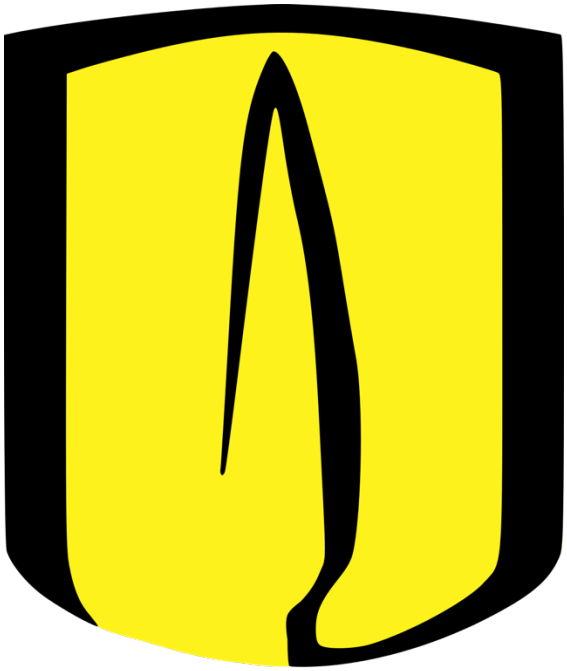
Thais Tamaio (202022213)

Santiago Tenjo (202113965)

Francisco Guzmán (202012332)

Caso de estudio 2: Memoria virtual



Infraestructura Computacional

2023-1

Contenido

1. Introducción.

Para el presente reporte, se documentará el desarrollo, implementación y prueba del caso de estudio número 2, con respecto a la construcción de un prototipo a escala del sistema de administración de memoria virtual, que permite evaluar el uso de la memoria virtual y reconocer la importancia de contar con una administración apropiada de la misma, según sean los recursos disponibles.

El problema por solucionar consiste en administrar el uso de memoria virtual con el fin de garantizar la ejecución de la suma de dos matrices, para que así los resultados de la matriz C y las matrices iniciales A y B, sean almacenadas junto a las referencias de página en un nuevo archivo para el módulo 1. Ahora para el módulo 2, se espera la carga de las referencias, el conteo de los fallos de página y la simulación del comportamiento del sistema de paginación junto al algoritmo de envejecimiento, el cual permite la toma de decisiones de remplazo.

2. Funcionamiento global del sistema.

El código cuenta con 5 clases en total:

- Actalizador.java
- AlgoritmoEnvejecimiento.java
- Main.java
- Modo1.java
- Modo2.java

Main.java, es la clase principal y esta recibe el archivo de entrada *inicial.txt* el cual es leído y se almacena los datos:

- TP: tamaño de página.
- NF: número de filas.
- NC: número de columnas.
- TE: tamaño del entero.
- MP: marcos de página.

Posterior a esto inicia el modo 1 invoca la función para generar las referencias de página, Luego realiza lo mismo con el modo 2 usando el archivo de entrada.txt y se almacena los datos:

- numPagina: páginas en el archivo.

Según fue propuesto el caso de estudio en el documento, se cuenta con el uso de dos modos independientes con resultados y parámetros distintos.

2.1 Modo 1.

El modo 1 es nuestro código es la clase “Modo1.java”, esta clase es responsable de generar las referencias de página y cuenta con un único método llamado *generarReferenciasDePagina()*.

2.1.1 Descripción del algoritmo para generar referencias de página.

A grandes rasgos lo que hace el algoritmo es calcular el total de referencias que se generarán, calcular la posición de referencia de la matriz, calcular el desplazamiento de la referencia, almacenar el número de página y la posición, agregar las referencias con el formato indicado y crear el archivo de salida para el módulo 1 *entrada.txt*. A continuación, se explica detalladamente los procesos más relevantes:

- Calcular las referencias según la cantidad de columnas y filas.

```
// Calcular total de referencias a generar
 int totalReferencias = NF * NC * 3; //
 Se calcula el total de referencias que se generarán multiplicando el número de filas
por el número de columnas por 3
```

- Genera las referencias, sin antes calcular la posición que garantiza el orden del tipo row-major-order y se calcula el desplazamiento necesario según el tamaño de página.

```java
// Generar las referencias de página para cada elemento de la matriz.
for (int i = 0; i < NF; i++) {
    for (int j = 0; j < NC; j++) {
        for (int k = 0; k < 3; k++) {
            // Se calcula la posición de la referencia en la matriz.
            int posicion = (i * NC + j) * TE + k * NF * NC * TE;
            // Se calcula el número de página de la referencia.
            int numPagina = posicion / TP;
            // Se calcula el desplazamiento de la referencia en la página.
            int desplazamiento = posicion % TP;
            //
 Se almacena el número de página y la posición de la referencia en los arreglos correspo
ndientes.
            numPaginas[i * NC * 3 + j * 3 + k] = numPagina;
            posiciones[i * NC * 3 + j * 3 + k] = desplazamiento;
        }
    }
}
```

2.2 Modo 2.

En nuestro código el modo 2 es la clase Modo2.java, esta clase se encarga de simular el comportamiento del proceso, por ende, carga referencias una por una y se encarga del sistema de paginación. Al ser esta sus funciones principales, tiene como ingreso los MP: marcos de página y las páginas únicamente.

Para esto, existen dos threads que dividen las tareas de este modo, estos se encuentran asociados a las clases *AlgoritmoEnjecimiento.java* y *Actualizador.java*, ambos son iniciados desde la clase del modo 2, conceptualmente en nuestro código ambas clases tienes las mismas funciones.

```java
public class AlgoritmoEnvejecimiento extends Thread {

    private static boolean ejecutarEnvejecimiento;

    public AlgoritmoEnvejecimiento(){
        ejecutarEnvejecimiento = true;
    }

    public static void detenerEnvejecimiento(){
        ejecutarEnvejecimiento = false;
    }

    public void run(){

        while (ejecutarEnvejecimiento){

            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {e.printStackTrace();}

            Modo2.algoritmoEnvejecimiento();
        }
    }
}
```

- Este código hace parte de *AlgoritmoEnvejecimiento.java*, esta clase actualiza el estado del algoritmo de envejecimiento para así determinar su inicio en True, su final en False y por último también se indica el tiempo en el que corre el thread que es cada milisegundo.

```java
public class Actualizador extends Thread {

    private static boolean updateTP;

    public Actualizador(){
        updateTP = true;
    }

    public static void detenerActualizador(){
        updateTP = false;
    }

    public void run(){

        while (updateTP){

            try {
                Thread.sleep(2);
            } catch (InterruptedException e) {e.printStackTrace();}

            Modo2.updateTP();

        }
    }
}
```

- Este código hace parte de la clase *Actualizador.java*, que cuenta también con funciones que actualizan el estado mediante True y False, solo que para este el tiempo en el que corre el thread es de dos milisegundos.

2.2.1 Descripción de las estructuras de datos usadas para simular el comportamiento del sistema de paginación.

Las estructuras utilizadas se encuentran en la clase *Modo2.java,* lo primero que se hace es declarar listas que indican la ubicación de las páginas en memoria real y virtual, y una tabla para el algoritmo de envejecimiento. Estas estructuras son editadas y consultadas a lo largo de la ejecución de los algoritmos que las involucran, por esto es importante que sean iniciadas con *static*.

```java
//Listas que indican la ubicación de las páginas en memoria real y virtual

public static List<Integer> paginasMemoriaReal = new ArrayList<Integer>();
public static List<Integer> paginasMemoriaVirtual = new ArrayList<Integer
>();
public static List<Integer> paginasEnUso = new ArrayList<Integer>();

//Tabla necesaria para ejecutar el algortimo de envejecimiento
private static Map<Integer, Integer> envejecimiento = new HashMap<Integer,
Integer>();
```

2.2.2 Esquema de sincronización.

El siguiente código es el orden en el cual se ejecuta el modo 2, teniendo en cuenta que para esto se inician los dos thread antes mencionados, para después hacer *cargaInicialPaginas()* y concluir con el cambio de estado de los dos threads.

```java
public Integer ejecutarModo2()
{
    new AlgoritmoEnvejecimiento().start();
    new Actualizador().start();
    cargaInicialPaginas();
    Actualizador.detenerActualizador();
    AlgoritmoEnvejecimiento.detenerEnvejecimiento();
    return fallosPagina;
}
```

Además, el código implementa un esquema de sincronización utilizando un objeto lock, definido como "privaste static final Object lock = new Object();" y un bloque synchronized(lock) {}. Se utiliza para proteger el acceso a estructuras de datos compartidas por múltiples hilos, como las listas de páginas en memoria real y virtual, la lista de páginas en uso y el mapa de envejecimiento. Al utilizar

la palabra clave synchronized, el bloque de código que sigue solo se puede ejecutar por un hilo a la vez, lo que garantiza que no se produzcan conflictos al acceder a los datos compartidos.

El código utiliza sincronización en los siguientes lugares:

- En el método cargaInicialPaginas(), cuando se accede a las listas paginasMemoriaReal y paginasEnUso dentro de un bloque synchronized(lock).

```
//
 Sincroniza el acceso a la lista de páginas en memoria real y de
páginas en uso
synchronized(paginasMemoriaReal)
{
    synchronized(paginasEnUso)
    {
```

- En el método updateTP(), cuando se accede a las listas paginasMemoriaVirtual, paginasEnUso y paginasMemoriaReal, así como al mapa de envejecimiento, dentro de un bloque synchronized(lock).

```
// Sincroniza el acceso a las estructuras de datos compartidas
synchronized (paginasMemoriaReal) {
    synchronized (paginasEnUso) {
        synchronized (paginasMemoriaVirtual) {
            synchronized (envejecimiento) {
```

Es necesario utilizar sincronización en estos lugares porque varias tareas concurrentes pueden intentar modificar estas estructuras de datos compartidas al mismo tiempo. Sin sincronización, esto podría dar lugar a condiciones de carrera y errores en el programa. Con la sincronización, cada tarea espera su turno para acceder a los datos compartidos y evita que otros hilos interfieran con su trabajo, lo que garantiza que el programa funcione correctamente.

2.2.3 Algoritmo de envejecimiento.

Este algoritmo es una implementación del algoritmo de envejecimiento de páginas, utilizado en sistemas operativos para gestionar la memoria virtual. El objetivo del algoritmo es mantener un registro de las páginas que se están utilizando en la memoria, y dar prioridad a aquellas páginas que hayan estado en uso durante más tiempo.

El algoritmo comienza bloqueando el objeto "envejecimiento" y luego el objeto "paginasEnUso" para asegurar que no haya condiciones de carrera durante la iteración sobre los mapas. Luego, itera sobre el mapa "envejecimiento" para cada entrada de la página, realiza un corrimiento hacia la izquierda de los bits, inserta un "1" en el segundo bit más significativo y actualiza el valor del contador de envejecimiento en el mapa. Si una página no se encuentra en el conjunto de "paginasEnUso", se agrega con un contador de envejecimiento inicial de 10000000. Si la página ya está en el mapa de "envejecimiento", se actualiza su contador de envejecimiento.

```
// Iterar sobre el mapa de envejecimiento
for (Map.Entry<Integer, Integer> paginasEnvejecimiento :
envejecimiento.entrySet()) {
    Integer paginaEnvejecimiento = paginasEnvejecimiento.getKey();
    Integer bits = paginasEnvejecimiento.getValue();
    if (!paginasEnUso.contains(paginaEnvejecimiento)) {
        Integer pagina = paginaEnvejecimiento;
        // Realizar el corrimiento hacia la izquierda
        bits <<= 10;
        //
 Modificar el primer dígito (poner un 1 en el segundo bit más signific
ativo)
        bits = (bits & 0x7F) | 0x40;
        //
 Actualizar el valor del contador de envejecimiento en el mapa
        envejecimiento.put(pagina, bits);
    }
}
```

Después, se itera sobre el conjunto de "paginasEnUso" y se actualiza el mapa "envejecimiento" con los valores correspondientes.

```
for (Integer pagina : paginasEnUso) {
    Integer bits = envejecimiento.get(pagina);
    if (bits == null) {
        //
 Si la página no está en el mapa de envejecimiento, se agrega con un c
ontador de envejecimiento inicial de 10000000
        envejecimiento.put(pagina, 1000000000);
    }
    else {
        //
 Si la página ya está en el mapa de envejecimiento, se actualiza su co
ntador de envejecimiento
        bits = bits / 10;  // Elimina el último dígito
        bits = 0b10000000 | bits;  // Agrega el 1 al principio
        envejecimiento.put(pagina,  bits / 2);
    }
}
```

Finalmente, se eliminan las páginas en uso del conjunto de "paginasEnUso".

```
    // Se borran las páginas en uso del conjunto de páginas en uso
    paginasEnUso.clear();
}
```

En resumen, este algoritmo utiliza un enfoque de "envejecimiento" para manejar la memoria virtual y dar prioridad a las páginas que han estado en uso durante más tiempo.

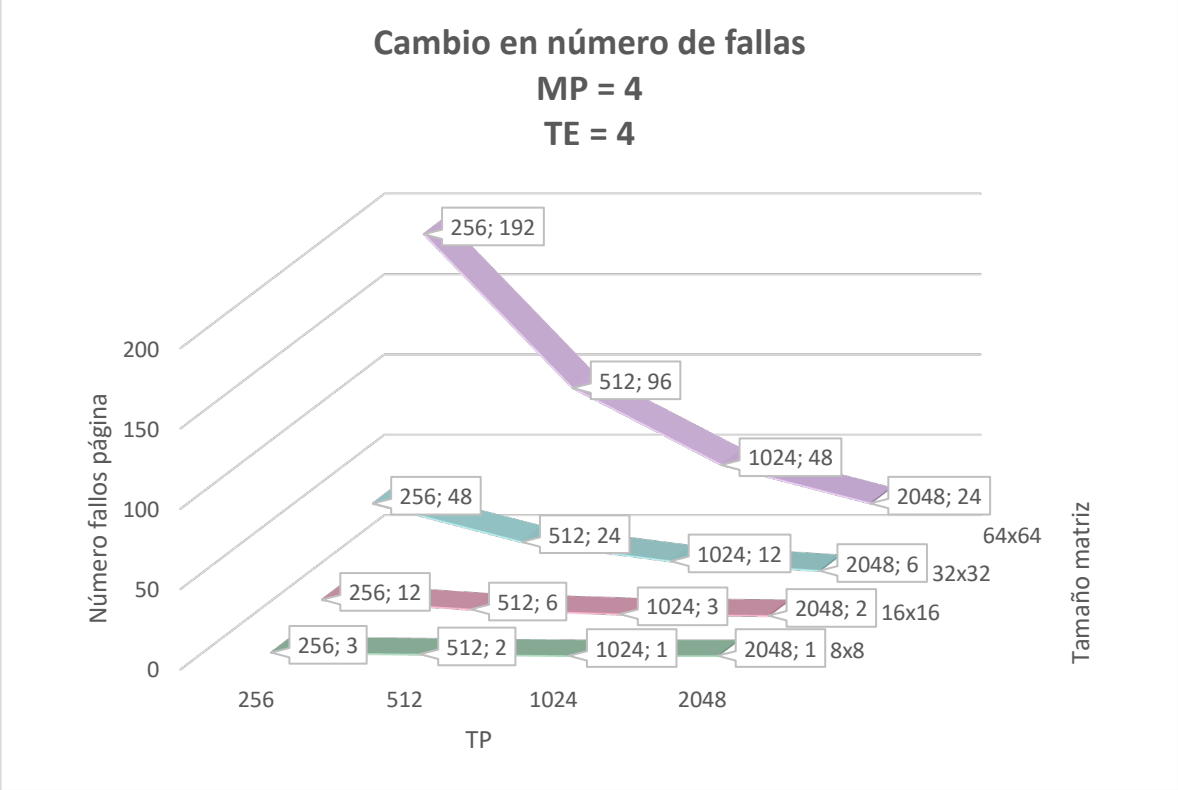
3. Resultados.

A continuación, se muestra el total de fallos de página por cada combinación de tamaños de matriz y los distintos tamaños de página utilizados:

		Tamaño de página			
		256	512	1024	2048
Tam. matriz	8x8	3	2	1	1
	16x16	12	6	3	2
	32x32	48	24	12	6
	64x64	192	96	48	24

4. Anexos y graficas.

Como se puede ver en la gráfica, la cantidad de fallos de página sigue un comportamiento esperado. Es decir, que a mayor tamaño de la matriz, más fallos de página serán generados a la hora de simular el proceso de memoria. De igual manera, entre menor sea el tamaño de cada página, más fallas de página serán generadas.



5. Consideraciones finales.

- ¿Qué pasaría si las matrices fueran almacenadas siguiendo column-major order?

Si las matrices fueran almacenadas siguiendo column-major order, en lugar de row-major order, se cambiaría el orden en que se acceden a los elementos de las matrices. En row-major order, los elementos de una fila se almacenan de manera contigua en la memoria, mientras que en column-major order, los elementos de una columna se almacenan de manera contigua en la memoria. Por lo tanto, el proceso de recorrido1 tendría que modificarse para adaptarse a este cambio en el orden de almacenamiento de los elementos de las matrices.

En el código actual, se generan las referencias de página de cada elemento de la matriz recorriendo primero las filas y luego las columnas. Si las matrices fueran almacenadas siguiendo column-major order, la generación de las referencias de página se realizaría recorriendo primero las columnas y luego las filas. Por lo tanto, los índices de las filas y las columnas cambiarían en la fórmula utilizada para calcular la posición de la referencia en la matriz, lo que daría lugar a diferentes valores de posición, número de página y desplazamiento en la página.

- ¿Cómo varía el número de fallas de página si el algoritmo estudiado es el de multiplicación de matrices?

El número de fallas de página en el algoritmo de multiplicación de matrices variará en función del tamaño de las matrices y de las páginas, así como del número de marcos asignados por el sistema. En general, se esperaría que el número de fallas de página aumente a medida que el tamaño de las matrices aumenta y disminuya a medida que el tamaño de las páginas aumenta y/o el número de marcos asignados por el sistema aumenta. Sin embargo, la variación exacta dependerá de las características específicas del sistema y del algoritmo de envejecimiento utilizado para la gestión de las páginas.

5.1 Interpretaciones finales:

En un proceso en el que se simula el guardado de tres matrices en memoria real, el tamaño de las matrices y el tamaño de las páginas de memoria son dos factores importantes que pueden afectar la cantidad de fallos de página que ocurren durante la simulación.

En términos generales, un fallo de página se produce cuando un programa intenta acceder a una página de memoria que no está actualmente en la memoria RAM y debe ser cargada desde el disco duro. Si la página que se necesita ya está en la memoria RAM, se puede acceder a ella rápidamente y sin problemas. Pero si la página no está en la memoria RAM, se produce un fallo de página y el sistema operativo debe cargar la página desde el disco duro, lo que puede ser un proceso lento y que consume muchos recursos.

En el caso de tener una matriz de mayor tamaño, se requerirá más espacio de memoria para guardarla en la memoria RAM. Si el tamaño de la página de memoria es pequeño, esto significa que cada página de memoria contendrá menos elementos de la matriz. Como resultado, se necesitarán más páginas para almacenar toda la matriz en la memoria RAM, lo que aumentará la probabilidad de que se produzcan fallos de página.

Por otro lado, si se tienen páginas de menor tamaño, cada página contendrá menos elementos de la matriz, lo que aumentará la cantidad de páginas que se necesitan para almacenar toda la matriz en la memoria RAM. Si el sistema operativo tiene que cargar muchas páginas pequeñas desde el disco duro, esto también puede aumentar la probabilidad de que se produzcan fallos de página.

En resumen, la relación entre el tamaño de las matrices y el tamaño de las páginas de memoria puede afectar la cantidad de fallos de página que ocurren durante un proceso de simulación. Si se tienen matrices de mayor tamaño, es recomendable utilizar páginas de memoria más grandes para reducir la cantidad de páginas necesarias para almacenar la matriz y, por lo tanto, reducir la probabilidad de que se produzcan fallos de página. Si se tienen matrices más pequeñas, se puede utilizar un tamaño de página de memoria más pequeño para minimizar la cantidad de espacio de memoria no utilizado.