Sistemas Operativos

Universidad Complutense de Madrid

2020-2021

Módulo 5: Gestión de memoria

Juan Carlos Sáez

Contenido

Contenido

Objetivos del sistema de gestión de memoria

El SO multiplexa recursos entre procesos

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

Objetivos deseables:

- Ofrecer a cada proceso un espacio lógico propio
- Proporcionar protección entre procesos
- Dar soporte a las regiones del proceso
- Permitir que procesos compartan memoria
- Proporcionar a los procesos mapas de memoria muy grandes
- Maximizar el grado de multiprogramación

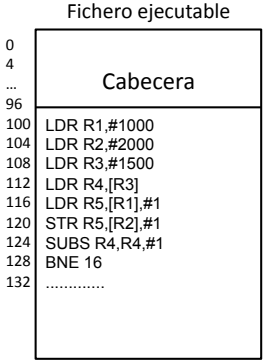
Espacios lógicos independientes

- No se conoce la posición de memoria donde un programa se ejecutará
- Código ejecutable genera *direcciones lógicas* entre 0 y N
- Ejemplo: Archivo ejecutable de un programa que copia un vector

Ejemplo

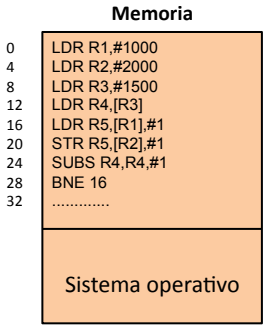
```
for (i=0; i < size; i++)
    B[i]=A[i];
```

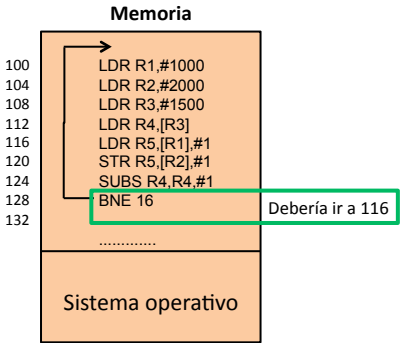
- Cabecera de 100 Bytes
- Instrucción ASM 4 Bytes
- Vector origen a partir de dirección 1000
- Vector destino a partir de dirección 2000
- Tamaño del vector en dirección 1500



Ejecución en SO monoprogramado

- Supongamos que el código del SO está cargado en las direcciones más altas de memoria principal
- Programa se carga en dirección 0
  - Si no se puede, habrá que ajustar las referencias
- Se le pasa el control al programa



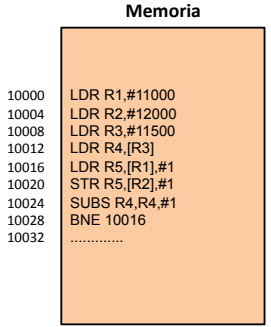


Reubicación

- Traducir direcciones lógicas a físicas
  - Dir. lógicas: direcciones de memoria generadas por el programa
  - Dir. físicas: direcciones de mem. principal asignadas al proceso
- Necesaria en SO con multiprogramación:
  - *Traducción(PID,dir_lógica)* $\rightarrow$ *dir_física*
- Reubicación crea espacio lógico independiente para proceso
  - SO debe poder acceder a espacios lógicos de los procesos
- Dos alternativas de reubicación:
  - Software
  - Hardware

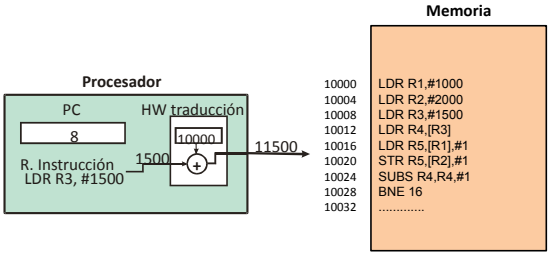
Reubicación software

- Traducción de direcciones durante carga del programa
- Programa en memoria distinto del ejecutable
- Desventajas:
  - No asegura protección
  - No permite mover programa en tiempo de ejecución



Reubicación hardware

- Hardware (MMU) encargado de traducción en tiempo de ejecución
- El programa se carga en memoria sin modificar
  - El programa/procesador genera direcciones lógicas
- SO se encarga de:
  - Almacenar por cada proceso su función de traducción
  - Especificar al hardware qué función debe aplicar para cada proceso



Protección

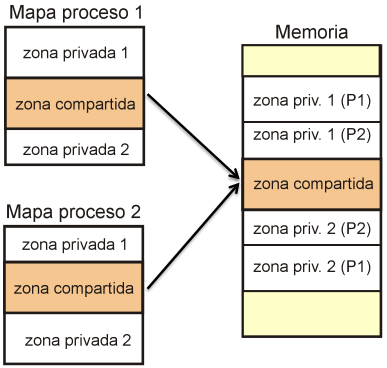
- Monoprogramación: Proteger al SO
- Multiprogramación: Además procesos entre sí

Requisitos mínimos:

1. La traducción debe crear espacios de direcciones disjuntos
2. Es necesario validar todas las direcciones que genera el programa
   - La detección se realiza por HW → generación de excepción
   - El tratamiento lo hace el SO

Compartición de memoria

- Direcciones lógicas de 2 o más procesos se corresponden con misma dirección física
- Bajo control del SO
- Beneficios:
  - Mecanismo de comunicación entre procesos muy rápido
  - Procesos ejecutando mismo programa comparten su código
    - El SO realiza esta optimización de forma transparente si la estrategia de gestión de memoria lo permite
- Requiere asignación no contigua



Soporte de regiones

- Mapa de proceso no homogéneo
  - Conjunto de regiones con distintas características
    - Código, pila, datos, ...
  - Ejemplo: Región de código no modificable
- Mapa de proceso dinámico
  - Regiones cambian de tamaño (p.ej. pila)
  - Se crean y destruyen regiones
  - Existen zonas sin asignar (huecos)
- SO debe guardar una tabla de regiones para cada proceso
- Gestor de memoria debe dar soporte a estas características:
  1. Detectar accesos no permitidos a una región
  2. Detectar accesos a huecos
  3. Evitar reservar espacio para huecos

Utilización de la memoria

Aprovechamiento de memoria óptimo pero irrealizable

Memoria

0	Dirección 50 del proceso 4
1	Dirección 10 del proceso 6
2	Dirección 95 del proceso 7
3	Dirección 56 del proceso 8
4	Dirección 0 del proceso 12
5	Dirección 5 del proceso 20
6	Dirección 0 del proceso 1
	.....
	.....
N-1	Dirección 88 del proceso 9
N	Dirección 51 del proceso 4

- Reparto de memoria maximizando grado de multiprogramación
- Desperdiciar algo de espacio en memoria es inevitable
- Uso de unidades de asignación mínima de memoria (p.ej. páginas)
  - Uso de memoria virtual para aumentar grado de multiprogramación del SO
    - Mapa de memoria dividido en páginas
    - Memoria principal dividida en marcos de página
- Se "desperdicia" memoria debido a:
  - "Restos" inutilizables (fragmentación)
  - Tablas requeridas por gestor de memoria

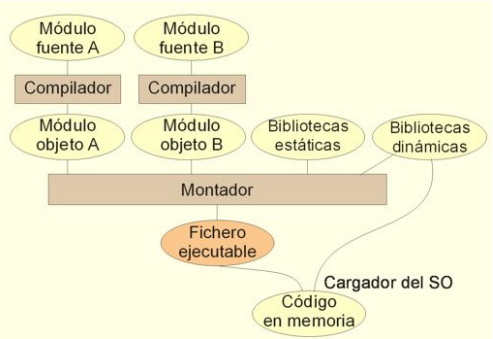
Mapas de memoria muy grandes para procesos

- Procesos necesitan cada vez mapas más grandes
  - Aplicaciones más avanzadas o novedosas
- Resuelto gracias al uso de memoria virtual
- Antes se usaban *overlays*:
  - Programa dividido en fases que se ejecutan sucesivamente
  - En cada momento sólo hay una fase residente en memoria
  - Cada fase realiza su labor y carga la siguiente
  - No es transparente ya que toda la labor la realizaba el programador

Contenido

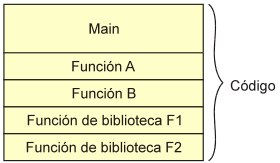
Construcción de un ejecutable

- Aplicación: conjunto de módulos en lenguaje de alto nivel
- Procesado en dos fases: Compilación y Montaje/Enlazado
- Compilación:
  - Genera módulo objeto
  - Resuelve referencias dentro cada módulo fuente
- Montaje (o enlazado):
  - Resuelve referencias entre módulos objeto
  - Resuelve referencias a símbolos de bibliotecas
  - Genera ejecutable incluyendo bibliotecas



Bibliotecas estáticas

- Colección de módulos objeto relacionados que exportan un conjunto de símbolos globales (funciones, variables, ...)
- **Enlazado estático**: el *linker* enlaza los módulos objeto del programa y de las bibliotecas creando un ejecutable autocontenido



Desventajas de las bibliotecas estáticas

- Ejecutables grandes
- Código de biblioteca replicado en muchos ejecutables y memoria
- Actualización de biblioteca implica volver a generar el ejecutable

Bibliotecas dinámicas

- Enlazado y carga en tiempo de ejecución
  - El ejecutable contiene:
    1. Nombre de la biblioteca
    2. Rutina de carga y montaje en tiempo de ejecución
  - La rutina de carga se invoca en la 1ª referencia a un símbolo de la bib.

Ventajas

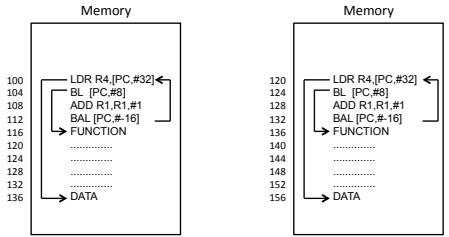
- Menor tamaño ejecutables
  - Código de rutinas de biblioteca sólo en archivo de biblioteca
- Procesos pueden compartir código de biblioteca dinámica
- Actualización automática de bibliotecas (Misma ABI)

Desventajas

- Ejecutable no es autocontenido
- Mayor tiempo de ejecución debido a carga y montaje
  - Tolerable, compensa el resto de las ventajas

Compartición de bibliotecas dinámicas

- Biblioteca dinámica contiene referencias internas (ej., `f()` invoca `g()`)
- ¿Cómo referenciar variables/funciones privadas dentro de la biblioteca?
  1. Reservar rango fijo de direcciones lógicas a la biblioteca
     - No flexible
     - Exige asignar rangos de direcciones disjuntos a distintas bibliotecas
  2. Permitir que el código de la biblioteca se "cargue" en cualquier posición del mapa de memoria de cualquier proceso
     - Simplifica carga de múltiples bibliotecas en mapa de memoria
     - Exige generar código independiente de posición (PIC)



Recuerda: Comandos útiles

Comandos

- `/sbin/ldconfig -p`
  - Muestra la lista de todas las librerías cargadas
- `ldd`
  - Permite ver las librerías dinámicas con las que hemos enlazado

Bibliotecas dinámicas: Ejemplo `ldd`

greetings.c

```c
#include <stdio.h>

int main (void) {
  char name[100];
  printf("Enter your name: ");

  if (scanf("%s", name) != 1) {
     printf("Error or EOF \n");
     return 1;
  } else {
     printf("Hi  %s!\n", name);
     return 0;
  }
}
```

Terminal

```
osuser@debian:~$ gcc -g greetings.c -o greetings
osuser@debian:~$ ldd greetings
    linux-vdso.so.1 =>  (0x00007fff39a35000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f80e3b95000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f80e3f3d000)
osuser@debian:~$
```

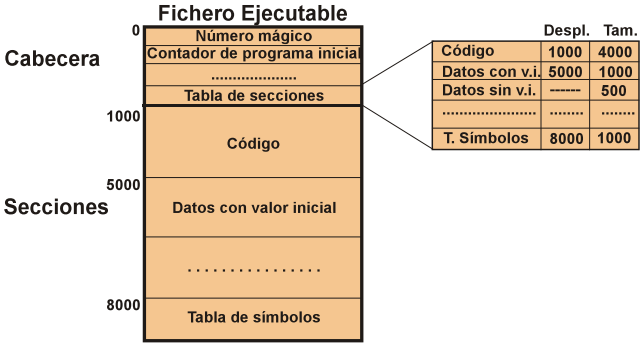
Formato del ejecutable

- En UNIX *Executable and Linkable Format* (ELF)

Estructura simplificada

1. Cabecera
   - Número mágico que identifica a ejecutable (0x7f+'ELF')
   - Punto de entrada del programa
   - Tabla de secciones
2. Secciones
   - Código (`.text`)
   - Datos inicializados (`.data`)
   - Datos no inicializados (`.bss`)
     - Sólo se especifica el tamaño
   - Tabla de símbolos de depuración (`.debug`)
   - Tabla de bibliotecas dinámicas (`.dynamic`, `.dynstr`, `.dynsym`)

Formato del ejecutable



Variables globales vs. locales

Variables globales (con sección)

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

Variables locales y parámetros (sin sección)

- Dinámicas
  - Se crean al invocar función
  - Se destruyen al retornar
- La dirección se calcula en tiempo de ejecución
- Recursividad: varias instancias de una variable

Variables globales vs. locales

Ejemplo

```
int x=8;      /* Variable global con valor inicial */
int y;        /* Variable global sin valor inicial */
int j=5;      /* Inicialización y reserva de espacio para var. global */

int f(int t) {   /* Parámetro */
 int z;          /* Variable local */
  ...
}

int main() {
  ...
  f(4);
  ...
}
```

Recuerda: Comandos útiles

Comandos

- `nm` / `objdump`
  - Permiten visualizar partes de un ejecutable
  - Opciones típicas de `objdump`: -S, -t, -f, -h

Ejemplo nm

```
osuser@debian:~$ nm -Dsv greetings
                 w __gmon_start__
                 U __isoc99_scanf
                 U __libc_start_main
                 U printf
                 U puts
osuser@debian:~$
```

Recuerda: objdump

Ejemplo objdump

```
osuser@debian:~$ objdump -x greetings
greetings:     file format elf64-x86-64
greetings
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00000000004004c0
...
Sections:
Idx Name Size VMA LMA
File off Algn
..
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
...
 13 .text         0000020c  00000000004004c0  00000000004004c0  000004c0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
...
 15 .rodata       0000002f  00000000004006d8  00000000004006d8  000006d8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
...
 21 .dynamic      000001e0  00000000006007f8  00000000006007f8  000007f8  2**3
                  CONTENTS, ALLOC, LOAD, DATA
...
 24 .data         00000010  0000000000600a18  0000000000600a18  00000a18  2**3
                  CONTENTS, ALLOC, LOAD, DATA
...
 25 .bss          00000008  0000000000600a28  0000000000600a28  00000a28  2**2
                  ALLOC
```

Mapa de memoria de un proceso (MMP)

- EL MMP es un **conjunto de regiones** de memoria donde se almacena todo lo necesario para que un programa pueda ejecutarse
  - También conocido como *espacio de direcciones del proceso*
- Cada región es una zona contigua tratada como unidad
- Cada región posee datos + metainformación (que mantiene el SO)
  - Dirección de comienzo y tamaño inicial
  - Soporte: dónde se almacena su contenido inicial si lo tuviese (ej. fichero ejecutable)
  - Protección: RWX
  - Uso compartido o privado
  - Tamaño fijo o variable (modo de crecimiento $\uparrow$ / $\downarrow$)

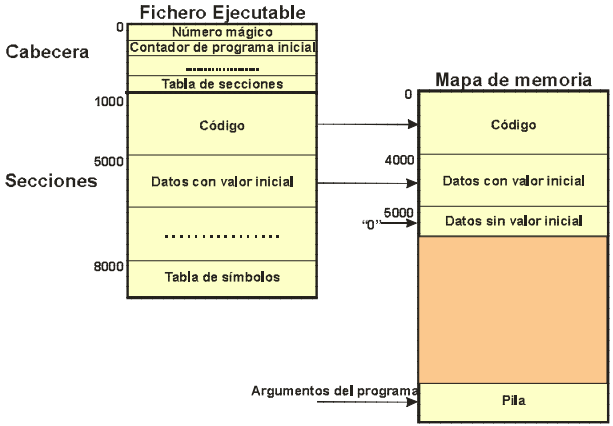
Crear mapa de memoria desde ejecutable

- Ejecución de un programa: Crea mapa a partir de ejecutable
  - Algunas secciones del ejecutable → Regiones de mapa inicial

Regiones básicas

1. **Código**
   - Compartida, RX, T. Fijo, Soporte en Ejecutable
2. **Datos con valor inicial**
   - Privada, RW, T. Fijo, Soporte en Ejecutable
3. **Datos sin valor inicial**
   - Privada, RW, T. Fijo, Sin Soporte (rellenar 0)
4. **Pila**
   - Privada, RW, T. Variable, Sin Soporte (rellenar 0)
   - Crece hacia direcciones más bajas
   - Pila inicial: argumentos del programa (`execvp()`)

Crear mapa de memoria desde ejecutable



Otras regiones del mapa de memoria

- Durante ejecución de proceso se crean nuevas regiones
  - Mapa de memoria tiene un carácter dinámico

Otras regiones

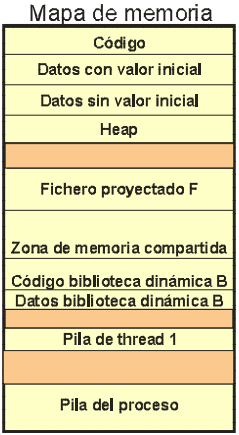
- **Región de *Heap***
  - Soporte de memoria dinámica (`malloc()` en C)
  - Privada, RW, T. Variable, Sin Soporte (rellenar 0's)
  - Crece hacia direcciones más altas
- **Regiones de bibliotecas dinámicas**
  - Se crean regiones asociadas al código y datos de la biblioteca
- **Pilas de *threads***
  - Cada pila de *thread* corresponde con una región
  - Mismas características que pila del proceso

Otras regiones del mapa de memoria

Otras regiones

- **Memoria compartida**
  - Región asociada a la zona de memoria compartida
  - Compartida, T. Variable, Sin Soporte (rellenar 0's)
  - Protección especificada en proyección
- **Fichero proyectado en memoria** (`mmap()`)
  - Región asociada al archivo proyectado
  - T. Variable, Soporte en fichero
  - Protección y carácter compartido o privado especificado en proyección

Características de las regiones



Región	Soporte	Protección	Comp/Priv	Tamaño
Código	Fichero	RX	Compartida	Fijo
Datos con v.i.	Fichero	RW	Privada	Fijo
Datos sin v.i.	Sin soporte	RW	Privada	Fijo
Pilas	Sin soporte	RW	Privada	Variable
Heap	Sin soporte	RW	Privada	Variable
Fich. proyect.	Fichero	Espec. usuario	Comp./Priv	Variable
Memoria comp.	Sin soporte	Espec. usuario	Compartido	Variable

Sistema de ficheros /proc

Terminal 1

```
osuser@debian:~$ ./greetings
Enter your name:
```

Terminal 2

```
osuser@debian:~$ ps -a | grep greetings
 7644 pts/5    00:00:00 greetings
osuser@debian:~$ cat /proc/7644/maps
00400000-00401000 r-xp 00000000 00:11 324                /home/osuser/greetings
00600000-00601000 rw-p 00000000 00:11 324                /home/osuser/greetings
7f34830e2000-7f3483263000 r-xp 00000000 08:01 675745     /lib/x86_64-linux-gnu/libc-2.13.so
7f3483263000-7f3483463000 ---p 00181000 08:01 675745     /lib/x86_64-linux-gnu/libc-2.13.so
7f3483463000-7f3483467000 r--p 00181000 08:01 675745     /lib/x86_64-linux-gnu/libc-2.13.so
7f3483467000-7f3483468000 rw-p 00185000 08:01 675745     /lib/x86_64-linux-gnu/libc-2.13.so
7f3483468000-7f348346d000 rw-p 00000000 00:00 0
7f348346d000-7f348348d000 r-xp 00000000 08:01 675742     /lib/x86_64-linux-gnu/ld-2.13.so
7f348366d000-7f3483670000 rw-p 00000000 00:00 0
7f3483688000-7f348368c000 rw-p 00000000 00:00 0
7f348368c000-7f348368d000 r--p 0001f000 08:01 675742     /lib/x86_64-linux-gnu/ld-2.13.so
7f348368d000-7f348368e000 rw-p 00020000 08:01 675742     /lib/x86_64-linux-gnu/ld-2.13.so
7f348368e000-7f348368f000 rw-p 00000000 00:00 0
7fff4de1b000-7fff4de3c000 rw-p 00000000 00:00 0          [stack]
7fff4de8a000-7fff4de8b000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0  [vsyscall]
```

Operaciones sobre regiones

- El SO puede alterar el mapa de memoria de un proceso durante su ciclo de vida

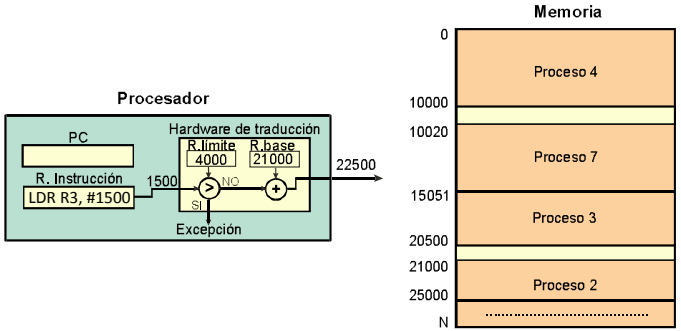
Operaciones posibles

- **Crear región**
  - Implícitamente al crear mapa inicial o por solicitud del programa en t. de ejecución (p.ej. proyectar un archivo con `mmap()`)
- **Eliminar región**
  - Implícitamente al terminar el proceso o por solicitud del programa en t. de ejecución (p.ej. `munmap()`)
- **Redimensionar región**
  - Implícitamente para la pila o por solicitud del programa para el *heap*
- **Duplicar región**
  - Operación requerida por el servicio fork de POSIX

Contenido

Asignación contigua (I)

- Mapa de memoria de cada proceso en zona contigua de memoria principal
- Hardware (MMU) encargado de traducción en tiempo de ejecución
- Hardware requerido: Regs. valla (R. base y R. límite)
  - Sólo accesibles en modo privilegiado (modo *kernel*)



Asignación contigua (II)

SO mantiene información sobre:

1. El valor de regs. valla de cada proceso en su BCP
   - En cambio de contexto SO carga en regs. valor adecuado
2. Estado de ocupación de la memoria
   - Estructuras de datos que identifiquen huecos y zonas asignadas en memoria principal
   - Tabla de regiones de cada proceso

Este esquema presenta fragmentación externa

- Se generan pequeños fragmentos libres entre zonas asignadas
- Posible solución: compactación $\rightarrow$ proceso costoso

Política de asignación de espacio

- *¿Qué hueco usar para almacenar el mapa de memoria de un nuevo proceso?*

Posibles políticas

- **Primer ajuste:** Asignar el primer hueco encontrado
- **Mejor ajuste:** Asignar el menor hueco posible
  - Lista de huecos ordenada ascendentemente por tamaño
- **Peor ajuste:** Asignar el mayor hueco con tamaño suficiente
  - Lista de huecos ordenada descendentemente por tamaño

- Primer ajuste es más eficiente y proporciona buen aprovechamiento de la memoria

Operaciones sobre regiones con a. contigua

- Al crear proceso se le asigna zona de memoria de tamaño fijo
  - Suficiente para albergar regiones iniciales (código, datos)
  - Con huecos para permitir cambios de tamaño y añadir nuevas regiones (p.ej. bibliotecas dinámicas o pilas de threads)
  - Difícil asignación adecuada
    - Si grande se desperdicia espacio, si pequeño se puede agotar
- Operaciones:
  - Crear/liberar/cambiar tamaño usan la tabla de regiones para gestionar la zona asignada al proceso
  - Duplicar región requiere crear región nueva y copiar contenido
    - Limitaciones del hardware impiden compartir memoria

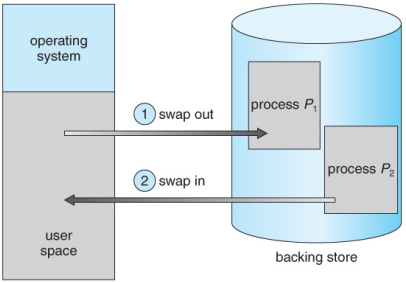
Valoración del esquema de a. contigua

¿Proporciona las funciones deseables en un gestor de memoria?

- Espacios independientes para procesos:
  - mediante registros valla
- Protección:
  - mediante registros valla
  - registros valla accesibles sólo en modo privilegiado (SO)
- Compartir memoria:
  - no es posible
- Soporte de regiones:
  - No es posible asignar permisos de acceso (RWX) a cada región
  - No es posible detectar accesos erróneos (huecos dentro del mapa de memoria) o desbordamiento de pila
- Maximizar utilización de la memoria y soporte de mapas grandes
  - mal aprovechamiento de memoria por fragmentación externa
  - no permite mapas grandes

Intercambio (I)

- *¿Qué hacer si no caben todos los procesos en memoria principal?* → swapping
- Usado en primeras versiones de UNIX
- Zona de Intercambio o *Swap*: partición (fichero) de disco que almacena imágenes de procesos
  - Imagen de memoria de proc. en swap → proc. suspendido
  - Al usar swapping puro, un proceso no puede ejecutarse si su mapa de memoria no está cargado completamente en memoria



Intercambio (II)

Swap out

- Cuando no caben en memoria los procesos activos, se expulsa un proceso de memoria copiando imagen a *swap*
- Diversos criterios de selección del proceso a expulsar
  - P.ej. Dependiendo de prioridad del proceso
  - Preferiblemente un proceso bloqueado
  - No expulsar si está activo DMA sobre mapa del proceso
- No es necesario copiar todo el mapa:
  - El código está en el fichero
  - Los huecos no contienen información del proceso

Intercambio (III)

Swap in

- Cuando haya espacio en memoria principal, se lee el proceso a memoria copiando la imagen desde *swap*
- También cuando un proceso lleva un cierto tiempo expulsado
  - En este caso antes de *swap in*, hay *swap out* de otro

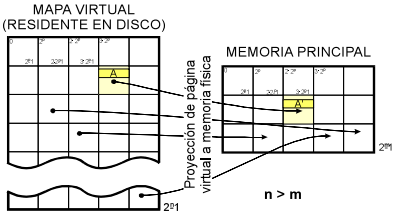
Intercambio (IV)

- Asignación de espacio en el dispositivo de swap
  - *Con preasignación*: se asigna espacio al crear el proceso
  - *Sin preasignación*: se asigna espacio al expulsarlo
- Los procesos activos han de residir completamente en MP
  - Grado de multiprogramación depende del tamaño de proc. y MP
- Solución general → Uso de esquemas de memoria virtual
  - Aunque se sigue usando integrado con esa técnica

Contenido

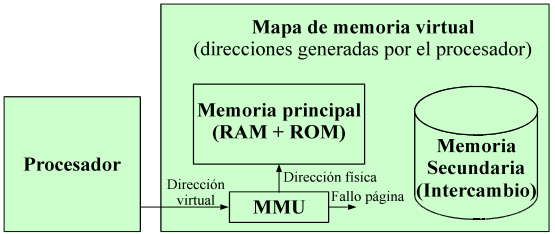
Fundamentos de la memoria virtual (I)

- Los procesos (y el SO) generan direcciones virtuales al ejecutarse
  - Dir. virtuales de un proceso pertenecen al su *mapa de memoria virtual* (imagen de memoria)
- El mapa de memoria virtual de un proceso se divide en *páginas*
  - Página: unidad mínima de asignación de MV (p.ej., 4KB)
- Parte del mapa de memoria virtual de un proceso está en MP (memoria principal) y parte en disco (*swap* o memoria secundaria)
- El SO se encarga de que estén en memoria principal las páginas "necesarias" del mapa de memoria virtual de cada proceso



Fundamentos de la memoria virtual (II)

- Cada proceso utiliza y genera direcciones virtuales
  - Cada dirección virtual → página asociada + desplazamiento (offset)
- La MMU (*Memory Management Unit*) traduce las direcciones virtuales en físicas
  - Si la página referenciada está en MP → Se realiza un acceso a MP
  - Si la página referenciada no está en MP → La MMU genera excepción de *fallo de página*
    - El SO trata el fallo de página transfiriendo la página desde disco (swap) a MP

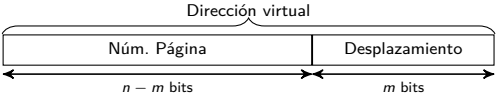


Fundamentos de la memoria virtual (III)

- Transferencia de páginas entre ambos niveles (*swap* y MP)
  - De M. secundaria a principal: bajo demanda
  - De M. principal a secundaria: por expulsión
- La memoria virtual es eficaz porque los procesos exhiben localidad de referencias
  - Localidad espacial (p.ej., página perteneciente a la región de código.)
  - Localidad temporal (p.ej., múltiples referencias a una misma variable)
  - En la práctica, los procesos sólo usan parte de su mapa de memoria en un intervalo de tiempo
    - Para maximizar el rendimiento, el SO intenta que parte usada de la memoria de un proceso (*conjunto de trabajo*) resida en MP (*conjunto residente*)
- Beneficios:
  - Aumenta el grado de multiprogramación
  - Permite ejecución de programas que no quepan en MP

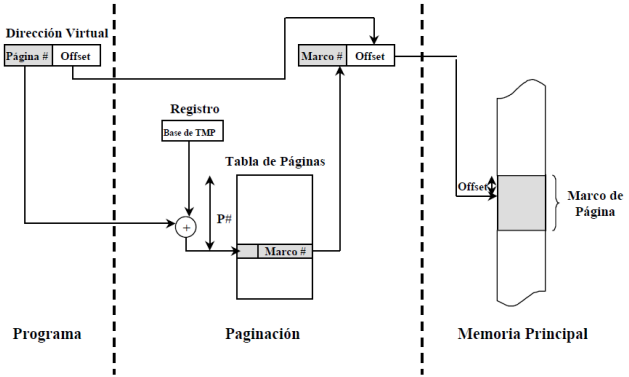
Memoria virtual: aspectos hardware

- Asignación no contigua
- Unidad de asignación: página (tamaño en bytes potencia de 2)
  - Mapa de memoria del proceso dividido en páginas
  - Memoria principal dividida en *marcos de página* (tam. marco=tam. pág)
- Dirección lógica (ó Dir. Virtual en este caso): nº página + desplazamiento
  - Direcciones virtuales de n bits y $T_{\text{página}} = 2^m$ bytes



- Tabla de páginas (TP):
  - Relaciona cada página con el marco que la contiene
  - MMU usa TP para traducir direcciones virtuales a físicas
  - Se almacena en MP

Ejemplo de traducción con tablas de páginas

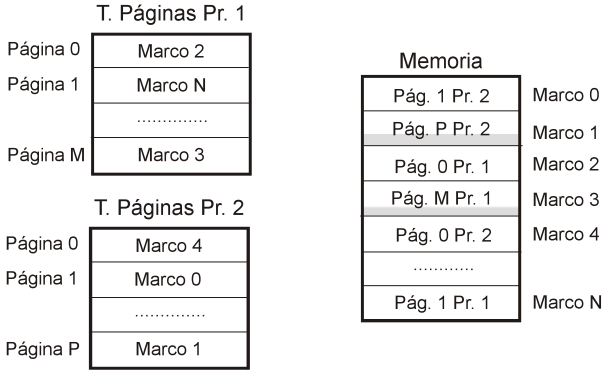


Contenido de entrada de TP

- Número de marco asociado
- Bit de página válida/inválida
  - Usado en mem. virtual para indicar si página presente
  - Si página no presente → Excepción de fallo de página
- Información de protección: WX (*Write/eXecute*)
  - Si operación no permitida → Excepción
- Información de nivel de privilegio: Usuario/Kernel
- Bit de página accedida (*Ref*)
  - MMU lo activa cuando se accede a esta página desde que la página se trajo a MP
- Bit de página modificada (*Mod*)
  - MMU lo activa cuando se escribe en esta página
- Bit de desactivación de cache

Fragmentación interna en paginación

- Tendremos *fragmentación interna* si Mem. asignada > Mem. requerida
  - Puede desperdiciarse parte de un marco asignado



Tamaño de página

Condicionado por diversos factores contrapuestos:

- Potencia de 2 y múltiplo del tamaño del bloque de E/S
- Mejor pequeño por:
  - Menor fragmentación interna
  - Se ajusta mejor al conjunto de trabajo
- Mejor grande por:
  - Tablas más pequeñas
  - Mejor rendimiento en las transferencias MP disp. E/S (disco)
- Compromiso: entre 2KB y 16KB
  - En SSOO tipo UNIX actuales consultar con: `getconf PAGESIZE`

Gestión del SO

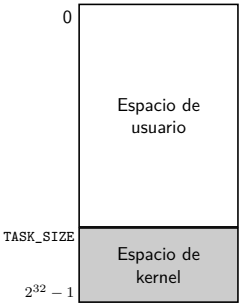
SO mantiene:

1. TP por cada proceso
   - En cambio de contexto notifica a MMU cuál debe usar (RIED)
   - RIED: Registro de Identificación de Espacio de Direccionamiento
     - Sólo el SO puede modificar este registro
     - Ejemplo: En x86, RIED es CR3 (Control register #3)
2. Una única TP para él mismo
   - En general, estas páginas sólo son accesibles cuando se ejecuta en modo kernel
3. Tabla de marcos de página
   - Estado de cada marco (libre u ocupado, ...)
4. Tabla de regiones por cada proceso

Páginas y mapa de memoria

- El SO gestiona dos tipos de página:
  - Página de usuario: almacena código o datos de un proceso
  - Página de sistema: almacena código o datos (variables y estructuras de datos) del SO
- Por simplicidad, el SO divide el mapa de memoria (virtual) del proceso en *espacio de kernel* (páginas de sistema) y *espacio de usuario* (páginas de usuario)

- Cuando el proceso ejecuta instrucciones en modo usuario solo puede generar direcciones (referenciar páginas) del espacio de usuario
  - En general, la MMU impide que el proceso acceda a direcciones del esp. de kernel
- Cuando el proceso invoca una llamada al sistema (paso a modo kernel) el SO accede a ambos espacios



Páginas y mapa de memoria

Características de esta organización

- Cada proceso tiene su propia tabla de páginas (TP) pero ...
  - Parte del espacio de kernel similar para todos los procesos
  - Hilos de un mismo proceso comparten TP
  - Tabla de páginas común para proceso de usuario en ejecución y kernel
    - El proceso de usuario no puede acceder a páginas del kernel
- Simplifica implementación de llamadas al sistema y gestión de excepciones
  - No exige cambiar tabla de páginas al entrar y salir de modo kernel
  - Fácil traducción de direcciones de parámetros tipo puntero en *syscalls*

Valoración de la paginación

¿Proporciona las funciones deseables en un gestor de memoria?

- Espacios independientes para procesos:
  - Mediante TP
- Protección:
  - Mediante TP
- Compartir memoria:
  - Entradas de las TPs de dos o más procesos corresponden con mismo marco (Varias dir. virtuales $\rightarrow$ 1 única dir. física)
- Soporte de regiones:
  - Bits de protección
  - Bit de validez: no se reserva espacio para huecos
- Maximizar utilización de la memoria y soporte de mapas grandes
  - permite mapas de memoria virtual $>>$ cantidad de memoria física
- *Problema:* Mucho mayor gasto en tablas del SO que con asignación contigua
  - Es el precio de mucha mayor funcionalidad

Problemática de las TPs

Eficiencia:

- Cada acceso lógico requiere dos accesos a memoria principal (uno detrás del otro):
  - A la tabla de páginas + al propio dato o instrucción
- Solución: Cache de traducciones (TLB)

Gasto de almacenamiento:

- Tablas muy grandes
  - Ejemplo: páginas 4KB, dir. virtuales de 32 bits y 4 bytes por entrada
    - Tamaño TP: $2^{20} * 4 = 4\text{MB/proceso}$
- Solución (impuesta por HW):
  - Tablas multinivel
  - Tablas invertidas

Tratamiento del fallo de página

Tratamiento de excepción

- El HW almacena la dirección de fallo en un registro
- Si dirección inválida → Aborta proceso o le manda señal
- Consulta T. marcos, si no hay ningún marco libre
  - Selección de víctima: pág P marco M
    - Marca P como inválida
  - Si P modificada (bit Mod de P activo)
    - Inicia escritura P en mem. secundaria
- Hay marco libre (se ha liberado o lo había previamente):
  - Inicia lectura de página en marco M
  - Marca entrada de página válida referenciando a M
  - Pone M como ocupado en T. marcos (si no lo estaba)

- Fallo de página puede implicar 2 accesos a disco en el caso peor

Políticas de administración de MV

Política de reemplazo:

- ¿Qué página reemplazar si fallo y no hay marco libre?
- Reemplazo local
  - Sólo puede usarse para reemplazo un marco asignado al proceso que causa fallo
- Reemplazo global
  - Puede usarse para reemplazo cualquier marco

Política de asignación de espacio a los procesos:

- ¿Cómo se reparten los marcos entre los procesos?
  - Asignación fija o dinámica

Algoritmos de reemplazo

- Objetivo: Minimizar la tasa de fallos de página
- Cada algoritmo descrito tiene versión local y global:
  - local: criterio se aplica a las páginas residentes del proceso
  - global: criterio se aplica a todas las páginas residentes
- Algoritmos presentados
  - Óptimo
  - FIFO
  - Reloj (o segunda oportunidad)
  - LRU
- Uso de técnicas de *buffering* de páginas

Algoritmo óptimo

- Criterio: se conoce la página residente que tardará más ser accedida → Irrealizable
- Interés para estudios analíticos comparativos
- Ejemplo:
  - Memoria física de 4 marcos de página
  - Referencias *a b g a d e a b a d e g d e*

Ref	**a**	**b**	**g**	**a**	**d**	**e**	**a**	**b**	**a**	**d**	**e**	**g**	**d**	**e**
0	a	a	a	a	a	a	a	a	a	a	a	g	g	g
1		b	b	b	b	b	b	b	b	b	b	b	b	b
2			g	g	g	e	e	e	e	e	e	e	e	e
3					d	d	d	d	d	d	d	d	d	d

6 fallos

Algoritmo FIFO

- Criterio: página que lleva más tiempo residente
- Implementación sencilla:
  - páginas residentes en orden FIFO → se expulsa la primera
  - no requiere el bit de página accedida (Ref)
- No es una buena estrategia:
  - Página que lleva mucho tiempo residente puede seguir accediéndose frecuentemente
  - Su criterio no se basa en el uso de la página
- Ejemplo:
  - Memoria física de 4 marcos de página
  - Referencias *a b g a d e a b a d e g d e*

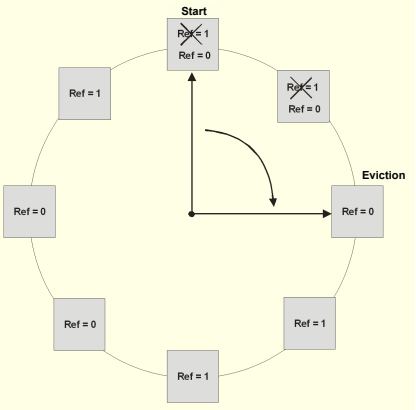
Ref	**a**	**b**	**g**	**a**	**d**	**e**	**a**	**b**	**a**	**d**	**e**	**g**	**d**	**e**
0	a_1	a	a	a	a	e_6	e	e	e	e	e	e	d_{13}	d
1		b_2	b	b	b	b	a_7	a	a	a	a	a	a	e_{14}
2			g_3	g	g	g	g	b_8	b	b	b	b	b	b
3					d_5	d	d	d	d	d	d	g_{12}	g	g

10 fallos

Algoritmo del reloj (o 2ª oportunidad)

- FIFO + uso de bit de referencia Ref (de página accedida)
- Criterio:
  - Si página elegida por FIFO no tiene activo Ref
    - Es la página expulsada
  - Si lo tiene activo (2ª oportunidad)
    - Se desactiva Ref
    - Se pone página al final de FIFO
    - Se aplica criterio a la siguiente página
- Se puede implementar orden FIFO como lista circular con una referencia a la primera página de la lista:
  - Se visualiza como un reloj donde la referencia a la primera página es la aguja del reloj

Algoritmo del reloj (o 2ª oportunidad)



Algoritmo del reloj (o 2ª oportunidad)

- Ejemplo:
  - Memoria física de 4 marcos de página
  - Referencias: *a b g a d e a b a d e g d e*

Ref	a	b	g	a	d	e		a	b	a	d	e	g	d	e
0	↓a_0	↓a_0	↓a_0	↓a_1	↓a_1	a_0	a_0	a_1	a_1	a_1	a_1	a_1	a_0	a_0	a_0
1		b_0	b_0	b_0	b_0	↓b_0	e_0	e_0	e_0	e_0	e_0	e_1	e_0	e_0	e_1
2			g_0	g_0	g_0	g_0	↓g_0	↓g_0	b_0	b_0	b_0	b_0	g_0	g_0	g_0
3					d_0	d_0	d_0	d_0	↓d_0	↓d_0	↓d_1	↓d_1	↓d_0	↓d_1	↓d_1

7 fallos

Algoritmo LRU

- Criterio: página residente menos recientemente usada
- Por proximidad de referencias:
  - Pasado reciente condiciona futuro próximo
- Difícil implementación estricta (hay aproximaciones):
  - Precisaría una MMU específica
- Posible implementación con HW específico:
  - La MMU mantiene un contador que se incrementa cada cierto tiempo
  - En entrada de TP hay una marca de tiempo (*timestamp*)
  - En cada acceso a memoria MMU copia contador del sistema a entrada referenciada
  - Reemplazo: Página con *timestamp* más bajo

Algoritmo LRU

- Ejemplo:
  - Memoria física de 4 marcos de página
  - Referencias: *a b g a d e a b a d e g d e*

Ref	**a**	**b**	**g**	**a**	**d**	**e**	**a**	**b**	**a**	**d**	**e**	**g**	**d**	**e**
0	a_1	a	a	a_4	a	a	a_7	a	a_9	a	a	a	a	a
1		b_2	b	b	b	e_6	e	e	e	e	e_{11}	e	e	e_{14}
2			g_3	g	g	g	g	b_8	b	b	b	g_{12}	g	g
3					d_5	d	d	d	d	d_{10}	d	d	d_{13}	d

7 fallos

Buffering de páginas

- Caso peor en tratamiento de fallo de página:
  - 2 accesos a disco
- Alternativa: mantener una reserva de marcos libres
  - Fallo de página: siempre usa marco libre (no reemplazo)
- Si número de marcos libres $<$ umbral
  - "Demonio de paginación" aplica repetidamente el algoritmo de reemplazo:
    - páginas no modificadas pasan a lista de marcos libres
    - 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 E/S)
  - puede arreglar el comportamiento de algoritmos "malos"

Retención de páginas en memoria

- Páginas marcadas como no reemplazables
- Se aplica a páginas del propio SO
  - SO con páginas fijas en memoria es más sencillo
- También se aplica mientras se hace DMA sobre una página
- Algunos sistemas ofrecen a aplicaciones un servicio para fijar en memoria una o más páginas de su mapa
  - Adecuado para procesos de tiempo real
  - Puede afectar al rendimiento del sistema
  - En POSIX servicio `mlock()`

Estrategia de asignación fija

- Número de marcos asignados al proceso (conjunto residente) es constante
- Puede depender de características del proceso:
  - tamaño, prioridad,...
- No se adapta a las distintas fases del programa
- Comportamiento relativamente predecible
- Sólo tiene sentido usar reemplazo local
- Arquitectura impone número mínimo:
  - Por ejemplo: instrucción `MOVE /DIR1, /DIR2`
    - Requiere un mínimo de 3 marcos en memoria:
    - Instrucción y dos operandos deben estar residentes para ejecutar la instrucción

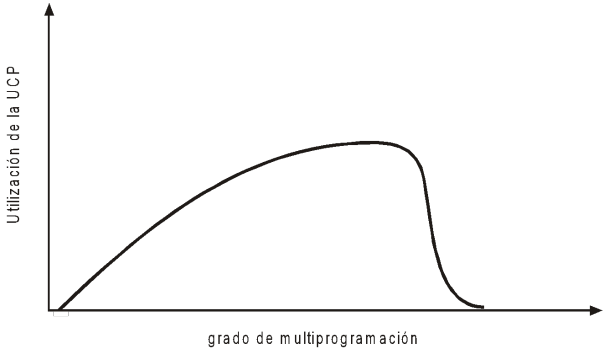
Estrategia de asignación dinámica

- Número de marcos varía dependiendo de comportamiento del proceso (y posiblemente de los otros procesos)
- Asignación dinámica + reemplazo local
  - Proceso va aumentando o disminuyendo su conjunto residente dependiendo de su comportamiento
  - Comportamiento relativamente predecible
- Asignación dinámica + reemplazo global
  - Procesos se quitan las páginas entre ellos
  - Comportamiento difícilmente predecible

Hyperpaginación (*Thrashing*)

- 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 núm. marcos disponibles $< \sum$ conjuntos de trabajo de todos
  - Grado de utilización de CPU cae drásticamente
  - Procesos están casi siempre en colas de dispositivo de paginación
  - Solución (similar al *swapping*): Control de carga
    1. Disminuir el grado de multiprogramación
    2. Suspender 1 o más procesos liberando sus páginas residentes
  - Problema: ¿Cómo detectar esta situación?

Hyperpaginación en el sistema

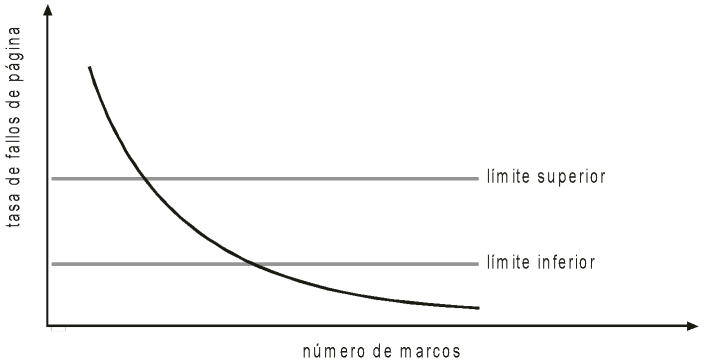


Estrategia del conjunto de trabajo

- Intentar conocer el conjunto de trabajo de cada proceso
  - Páginas usadas por el proceso en las últimas N referencias
- Si conjunto de trabajo decrece se liberan marcos
- Si conjunto de trabajo crece se asignan nuevos marcos
  - Si no hay disponibles: suspender proceso(s)
  - Se reactivan cuando hay marcos suficientes para c. de trabajo
- Asignación dinámica con reemplazo local
- Difícil implementación estricta
  - Precisaría una MMU específica
- Se pueden implementar aproximaciones:
  - Estrategia basada en frecuencia de fallos de página (PFF):
    - Controlar tasa de fallos de página de cada proceso

Estrategia basada en frecuencia de fallos

- Si tasa $<$ límite inferior se liberan marcos aplicando un algoritmo de reemplazo
- Si tasa $>$ límite superior se asignan nuevos marcos
  - Si no marcos libres se suspende algún proceso



Control de carga y reemplazo global

- Algoritmos de reemplazo global no controlan hiperpaginación
  - ¡Incluso el óptimo!
  - Necesitan cooperar con un algoritmo de control de carga
- Ejemplo: UNIX 4.3 BSD
  - Reemplazo global con algoritmo del reloj
    - Variante con dos "manecillas"
  - Uso de buffering de páginas
    - "demonio de paginación" controla no de marcos libres
  - Si número de marcos libres $<$ umbral
    - "demonio de paginación" aplica reemplazo
  - Si se repite con frecuencia la falta de marcos libres:
    - Proceso "swapper" suspende procesos

Contenido

Gestión de memoria en Linux

En casi todos los SSOO hay 2 esquemas de gestión de memoria:

1. Sistema de paginación (MV) para los procesos
2. Gestión de memoria dinámica del kernel

Gestión del espacio de usuario en Linux

Sistema de paginación para la gestión de memoria virtual

- En una máquina de 32 bits cada proceso tiene 3GB para direcciones virtuales de usuario y 1GB para las de núcleo
- La organización de la TP depende de la arquitectura (viene impuesto)
- Asigna un conjunto de páginas contiguas usando el algoritmo *buddy*
- Algoritmo de reemplazamiento usado: Variante del alg. del reloj pero con dos manecillas
- Gestión del heap: La realiza la biblioteca del lenguaje de programación concreto con soporte del SO

Gestión del heap de C en Linux

- La biblioteca de gestión del heap solicita una nueva página al SO cuando necesita más memoria
  - Llamada al sistema `brk()`
- Normalmente no la devuelve cuando le sobra.
- La asignación debe ser muy eficiente porque hay un GRAN NUMERO de peticiones:
  - Se utiliza un algoritmo de múltiples listas con huecos de tamaño variable "buddy perezoso"
    - No se juntan huecos adyacentes.
- Para sistemas *multithread* hay versiones con varios heaps
  - Para que el acceso a las estructuras de datos que mantienen el estado del heap no sea un cuello de botella

Gestión del espacio de kernel en Linux

Sistema de gestión de áreas de memoria del kernel (Slab Allocator)

- Para las estructuras de datos y buffers que crea el kernel.
- Solicita un grupo de marcos páginas contiguas de memoria al sistema de paginación y gestiona el espacio.
- Asignador basado en objetos: tiene caches de objetos usados con frecuencia
  - El kernel crea y destruye objetos del mismo tipo, así evita destruirlos e inicializarlos