Administración de memoria

Sistemas Operativos
DC - UBA - FCEN

29 de septiembre de 2025

Menú para hoy

Administración de memoria

- Hasta ahora vimos cómo ejecutar muchos procesos con una sola CPU.

Administración de memoria

- Hasta ahora vimos cómo ejecutar muchos procesos con una sola CPU.
- La CPU sólo puede ejecutar instrucciones que lee de memoria principal.

Administración de memoria

- Hasta ahora vimos cómo ejecutar muchos procesos con una sola CPU.
- La CPU sólo puede ejecutar instrucciones que lee de memoria principal.
- Los programas y datos viven en almacenamiento secundario, así que hay que cargarlos en memoria principal para ejecutarlos.

Administración de memoria

- Hasta ahora vimos cómo ejecutar muchos procesos con una sola CPU.
- La CPU sólo puede ejecutar instrucciones que lee de memoria principal.
- Los programas y datos viven en almacenamiento secundario, así que hay que cargarlos en memoria principal para ejecutarlos.
- Necesitamos **memory management.**

Administración de memoria

- Hasta ahora vimos cómo ejecutar muchos procesos con una sola CPU.
- La CPU sólo puede ejecutar instrucciones que lee de memoria principal.
- Los programas y datos viven en almacenamiento secundario, así que hay que cargarlos en memoria principal para ejecutarlos.
- Necesitamos **memory management.**

Administración de memoria

- Hasta ahora vimos cómo ejecutar muchos procesos con una sola CPU.
- La CPU sólo puede ejecutar instrucciones que lee de memoria principal.
- Los programas y datos viven en almacenamiento secundario, así que hay que cargarlos en memoria principal para ejecutarlos.
- Necesitamos **memory management.**

El sistema operativo es responsable de:

- Saber qué partes de la memoria están en uso.
- Saber qué proceso usa cada parte de la memoria.
- Asignar y liberar espacios de memoria.

Menú para hoy

Memoria virtual

- Toda dirección generada por un programa es una dirección virtual.

Memoria virtual

- Toda dirección generada por un programa es una dirección virtual.
- El SO provee una ilusión de que tiene toda la memoria necesaria disponible.

Memoria virtual

- Toda dirección generada por un programa es una dirección virtual.
- El SO provee una ilusión de que tiene toda la memoria necesaria disponible.
- Un componente de hardware, la MMU (*Memory Management Unit*), se ocupa de la traducción de direcciones virtuales a físicas.

Memoria virtual

- Toda dirección generada por un programa es una dirección virtual.
- El SO provee una ilusión de que tiene toda la memoria necesaria disponible.
- Un componente de hardware, la MMU (*Memory Management Unit*), se ocupa de la traducción de direcciones virtuales a físicas.

Memoria virtual

- Toda dirección generada por un programa es una dirección virtual.
- El SO provee una ilusión de que tiene toda la memoria necesaria disponible.
- Un componente de hardware, la MMU (*Memory Management Unit*), se ocupa de la traducción de direcciones virtuales a físicas.

Objetivos:

- Facilitar el uso de la memoria: los programadores no tienen que gestionar manualmente la ubicación de código y datos.

Memoria virtual

- Toda dirección generada por un programa es una dirección virtual.
- El SO provee una ilusión de que tiene toda la memoria necesaria disponible.
- Un componente de hardware, la MMU (*Memory Management Unit*), se ocupa de la traducción de direcciones virtuales a físicas.

Objetivos:

- Facilitar el uso de la memoria: los programadores no tienen que gestionar manualmente la ubicación de código y datos.
- Transparencia: los programas no saben que su memoria es virtual.

Memoria virtual

- Toda dirección generada por un programa es una dirección virtual.
- El SO provee una ilusión de que tiene toda la memoria necesaria disponible.
- Un componente de hardware, la MMU (*Memory Management Unit*), se ocupa de la traducción de direcciones virtuales a físicas.

Objetivos:

- Facilitar el uso de la memoria: los programadores no tienen que gestionar manualmente la ubicación de código y datos.
- Transparencia: los programas no saben que su memoria es virtual.
- Eficiencia: traducción rápida con poco *overhead*.

Memoria virtual

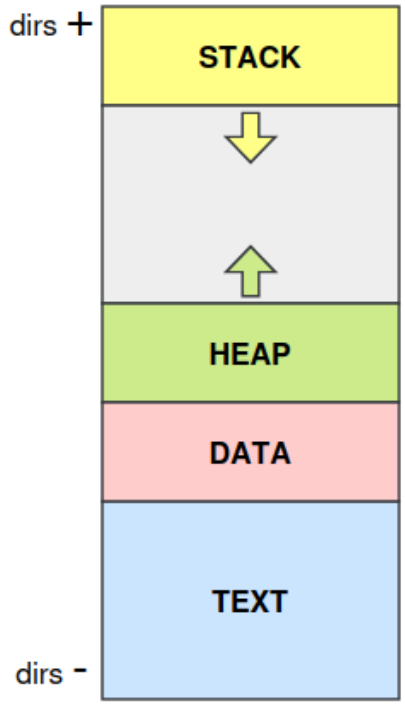
- Toda dirección generada por un programa es una dirección virtual.
- El SO provee una ilusión de que tiene toda la memoria necesaria disponible.
- Un componente de hardware, la MMU (*Memory Management Unit*), se ocupa de la traducción de direcciones virtuales a físicas.

Objetivos:

- Facilitar el uso de la memoria: los programadores no tienen que gestionar manualmente la ubicación de código y datos.
- Transparencia: los programas no saben que su memoria es virtual.
- Eficiencia: traducción rápida con poco *overhead*.
- Protección: los procesos no pueden dañarse entre sí.

El espacio de direcciones

- Cada proceso tiene su propio espacio de direcciones: una vista privada de la memoria.
- Contiene: code, stack, heap, data.
- El stack y el heap se ubican de esta manera por convención, pero puede cambiar (ejemplo: múltiples threads).
- Los programas no se cargan en una dirección física fija: el SO carga los programas en ubicaciones arbitrarias de la memoria física.
- La MMU maneja la traducción de direcciones.



Traducción de direcciones

- Cada acceso a memoria (*load*, *store*, *fetch*) se debe traducir a dirección física.

Traducción de direcciones

- Cada acceso a memoria (*load*, *store*, *fetch*) se debe traducir a dirección física.
- El SO es responsable de administrar el uso de memoria entre todos los procesos.

Traducción de direcciones

- Cada acceso a memoria (*load*, *store*, *fetch*) se debe traducir a dirección física.
- El SO es responsable de administrar el uso de memoria entre todos los procesos.

Traducción de direcciones

- Cada acceso a memoria (*load*, *store*, *fetch*) se debe traducir a dirección física.
- El SO es responsable de administrar el uso de memoria entre todos los procesos.

Estrategias de pasaje de direcciones

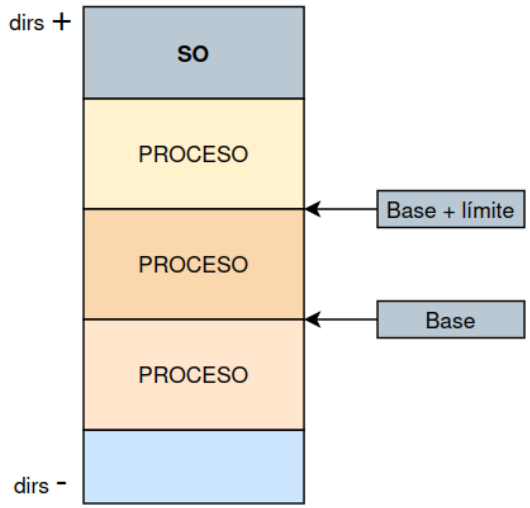
Repasemos tres formas de encarar este problema.

- Base y límite
- Segmentación
- Paginación

Traducción de direcciones

Base y límite

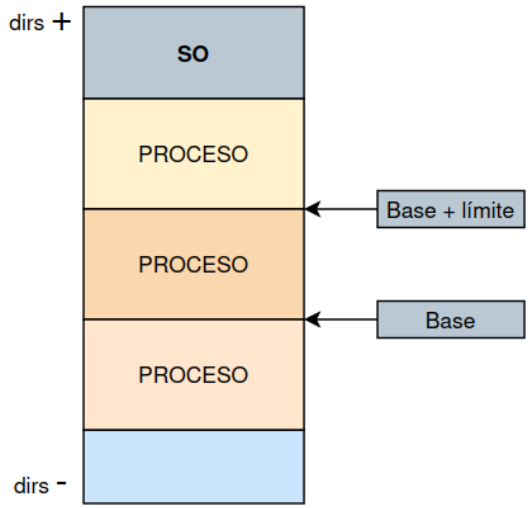
- Cada proceso tiene:
  - Registro Base: dirección física donde se carga el programa, corresponde a la dirección virtual 0.
  - Registro Límite: tamaño de la región de memoria válida para el proceso, asegura que el proceso no puede acceder a memoria por fuera de su rango.



Traducción de direcciones

Base y límite

- Cada proceso tiene:
  - Registro Base: dirección física donde se carga el programa, corresponde a la dirección virtual 0.
  - Registro Límite: tamaño de la región de memoria válida para el proceso, asegura que el proceso no puede acceder a memoria por fuera de su rango.

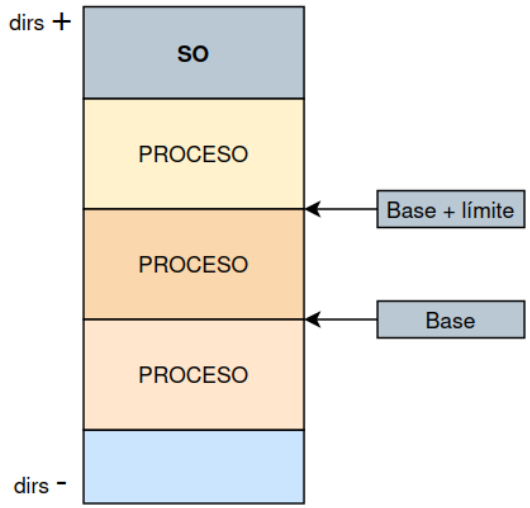


- La CPU tiene un único set de registros base y límite. Durante un context switch, el SO debe cargar los valores del nuevo proceso.

Traducción de direcciones

Base y límite

- Cada proceso tiene:
  - Registro Base: dirección física donde se carga el programa, corresponde a la dirección virtual 0.
  - Registro Límite: tamaño de la región de memoria válida para el proceso, asegura que el proceso no puede acceder a memoria por fuera de su rango.



- La CPU tiene un único set de registros base y límite. Durante un context switch, el SO debe cargar los valores del nuevo proceso.
- Fragmentación interna: Si se asigna memoria de más, la memoria no utilizada es desperdiciada ya que no pueden usarla otros procesos.

Traducción de direcciones

Segmentación

- Cada espacio de direcciones se separa en segmentos lógicos de distinto tamaño: *code*, *heap*, *stack*.

Traducción de direcciones

Segmentación

- Cada espacio de direcciones se separa en segmentos lógicos de distinto tamaño: *code*, *heap*, *stack*.
- La MMU tiene un par de registros base y límite por segmento.

Traducción de direcciones

Segmentación

- Cada espacio de direcciones se separa en segmentos lógicos de distinto tamaño: *code*, *heap*, *stack*.
- La MMU tiene un par de registros base y límite por segmento.
- Reduce la cantidad de memoria desperdiciada en el espacio físico.

Traducción de direcciones

Segmentación

- Cada espacio de direcciones se separa en segmentos lógicos de distinto tamaño: *code*, *heap*, *stack*.
- La MMU tiene un par de registros base y límite por segmento.
- Reduce la cantidad de memoria desperdiciada en el espacio físico.
- Segmentation Fault: se genera al intentar acceder a memoria por fuera de un segmento válido. Es detectado por el hardware y transfiere el control al SO.

Traducción de direcciones

Segmentación

- Cada espacio de direcciones se separa en segmentos lógicos de distinto tamaño: *code, heap, stack*.
- La MMU tiene un par de registros base y límite por segmento.
- Reduce la cantidad de memoria desperdiciada en el espacio físico.
- Segmentation Fault: se genera al intentar acceder a memoria por fuera de un segmento válido. Es detectado por el hardware y transfiere el control al SO.
- Durante un *context switch*, se guardan y cargan los base-límite de cada segmento, y se actualiza el tamaño si el segmento creció (ej: uso de `malloc()`).

Traducción de direcciones

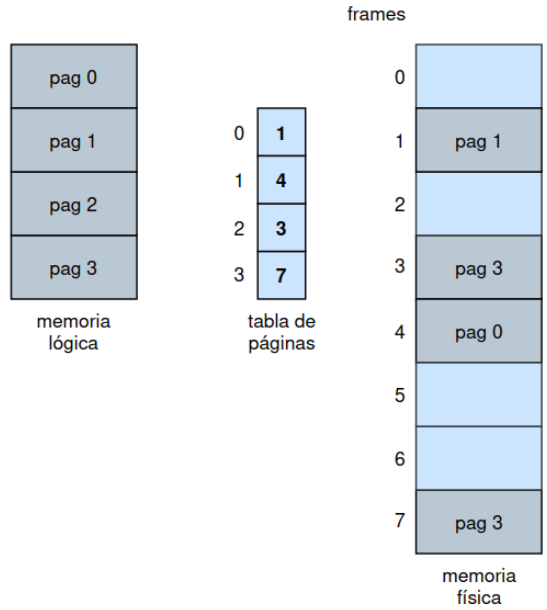
Segmentación

- Cada espacio de direcciones se separa en segmentos lógicos de distinto tamaño: *code, heap, stack*.
- La MMU tiene un par de registros base y límite por segmento.
- Reduce la cantidad de memoria desperdiciada en el espacio físico.
- Segmentation Fault: se genera al intentar acceder a memoria por fuera de un segmento válido. Es detectado por el hardware y transfiere el control al SO.
- Durante un *context switch*, se guardan y cargan los base-límite de cada segmento, y se actualiza el tamaño si el segmento creció (ej: uso de `malloc()`).
- Fragmentación externa: hay suficiente espacio libre pero está dispersa, dificultando asignar segmentos grandes de memoria.

Traducción de direcciones

Paginación

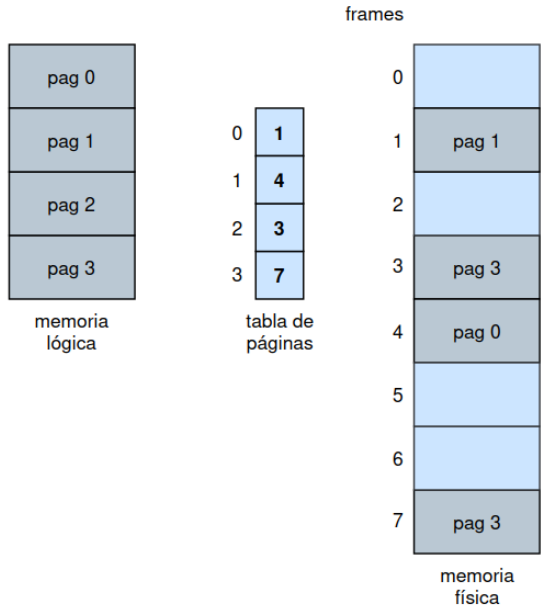
- La memoria virtual se divide en páginas de tamaño fijo (comunmente 4KB).
- La memoria física se divide en marcos de páginas del mismo tamaño.
- Cada página se mapea a un marco de página de forma independiente, así que la memoria virtual y física no necesita ser contigua.



Traducción de direcciones

Paginación

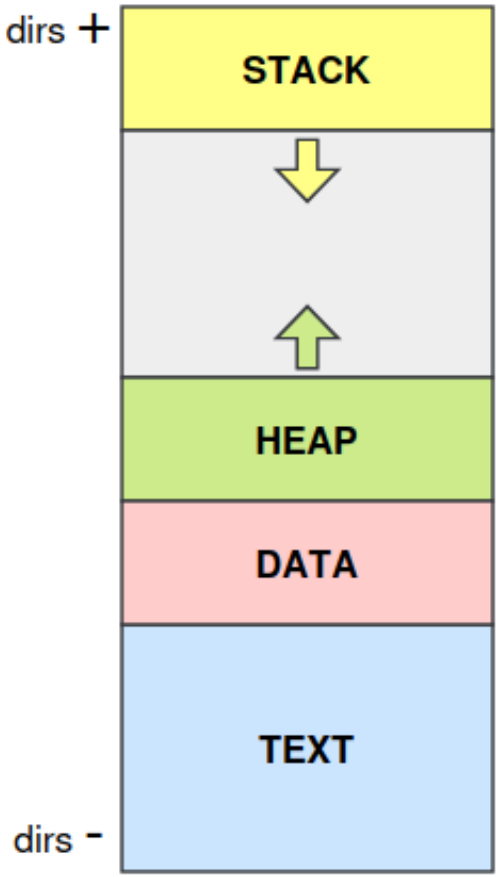
- La memoria virtual se divide en páginas de tamaño fijo (comunmente 4KB).
- La memoria física se divide en marcos de páginas del mismo tamaño.
- Cada página se mapea a un marco de página de forma independiente, así que la memoria virtual y física no necesita ser contigua.



- Evita la fragmentación externa y simplifica la asignación de memoria.
- Vuelve a aparecer la fragmentación interna.

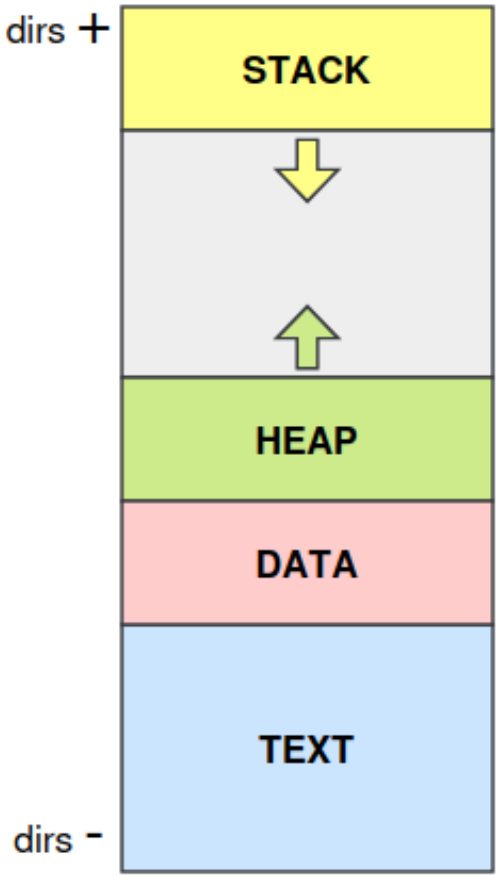
Menú para hoy

Memory API



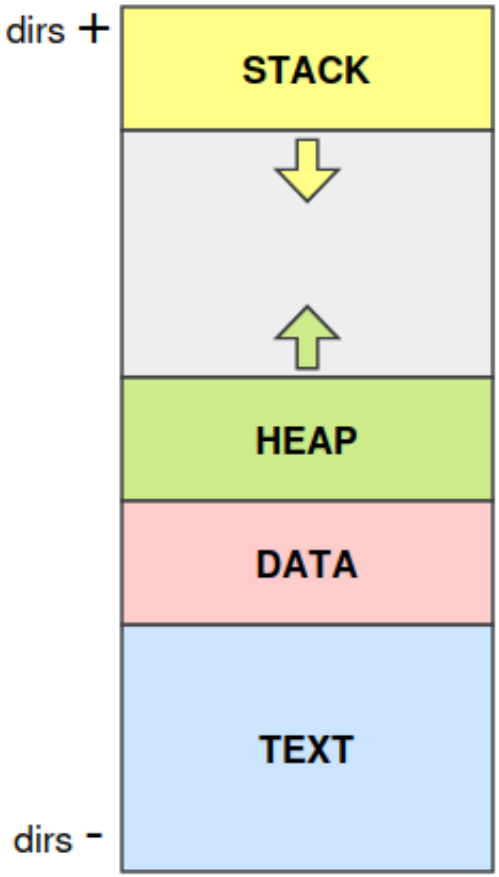
- El SO virtualiza la memoria, y las aplicaciones necesitan una forma de pedir más memoria.

Memory API



- El SO virtualiza la memoria, y las aplicaciones necesitan una forma de pedir más memoria.
- En C (y otros lenguajes) se provee una Memory API que permite a los programadores pedir y liberar memoria.

Memory API



- El SO virtualiza la memoria, y las aplicaciones necesitan una forma de pedir más memoria.
- En C (y otros lenguajes) se provee una Memory API que permite a los programadores pedir y liberar memoria.
- Existen dos "tipos de memoria": el stack, que se maneja de manera implícita, y la heap, que se maneja de manera explícita.

Stack vs Heap

- Memoria Stack:

Stack vs Heap

- Memoria Stack:
  - Es administrada implícitamente por el compilador.

Stack vs Heap

- Memoria Stack:
  - Es administrada implícitamente por el compilador.
  - Se reserva memoria para variables locales dentro de las funciones.

Stack vs Heap

- Memoria Stack:
  - Es administrada implícitamente por el compilador.
  - Se reserva memoria para variables locales dentro de las funciones.
  - Se libera automáticamente al retornar.

Stack vs Heap

- Memoria Stack:
  - Es administrada implícitamente por el compilador.
  - Se reserva memoria para variables locales dentro de las funciones.
  - Se libera automáticamente al retornar.
  - Es de tamaño limitado y no puede usarse para estructuras dinámicas.

Stack vs Heap

- Memoria Stack:
  - Es administrada implícitamente por el compilador.
  - Se reserva memoria para variables locales dentro de las funciones.
  - Se libera automáticamente al retornar.
  - Es de tamaño limitado y no puede usarse para estructuras dinámicas.
- Memoria Heap:

Stack vs Heap

- Memoria Stack:
  - Es administrada implícitamente por el compilador.
  - Se reserva memoria para variables locales dentro de las funciones.
  - Se libera automáticamente al retornar.
  - Es de tamaño limitado y no puede usarse para estructuras dinámicas.
- Memoria Heap:
  - El programador reserva y libera memoria manualmente.

Stack vs Heap

- Memoria Stack:
  - Es administrada implícitamente por el compilador.
  - Se reserva memoria para variables locales dentro de las funciones.
  - Se libera automáticamente al retornar.
  - Es de tamaño limitado y no puede usarse para estructuras dinámicas.
- Memoria Heap:
  - El programador reserva y libera memoria manualmente.
    - `malloc()` para reservar.

Stack vs Heap

- Memoria Stack:
  - Es administrada implícitamente por el compilador.
  - Se reserva memoria para variables locales dentro de las funciones.
  - Se libera automáticamente al retornar.
  - Es de tamaño limitado y no puede usarse para estructuras dinámicas.
- Memoria Heap:
  - El programador reserva y libera memoria manualmente.
    - `malloc()` para reservar.
    - `free()` para liberar.

Uso de malloc()

- `void* malloc(size_t size);`

Uso de `malloc()`

- `void* malloc(size_t size);`
- Pide una cantidad de bytes determinada por parámetro.

Uso de `malloc()`

- `void* malloc(size_t size);`
- Pide una cantidad de bytes determinada por parámetro.
- El valor `size` no suele proveerse con un valor directo sino con ayuda del operador `sizeof()`.

Uso de `malloc()`

- `void* malloc(size_t size);`
- Pide una cantidad de bytes determinada por parámetro.
- El valor `size` no suele proveerse con un valor directo sino con ayuda del operador `sizeof()`.
- Retorna un puntero a la memoria solicitado, o `NULL` si falla.

Uso de `malloc()`

- `void* malloc(size_t size);`
- Pide una cantidad de bytes determinada por parámetro.
- El valor `size` no suele proveerse con un valor directo sino con ayuda del operador `sizeof()`.
- Retorna un puntero a la memoria solicitado, o `NULL` si falla.
- Se debe castear el puntero `void*` al tipo de datos corresondiente.

Uso de `malloc()`

- `void* malloc(size_t size);`
- Pide una cantidad de bytes determinada por parámetro.
- El valor `size` no suele proveerse con un valor directo sino con ayuda del operador `sizeof()`.
- Retorna un puntero a la memoria solicitado, o `NULL` si falla.
- Se debe castear el puntero `void*` al tipo de datos corresondiente.
- Es una función de librería, no una *system call*. Internamente utiliza la *syscall* `brk()`, que maneja memoria dentro del espacio virtual de direcciones.

Uso de `malloc()`

- `void* malloc(size_t size);`
- Pide una cantidad de bytes determinada por parámetro.
- El valor `size` no suele proveerse con un valor directo sino con ayuda del operador `sizeof()`.
- Retorna un puntero a la memoria solicitado, o `NULL` si falla.
- Se debe castear el puntero `void*` al tipo de datos corresondiente.
- Es una función de librería, no una *system call*. Internamente utiliza la *syscall* `brk()`, que maneja memoria dentro del espacio virtual de direcciones.
- No usar nunca `brk()` o `sbrk()` directamente.

Uso de `malloc()`

- `void* malloc(size_t size);`
- Pide una cantidad de bytes determinada por parámetro.
- El valor `size` no suele proveerse con un valor directo sino con ayuda del operador `sizeof()`.
- Retorna un puntero a la memoria solicitado, o `NULL` si falla.
- Se debe castear el puntero `void*` al tipo de datos corresondiente.
- Es una función de librería, no una *system call*. Internamente utiliza la *syscall* `brk()`, que maneja memoria dentro del espacio virtual de direcciones.
- No usar nunca `brk()` o `sbrk()` directamente.

Uso de `malloc()`

- `void* malloc(size_t size);`
- Pide una cantidad de bytes determinada por parámetro.
- El valor `size` no suele proveerse con un valor directo sino con ayuda del operador `sizeof()`.
- Retorna un puntero a la memoria solicitado, o `NULL` si falla.
- Se debe castear el puntero `void*` al tipo de datos corresondiente.
- Es una función de librería, no una *system call*. Internamente utiliza la *syscall* `brk()`, que maneja memoria dentro del espacio virtual de direcciones.
- No usar nunca `brk()` o `sbrk()` directamente.

¿Qué hace el siguiente código?

```
int *arr = (int *)malloc(10 * sizeof(int));
```

Problemas con `malloc()`

- Buffer overflow: No reservar suficiente espacio (ejemplo: olvidarse del caracter nulo en strings). Representa un gran problema de seguridad.

Problemas con `malloc()`

- Buffer overflow: No reservar suficiente espacio (ejemplo: olvidarse del caracter nulo en strings). Representa un gran problema de seguridad.
- Lecturas no inicializadas: Usar la memoria pedida antes de asignarle un valor.

Problemas con `malloc()`

- Buffer overflow: No reservar suficiente espacio (ejemplo: olvidarse del caracter nulo en strings). Representa un gran problema de seguridad.
- Lecturas no inicializadas: Usar la memoria pedida antes de asignarle un valor.
- Memory leaks: No llamar a `free()`. Con el tiempo degradan el rendimiento. Los lenguajes con garbage collection no lo resuelven por completo.

Problemas con `malloc()`

- Buffer overflow: No reservar suficiente espacio (ejemplo: olvidarse del caracter nulo en strings). Representa un gran problema de seguridad.
- Lecturas no inicializadas: Usar la memoria pedida antes de asignarle un valor.
- Memory leaks: No llamar a `free()`. Con el tiempo degradan el rendimiento. Los lenguajes con garbage collection no lo resuelven por completo.
- Dangling pointers: Intentar usar la memoria después de haberla liberado.

Problemas con `malloc()`

- Buffer overflow: No reservar suficiente espacio (ejemplo: olvidarse del caracter nulo en strings). Representa un gran problema de seguridad.
- Lecturas no inicializadas: Usar la memoria pedida antes de asignarle un valor.
- Memory leaks: No llamar a `free()`. Con el tiempo degradan el rendimiento. Los lenguajes con garbage collection no lo resuelven por completo.
- Dangling pointers: Intentar usar la memoria después de haberla liberado.
- Doble `free()`: Llamar a `free()` más de una vez sobre el mismo puntero.

Problemas con `malloc()`

- Buffer overflow: No reservar suficiente espacio (ejemplo: olvidarse del caracter nulo en strings). Representa un gran problema de seguridad.
- Lecturas no inicializadas: Usar la memoria pedida antes de asignarle un valor.
- Memory leaks: No llamar a `free()`. Con el tiempo degradan el rendimiento. Los lenguajes con garbage collection no lo resuelven por completo.
- Dangling pointers: Intentar usar la memoria después de haberla liberado.
- Doble `free()`: Llamar a `free()` más de una vez sobre el mismo puntero.
- Free inválido: Liberar un puntero que no fue reservado dinámicamente.

Más funciones de heap

- `calloc()`: reserva memoria y la inicializa con ceros.
- `realloc()`: cambia el tamaño de un bloque de memoria reservada previamente.
- `free()`: libera memoria previamente reservada con `malloc()`, `calloc()` o `realloc()`.

Más funciones de heap

- `calloc()`: reserva memoria y la inicializa con ceros.
- `realloc()`: cambia el tamaño de un bloque de memoria reservada previamente.
- `free()`: libera memoria previamente reservada con `malloc()`, `calloc()` o `realloc()`.

¿Qué hace el siguiente código?

```
int *p1 = malloc(100);
int *p2 = calloc(25, sizeof(int));
int *p3 = realloc(p1, 200);
free(p2);
free(p3);
```

Menú para hoy

Gestión del espacio libre

- El SO debe:

Gestión del espacio libre

- El SO debe:
  - Administrar eficientemente la memoria no utilizada.

Gestión del espacio libre

- El SO debe:
  - Administrar eficientemente la memoria no utilizada.
  - Manejar los pedidos de asignación y liberación.

Gestión del espacio libre

- El SO debe:
  - Administrar eficientemente la memoria no utilizada.
  - Manejar los pedidos de asignación y liberación.
  - Evitar y reducir la fragmentación.

Gestión del espacio libre

- El SO debe:
  - Administrar eficientemente la memoria no utilizada.
  - Manejar los pedidos de asignación y liberación.
  - Evitar y reducir la fragmentación.
- Esto es especialmente desafiante al usar memoria dinámica.

Gestión del espacio libre

- El SO debe:
  - Administrar eficientemente la memoria no utilizada.
  - Manejar los pedidos de asignación y liberación.
  - Evitar y reducir la fragmentación.
- Esto es especialmente desafiante al usar memoria dinámica.
- El SO mantiene una free list (bitmap, lista enlazada, etc.) con metadata sobre los bloques de memoria disponibles.

Gestión del espacio libre

- El SO debe:
  - Administrar eficientemente la memoria no utilizada.
  - Manejar los pedidos de asignación y liberación.
  - Evitar y reducir la fragmentación.
- Esto es especialmente desafiante al usar memoria dinámica.
- El SO mantiene una free list (bitmap, lista enlazada, etc.) con metadata sobre los bloques de memoria disponibles.
- Al momento de asignar memoria, se busca en esta estructura un "cacho" de memoria libre que alcance para el pedido.

Estrategias de asignación

- Estrategias comunes:

Estrategias de asignación

- Estrategias comunes:
  - First Fit: encuentra la primer porción de memoria lo suficientemente grande.

Estrategias de asignación

- Estrategias comunes:
  - First Fit: encuentra la primer porción de memoria lo suficientemente grande.
  - Best Fit: encuentra la porción más chica de memoria libre que alcanza para el pedido.

Estrategias de asignación

- Estrategias comunes:
  - First Fit: encuentra la primer porción de memoria lo suficientemente grande.
  - Best Fit: encuentra la porción más chica de memoria libre que alcanza para el pedido.
  - Worst Fit: encuentra la porción más grande.

Estrategias de asignación

- Estrategias comunes:
  - First Fit: encuentra la primer porción de memoria lo suficientemente grande.
  - Best Fit: encuentra la porción más chica de memoria libre que alcanza para el pedido.
  - Worst Fit: encuentra la porción más grande.
  - Next Fit: continúa la búsqueda desde la última posición.

Estrategias de asignación

- Estrategias comunes:
  - First Fit: encuentra la primer porción de memoria lo suficientemente grande.
  - Best Fit: encuentra la porción más chica de memoria libre que alcanza para el pedido.
  - Worst Fit: encuentra la porción más grande.
  - Next Fit: continúa la búsqueda desde la última posición.
- Otros más complejos:

Estrategias de asignación

- Estrategias comunes:
  - First Fit: encuentra la primer porción de memoria lo suficientemente grande.
  - Best Fit: encuentra la porción más chica de memoria libre que alcanza para el pedido.
  - Worst Fit: encuentra la porción más grande.
  - Next Fit: continúa la búsqueda desde la última posición.
- Otros más complejos:
  - Listas segregadas para los tamaños más comunes, Slab allocator para pre-asignar memoria: usados en kernels.

Estrategias de asignación

- Estrategias comunes:
  - First Fit: encuentra la primer porción de memoria lo suficientemente grande.
  - Best Fit: encuentra la porción más chica de memoria libre que alcanza para el pedido.
  - Worst Fit: encuentra la porción más grande.
  - Next Fit: continúa la búsqueda desde la última posición.
- Otros más complejos:
  - Listas segregadas para los tamaños más comunes, Slab allocator para pre-asignar memoria: usados en kernels.
  - Buddy allocator: divide la memoria en potencias de 2, *splitting* y *coalescing* recursivo.

Estrategias de asignación

- Estrategias comunes:
  - First Fit: encuentra la primer porción de memoria lo suficientemente grande.
  - Best Fit: encuentra la porción más chica de memoria libre que alcanza para el pedido.
  - Worst Fit: encuentra la porción más grande.
  - Next Fit: continúa la búsqueda desde la última posición.
- Otros más complejos:
  - Listas segregadas para los tamaños más comunes, Slab allocator para pre-asignar memoria: usados en kernels.
  - Buddy allocator: divide la memoria en potencias de 2, *splitting* y *coalescing* recursivo.
- Cada estrategia tiene sus *trade-offs* entre velocidad y fragmentación.

Estrategias de asignación

Ejercicio

Enunciado

Tengo un sistema con 16 MB de memoria **sin particionar** que direcciona a byte. El estado actual de la memoria es el siguiente (cuadrado = 1MB):

Llegan los siguientes pedidos de memoria en ese orden:
512 KB, 3 MB, 1 MB, 2MB, 512 KB.
Indicar qué bloques se asignan para cada pedido utilizando *first-fit*, *best-fit* y *worst-fit*.

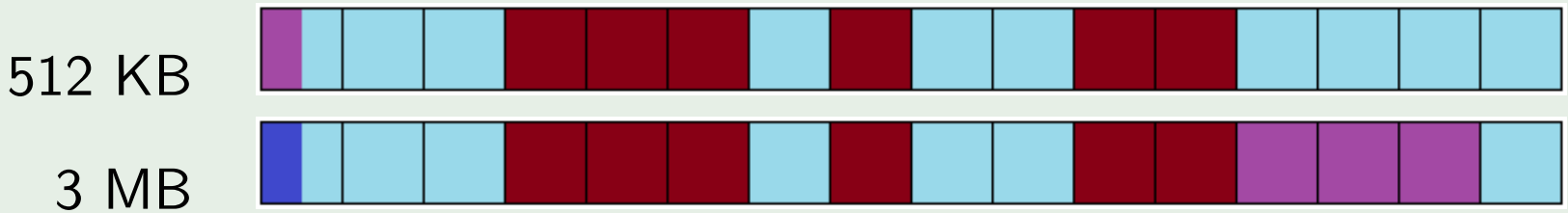
Solución First-Fit

512 KB

Estrategias de asignación

Ejercicio

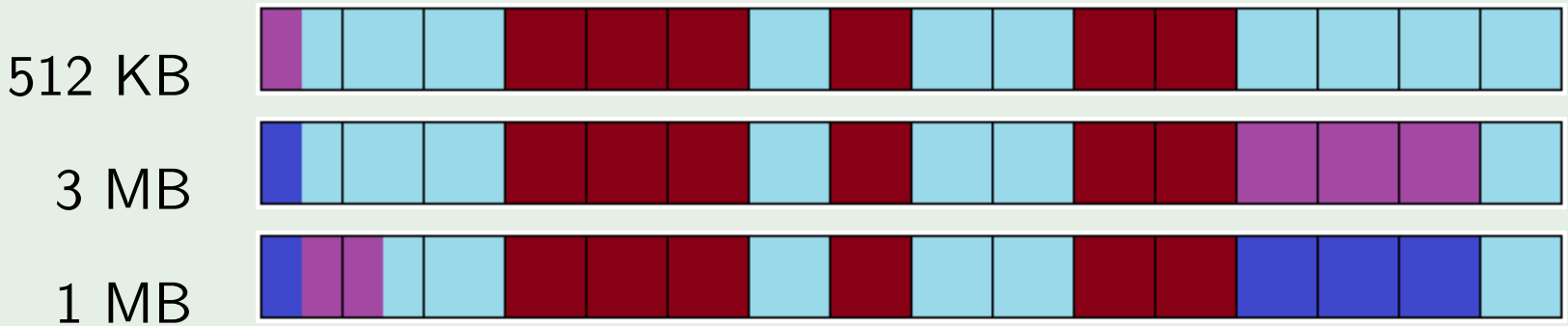
Solución First-Fit



Estrategias de asignación

Ejercicio

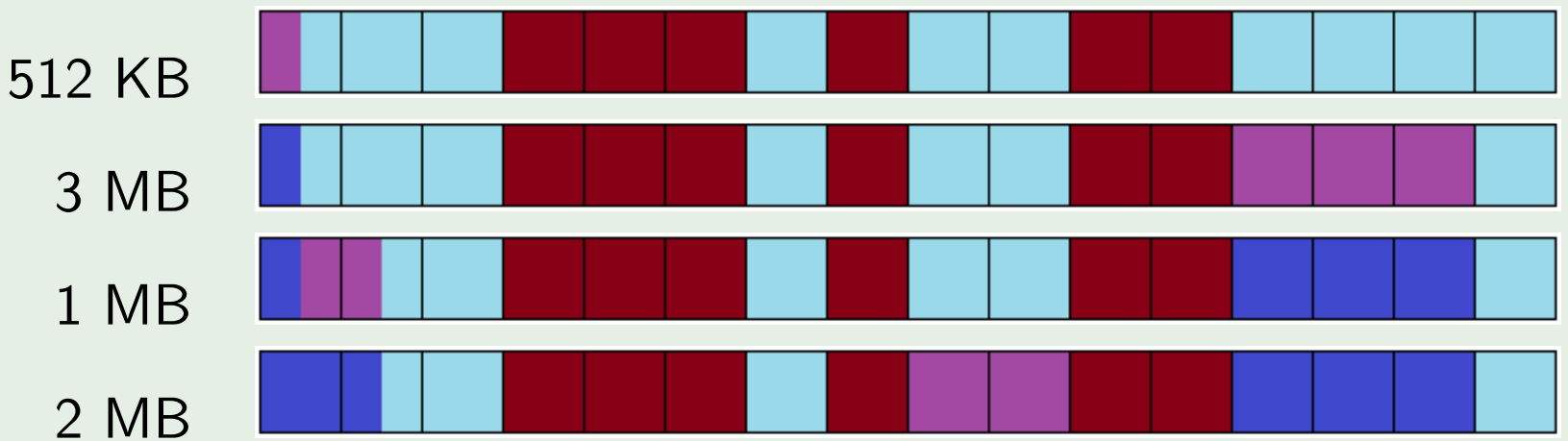
Solución First-Fit



Estrategias de asignación

Ejercicio

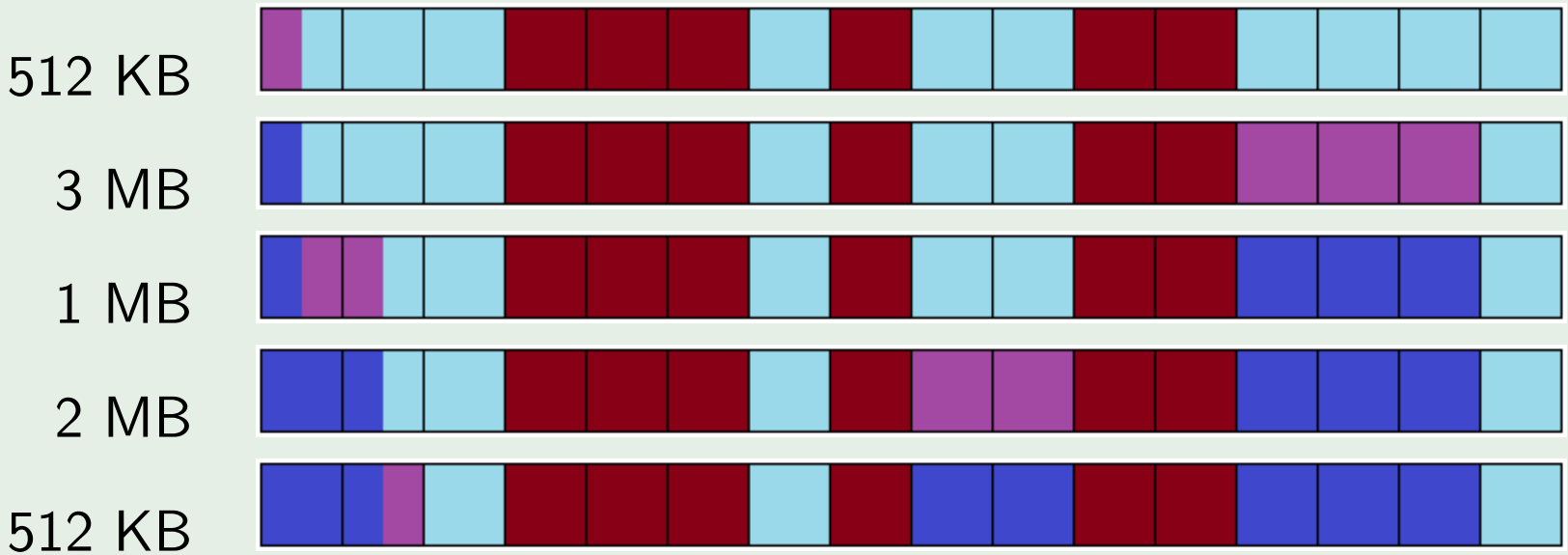
Solución First-Fit



Estrategias de asignación

Ejercicio

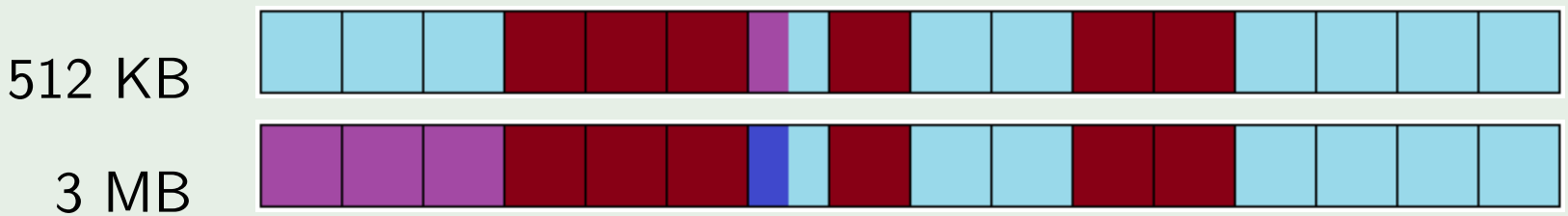
Solución First-Fit



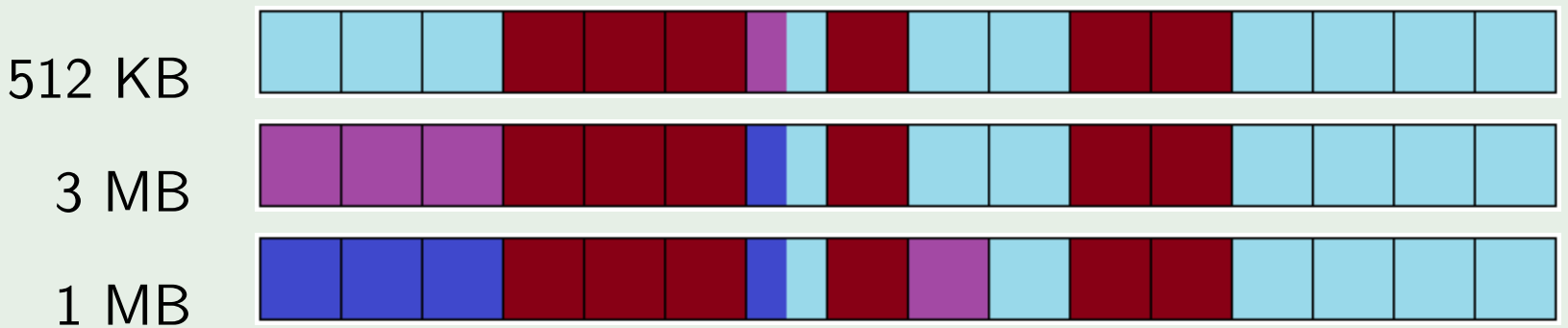
Solución Best-Fit

512 KB

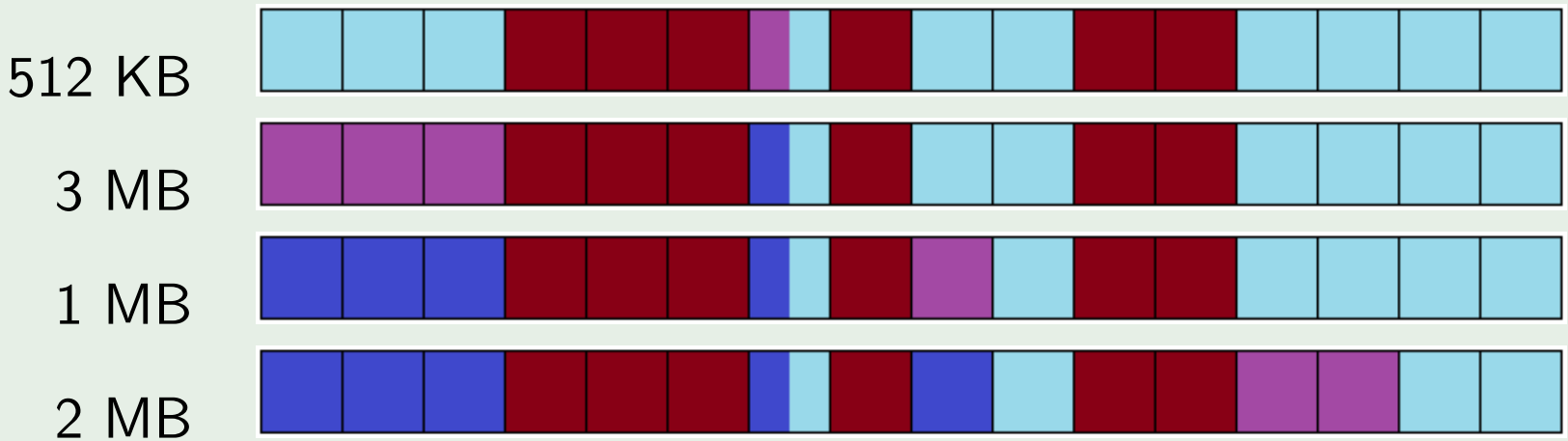
Solución Best-Fit



Solución Best-Fit



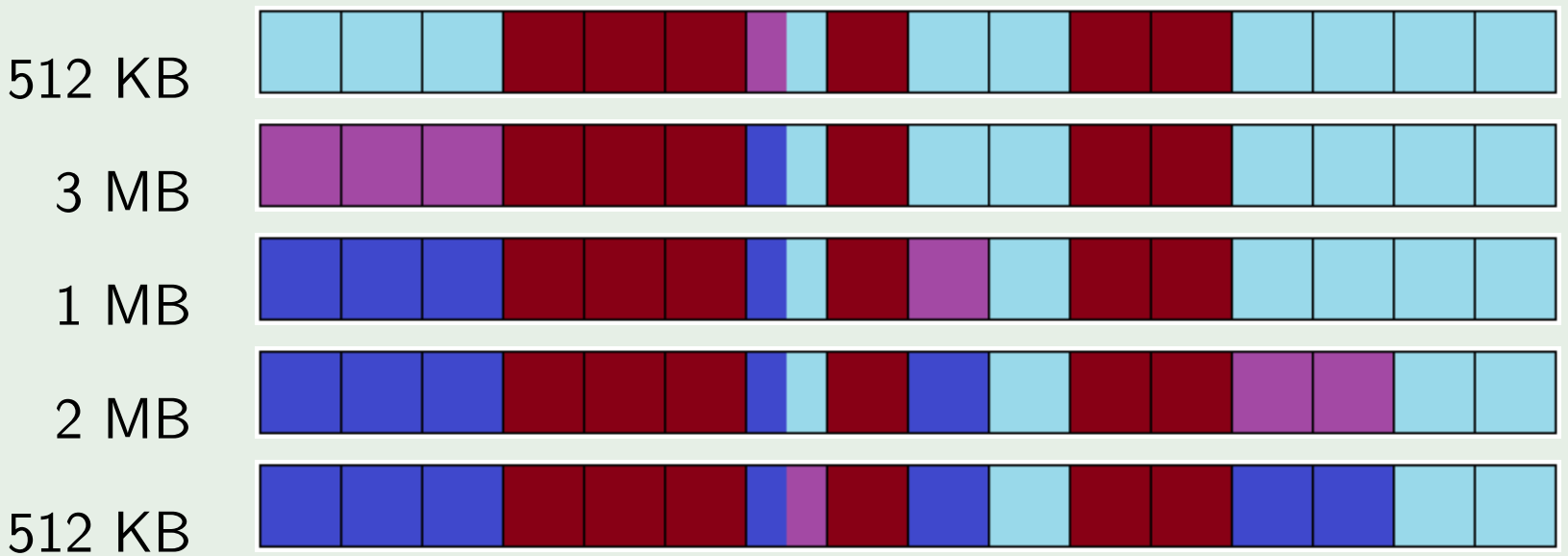
Solución Best-Fit



Estrategias de asignación

Ejercicio

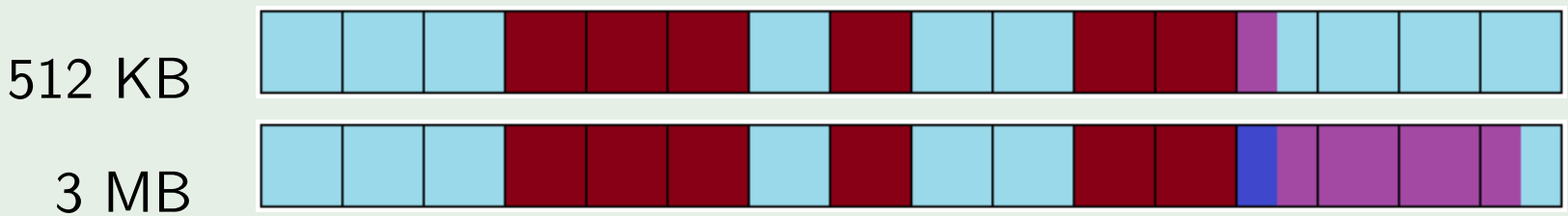
Solución Best-Fit



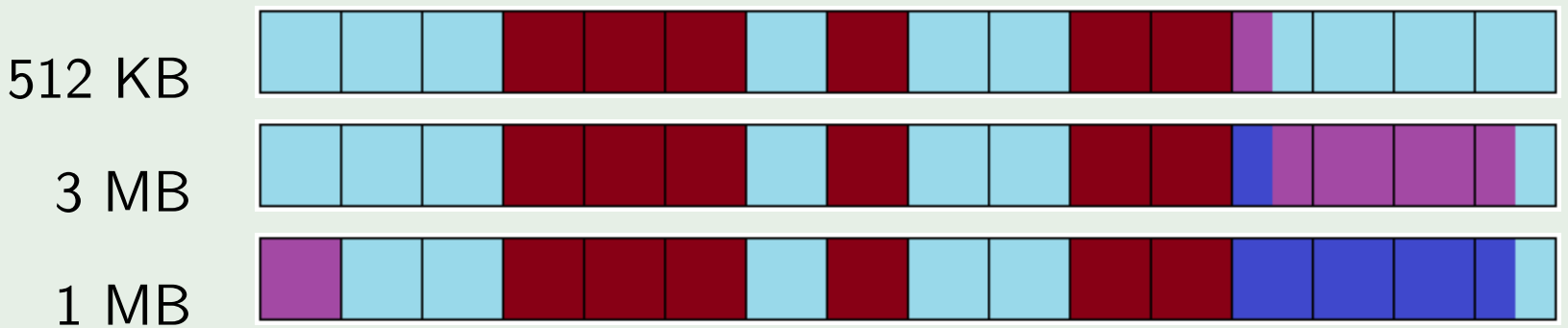
Solución Worst-Fit

512 KB

Solución Worst-Fit



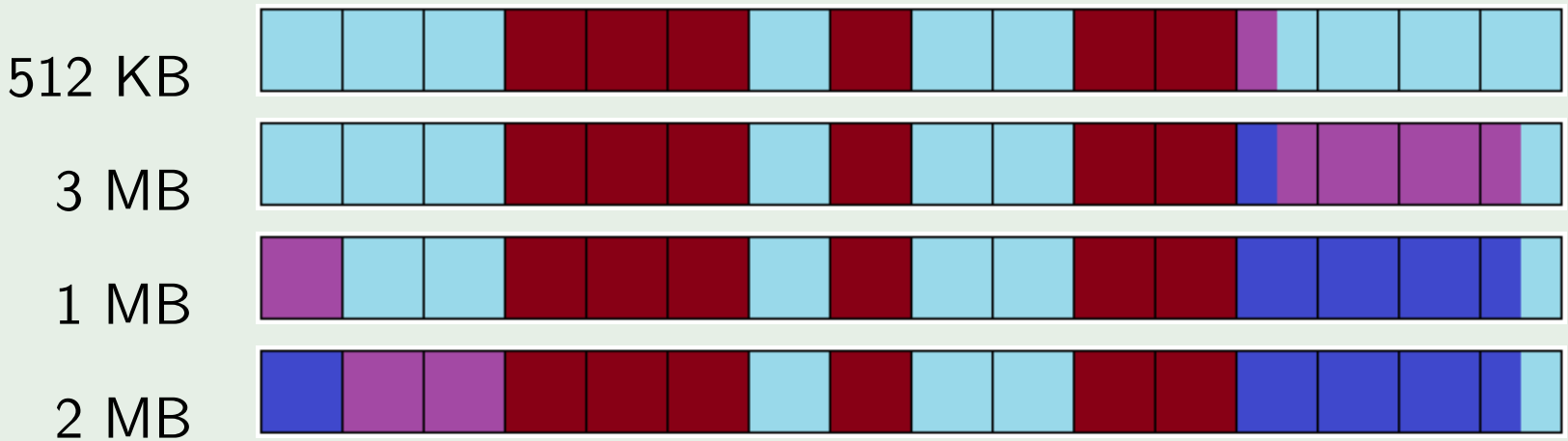
Solución Worst-Fit



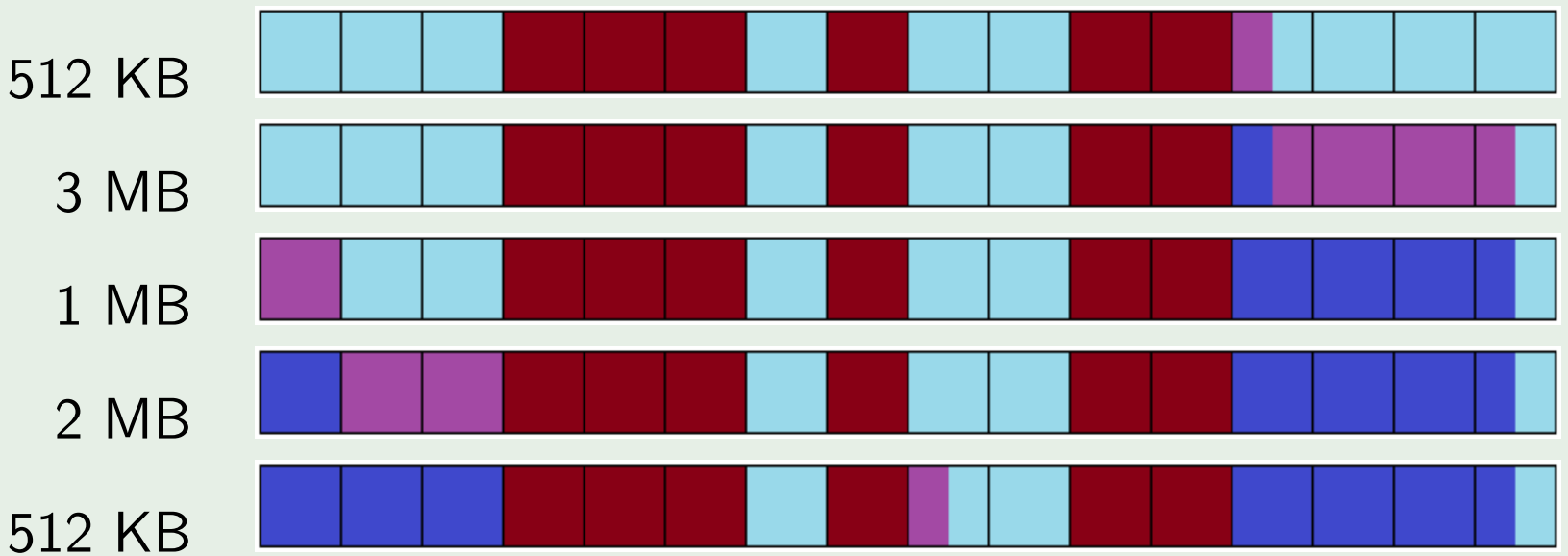
Estrategias de asignación

Ejercicio

Solución Worst-Fit



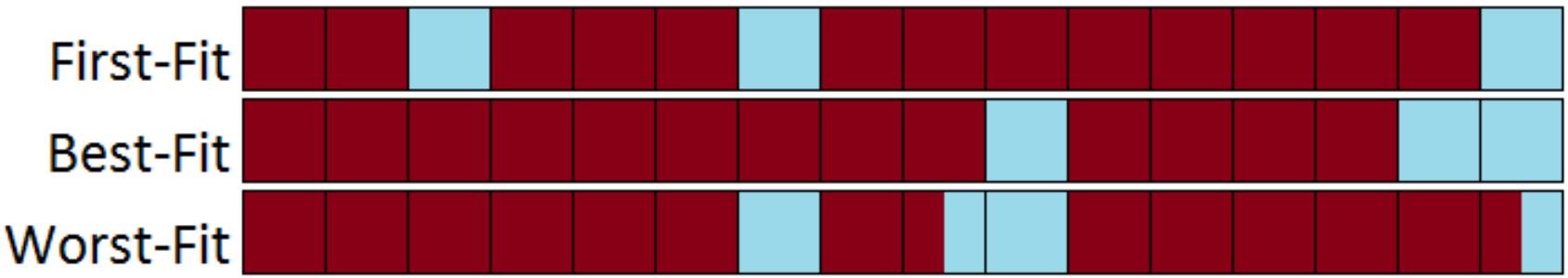
Solución Worst-Fit



Estrategias de asignación

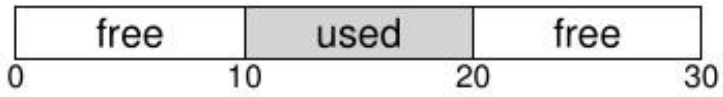
Ejercicio

Entonces, ¿Cuál es mejor?



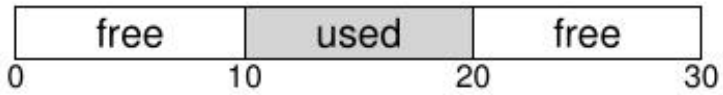
Splitting

- Supongamos que tenemos un heap de 30 bytes con esta pinta:

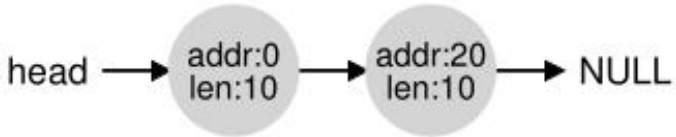


Splitting

- Supongamos que tenemos un heap de 30 bytes con esta pinta:

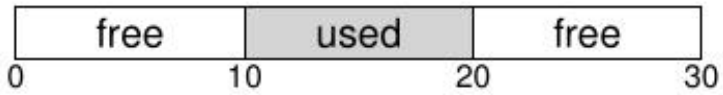


- La *free list* para este heap tendrá 2 elementos, reprentando los espacios libres:

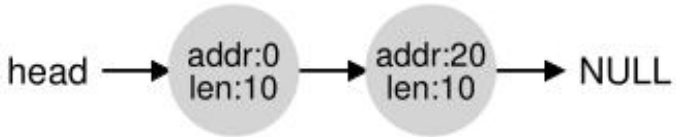


Splitting

- Supongamos que tenemos un heap de 30 bytes con esta pinta:



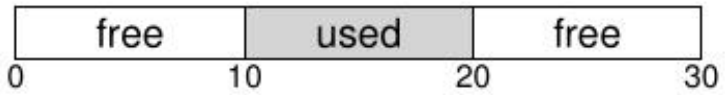
- La *free list* para este heap tendrá 2 elementos, reprentando los espacios libres:



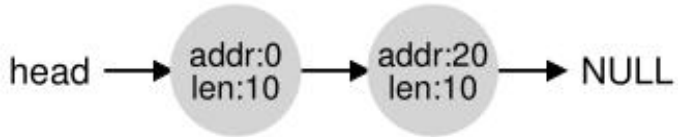
- Cualquier pedido de más de 10 bytes va a fallar.

Splitting

- Supongamos que tenemos un heap de 30 bytes con esta pinta:



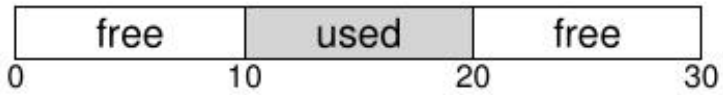
- La *free list* para este heap tendrá 2 elementos, reprentando los espacios libres:



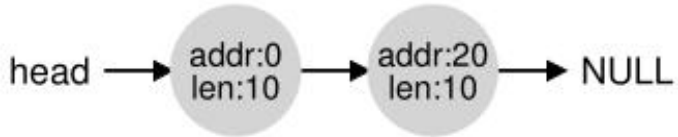
- Cualquier pedido de más de 10 bytes va a fallar.
- Para atender cualquier pedido menor a 10 bytes, se hace split.

Splitting

- Supongamos que tenemos un heap de 30 bytes con esta pinta:



- La *free list* para este heap tendrá 2 elementos, reprentando los espacios libres:



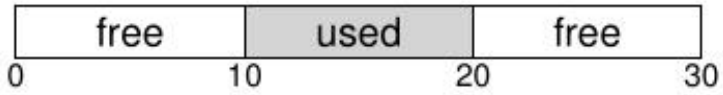
- Cualquier pedido de más de 10 bytes va a fallar.
- Para atender cualquier pedido menor a 10 bytes, se hace split.

Splitting

Si un requerimiento de memoria es menor que una porción de memoria libre, se retorna la primera parte y se mantiene el resto en la *free list*.

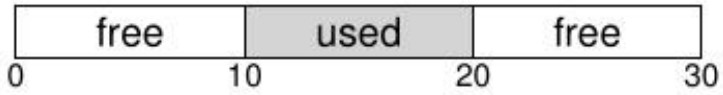
Splitting

- Supongamos que tenemos un heap de 30 bytes con esta pinta:

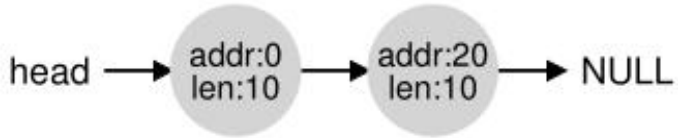


Splitting

- Supongamos que tenemos un heap de 30 bytes con esta pinta:

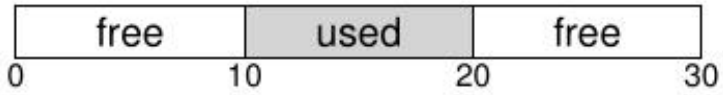


- La *free list* para este heap tendrá 2 elementos, reprentando los espacios libres:

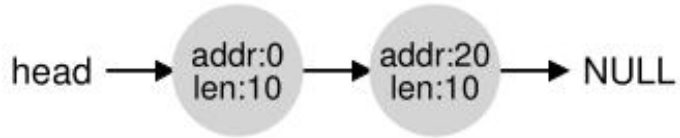


Splitting

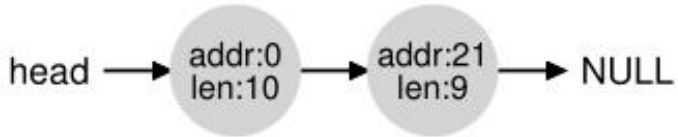
- Supongamos que tenemos un heap de 30 bytes con esta pinta:



- La *free list* para este heap tendrá 2 elementos, reprentando los espacios libres:

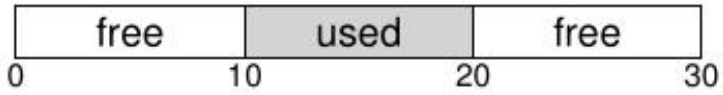


- Si se solicita 1 byte, y el mecanismo de asignación decide usar el segundo "cacho" libre, la solicitud se responde con un puntero a la posición 20 y la *free list* quedaría como sigue:

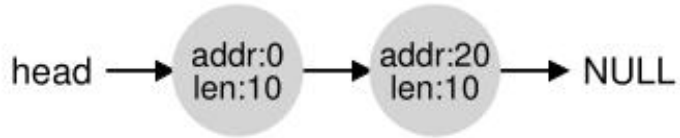


Splitting

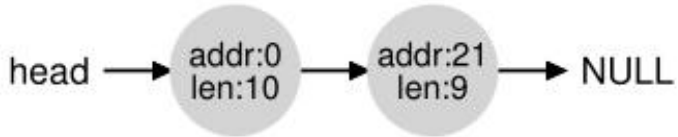
- Supongamos que tenemos un heap de 30 bytes con esta pinta:



- La *free list* para este heap tendrá 2 elementos, reprentando los espacios libres:



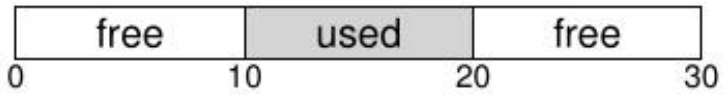
- Si se solicita 1 byte, y el mecanismo de asignación decide usar el segundo "cacho" libre, la solicitud se responde con un puntero a la posición 20 y la *free list* quedaría como sigue:



- El *splitting* ayuda a prevenir el desperdicio grandes porciones de memoria en pequeñas solicitudes.

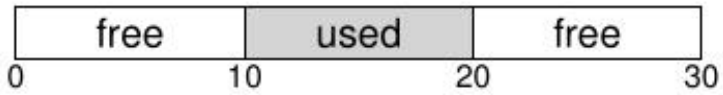
Coalescing

- Supongamos que tenemos un heap de 30 bytes con esta pinta:

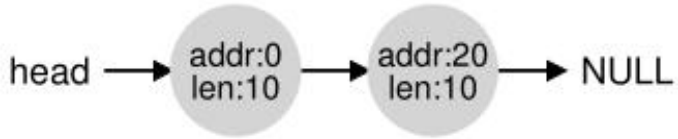


Coalescing

- Supongamos que tenemos un heap de 30 bytes con esta pinta:

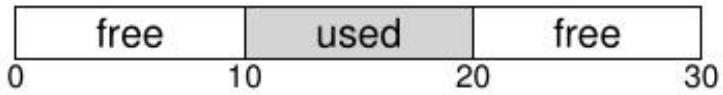


- La *free list* para este heap tendrá 2 elementos, reprentando los espacios libres:

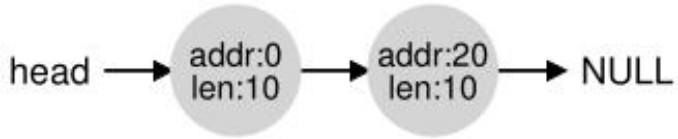


Coalescing

- Supongamos que tenemos un heap de 30 bytes con esta pinta:



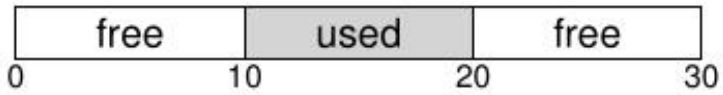
- La *free list* para este heap tendrá 2 elementos, reprentando los espacios libres:



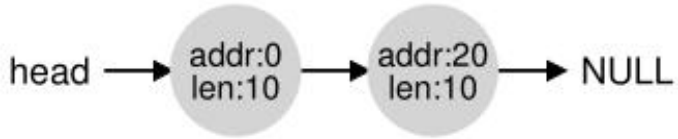
- Supongamos que se hace `free(10)`, liberando el espacio asignado del medio del heap.

Coalescing

- Supongamos que tenemos un heap de 30 bytes con esta pinta:



- La *free list* para este heap tendrá 2 elementos, reprentando los espacios libres:



- Supongamos que se hace `free(10)`, liberando el espacio asignado del medio del heap.
- Si simplemente añadimos este nuevo espacio libre a la *free list*, tendríamos algo como esto:

Coalescing

- Ahora todo el heap está ahora libre, pero parece dividirse en tres bloques de 10 bytes. Por lo tanto, una solicitud de 20 bytes fallará, aunque la memoria está disponible.

Coalescing

- Ahora todo el heap está ahora libre, pero parece dividirse en tres bloques de 10 bytes. Por lo tanto, una solicitud de 20 bytes fallará, aunque la memoria está disponible.
- Para evitar esto, se suele juntar (coalesce) el espacio libre cuando se libera memoria.

Coalescing

- Ahora todo el heap está ahora libre, pero parece dividirse en tres bloques de 10 bytes. Por lo tanto, una solicitud de 20 bytes fallará, aunque la memoria está disponible.
- Para evitar esto, se suele juntar (coalesce) el espacio libre cuando se libera memoria.

Coalescing

Cuando la memoria es liberada, se verifica si las porciones aledañas también están libres, y en ese caso se *mergean* en una única porción más grande.

Coalescing

- Ahora todo el heap está ahora libre, pero parece dividirse en tres bloques de 10 bytes. Por lo tanto, una solicitud de 20 bytes fallará, aunque la memoria está disponible.
- Para evitar esto, se suele juntar (coalesce) el espacio libre cuando se libera memoria.

Coalescing

Cuando la memoria es liberada, se verifica si las porciones aledañas también están libres, y en ese caso se *mergean* en una única porción más grande.

- Así nuestra *free list* quedaría como sigue:

Coalescing

- Ahora todo el heap está ahora libre, pero parece dividirse en tres bloques de 10 bytes. Por lo tanto, una solicitud de 20 bytes fallará, aunque la memoria está disponible.
- Para evitar esto, se suele juntar (coalesce) el espacio libre cuando se libera memoria.

Coalescing

Cuando la memoria es liberada, se verifica si las porciones aledañas también están libres, y en ese caso se *mergean* en una única porción más grande.

- Así nuestra *free list* quedaría como sigue:

- El *coalescing* reduce la fragmentación externa.

Headers en memoria asignada

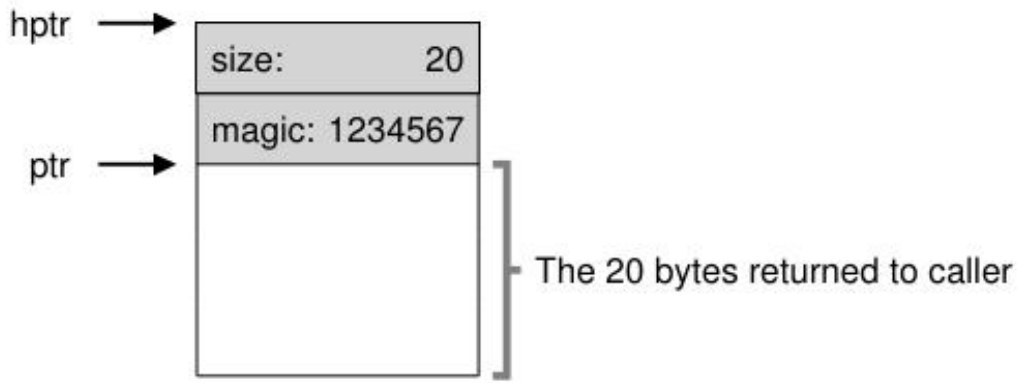
- `free(void *ptr)` no tiene el size como parámetro, sólo el puntero al inicio de la porción de memoria asignada.

Headers en memoria asignada

- `free(void *ptr)` no tiene el size como parámetro, sólo el puntero al inicio de la porción de memoria asignada.
- Para abordar esto, al asignar memoria se suele almacenar un *header* que con el tamaño del bloque y alguna metadata más.

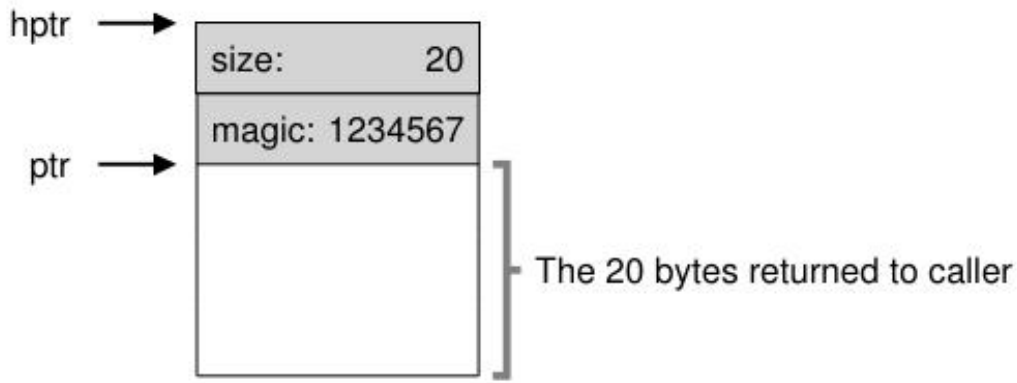
Headers en memoria asignada

- `free(void *ptr)` no tiene el size como parámetro, sólo el puntero al inicio de la porción de memoria asignada.
- Para abordar esto, al asignar memoria se suele almacenar un *header* que con el tamaño del bloque y alguna metadata más.
- Por ejemplo, si se reservan 20 bytes, se devuelve un puntero al inicio de esos 20 bytes pero se ocupa una cantidad fija de bytes al inicio para el header.



Headers en memoria asignada

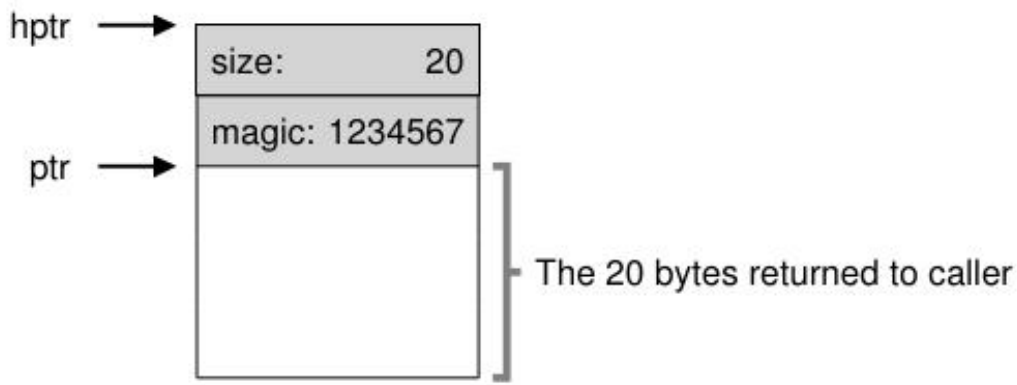
- `free(void *ptr)` no tiene el size como parámetro, sólo el puntero al inicio de la porción de memoria asignada.
- Para abordar esto, al asignar memoria se suele almacenar un *header* que con el tamaño del bloque y alguna metadata más.
- Por ejemplo, si se reservan 20 bytes, se devuelve un puntero al inicio de esos 20 bytes pero se ocupa una cantidad fija de bytes al inicio para el header.



- Al llamar a `free()`, se utiliza aritmética de punteros para encontrar el *header*.

Headers en memoria asignada

- `free(void *ptr)` no tiene el size como parámetro, sólo el puntero al inicio de la porción de memoria asignada.
- Para abordar esto, al asignar memoria se suele almacenar un *header* que con el tamaño del bloque y alguna metadata más.
- Por ejemplo, si se reservan 20 bytes, se devuelve un puntero al inicio de esos 20 bytes pero se ocupa una cantidad fija de bytes al inicio para el header.



- Al llamar a `free()`, se utiliza aritmética de punteros para encontrar el *header*.
- Ayuda a verificar integridad y llevar a cabo el *coalescing*.

Gestión de espacio libre con paginación

- El SO mantiene una lista de marcos de página libres.

Gestión de espacio libre con paginación

- El SO mantiene una lista de marcos de página libres.
- Cuando un proceso necesita memoria, se asignan páginas (no necesariamente congiguas) de la *free list* y se mapean mediante una tabla de páginas.

Gestión de espacio libre con paginación

- El SO mantiene una lista de marcos de página libres.
- Cuando un proceso necesita memoria, se asignan páginas (no necesariamente congiguas) de la *free list* y se mapean mediante una tabla de páginas.
- Cada proceso tiene su propia tabla de páginas, que es consultada por la MMU al intentar acceder a una dirección virtual.

Gestión de espacio libre con paginación

- El SO mantiene una lista de marcos de página libres.
- Cuando un proceso necesita memoria, se asignan páginas (no necesariamente congiguas) de la *free list* y se mapean mediante una tabla de páginas.
- Cada proceso tiene su propia tabla de páginas, que es consultada por la MMU al intentar acceder a una dirección virtual.
- Si un proceso ocupa N páginas, en un primer enfoque al menos N frames deben estar disponibles en memoria para poder asignarle.

Gestión de espacio libre con paginación

- El SO mantiene una lista de marcos de página libres.
- Cuando un proceso necesita memoria, se asignan páginas (no necesariamente congiguas) de la *free list* y se mapean mediante una tabla de páginas.
- Cada proceso tiene su propia tabla de páginas, que es consultada por la MMU al intentar acceder a una dirección virtual.
- Si un proceso ocupa N páginas, en un primer enfoque al menos N frames deben estar disponibles en memoria para poder asignarle.
- Si no hay suficiente memoria, se puede bajar otra página a disco y usar el *frame* liberado. Si se vuelve a necesitar esa página más tarde, se vuelve a cargar.

Gestión de espacio libre con paginación

- El SO mantiene una lista de marcos de página libres.
- Cuando un proceso necesita memoria, se asignan páginas (no necesariamente congiguas) de la *free list* y se mapean mediante una tabla de páginas.
- Cada proceso tiene su propia tabla de páginas, que es consultada por la MMU al intentar acceder a una dirección virtual.
- Si un proceso ocupa N páginas, en un primer enfoque al menos N frames deben estar disponibles en memoria para poder asignarle.
- Si no hay suficiente memoria, se puede bajar otra página a disco y usar el *frame* liberado. Si se vuelve a necesitar esa página más tarde, se vuelve a cargar.
- Así es como la memoria física parece más grande de lo que es.

Gestión de espacio libre con paginación

- El SO mantiene una lista de marcos de página libres.
- Cuando un proceso necesita memoria, se asignan páginas (no necesariamente congiguas) de la *free list* y se mapean mediante una tabla de páginas.
- Cada proceso tiene su propia tabla de páginas, que es consultada por la MMU al intentar acceder a una dirección virtual.
- Si un proceso ocupa N páginas, en un primer enfoque al menos N frames deben estar disponibles en memoria para poder asignarle.
- Si no hay suficiente memoria, se puede bajar otra página a disco y usar el *frame* liberado. Si se vuelve a necesitar esa página más tarde, se vuelve a cargar.
- Así es como la memoria física parece más grande de lo que es.
- Paginación a demanda: en lugar de cargar el programa entero en memoria física para poder ejecutarlo, cargar sólo las páginas que son necesarias en cada momento.

Gestión de espacio libre con paginación

- El SO mantiene una lista de marcos de página libres.
- Cuando un proceso necesita memoria, se asignan páginas (no necesariamente congiguas) de la *free list* y se mapean mediante una tabla de páginas.
- Cada proceso tiene su propia tabla de páginas, que es consultada por la MMU al intentar acceder a una dirección virtual.
- Si un proceso ocupa N páginas, en un primer enfoque al menos N frames deben estar disponibles en memoria para poder asignarle.
- Si no hay suficiente memoria, se puede bajar otra página a disco y usar el *frame* liberado. Si se vuelve a necesitar esa página más tarde, se vuelve a cargar.
- Así es como la memoria física parece más grande de lo que es.
- Paginación a demanda: en lugar de cargar el programa entero en memoria física para poder ejecutarlo, cargar sólo las páginas que son necesarias en cada momento.
- Esto permite la ejecución de procesos que están sólo parcialmente en memoria.

Menú para hoy

Page Fault

- No todas las páginas están en memoria todo el tiempo.

Page Fault

- No todas las páginas están en memoria todo el tiempo.
- Si se solicita una página que no está cargada, se genera un page fault.

Page Fault

- No todas las páginas están en memoria todo el tiempo.
- Si se solicita una página que no está cargada, se genera un page fault.
- En ese caso, el SO:

Page Fault

- No todas las páginas están en memoria todo el tiempo.
- Si se solicita una página que no está cargada, se genera un page fault.
- En ese caso, el SO:
  1. Determina si la referencia a memoria es válida o no.

Page Fault

- No todas las páginas están en memoria todo el tiempo.
- Si se solicita una página que no está cargada, se genera un page fault.
- En ese caso, el SO:
  1. Determina si la referencia a memoria es válida o no.
  2. Si es inválida, termina el proceso. Si es válida pero no está en memoria, hay que cargarla.

Page Fault

- No todas las páginas están en memoria todo el tiempo.
- Si se solicita una página que no está cargada, se genera un page fault.
- En ese caso, el SO:
  1. Determina si la referencia a memoria es válida o no.
  2. Si es inválida, termina el proceso. Si es válida pero no está en memoria, hay que cargarla.
  3. Encuentra un *frame* libre.

Page Fault

- No todas las páginas están en memoria todo el tiempo.
- Si se solicita una página que no está cargada, se genera un page fault.
- En ese caso, el SO:
  1. Determina si la referencia a memoria es válida o no.
  2. Si es inválida, termina el proceso. Si es válida pero no está en memoria, hay que cargarla.
  3. Encuentra un *frame* libre.
  4. Lee la página en ese *frame*.

Page Fault

- No todas las páginas están en memoria todo el tiempo.
- Si se solicita una página que no está cargada, se genera un page fault.
- En ese caso, el SO:
  1. Determina si la referencia a memoria es válida o no.
  2. Si es inválida, termina el proceso. Si es válida pero no está en memoria, hay que cargarla.
  3. Encuentra un *frame* libre.
  4. Lee la página en ese *frame*.
  5. Actualiza las tablas de páginas.

Page Fault

- No todas las páginas están en memoria todo el tiempo.
- Si se solicita una página que no está cargada, se genera un page fault.
- En ese caso, el SO:
  1. Determina si la referencia a memoria es válida o no.
  2. Si es inválida, termina el proceso. Si es válida pero no está en memoria, hay que cargarla.
  3. Encuentra un *frame* libre.
  4. Lee la página en ese *frame*.
  5. Actualiza las tablas de páginas.
  6. Reaunida la ejecución desde la instrucción que causó el *page fault*.

Page Fault

- No todas las páginas están en memoria todo el tiempo.
- Si se solicita una página que no está cargada, se genera un page fault.
- En ese caso, el SO:
  1. Determina si la referencia a memoria es válida o no.
  2. Si es inválida, termina el proceso. Si es válida pero no está en memoria, hay que cargarla.
  3. Encuentra un *frame* libre.
  4. Lee la página en ese *frame*.
  5. Actualiza las tablas de páginas.
  6. Reanuda la ejecución desde la instrucción que causó el *page fault*.
- Si la memoria está llena, el SO debe desalojar una página siguiendo una política de reemplazo de páginas.

Page Fault

- No todas las páginas están en memoria todo el tiempo.
- Si se solicita una página que no está cargada, se genera un page fault.
- En ese caso, el SO:
  1. Determina si la referencia a memoria es válida o no.
  2. Si es inválida, termina el proceso. Si es válida pero no está en memoria, hay que cargarla.
  3. Encuentra un *frame* libre.
  4. Lee la página en ese *frame*.
  5. Actualiza las tablas de páginas.
  6. Reanuda la ejecución desde la instrucción que causó el *page fault*.
- Si la memoria está llena, el SO debe desalojar una página siguiendo una política de reemplazo de páginas.
- Elegir la página correcta para desalojar es crítico para el rendimiento: una decisión equivocada puede ocasionar repetidos *faults*, lo que produce más accesos a disco.

Algoritmos de reemplazo

- Óptimo: Reemplaza la página que se va a usar más lejos en el futuro. Es teórico, no implementable.

Algoritmos de reemplazo

- Óptimo: Reemplaza la página que se va a usar más lejos en el futuro. Es teórico, no implementable.
- FIFO: *First In - First Out*. Asocia a cada página el tiempo en el que fue cargada en memoria. Cuando se tiene que reemplazar una página, se elige la más antigua.

Algoritmos de reemplazo

- Óptimo: Reemplaza la página que se va a usar más lejos en el futuro. Es teórico, no implementable.
- FIFO: *First In - First Out*. Asocia a cada página el tiempo en el que fue cargada en memoria. Cuando se tiene que reemplazar una página, se elige la más antigua.
- LRU: *Least recently used*. Asocia a cada página el tiempo de la última vez que se usó. Cuando se tiene que reemplazar una página, se elige la que hace más tiempo que no se usa.

Algoritmos de reemplazo

- Óptimo: Reemplaza la página que se va a usar más lejos en el futuro. Es teórico, no implementable.
- FIFO: *First In - First Out*. Asocia a cada página el tiempo en el que fue cargada en memoria. Cuando se tiene que reemplazar una página, se elige la más antigua.
- LRU: *Least recently used*. Asocia a cada página el tiempo de la última vez que se usó. Cuando se tiene que reemplazar una página, se elige la que hace más tiempo que no se usa.
- Second Chance: Rotar a través de las páginas hasta encontrar la que no ha sido referenciada desde la última vez que chequeamos.

Algoritmos de reemplazo

- Óptimo: Reemplaza la página que se va a usar más lejos en el futuro. Es teórico, no implementable.
- FIFO: *First In - First Out*. Asocia a cada página el tiempo en el que fue cargada en memoria. Cuando se tiene que reemplazar una página, se elige la más antigua.
- LRU: *Least recently used*. Asocia a cada página el tiempo de la última vez que se usó. Cuando se tiene que reemplazar una página, se elige la que hace más tiempo que no se usa.
- Second Chance: Rotar a través de las páginas hasta encontrar la que no ha sido referenciada desde la última vez que chequeamos.

  Cuando una página es seleccionada para desalojar, se mira el bit de referenciada. Si es 0, se desaloja. Si es 1, se le da una segunda oportunidad, se limpia el bit de referenciada, y se actualiza el tiempo de llegada con el tiempo actual.

Algoritmos de reemplazo

Ejercicio

Enunciado

Tengo un sistema con 6 páginas y sólo 4 marcos de página. La memoria comienza vacía.
Llegan los siguientes pedidos de memoria (número de página) en ese orden:

1, 2, 1, 3, 4, 3, 5, 6, 2

Indicar qué página se desaloja tras cada pedido utilizando el algoritmo FIFO, LRU y SC.

Algoritmos de reemplazo

Ejercicio

Solución FIFO

Pag Pedidas	Frames				Pags a desalojar
1					
2					
1					
3					
4					
3					
5					
6					
2					

Algoritmos de reemplazo

Ejercicio

Solución FIFO

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2					
1					
3					
4					
3					
5					
6					
2					

Algoritmos de reemplazo

Ejercicio

Solución FIFO

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1					
3					
4					
3					
5					
6					
2					

Algoritmos de reemplazo

Ejercicio

Solución FIFO

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1	**1**	2			1 2
3					
4					
3					
5					
6					
2					

Solución FIFO

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1	**1**	2			1 2
3	1	2	**3**		1 2 3
4					
3					
5					
6					
2					

Solución FIFO

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1	**1**	2			1 2
3	1	2	**3**		1 2 3
4	1	2	3	**4**	1 2 3 4
3					
5					
6					
2					

Solución FIFO

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1	**1**	2			1 2
3	1	2	**3**		1 2 3
4	1	2	3	**4**	1 2 3 4
3	1	2	**3**	4	1 2 3 4
5					
6					
2					

Solución FIFO

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1	**1**	2			1 2
3	1	2	**3**		1 2 3
4	1	2	3	**4**	1 2 3 4
3	1	2	**3**	4	1 2 3 4
5	**5**	2	3	4	2 3 4 5
6					
2					

Solución FIFO

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1	**1**	2			1 2
3	1	2	**3**		1 2 3
4	1	2	3	**4**	1 2 3 4
3	1	2	**3**	4	1 2 3 4
5	**5**	2	3	4	2 3 4 5
6	5	**6**	3	4	3 4 5 6
2					

Solución FIFO

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1	**1**	2			1 2
3	1	2	**3**		1 2 3
4	1	2	3	**4**	1 2 3 4
3	1	2	**3**	4	1 2 3 4
5	**5**	2	3	4	2 3 4 5
6	5	**6**	3	4	3 4 5 6
2	5	6	**2**	4	4 5 6 2

Solución LRU

Pag Pedidas	Frames				Pags a desalojar
1					
2					
1					
3					
4					
3					
5					
6					
2					

Solución LRU

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2					
1					
3					
4					
3					
5					
6					
2					

Algoritmos de reemplazo

Ejercicio

Solución LRU

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1					
3					
4					
3					
5					
6					
2					

Algoritmos de reemplazo

Ejercicio

Solución LRU

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1	**1**	2			2 1
3					
4					
3					
5					
6					
2					

Solución LRU

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1	**1**	2			2 1
3	1	2	**3**		2 1 3
4					
3					
5					
6					
2					

Solución LRU

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1	**1**	2			2 1
3	1	2	**3**		2 1 3
4	1	2	3	**4**	2 1 3 4
3					
5					
6					
2					

Solución LRU

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1	**1**	2			2 1
3	1	2	**3**		2 1 3
4	1	2	3	**4**	2 1 3 4
3	1	2	**3**	4	2 1 4 3
5					
6					
2					

Solución LRU

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1	**1**	2			2 1
3	1	2	**3**		2 1 3
4	1	2	3	**4**	2 1 3 4
3	1	2	**3**	4	2 1 4 3
5	1	**5**	3	4	1 4 3 5
6					
2					

Solución LRU

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1	**1**	2			2 1
3	1	2	**3**		2 1 3
4	1	2	3	**4**	2 1 3 4
3	1	2	**3**	4	2 1 4 3
5	1	**5**	3	4	1 4 3 5
6	**6**	5	3	4	4 3 5 6
2					

Solución LRU

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1	**1**	2			2 1
3	1	2	**3**		2 1 3
4	1	2	3	**4**	2 1 3 4
3	1	2	**3**	4	2 1 4 3
5	1	**5**	3	4	1 4 3 5
6	**6**	5	3	4	4 3 5 6
2	6	5	3	**2**	3 5 6 2

Algoritmos de reemplazo

Ejercicio

Solución Second Chance

Pag Pedidas	Frames				Pags a desalojar
1					
2					
1					
3					
4					
3					
5					
6					
2					

Algoritmos de reemplazo

Ejercicio

Solución Second Chance

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2					
1					
3					
4					
3					
5					
6					
2					

Solución Second Chance

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1					
3					
4					
3					
5					
6					
2					

Algoritmos de reemplazo

Ejercicio

Solución Second Chance

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1	**1**	2			1 2
3					
4					
3					
5					
6					
2					

Solución Second Chance

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1	**1**	2			1 2
3	1	2	**3**		1 2 3
4					
3					
5					
6					
2					

Solución Second Chance

Pag Pedidas	Frames				Pags a desalojar
1	**1**				1
2	1	**2**			1 2
1	**1**	2			1 2
3	1	2	**3**		1 2 3
4	1	2	3	**4**	1 2 3 4
3					
5					
6					
2					

Solución Second Chance

Pag Pedidas	Frames				Pags a desalojar			
1	**1**				1			
2	1	**2**			1	2		
1	**1**	2			1	2		
3	1	2	**3**		1	2	3	
4	1	2	3	**4**	1	2	3	4
3	1	2	**3**	4	1	2	3	4
5								
6								
2								

Solución Second Chance

Pag Pedidas	Frames				Pags a desalojar			
1	**1**				1			
2	1	**2**			1	2		
1	**1**	2			1	2		
3	1	2	**3**		1	2	3	
4	1	2	3	**4**	1	2	3	4
3	1	2	**3**	4	1	2	3	4
5	1	2	3	4	2	3	4	1
	1	**5**	3	4	3	4	1	5
6								
2								

Solución Second Chance

Pag Pedidas	Frames				Pags a desalojar			
1	**1**				1			
2	1	**2**			1	2		
1	**1**	2			1	2		
3	1	2	**3**		1	2	3	
4	1	2	3	**4**	1	2	3	4
3	1	2	**3**	4	1	2	3	4
5	1	2	3	4	2	3	4	1
	1	**5**	3	4	3	4	1	5
6	1	5	3	4	4	1	5	3
	1	5	3	**6**	1	5	3	6
2								

Solución Second Chance

Pag Pedidas	Frames				Pags a desalojar			
1	**1**				1			
2	1	**2**			1	2		
1	**1**	2			1	2		
3	1	2	**3**		1	2	3	
4	1	2	3	**4**	1	2	3	4
3	1	2	**3**	4	1	2	3	4
5	1	2	3	4	2	3	4	1
	1	**5**	3	4	3	4	1	5
6	1	5	3	4	4	1	5	3
	1	5	3	**6**	1	5	3	6
2	**2**	5	3	6	5	3	6	2

Algoritmos de reemplazo

¿Qué algoritmo de reemplazo de páginas es es mejor?

Algoritmos de reemplazo

¿Qué algoritmo de reemplazo de páginas es es mejor?

- Los algoritmos se evalúan ejecutándolos sobre una secuencia particular de referencias a memoria y computando el número de *page-faults*.

Algoritmos de reemplazo

¿Qué algoritmo de reemplazo de páginas es es mejor?

- Los algoritmos se evalúan ejecutándolos sobre una secuencia particular de referencias a memoria y computando el número de *page-faults*.
- En general queremos el que tenga la menor tasa de *page-faults*.

Algoritmos de reemplazo

¿Qué algoritmo de reemplazo de páginas es es mejor?

- Los algoritmos se evalúan ejecutándolos sobre una secuencia particular de referencias a memoria y computando el número de *page-faults*.
- En general queremos el que tenga la menor tasa de *page-faults*.
- Page Fault Rate $= \frac{\text{páginas qué pedí y no estaban cargadas en memoria}}{\text{páginas totales}}$

Algoritmos de reemplazo

¿Qué algoritmo de reemplazo de páginas es es mejor?

- Los algoritmos se evalúan ejecutándolos sobre una secuencia particular de referencias a memoria y computando el número de *page-faults*.
- En general queremos el que tenga la menor tasa de *page-faults*.
- Page Fault Rate $= \frac{\text{páginas qué pedí y no estaban cargadas en memoria}}{\text{páginas totales}}$

Algoritmos de reemplazo

¿Qué algoritmo de reemplazo de páginas es es mejor?

- Los algoritmos se evalúan ejecutándolos sobre una secuencia particular de referencias a memoria y computando el número de *page-faults*.
- En general queremos el que tenga la menor tasa de *page-faults*.
- Page Fault Rate $= \frac{\text{páginas qué pedí y no estaban cargadas en memoria}}{\text{páginas totales}}$

Ejercicio

Para los ejercicios anteriores, calcular el *Page fault rate* en cada caso.

Algoritmos de reemplazo

¿Qué algoritmo de reemplazo de páginas es es mejor?

- Los algoritmos se evalúan ejecutándolos sobre una secuencia particular de referencias a memoria y computando el número de *page-faults*.
- En general queremos el que tenga la menor tasa de *page-faults*.
- Page Fault Rate $= \frac{\text{páginas qué pedí y no estaban cargadas en memoria}}{\text{páginas totales}}$

Ejercicio

Para los ejercicios anteriores, calcular el *Page fault rate* en cada caso.

Solución

- Page Fault Rate (FIFO) $= 7 \ / \ 9$
- Page Fault Rate (LRU) $= 7 \ / \ 9$
- Page Fault Rate (SC) $= 7 \ / \ 9$

Dirty bit

- Escribir a disco es lento.

Dirty bit

- Escribir a disco es lento.
- Si una página no fue modificada, se puede desalojar sin escribirla a disco.

Dirty bit

- Escribir a disco es lento.
- Si una página no fue modificada, se puede desalojar sin escribirla a disco.
- Una página que sí fue modificada, debe ser actualizada en disco.

Dirty bit

- Escribir a disco es lento.
- Si una página no fue modificada, se puede desalojar sin escribirla a disco.
- Una página que sí fue modificada, debe ser actualizada en disco.
- El dirty bit se utiliza para indicar que una página fue modificada.

Dirty bit

- Escribir a disco es lento.
- Si una página no fue modificada, se puede desalojar sin escribirla a disco.
- Una página que sí fue modificada, debe ser actualizada en disco.
- El dirty bit se utiliza para indicar que una página fue modificada.
- Se puede utilizar el *dirty bit* para optimizar los algoritmos de reemplazo (ej: desalojar primero las páginas no modificadas).

Thrashing

- Un proceso hace thrashing si pasa más tiempo cargando y descargando páginas que ejecutando.

Thrashing

- Un proceso hace thrashing si pasa más tiempo cargando y descargando páginas que ejecutando.
- Ejemplo real:

Thrashing

- Un proceso hace thrashing si pasa más tiempo cargando y descargando páginas que ejecutando.
- Ejemplo real:
  1. Si el uso de CPU es muy bajo, el *scheduler* introduce más procesos al sistema.

Thrashing

- Un proceso hace thrashing si pasa más tiempo cargando y descargando páginas que ejecutando.
- Ejemplo real:
  1. Si el uso de CPU es muy bajo, el *scheduler* introduce más procesos al sistema.
  2. Supongamos que un proceso empieza a necesitar más frames.

Thrashing

- Un proceso hace thrashing si pasa más tiempo cargando y descargando páginas que ejecutando.
- Ejemplo real:
  1. Si el uso de CPU es muy bajo, el *scheduler* introduce más procesos al sistema.
  2. Supongamos que un proceso empieza a necesitar más frames.
  3. Con un enfoque de reemplazo de paginas global (reemplazando páginas sin importar a qué proceso pertenecen) el proceso empieza a fallar y a tomar frames de otros procesos.

Thrashing

- Un proceso hace thrashing si pasa más tiempo cargando y descargando páginas que ejecutando.
- Ejemplo real:
  1. Si el uso de CPU es muy bajo, el *scheduler* introduce más procesos al sistema.
  2. Supongamos que un proceso empieza a necesitar más frames.
  3. Con un enfoque de reemplazo de paginas global (reemplazando páginas sin importar a qué proceso pertenecen) el proceso empieza a fallar y a tomar frames de otros procesos.
  4. Esos procesos necesitan esas páginas así que fallan y toman frames de otros procesos.

Thrashing

- Un proceso hace thrashing si pasa más tiempo cargando y descargando páginas que ejecutando.
- Ejemplo real:
  1. Si el uso de CPU es muy bajo, el *scheduler* introduce más procesos al sistema.
  2. Supongamos que un proceso empieza a necesitar más frames.
  3. Con un enfoque de reemplazo de paginas global (reemplazando páginas sin importar a qué proceso pertenecen) el proceso empieza a fallar y a tomar frames de otros procesos.
  4. Esos procesos necesitan esas páginas así que fallan y toman frames de otros procesos.
  5. Y así, y así...

Thrashing

- Un proceso hace thrashing si pasa más tiempo cargando y descargando páginas que ejecutando.
- Ejemplo real:
  1. Si el uso de CPU es muy bajo, el *scheduler* introduce más procesos al sistema.
  2. Supongamos que un proceso empieza a necesitar más frames.
  3. Con un enfoque de reemplazo de paginas global (reemplazando páginas sin importar a qué proceso pertenecen) el proceso empieza a fallar y a tomar frames de otros procesos.
  4. Esos procesos necesitan esas páginas así que fallan y toman frames de otros procesos.
  5. Y así, y así...
  6. Mientras están todos esperando que se atiendan sus *page-faults*, la cola *ready* para ejecutar se vacía y decrementa el uso de CPU.

Thrashing

- Un proceso hace thrashing si pasa más tiempo cargando y descargando páginas que ejecutando.
- Ejemplo real:
  1. Si el uso de CPU es muy bajo, el *scheduler* introduce más procesos al sistema.
  2. Supongamos que un proceso empieza a necesitar más frames.
  3. Con un enfoque de reemplazo de paginas global (reemplazando páginas sin importar a qué proceso pertenecen) el proceso empieza a fallar y a tomar frames de otros procesos.
  4. Esos procesos necesitan esas páginas así que fallan y toman frames de otros procesos.
  5. Y así, y así...
  6. Mientras están todos esperando que se atiendan sus *page-faults*, la cola *ready* para ejecutar se vacía y decrementa el uso de CPU.
  7. El *scheduler* ve esta baja en el uso de CPU e incrementa el grado de multiprogramación.

Thrashing

- Un proceso hace thrashing si pasa más tiempo cargando y descargando páginas que ejecutando.
- Ejemplo real:
  1. Si el uso de CPU es muy bajo, el *scheduler* introduce más procesos al sistema.
  2. Supongamos que un proceso empieza a necesitar más frames.
  3. Con un enfoque de reemplazo de paginas global (reemplazando páginas sin importar a qué proceso pertenecen) el proceso empieza a fallar y a tomar frames de otros procesos.
  4. Esos procesos necesitan esas páginas así que fallan y toman frames de otros procesos.
  5. Y así, y así...
  6. Mientras están todos esperando que se atiendan sus *page-faults*, la cola *ready* para ejecutar se vacía y decrementa el uso de CPU.
  7. El *scheduler* ve esta baja en el uso de CPU e incrementa el grado de multiprogramación.
  8. Y todo empeora. Y el sistema colapsa.

Thrashing

- Mitigación:

Thrashing

- Mitigación:
  - En una computadora personal, un usuario puede darse cuenta de que está ocurriendo *thrashing* y matar algunos procesos “a mano”.

Thrashing

- Mitigación:
  - En una computadora personal, un usuario puede darse cuenta de que está ocurriendo *thrashing* y matar algunos procesos "a mano".
  - Desde el SO, se pueden limitar los efectos del *thrashing* usando un reemplazo de páginas local, es decir, que cada proceso sólo pueda tomar *frames* de los que ya tiene asignados, y no pueda "robarle" a otro proceso.

Thrashing

- Mitigación:
  - En una computadora personal, un usuario puede darse cuenta de que está ocurriendo *thrashing* y matar algunos procesos "a mano".
  - Desde el SO, se pueden limitar los efectos del *thrashing* usando un reemplazo de páginas local, es decir, que cada proceso sólo pueda tomar *frames* de los que ya tiene asignados, y no pueda "robarle" a otro proceso.
  - Algunos SO corren un proceso Out-Of-Memory Killer.

Thrashing

- Mitigación:
  - En una computadora personal, un usuario puede darse cuenta de que está ocurriendo *thrashing* y matar algunos procesos "a mano".
  - Desde el SO, se pueden limitar los efectos del *thrashing* usando un reemplazo de páginas local, es decir, que cada proceso sólo pueda tomar *frames* de los que ya tiene asignados, y no pueda "robarle" a otro proceso.
  - Algunos SO corren un proceso Out-Of-Memory Killer.
  - Pero para prevenir el *thrashing* deberíamos proveer a un proceso de tantos *frames* como necesite.

Principio de localidad

- Una localidad es un conjunto de páginas que se usan activamente al mismo tiempo.

Principio de localidad

- Una localidad es un conjunto de páginas que se usan activamente al mismo tiempo.
- Localidad temporal: las páginas más recientemente usadas tienden a ser reusadas en el corto plazo.

Principio de localidad

- Una localidad es un conjunto de páginas que se usan activamente al mismo tiempo.
- Localidad temporal: las páginas más recientemente usadas tienden a ser reusadas en el corto plazo.
- Localidad espacial: las direcciones cercanas entre sí suelen accederse juntas.

Principio de localidad

- Una localidad es un conjunto de páginas que se usan activamente al mismo tiempo.
- Localidad temporal: las páginas más recientemente usadas tienden a ser reusadas en el corto plazo.
- Localidad espacial: las direcciones cercanas entre sí suelen accederse juntas.
- La localidad es clave para el diseño inteligente de reemplazos de páginas.

Principio de localidad

- Una localidad es un conjunto de páginas que se usan activamente al mismo tiempo.
- Localidad temporal: las páginas más recientemente usadas tienden a ser reusadas en el corto plazo.
- Localidad espacial: las direcciones cercanas entre sí suelen accederse juntas.
- La localidad es clave para el diseño inteligente de reemplazos de páginas.
  - Supongamos que asignamos suficientes *frames* a un proceso para acomodar su localidad actual.

Principio de localidad

- Una localidad es un conjunto de páginas que se usan activamente al mismo tiempo.
- Localidad temporal: las páginas más recientemente usadas tienden a ser reusadas en el corto plazo.
- Localidad espacial: las direcciones cercanas entre sí suelen accederse juntas.
- La localidad es clave para el diseño inteligente de reemplazos de páginas.
  - Supongamos que asignamos suficientes *frames* a un proceso para acomodar su localidad actual.
  - Fallará por páginas en esa localidad hasta que estén todas en memoria, y luego ya no fallará hasta que cambie de localidad.

Principio de localidad

- Una localidad es un conjunto de páginas que se usan activamente al mismo tiempo.
- Localidad temporal: las páginas más recientemente usadas tienden a ser reusadas en el corto plazo.
- Localidad espacial: las direcciones cercanas entre sí suelen accederse juntas.
- La localidad es clave para el diseño inteligente de reemplazos de páginas.
  - Supongamos que asignamos suficientes *frames* a un proceso para acomodar su localidad actual.
  - Fallará por páginas en esa localidad hasta que estén todas en memoria, y luego ya no fallará hasta que cambie de localidad.
  - Si no asignamos suficientes frames para el tamaño de su localidad actual, el proceso hará *thrashing*.

Principio de localidad

- Una localidad es un conjunto de páginas que se usan activamente al mismo tiempo.
- Localidad temporal: las páginas más recientemente usadas tienden a ser reusadas en el corto plazo.
- Localidad espacial: las direcciones cercanas entre sí suelen accederse juntas.
- La localidad es clave para el diseño inteligente de reemplazos de páginas.
  - Supongamos que asignamos suficientes *frames* a un proceso para acomodar su localidad actual.
  - Fallará por páginas en esa localidad hasta que estén todas en memoria, y luego ya no fallará hasta que cambie de localidad.
  - Si no asignamos suficientes frames para el tamaño de su localidad actual, el proceso hará *thrashing*.
- Algoritmos como LRU tienden a funcionar bien porque aproximan estos patrones de uso.

Momento para preguntas

Menú para hoy

Resumen

Hoy vimos...

- Repaso de memoria virtual

Resumen

Hoy vimos...

- Repaso de memoria virtual
  - Base y límite

Resumen

Hoy vimos...

- Repaso de memoria virtual
  - Base y límite
  - Segmentación

Resumen

Hoy vimos...

- Repaso de memoria virtual
  - Base y límite
  - Segmentación
  - Paginación

Resumen

Hoy vimos...

- Repaso de memoria virtual
  - Base y límite
  - Segmentación
  - Paginación
  - Problemas de fragmentación

Resumen

Hoy vimos...

- Repaso de memoria virtual
  - Base y límite
  - Segmentación
  - Paginación
  - Problemas de fragmentación
- Memory API

Resumen

Hoy vimos...

- Repaso de memoria virtual
  - Base y límite
  - Segmentación
  - Paginación
  - Problemas de fragmentación
- Memory API
  - Memoria Stack vs Memoria Heap

Resumen

Hoy vimos...

- Repaso de memoria virtual
  - Base y límite
  - Segmentación
  - Paginación
  - Problemas de fragmentación
- Memory API
  - Memoria Stack vs Memoria Heap
  - `malloc()` y `free()`

Resumen

Hoy vimos...

- Repaso de memoria virtual
  - Base y límite
  - Segmentación
  - Paginación
  - Problemas de fragmentación
- Memory API
  - Memoria Stack vs Memoria Heap
  - `malloc()` y `free()`
- Gestión del espacio libre

Resumen

Hoy vimos...

- Repaso de memoria virtual
  - Base y límite
  - Segmentación
  - Paginación
  - Problemas de fragmentación
- Memory API
  - Memoria Stack vs Memoria Heap
  - `malloc()` y `free()`
- Gestión del espacio libre
  - Estrategias de asignación

Resumen

Hoy vimos...

- Repaso de memoria virtual
  - Base y límite
  - Segmentación
  - Paginación
  - Problemas de fragmentación
- Memory API
  - Memoria Stack vs Memoria Heap
  - `malloc()` y `free()`
- Gestión del espacio libre
  - Estrategias de asignación
  - Splitting y coalescing

Resumen

Hoy vimos...

- Repaso de memoria virtual
  - Base y límite
  - Segmentación
  - Paginación
  - Problemas de fragmentación
- Memory API
  - Memoria Stack vs Memoria Heap
  - `malloc()` y `free()`
- Gestión del espacio libre
  - Estrategias de asignación
  - Splitting y coalescing
  - Paginación a demanda

Resumen

Hoy vimos...

- Repaso de memoria virtual
  - Base y límite
  - Segmentación
  - Paginación
  - Problemas de fragmentación
- Memory API
  - Memoria Stack vs Memoria Heap
  - `malloc()` y `free()`
- Gestión del espacio libre
  - Estrategias de asignación
  - Splitting y coalescing
  - Paginación a demanda
- Reemplazo de páginas

Resumen

Hoy vimos...

- Repaso de memoria virtual
  - Base y límite
  - Segmentación
  - Paginación
  - Problemas de fragmentación
- Memory API
  - Memoria Stack vs Memoria Heap
  - `malloc()` y `free()`
- Gestión del espacio libre
  - Estrategias de asignación
  - Splitting y coalescing
  - Paginación a demanda
- Reemplazo de páginas
  - Page Fault

Resumen

Hoy vimos...

- Repaso de memoria virtual
  - Base y límite
  - Segmentación
  - Paginación
  - Problemas de fragmentación
- Memory API
  - Memoria Stack vs Memoria Heap
  - `malloc()` y `free()`
- Gestión del espacio libre
  - Estrategias de asignación
  - Splitting y coalescing
  - Paginación a demanda
- Reemplazo de páginas
  - Page Fault
  - Algoritmos de reemplazo

Resumen

Hoy vimos...

- Repaso de memoria virtual
  - Base y límite
  - Segmentación
  - Paginación
  - Problemas de fragmentación
- Memory API
  - Memoria Stack vs Memoria Heap
  - `malloc()` y `free()`
- Gestión del espacio libre
  - Estrategias de asignación
  - Splitting y coalescing
  - Paginación a demanda
- Reemplazo de páginas
  - Page Fault
  - Algoritmos de reemplazo
  - Thrashing y localidad

Resumen

Cómo seguimos...

Con esto se puede resolver toda la guía práctica 4.