos modificados
      - ► cuando se escriban a disco pasan a lista de libres
- Si se referencia una página mientras está en estas listas:
  - fallo de página la recupera directamente de la lista (no E/S)

Regiones duplicadas: *Copy-on-write* (COW)

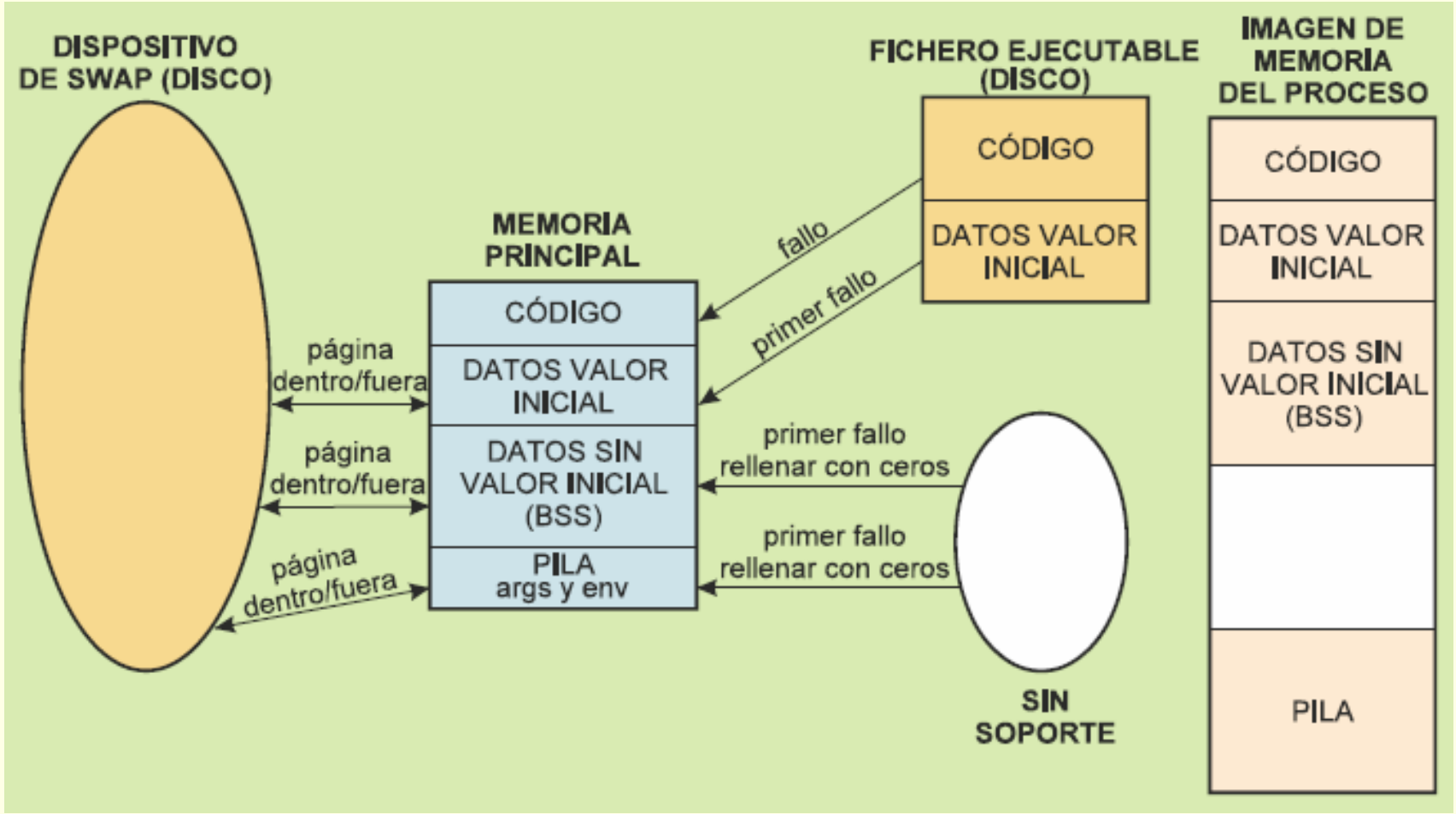
- Procesos usan regiones duplicadas:
  - Regiones privadas después de *fork*
  - Datos con valor inicial de programas y bibliotecas
- Operación de duplicado es costosa
  - Requiere copiar los marcos de página de la región
- En ocasiones, procesos no modifican todas las páginas de la región
  - Copia innecesaria (p.e. *exec* después de *fork*)
- Optimización: Duplicado por demanda (*copy-on-write*, COW)
  - Se comparte una página de la región mientras no se modifique
  - Se fijan como no escritura entradas de las páginas de la región
  - Si proceso la modifica (**fallo protección**) se crea copia para él
    - Siempre que haya permiso de escritura en la región
- **Recordatorio importante**: Con memoria virtual, SO
  - SO solo asigna espacio (marcos) en fallo página y protección
    - **Nunca** al crear el mapa o una región del mismo

Memoria secundaria: dispositivo de *swap*

- ¿Dónde se escribe la página reemplazada modificada?
- Si página remplazada ∈ región compartida y soporte en fichero
  - Se escribe en el fichero
- Si no, se escribe en memoria secundaria
  - Una especie de extensión de la memoria en el disco
    - Donde se almacenan páginas modificadas reemplazadas
  - Si vuelve a accederse, fallo de página y leer de disco
  - Pero recuerde que disco es muchísimo más lento que memoria
  - En 1ª expulsión de la página se asigna espacio en m. secundaria
    - Alternativa preasignación:
      - Cuando se crea la región se reserva ya espacio en m. secundaria
- En UNIX, suele denominarse dispositivo/partición de *swap*



Evolución del mapa del proceso

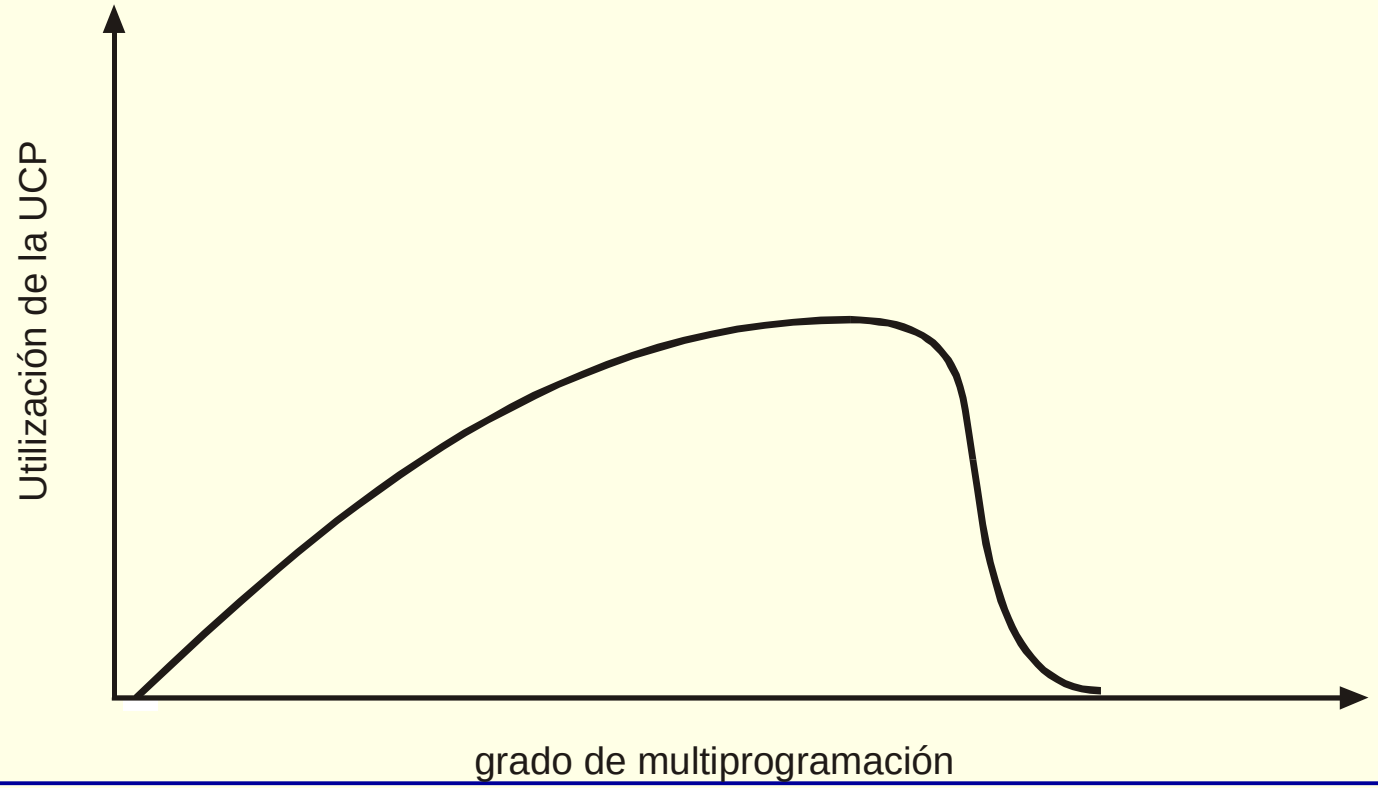


Hiperpaginación (*Thrashing*)

Tasa excesiva de fallos de página en el sistema

Si demasiados procesos, no caben en memoria sus conjuntos de trabajo

- Rendimiento del sistema cae en picado
- Hay que suspender la ejecución de uno o más procesos



Contenido

- Aspectos generales de la gestión de memoria
- Mapa de memoria de un proceso
- Gestión de la memoria del sistema
  - Paginación
  - Memoria virtual
- **Proyección de archivos**
- Bibliotecas dinámicas

Ficheros proyectados en memoria

- Programa solicita proyección de fichero (o parte) en su mapa
  - SO crea una región asociada al fichero
- Características de la región: las especificadas en la solicitud
  - Protección; Privada o compartida
    - Si privada, cambios no se reflejan en el fichero
  - Dirección prefijada o libre; Basada en fichero o sin soporte
- Programa accede a región de memoria asociada a fichero
  - Fallo de página trae la página → Está accediendo al fichero
  - Al expulsar página modificada de proyección compartida
    - Está escribiendo en el fichero
- Forma alternativa de acceso a ficheros frente a *read*/*write*
  - Menos llamadas al sistema (no *read*, *write*, *lseek*): más eficiente
  - Se facilita programación: acceso directo a estructuras de datos

Servicios de proyección de UNIX

*void **__mmap__*(void *dir, size_t lon, int prot, int flags, int fd, off_t desp);*

- Establece proyección entre espacio de dir. de proceso y fichero
  - Devuelve dirección de memoria donde se ha proyectado fichero
  - *dir* dirección lógica donde proyectar. Si *NULL* SO elige una
  - *lon* especifica el número de bytes a proyectar
  - *prot* protección de acceso: *PROT_READ PROT_WRITE PROT_EXEC*
  - *flags* info. sobre proyección
    - Compartida (*MAP_SHARED*) o privada (*MAP_PRIVATE*)
    - Anónima (*MAP_ANON*): sin soporte
    - Proyecta en dirección lógica especificada en *dir* (*MAP_FIXED*)
  - *fd* el descriptor de fichero a proyectar (si no *MAP_ANON*)
  - *desp* desplazamiento en fichero a partir del que realiza proyección

void __*munmap*__*(void *dir, size_t lon);*

- Desproyecta desde la dirección lógica *dir* hasta *dir+lon*
- NOTA: *dir* y *desp* deben ser múltiplos del tamaño de página

Ejemplos de uso de *mmap*

- Contar cuántas veces aparece un carácter en un fichero de texto
- Cambiar minúsculas por mayúsculas en un fichero de texto
- Copiar un fichero cambiando minúsculas por mayúsculas
- Gestión de una plantilla como un vector de empleados
  - Con problema de autorreferencias
  - Resuelto cambiando puntero por entero
- Juego de tablero compartido y con persistencia

- NOTA:
  - Por simplicidad, programas sin control de errores

Cuántas veces aparece un carácter en un fichero

fperez@box1: ~/GMI/ejemplos_mmap/1_contarcar

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main(int  argc, char **argv) {
        int i, fd, contador=0;
        char caracter;
        char *p;
        struct stat bstat;

        caracter=argv[1][0];
        fd=open(argv[2], O_RDONLY);
        fstat(fd, &bstat);
        p=mmap((caddr_t) 0, bstat.st_size, PROT_READ, MAP_SHARED, fd, 0);
        close(fd);
        for (i=0; i<bstat.st_size; i++)
                if (p[i]==caracter) contador++;
                //if (*p++==caracter) contador++; // EQUIVALENTE
        munmap(p, bstat.st_size);
        printf("%d\n", contador);
        return(0);
}
                                                        1,1           Todo
```

Pasar a mayúsculas un fichero de texto

fperez@box1: ~/GMI/ejemplos_mmap/2_amayusculas/1_mismo_fichero/mmap

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <ctype.h>

int main(int  argc, char **argv) {
        int i, fd;
        char *p;
        struct stat bstat;

        fd=open(argv[1], O_RDWR);
        fstat(fd, &bstat);
        p=mmap((caddr_t) 0, bstat.st_size, PROT_READ|PROT_WRITE,
                        MAP_SHARED, fd, 0);
        close(fd);
        for (i=0; i<bstat.st_size; i++)
                if (islower(p[i]))
                        p[i]=toupper(p[i]);
        munmap(p, bstat.st_size);
        return(0);
}
                                                        6,1           Todo
```

Copia fichero de texto pasando a mayúsculas

fperez@box1: ~/GMI/ejemplos_mmap/2_amayusculas/2_otro_fichero

```
int main(int  argc, char **argv) {
        int i, fd, fdd;
        char *org, *dst;
        struct stat bstat;

        fd=open(argv[1], O_RDONLY);
        fstat(fd, &bstat);
        org=mmap((caddr_t) 0, bstat.st_size, PROT_READ,
                        MAP_SHARED, fd, 0);
        close(fd);
        fdd=open(argv[2], O_CREAT|O_TRUNC|O_RDWR, 0640);
        ftruncate(fdd, bstat.st_size);
        dst=mmap((caddr_t) 0, bstat.st_size, PROT_WRITE,
                        MAP_SHARED, fdd, 0);
        close(fdd);
        for (i=0; i<bstat.st_size; i++)
                if (islower(org[i])) dst[i]=toupper(org[i]);
                else dst[i]=org[i];
        munmap(org, bstat.st_size);
        munmap(dst, bstat.st_size);

        return(0);
}
```

31,2 Final

Creación empleado (con problema autorreferencias)

fperez@box1: ~/GMI/ejemplos_mmap/3_empleados/3_map_problema_autorreferencias

```
struct empleado {
        char nombre[32];
        int nhijos;
        struct empleado *jefe; // etc.
};
char *fich_plantilla ="plantilla.dat";
int main(int argc, char *argv[]) {
        struct stat bstat;
        int fd, tam, nempl, nempls;
        struct empleado *plantilla;

        fd=open(fich_plantilla, O_RDWR|O_CREAT, 0600);
        fstat(fd, &bstat);
        tam= bstat.st_size+sizeof(struct empleado);
        ftruncate(fd, tam);
        plantilla=mmap((caddr_t) 0, tam, PROT_WRITE, MAP_SHARED, fd, 0);
        nempl=bstat.st_size/sizeof(struct empleado);
        strcpy(plantilla[nempl].nombre, argv[1]);
        plantilla[nempl].nhijos=atoi(argv[2]);
        plantilla[nempl].jefe=NULL;
        if (argc==4) { // tiene jefe
                nempls=bstat.st_size/sizeof(struct empleado);
                for (int i=0; i<nempls; i++)
                        if (strcmp(plantilla[i].nombre, argv[3])==0){
                                plantilla[nempl].jefe=&plantilla[i];
                                break;
                        }
        }
        munmap(plantilla, tam);
        return 0;
}
```

40,2 Final

Imprimir plantilla (con problema autorreferencias)

fperez@box1: ~/GMI/ejemplos_mmap/3_empleados/3_map_problema_autorreferencias

```
struct empleado {
        char nombre[32];
        int nhijos;
        struct empleado *jefe; // etc.
};
char *fich_plantilla ="plantilla.dat";
int main(int argc, char *argv[]) {
        struct stat bstat;
        int fd, tam, nempls;
        struct empleado *plantilla;

        fd=open(fich_plantilla, O_RDONLY);
        fstat(fd, &bstat);
        tam= bstat.st_size;
        plantilla=mmap((caddr_t) 0, tam, PROT_READ,
                        MAP_SHARED, fd, 0);
        nempls=bstat.st_size/sizeof(struct empleado);
        for (int i=0; i<nempls; i++){
                printf("Empleado: nombre %s nhijos %d\n",
                        plantilla[i].nombre, plantilla[i].nhijos);
                if (plantilla[i].jefe)
                        printf("\tJefe: nombre %s\n", plantilla[i].jefe->nombre)
;
        }
        munmap(plantilla, tam);
        return 0;
}
```

36,2 Final

Incrementar hijos de empleado

fperez@box1: ~/GMI/ejemplos_mmap/3_empleados/3_map_problema_autorreferencias

```
struct empleado {
        char nombre[32];
        int nhijos;
        struct empleado *jefe; // etc.
};
char *fich_plantilla ="plantilla.dat";
int main(int argc, char *argv[]) {
        struct stat bstat;
        int fd, tam, nempls;
        struct empleado *plantilla;

        fd=open(fich_plantilla, O_RDWR|O_CREAT, 0600);
        fstat(fd, &bstat);
        tam= bstat.st_size;
        plantilla=mmap((caddr_t) 0, tam, PROT_READ|PROT_WRITE,
                        MAP_SHARED, fd, 0);
        nempls=bstat.st_size/sizeof(struct empleado);
        for (int i=0; i<nempls; i++)
                if (strcmp(plantilla[i].nombre, argv[1])==0){
                        plantilla[i].nhijos++;
                        break;
                }
        munmap(plantilla, tam);
        return 0;
}
```

35,2 Final

Creación empleado (sin problema autorreferencias)

fperez@box1: ~/GMI/ejemplos_mmap/3_empleados/4_mmap_problema_autorreferencias

```
struct empleado {
        char nombre[32];
        int nhijos;
        int jefe; // etc.
};
char *fich_plantilla ="plantilla.dat";
int main(int argc, char *argv[]) {
        struct stat bstat;
        int fd, tam, nempl, nempls;
        struct empleado *plantilla;

        fd=open(fich_plantilla, O_RDWR|O_CREAT, 0600);
        fstat(fd, &bstat);
        tam= bstat.st_size+sizeof(struct empleado);
        ftruncate(fd, tam);
        plantilla=mmap((caddr_t) 0, tam, PROT_WRITE, MAP_SHARED, fd, 0);
        nempl=bstat.st_size/sizeof(struct empleado);
        strcpy(plantilla[nempl].nombre, argv[1]);
        plantilla[nempl].nhijos=atoi(argv[2]);
        plantilla[nempl].jefe=-1;
        if (argc==4) { // tiene jefe
                nempls=bstat.st_size/sizeof(struct empleado);
                for (int i=0; i<nempls; i++)
                        if (strcmp(plantilla[i].nombre, argv[3])==0){
                                plantilla[nempl].jefe=i;
                                break;
                        }
        }
        munmap(plantilla, tam);
        return 0;
}
```

41,2 Final

Imprimir plantilla (sin problema autorreferencias)

fperez@box1: ~/GMI/ejemplos_mmap/3_empleados/4_mmap_problema_autorreferencias

```
#include <fcntl.h>
#include <ctype.h>
#include <string.h>


struct empleado {
        char nombre[32];
        int nhijos;
        int jefe; // etc.
};
char *fich_plantilla ="plantilla.dat";
int main(int argc, char *argv[]) {
        struct stat bstat;
        int fd, tam, nempls;
        struct empleado *plantilla;

        fd=open(fich_plantilla, O_RDONLY);
        fstat(fd, &bstat);
        tam= bstat.st_size;
        plantilla=mmap((caddr_t) 0, tam, PROT_READ,
                        MAP_SHARED, fd, 0);
        nempls=bstat.st_size/sizeof(struct empleado);
        for (int i=0; i<nempls; i++){
                printf("Empleado: nombre %s nhijos %d\n",
                        plantilla[i].nombre, plantilla[i].nhijos);
                if (plantilla[i].jefe!=-1)
                        printf("\tJefe: nombre %s\n", plantilla[plantilla[i].jefe].nombre);
        }
        munmap(plantilla, tam);
        return 0;
}
```

36,2 Final

Juego con tablero compartido y persistente

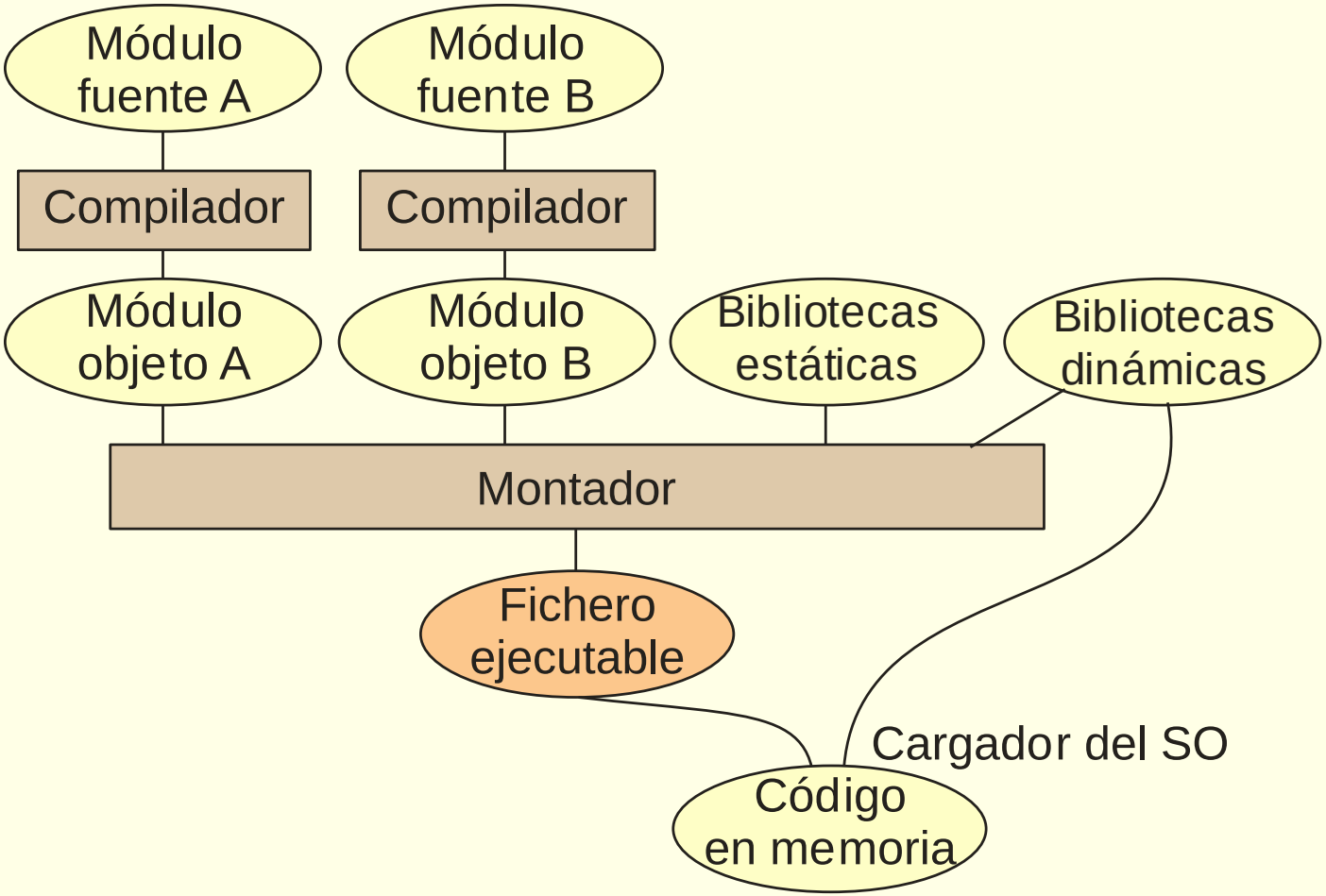
fperez@box1: ~/SII/ejemplos_mmap

```
/* Definición convencional de una matriz de nxn */
struct casilla {
        int abierta;
        int valor;
};
int n = atoi(argv[1]);
struct casilla (*tablero)[n];

/* reserva convencional en memoria dinámica versus */
tablero = malloc(n * n * sizeof(struct casilla));

/* asociar matriz a un fichero */
tablero=mmap((caddr_t) 0, tam, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);

/* en ambos casos la forma de acceso es la misma */
for (i=0; i<n; i++) {
        for (j=0; j<n; j++) {
                tablero[i][j].valor = (int)(((double)rand() * n * n) / RAND_MAX)
 + 1;
                tablero[i][j].abierta = 0;
        }
}
                                                              1,0-1    Comienzo
```

Contenido

- Aspectos generales de la gestión de memoria
- Mapa de memoria de un proceso
- Gestión de la memoria del sistema
  - Paginación
  - Memoria virtual
- Proyección de archivos
- **Bibliotecas dinámicas**

Ciclo de vida de un programa



Bibliotecas estáticas (Linux fichero extensión *.a*)

- Compilador: genera código objeto a partir de fuente
- Montador: genera ejecutable a partir de módulos objeto
  - Resolviendo referencias a símbolos entre módulos
- Biblioteca: colección de módulos objeto relacionados
- Bibliotecas de SO, de cada lenguaje, creadas por usuario...
- Usuario especifica bibliotecas requeridas en montaje
- Si programa incluye referencia a símbolo en biblioteca
  - Montador extrae objeto(s) requerido(s) de biblioteca
- Una vez extraídos objetos requeridos, montaje convencional
- Desventajas del montaje estático de bibliotecas:
  - Ejecutables grandes
  - Código de biblioteca repetido en ejecutables y memoria
  - Actualización de biblioteca implica volver a montar

Bibliotecas dinámicas (Linux fichero extensión *.so*)

- *Dynamically Linked Library (DLL)*
  - Carga y montaje en tiempo de ejecución
  - Transparente y con sobrecarga tolerable
- Montaje de programa que usa biblioteca dinámica
  - No se realiza resolución ni se incluye código de biblioteca
  - En ejecutable: se anota su uso (*ldd programa*)
- Cuando proceso hace primera referencia a símbolo en ejecución
  - Se busca en qué biblioteca está definido y
  - si todavía no está cargada, se carga y se resuelve el símbolo
- Ventajas: las desventajas de las estáticas
- Desventajas: Ejecutable no autocontenido

Bibliotecas estáticas vs. dinámicas

```
fperez@box1: ~/SII/experimentos_memoria/4_experimento
fperez@box1:~/SII/experimentos_memoria/4_experimento$ ls /usr/lib/x86_64-linux-g
nu/libpthread.*
/usr/lib/x86_64-linux-gnu/libpthread.a  /usr/lib/x86_64-linux-gnu/libpthread.so
fperez@box1:~/SII/experimentos_memoria/4_experimento$ cc -Wall -static   program
a.c  -lpthread -o programa
fperez@box1:~/SII/experimentos_memoria/4_experimento$ ldd programa
        no es un ejecutable dinámico
fperez@box1:~/SII/experimentos_memoria/4_experimento$ size programa
   text    data     bss     dec     hex filename
 884716    7364   26008  918088   e0248 programa
fperez@box1:~/SII/experimentos_memoria/4_experimento$ cc -Wall  programa.c  -lpt
hread -o programa
fperez@box1:~/SII/experimentos_memoria/4_experimento$ ldd programa
        linux-vdso.so.1 =>  (0x00007ffc99177000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f35be7f
0000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f35be426000)
        /lib64/ld-linux-x86-64.so.2 (0x000056118df93000)
fperez@box1:~/SII/experimentos_memoria/4_experimento$ size programa
   text    data     bss     dec     hex filename
   2178     616       8    2802     af2 programa
fperez@box1:~/SII/experimentos_memoria/4_experimento$ ldd /usr/lib/libreoffice/p
rogram/soffice.bin | wc -l
119
fperez@box1:~/SII/experimentos_memoria/4_experimento$
```

Carga explícita de bibliotecas dinámicas

- No se especifica biblioteca en mandato de montaje
- Programa solicita carga de biblioteca mediante servicio del sistema
  - UNIX *dlopen*
- Acceso "no" transparente a símbolos de la biblioteca
  - Programa solicita explícitamente la dirección de un símbolo
    - UNIX *dlsym*
  - A partir de ese punto, uso convencional
- Permite carga dinámica de código
  - En ejecución el proceso "aprende a hacer nuevas cosas"
  - Permite implementar componentes de tipo *plugin*

Servicios de bibliotecas dinámicas de UNIX

*void **__dlopen__*(const char *biblioteca, int flags);*

- Carga y montaje de una biblioteca dinámica
  - Devuelve descriptor de biblioteca cargada
  - *biblioteca* indica la biblioteca a cargar
  - *flags*: opciones de carga. solo es obligatorio:
    - *RTLD_LAZY*, *RTLD_NOW*: resolución símb. perezosa o inmediata
    - *lazy*: símbolos en biblioteca no se revuelven hasta primer acceso

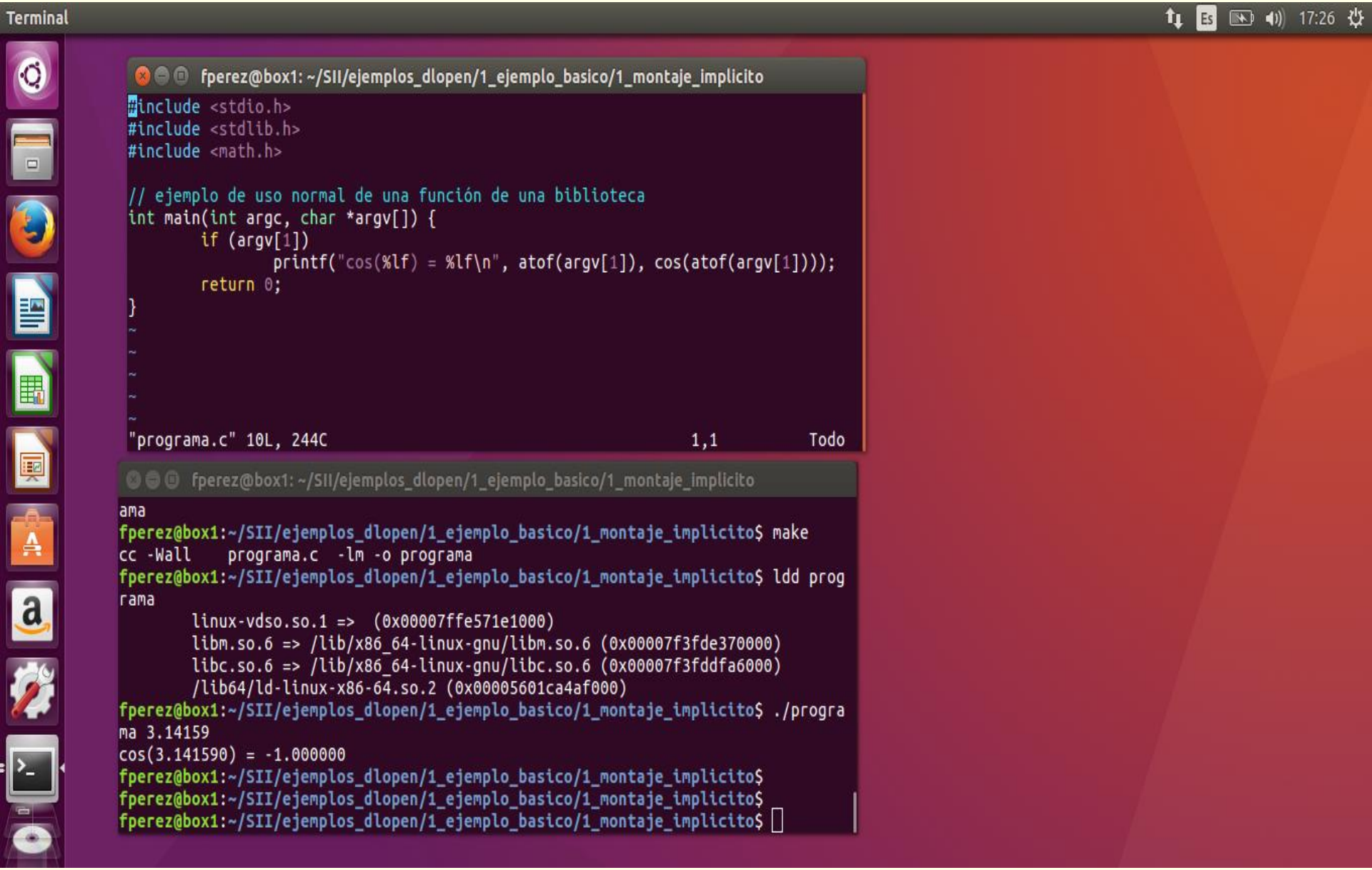
*void **__dlsym__*(void *descriptor, char *simbolo);*

- Permite acceder a símbolo exportado por biblioteca
  - Devuelve dirección de memoria de ese símbolo
  - *descriptor* de la biblioteca
    - *RTLD_NEXT* busca en la siguiente biblioteca especificada
  - *simbolo* a buscar

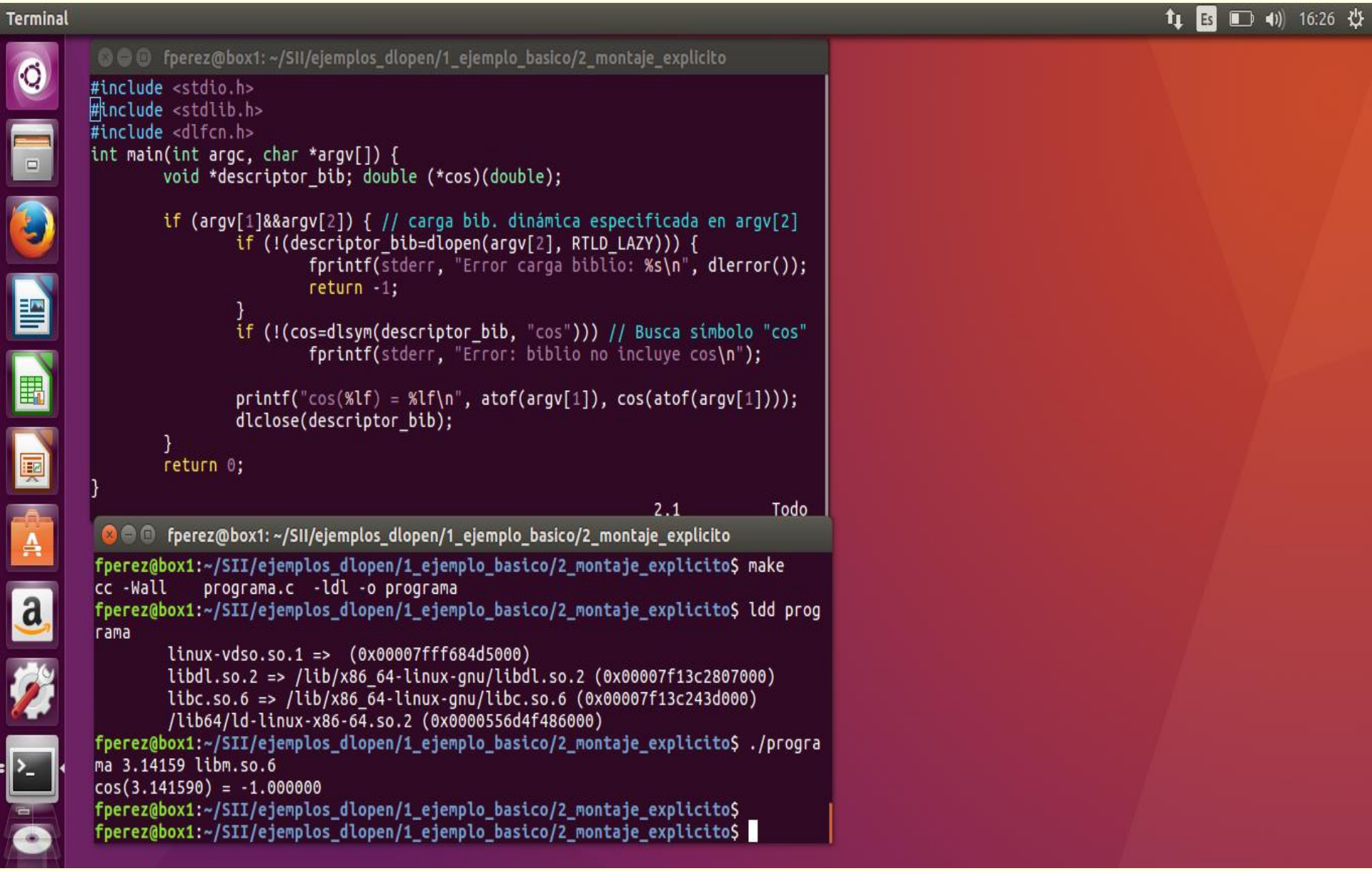
int __dlclose__*(void *descriptor);*

- Descarga biblioteca especificada por *descriptor*

Uso convencional de bibliotecas dinámicas



Uso de carga explícita de bibliotecas dinámicas



Ejemplo de uso de carga explícita de bib. dinámicas

- El ejemplo previo no parece demasiado útil…
  - ¿Para qué nos puede servir este mecanismo?
- Aplicación tipo navegador que presenta distintos tipos de objetos
- Debe aprender a manejar nuevos tipos de objetos sin re-arrancar
- Fichero de configuración especifica acción para cada tipo de objeto

  HTML → *acción1*
  TXT → *acción2*
  JPG → *acción3*
- Nuevo tipo de objeto (*XXX*)
  - Sin parar *App*, se añade nueva entrada a fichero de configuración

    XXX → *acciónX*
  - Cuando llega objeto de nuevo tipo:
    - *App* busca en f. configuración la acción asociada a ese objeto
    - Activa esa "acción" para presentarlo

Ejemplo de carga explícita: Solución “clásica”

- Un programa para presentar cada tipo de objeto
- Fichero de configuración define qué programa maneja cada objeto
- Nuevo tipo de objeto:
  - añadir en f. configuración nuevo programa asociado a ese tipo
- Cuando llega objeto de nuevo tipo:
  - *App* busca en f. configuración programa que maneja ese objeto
  - Arranca ese programa (*fork*+*exec*) para presentarlo
- Requiere crear un nuevo proceso por cada objeto a presentar
- Dificultad de compartir recursos entre *App* y programa arrancado
  - P.e. compartir la ventana donde se presenta el objeto

Ejemplo de carga explícita: Solución DLL

- Una DLL para presentar cada tipo de objeto
- DLL define una función con un nombre predefinido
  - P.e. *display* que recibe el nombre del fichero que contiene objeto
- Fichero configuración define qué DLL maneja cada objeto
- Nuevo tipo de objeto:
  - añadir en f. configuración nueva DLL asociada a ese tipo
- Cuando llega objeto de nuevo tipo:
  - *App* busca en f. configuración la DLL que maneja ese objeto
  - Carga DLL (*dlopen*) y busca dirección de función *display* (*dlsym*)
  - Llama a función, pasándole el objeto, para presentarlo
- Solución más eficiente: es el mismo proceso
- Solución más integrada: P.e. *App* y DLL comparten la ventana
- Véase ejemplo *procesa_archivos* en material de apoyo