Administración de memoria

Gisela Confalonieri

Departamento de Computación - FCEyN

25 de abril de 2024

Estructura de la clase

Motivación

La CPU sólo puede cargar instrucciones desde memoria, así que cualquier programa primero se tiene que cargar en memoria para ejecutar.

Los programas y datos viven en almacenamiento secundario, y deben cargarse memoria para ser utilizados por la CPU.

Con multiprogramación, ¡se cargan muchos programas! $\rightarrow$ necesitamos **memory management.**

Administración de memoria

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

Repaso: Memoria física vs Memoria lógica

Dirección de memoria lógica: la que ve la CPU.

Dirección de memoria física: la que ve la unidad de memoria.

Espacio de memoria lógica: conjunto de direcciones lógicas.

Espacio de memoria física: conjunto de direcciones físicas correspondientes a estas direcciones lógicas.

El programa de usuario **nunca** accede la dirección física real.
El programa de usuario trabaja con direcciones lógicas.

Asignación de memoria contigua

Esquema de memoria contigua

Se asignan los procesos a partes de memoria de tamaño variable, donde cada parte contiene exactamente un proceso.

El sistema operativo mantiene una estructura (puede ser un bitmap o una lista enlazada) indicando qué partes de la memoria están disponibles y cuáles están ocupados.

Cuando un proceso entra, el SO se fija sus requerimientos de memoria y la cantidad de memoria disponible.

Cuando un proceso termina, libera su memoria.

Estrategias de asignación de memoria contigua

Los espacios de memoria disponible se ven como un conjunto de "huecos" de varios tamaños dispersos por la memoria. Cuando un proceso llega y necesita memoria, el sistema busca un "hueco" con espacio suficiente para este proceso.

Estrategias para asignar memoria:

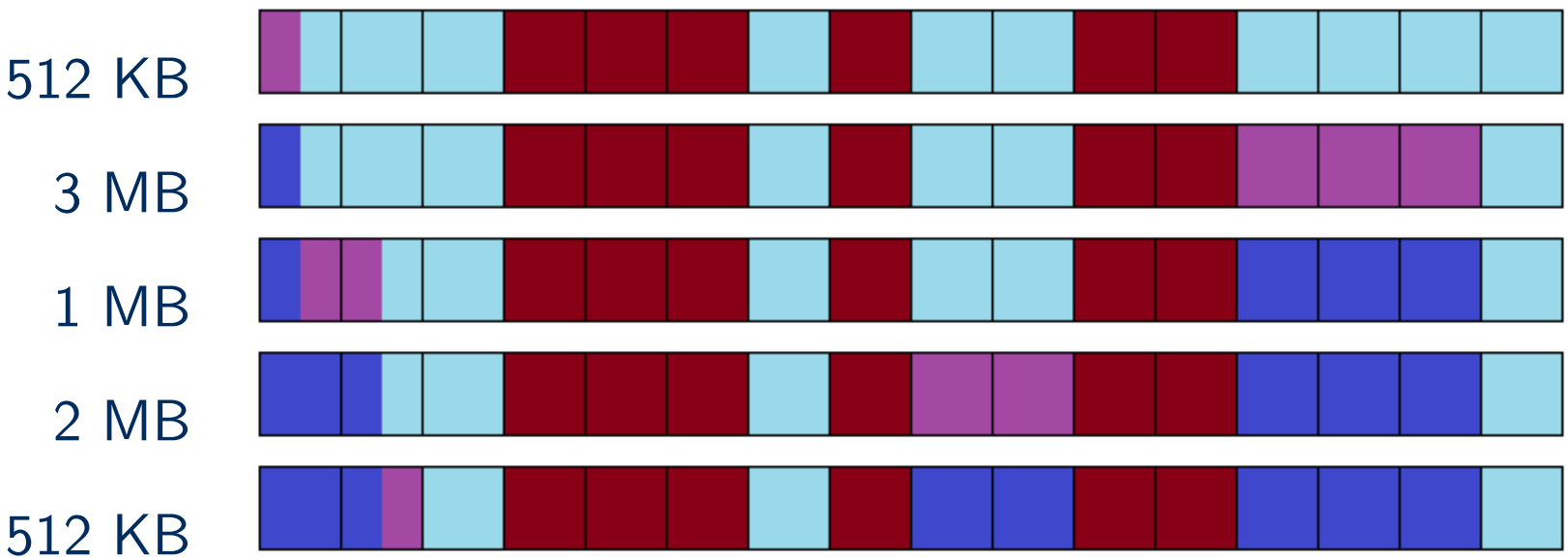
First-fit. Asigna el primer "hueco" lo suficientemente grande.
Best-fit. Asigna el "hueco" más chico que sea lo suficientemente grande.
Worst-fit. Asigna el "hueco" más grande.

Ejercicio:

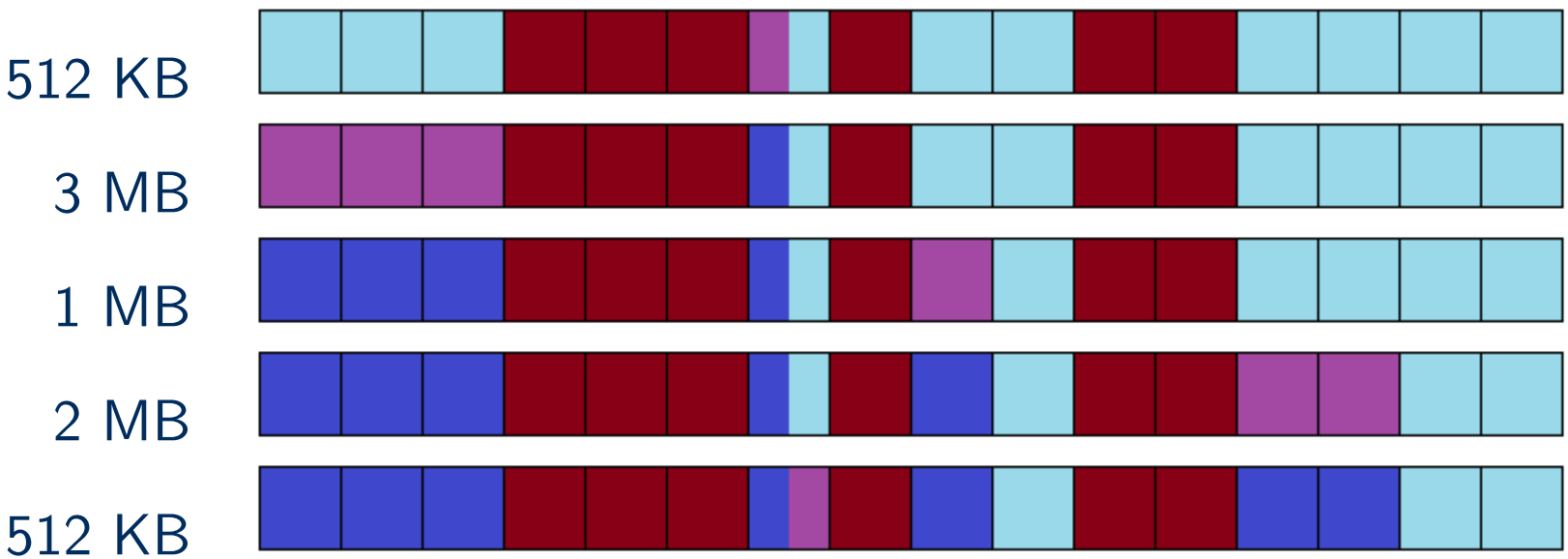
Tengo un sistema con 16 MB de memoria **sin particionar** que direcciona a byte. El estado actual de la memoria es el siguiente (cuadrado= 1MB):

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.

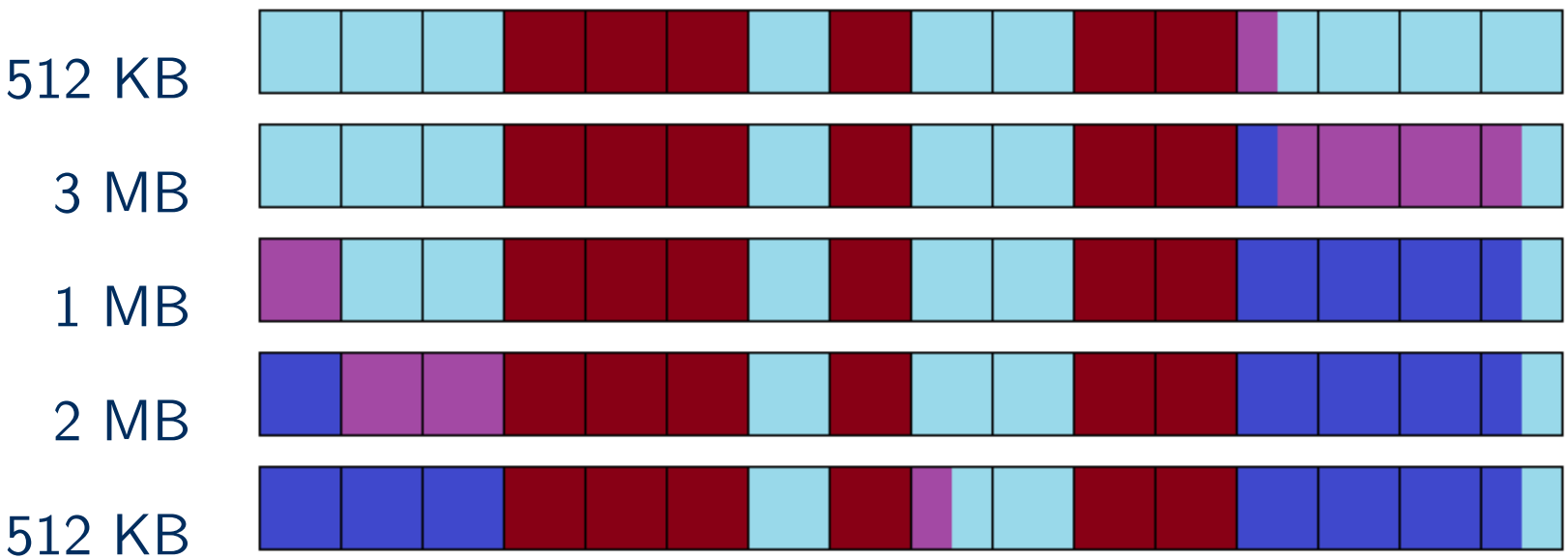
Ejercicio: Solución First-Fit



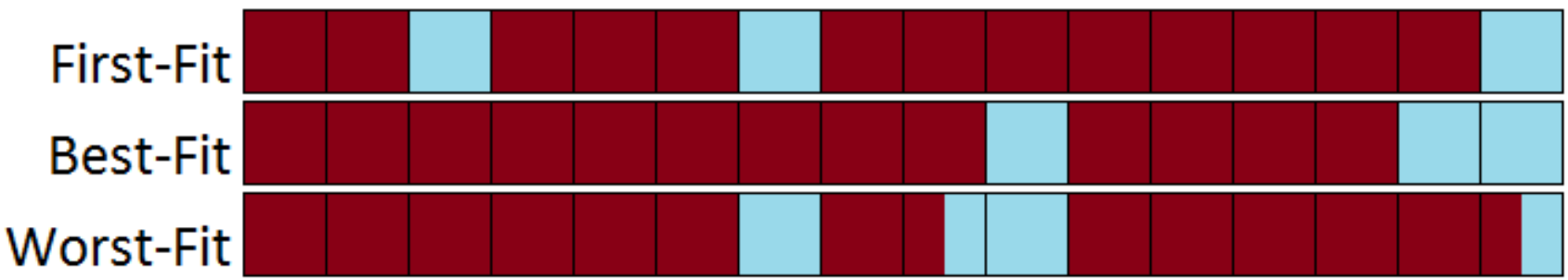
Ejercicio: Solución Best-Fit



Ejercicio: Solución Worst-Fit



Ejercicio: Entonces, ¿Cuál es mejor?



Fragmentación

Mientras los procesos se cargan y quitan de memoria, el espacio de memoria libre se fragmenta en pedacitos.

¿Y si hay suficiente memoria para satisfacer un pedido pero los espacios disponibles no son contiguos? → **fragmentación externa**.

¿Cómo evitar esto?

1. **Partir la memoria física en bloques de tamaño fijo** y asignar memoria en unidades basadas en el tamaño de bloque.
2. **Permitir que el espacio de direcciones físicas de los procesos no sea contiguo**, permitiendo que a un proceso se le asigne memoria siempre que haya disponible.

Paginación

La memoria física se particiona en bloques de tamaño fijo llamados **marcos** o **frames**, y la memoria lógica se particiona en bloques del mismo tamaño llamados **páginas. Las páginas de un proceso se cargan en marcos de memoria.**

Permite que el espacio de direcciones de memoria física de un proceso sea no contiguo.

Evita la fragmentación externa: cualquier marco libre puede ser asignado a un proceso que lo necesite.

Tiene fragmentación interna: Como los frames se asignan como unidades, el último frame asignado puede no estar completamente lleno.

Páginas compartidas

Ejemplo: biblioteca estándar de C (libc)

- La mayoría de los procesos de usuario requieren la libc.
- Suponer que la libc ocupa 2MB, y que un sistema tiene 40 procesos de usuario que la usan.
- Si cada proceso carga su propia copia de libc en su espacio de direcciones... se requeriría 80MB de memoria.

Con paginación, el **código reentrante** (que nunca cambia durante la ejecución) puede ser **compartido**, y dos o más procesos pueden **ejecutar el mismo código al mismo tiempo**.

- Sólo una copia de la libc necesita estar en memoria física: **la tabla de páginas para cada proceso de usuario mapea a la misma copia física de la libc.**
- Entonces, para soportar 40 procesos se requeire 2MB en lugar de 80MB.

Memoria virtual

Si un proceso ocupa n páginas, en un primer enfoque al menos n frames deben estar disponibles en memoria para poder asignarle $\rightarrow$ esto limita el tamaño de un programa al tamaño de la memoria física.

Sin embargo, los programas no siempre se necesitan completos (opciones que no se usan tanto o no todas al mismo tiempo, manejo de errores inusuales, etc).

Esquema de memoria virtual

Permite la **ejecución de procesos que están sólo parcialmente en memoria**. Así, los programas pueden ser más grandes que la cantidad de memoria física disponible. Además, como cada programa toma menos memoria física, más programas pueden correr al mismo tiempo.

Demand Paging

En lugar de cargar el programa entero en memoria física para poder ejecutarlo, **cargar sólo las páginas que son necesarias en cada momento**.

Como algunas páginas estarán en memoria y algunas en almacenamiento secundario, necesitamos soporte de hardware (ejemplo: bit válido-inválido).

Intentar acceder a una página que no está en memoria causa un *page-fault*.

Mientras no haya *page-faults*, el tiempo de acceso efectivo es igual al tiempo de acceso a memoria.
Si ocurre un *page-fault*, se debe primero leer la página desde almacenamiento secundario y luego accederla en memoria.
Entonces, **el tiempo de acceso efectivo es directamente proporcional a la tasa de page-faults**.

Reemplazo de páginas

Mientras un proceso está ejecutando, ocurre un *page-fault*. El sistema operativo determina dónde está la página deseada pero se encuentra con que ya no hay frames libres, toda la memoria está en uso. ¿Qué puede hacer?

Matar el proceso =(. No mejora la utilización de CPU ni el trhoughput. Y la paginación es invisible a los usuarios, no deberían darse cuenta.

Intercambiar páginas =). Liberar frames ocupados para poder cargar las páginas solicitadas.

Utilizando **reemplazo de páginas**, si no hay frames libres se requieren 2 transferencias de páginas (una para la que sale y una para la que entra). Esto duplica el tiempo para atender *page-faults* y el tiempo de acceso efectivo.

Mejora: usar un bit de modificación (*dirty bit*), y sólo escribir la página a disco si fue modificada desde que se leyó a memoria.

Reemplazo de páginas

Mecanismo (resumidísimo) para atender un *page-fault* con reemplazo de páginas:

1. Encontrar la página deseada en almacenamiento secundario.
2. Encontrar un frame libre:
   - 2.1 Si hay un frame libre, usarlo.
   - 2.2 Si no hay un frame libre, usar un algoritmo de reemplazo de páginas para seleccionar un frame *víctima*.
   - 2.3 Escribir el contenido del frame *víctima* a almacenamiento secundario (si hace falta) (en general, a espacio de swap); actualizar la tabla de páginas (y demás tablas).
3. Leer la página deseada en el recién liberado frame; actualizar la tabla de páginas (y demás tablas).
4. Continuar el proceso desde donde ocurrió el *page-fault*.

Algoritmos de reemplazo

FIFO: 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: 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: Es como FIFO, pero 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

¿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*.

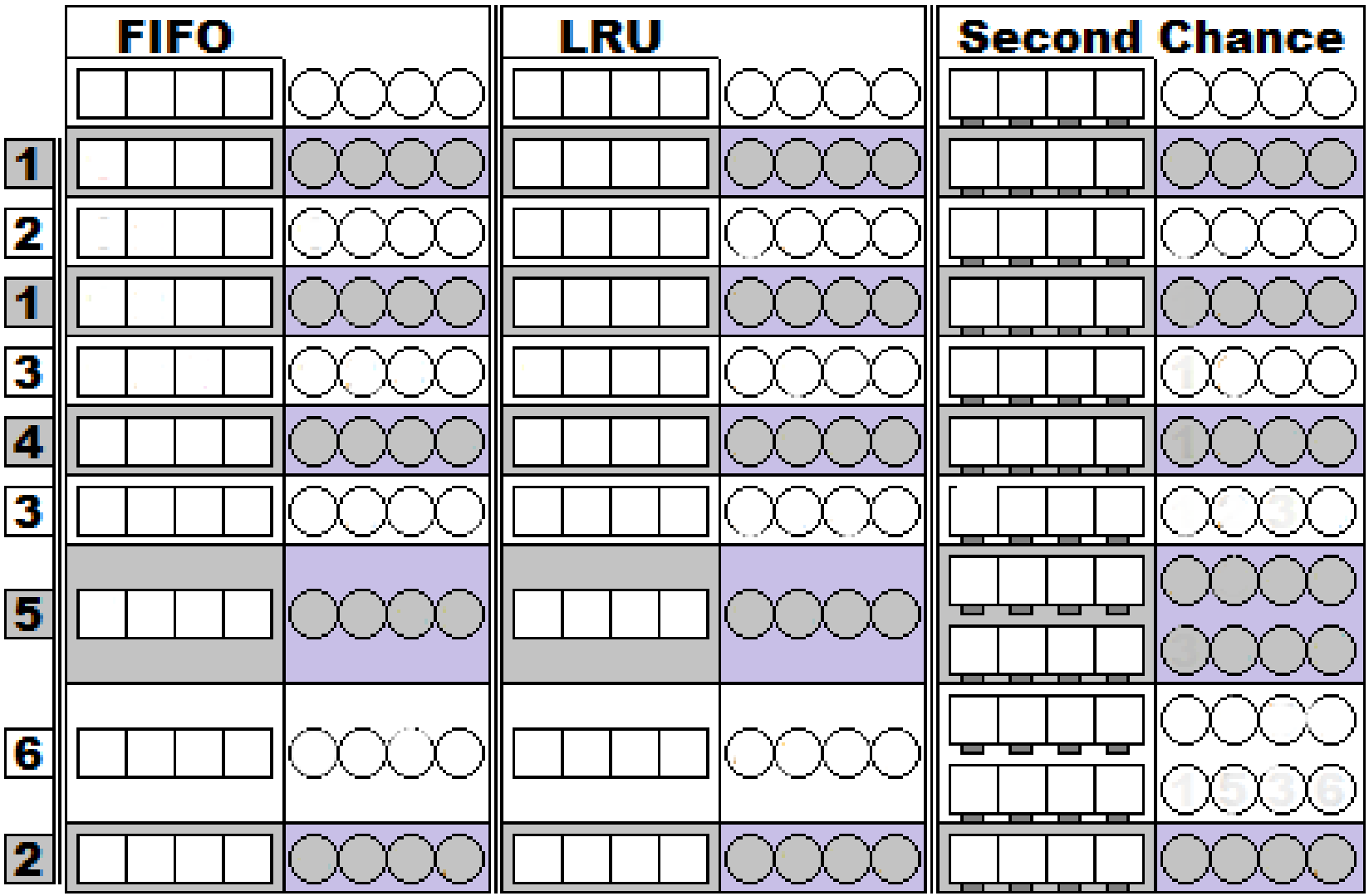
Ejercicio:

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:

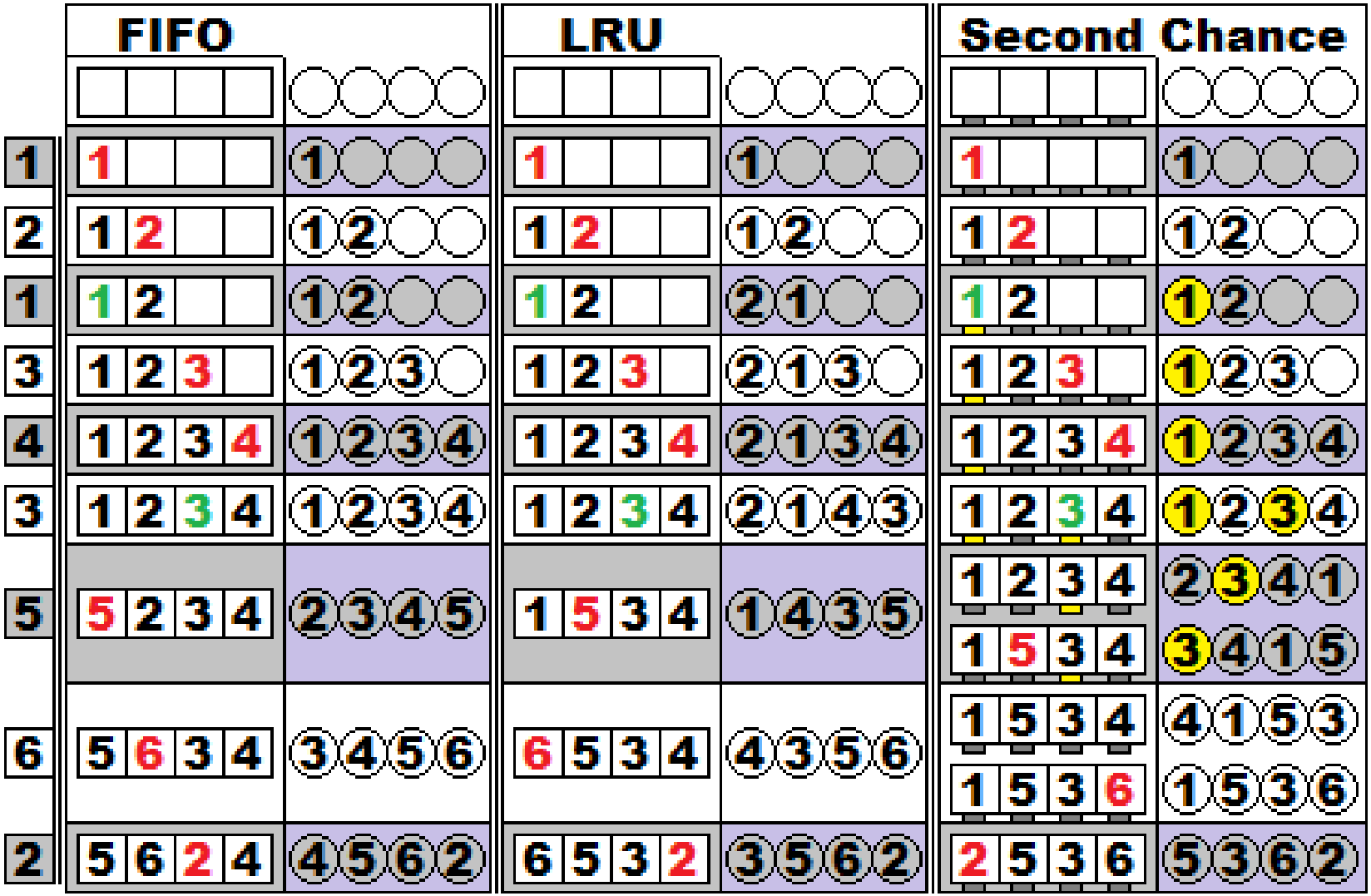
1, 2, 1, 3, 4, 3, 5, 6, 2

Indicar qué página se desaloja tras cada pedido utilizando los algoritmos FIFO, LRU y Second Chance, y calcular el *Page fault rate* en cada caso.
Page Fault Rate $= \frac{\text{páginas qué pedí y no estaban cargadas en memoria}}{\text{páginas totales}}$

Solución



Solución



Solución

Page Fault Rate (FIFO) = 7 / 9

Page Fault Rate (LRU) = 7 / 9

Page Fault Rate (SC) = 7 / 9

Thrashing

Un proceso hace **thrashing** si pasa más tiempo cargando y descargando páginas que ejecutando procesos de usuario.

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.

Prevención de thrashing

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.

Pero para prevenir el *thrashing* deberíamos proveer a un proceso de tantos *frames* como necesite.

Localidad

¿Cuántos *frames* necesita un proceso? → definir un modelo de localidad de la ejecución.

Una **localidad** es un conjunto de páginas que se usan activamente al mismo tiempo.
El **modelo de localidad** dice que mientras un proceso ejecuta, se mueve de localidad a localidad.

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*.

Resumen de la clase:

- Estrategias de asignación de memoria en esquemas de memoria contigua.
- Problema de la fragmentación externa e interna.
- Ventajas de la paginación y páginas compartidas.
- Esquema de memoria virtual y demand paging.
- Algoritmos de reemplazo de páginas.
- Thrashing.

Con esto se puede resolver toda la guía práctica 4.


SO ASK ME
QUESTIONS
quickmeme.com