CASO 2 - MEMORIA VIRTUAL

MANUELA LOVERA - 202320973 Y SHAIEL MATEO JIMENEZ - 202323846

1.Algoritmo para generar las referencias de página

Para generar las referencias de página, se siguen varios pasos. Para empezar, se obtiene el tamaño de página `TP`, el número de procesos a ejecutar `NPROC` y los distintos tamaños de las matrices que procesa cada proceso `TAMS`. Para cada proceso, se saca el número de filas `NF` y columnas `NC` de las matrices que manejan. Con base a estos números, se calcula el número de referencias `NR` (número total de enteros utilizados por las 3 matrices en la suma de matrices) y el número de páginas `NP` usadas por cada proceso de acuerdo al espacio total ocupado por las matrices dividio entre el tamaño de página.

```
    for (int i = 0; i <NPROC; i++) {
        int NF = TAMS.get(i).get(0);
        int NC = TAMS.get(i).get(1);
        int NR = NF*NC*3;
        int totalSpace = NR*4;
        int NP = (int) Math.ceil(totalSpace*1.0/TP);
        ....
    }
```

Para cada proceso, se genera un archivo nuevo donde se escribe toda la información anterior junto con las direcciones virtuales de cada una de las matrices referenciadas en el proceso. Las direcciones virtuales de los enteros de cada matriz se calcula a partir de la referencia exacta (sin paginación) de cada entero. Esto se hace mediante una función auxiliar que saca la página a la que pertenece por medio una división entera de la referencia entre el tamaño de página y su desplazamiento dentro de la página sacando el residuo de dicha división.

```
    public static int[] calculatePageAndReference(int currentReference
    , int pageSize) {
        int page = currentReference/pageSize;
        int ref = currentReference%pageSize;
        return new int[]{page, ref};
    }
```

Para cada proceso, se tienen 3 referencias al inicio de cada una de las matrices `M1 M2 M3`. Estas referencias, como se menciono anteriormente, son las referencias exactas consecutivas de los enteros de cada matriz (de 0 a `NR`). Por cada entero, se escribe su matriz correspondiente, la fila y columna en la que esta ubicado, la página en la que se encuentra y el desplazamiento para llegar hasta él (ambos calculados con la función auxiliar en base a la `refM#` correspondiente) y si se accede para leer `r` o escribir `w` (`M1` y `M2` son `r` y `M3` es `w`).

```
    for (int i = 0; i <NPROC; i++) {
        ....
        int[] pr;
        int refM1 = 0;
        int refM2 = NC*NF*4;
        int refM3 = NC*NF*4*2;

        try (FileWriter writer = new FileWriter(fileName)) {
                writer.write("TP=" + TP +"\n");
                writer.write("NF=" + NF +"\n");
                writer.write("NC=" + NC +"\n");
                writer.write("NR=" + NR +"\n");
                writer.write("NP=" + NP +"\n");
                for (int j = 0; j < NF; j++) {
                    for (int k = 0; k < NC; k++) {
                        pr = calculatePageAndReference(refM1, TP);
                        writer.write("M1: ["+j+"-"+k+"], "+pr[0]+", "+
    pr[1]+", r\n");
                        pr = calculatePageAndReference(refM2, TP);
                        writer.write("M2: ["+j+"-"+k+"], "+pr[0]+", "+
    pr[1]+", r\n");
                        pr = calculatePageAndReference(refM3, TP);
                        writer.write("M3: ["+j+"-"+k+"], "+pr[0]+", "+
    pr[1]+", w\n");
                        refM1+=4;
                        refM2+=4;
                        refM3+=4;
                    }
                }
            }
        ....
    }
```

2. Estructuras de datos y uso en la simulación

En la implementación de la Opción 2 se emplean varias estructuras de datos para modelar la tabla de páginas, los marcos físicos asignados a cada proceso, la política LRU y la cola de ejecución (round-robin). A continuación se describen con detalle cada una de ellas y se explica *cuándo* y *cómo* se actualizan.

1. pageTable (tabla de páginas).

- **Tipo / representación:** `List<List<Integer>>` donde cada entrada es una lista de tres enteros: `[frameNumber, rBit, wBit]`.
- **Semántica:**
  - `frameNumber`: índice del marco físico donde está cargada la página; `-1` indica que la página no está en memoria.
  - `rBit`: bit de referencia (1 si fue referenciada recientemente).
  - `wBit`: bit de escritura (1 si fue escrita).
- **Inicialización:** al leer `NP` se crea la lista y se inicializa cada entrada a `[-1,0,0]`.
- **Actualizaciones:**

1. *Al cargar una página* (page fault resuelto): `frameNumber = f`, `rBit = 1`; si la referencia es escritura, `wBit = 1`.
2. *En un hit*: se pone `rBit = 1` y si la operación es escritura se pone `wBit = 1`.
3. *Al expulsar (reemplazo)*: la entrada de la página victim se restaura a `[-1,0,0]`.

- **Ejemplo (antes/ después de cargar página 5 en marco 10):**

```
Antes: pageTable[5] = [-1,0,0]
Acción: cargar página 5 en marco 10 (referencia 'r')
Después: pageTable[5] = [10,1,0]
```

2. assignedFrames (marcos asignados).

- **Tipo / representación:** `Set<Integer>` que contiene los índices de marcos físicos asignados a ese proceso.
- **Semántica:** define el *working set* máximo de marcos que el proceso puede usar.
- **Inicialización:** al inicio de la simulación cada proceso recibe `framesPerProcess` marcos (p. ej. marcos contiguos calculados con la fórmula del main).
- **Actualizaciones:**
  - *Al finalizar un proceso*: se liberan los marcos (el conjunto se vacía) y se pasan a reasignación.
  - *Al reasignar*: se añaden marcos liberados a `assignedFrames` del proceso receptor.
- **Nota operativa:** para decidir si existe un marco libre al manejar un page fault, el código compara el tamaño de `lruMap` con `assignedFrames.size()`; si `lruMap.size() <assignedFrames.size()` existe al menos un marco libre.

3. lruMap (implementación de LRU).

- **Tipo / representación:** `LinkedHashMap<Integer,Integer>` con `accessOrder=true`, donde la *clave* es el número de página y el *valor* es el número de marco.
- **Semántica:** mantiene las páginas actualmente cargadas por el proceso en orden de uso; la primera clave es la página LRU (menos recientemente usada).
- **Inicialización:** vacío al inicio; se `put(page,frame)` cuando se carga una página.
- **Actualizaciones:**
  - *En hit:* una llamada a `get(page)` reordena internamente la entrada y la convierte en la más recientemente usada.
  - *En carga sin reemplazo:* `put(page,frame)` añade la entrada; ahora la página es la más reciente.
  - *En reemplazo:* se obtiene la primera clave (LRU) y se `remove(lruPage)`; luego se `put(newPage,victimFrame)`.
- **Ejemplo:** si `lruMap` contiene en orden [3→5, 7→2, 9→4] entonces la página LRU es 3 y su marco es 5.

4. references (lista de referencias).

- **Tipo / representación:** `List<Reference>` donde `Reference` almacena `page, offset, type` (`'r'/'w'`).
- **Semántica:** secuencia exacta de accesos generados por la Opción 1; se procesan en orden secuencial.
- **Actualización:** no cambia durante la ejecución; lo que cambia es un índice externo `currentRefIndex[pid]` que indica la próxima referencia a procesar.

5. *Cola de procesos y control de reintentos.*
   - **Tipo / representación:** `Queue<Integer>` (round-robin) y un arreglo `boolean[] retrying` por proceso.
   - **Semántica:** la cola contiene los PIDs de procesos listos. Si un proceso sufre page fault, se marca `retrying[pid]=true` y se vuelve a encolar sin avanzar su índice; en la siguiente vez que tome CPU reintentará la misma referencia.
   - **Actualización:** en cada turno se `poll()` un PID; tras procesar (hit o reintento resuelto) se decide si reencolar o descartar (si terminó).

6. *Contadores y métricas.*
   - **Campos:** `pageFaults`, `pageHits`, `writes`, y arreglo global `swapAccesses[pid]`.
   - **Reglas de actualización:**
     - `pageHits++` cuando la página está en memoria (hit) y `retrying[pid]==false`.
     - `pageFaults++` cuando la página no está cargada.
     - `swapAccesses[pid] += 1` si page fault se resuelve asignando un marco libre (carga simple).
     - `swapAccesses[pid] += 2` si page fault requiere reemplazo (lectura + posible escritura del victim según simplificación del enunciado).
     - `writes++` se incrementa si la página expulsada tenía `wBit==1`.

```
        Reference ref = pdata.references.get(idx);
        int page = ref.page;
        char type = ref.type;

        boolean pageFault = false;
        boolean replacement = false;

        System.out.println("PROC " + pid + " analizando linea_: " +
    idx);

        // Verificar si la p gina est  cargada (page hit)
        if (pdata.lruMap.containsKey(page)) {
            if (!retrying[pid]) {
                pdata.pageHits++;
            }
            pdata.lruMap.get(page); // Actualizar orden LRU
            pdata.pageTable.get(page).set(1, 1); // ref bit
            if (type == 'w') pdata.pageTable.get(page).set(2, 1);
            System.out.println("PROC " + pid + " hits: " + pdata.
    pageHits);
        } else {
            pageFault = true;
            pdata.pageFaults++;
            // Si hay espacio libre, asignar; si no, reemplazar LRU
```

```
23              if (pdata.lruMap.size() < pdata.assignedFrames.size()) {
24                  // Buscar un marco libre
25                  for (int frame : pdata.assignedFrames) {
26                      if (!pdata.lruMap.containsValue(frame)) {
27                          pdata.lruMap.put(page, frame);
28                          pdata.pageTable.get(page).set(0, frame);
29                          break;
30                      }
31                  }
32                  swapAccesses[pid] += 1; // Page fault sin reemplazo
33              } else {
34                  // Reemplazo LRU
35                  replacement = true;
36                  int lruPage = pdata.lruMap.keySet().iterator().next();
37                  int victimFrame = pdata.lruMap.remove(lruPage);
38                  if (pdata.pageTable.get(lruPage).get(2) == 1) pdata.
    writes++;
39                  pdata.pageTable.get(lruPage).set(0, -1);
40                  pdata.pageTable.get(lruPage).set(1, 0);
41                  pdata.pageTable.get(lruPage).set(2, 0);
42                  pdata.lruMap.put(page, victimFrame);
43                  pdata.pageTable.get(page).set(0, victimFrame);
44                  swapAccesses[pid] += 2; // Page fault con reemplazo
45              }
46              pdata.pageTable.get(page).set(1, 1);
47              if (type == 'w') pdata.pageTable.get(page).set(2, 1);
48              System.out.println("PROC " + pid + " falla de pag: " +
    pdata.pageFaults);
49          }
50
51          System.out.println("PROC " + pid + " envejecimiento");
52
53          if (pageFault && !retrying[pid]) {
54              retrying[pid] = true;
55              processQueue.add(pid); // Re-queue para el siguiente turno
    , misma referencia
56          } else {
57              retrying[pid] = false;
58              currentRefIndex[pid]++;
59              if (currentRefIndex[pid] < pdata.references.size()) {
60                  processQueue.add(pid); // M s referencias para
    procesar
61              }
62          }
63      }
```

3. Escenarios de prueba

Para comprobar el correcto funcionamiento del programa, se diseñaron varios escenarios de prueba con diferentes valores iniciales, para evidenciar la diferencia en el procesamiento del programa. Para un escenario donde se procesan 2 matrices de tamaño 100x100, tenemos diferentes tamaños de página:

- Con un tamaño de página de 16 B:

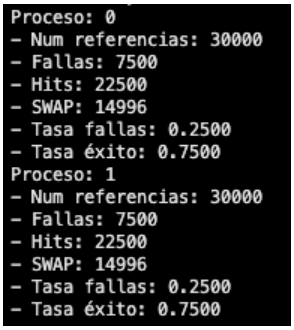

FIGURA 1. Resultados con TP = 16

- Con un tamaño de página de 32 B:

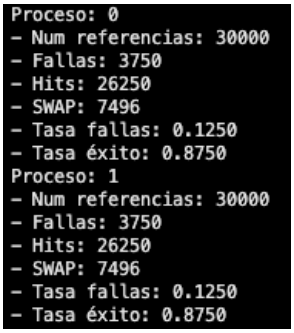

FIGURA 2. Resultados con TP = 32

- Con un tamaño de página de 64 B:

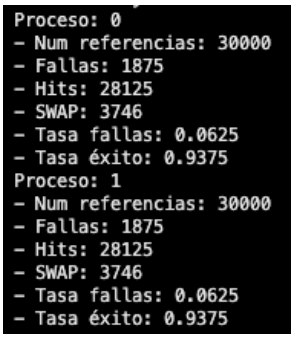

FIGURA 3. Resultados con TP = 64

- Con un tamaño de página de 128 B:

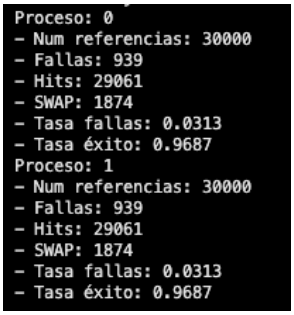

FIGURA 4. Resultados con TP = 128

- Con un tamaño de página de 256 B:

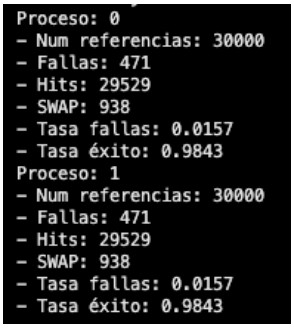

FIGURA 5. Resultados con TP = 256

- Con un tamaño de página de 512 B:

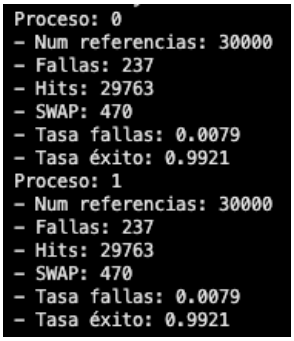

FIGURA 6. Resultados con TP = 512

- Con un tamaño de página de 1024 B:

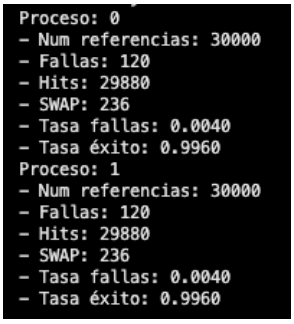

FIGURA 7. Resultados con TP = 1024

A continuación se muestra una gráfica comparativa entre los resultados obtenidos, comparando el número de fallas de página con respecto al tamaño de página:

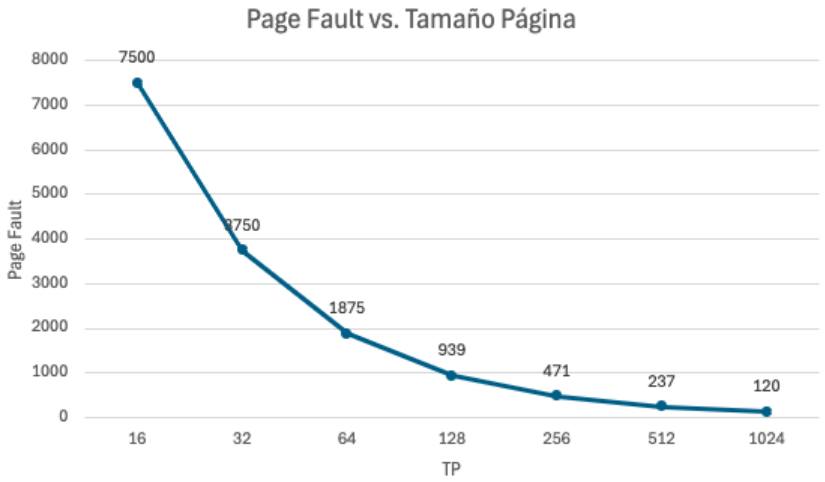

FIGURA 8. Comparación entre *TP* y *PF*

Adicionalmente, tenemos diferentes escenarios los cuales ejecutan con diferentes tamaños de matrices y parámetros:

- a) Utilizando la configuración de 8 *page frames*, TP=128, $NPROC$=2 y $TAMS$=50x50,500x500:

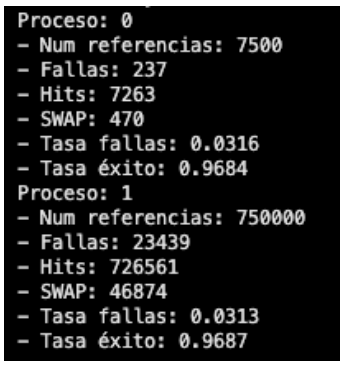

FIGURA 9. Resultados del escenario de prueba *a*

- b) Utilizando la configuración de 8 *page frames* TP=128, $NPROC$=4 y $TAMS$=50x50,50x50,50x50,50x50:

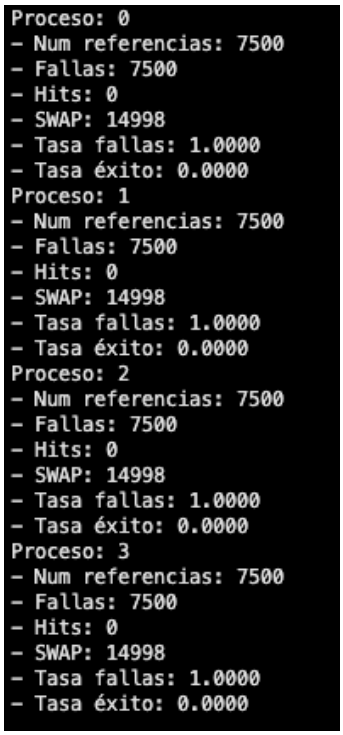

FIGURA 10. Resultados del escenario de prueba *b*

- c) Utilizando la configuración de 16 *page frames*, TP=128, $NPROC$=4 y $TAMS$=50x50,100x100,200x200,400x400:

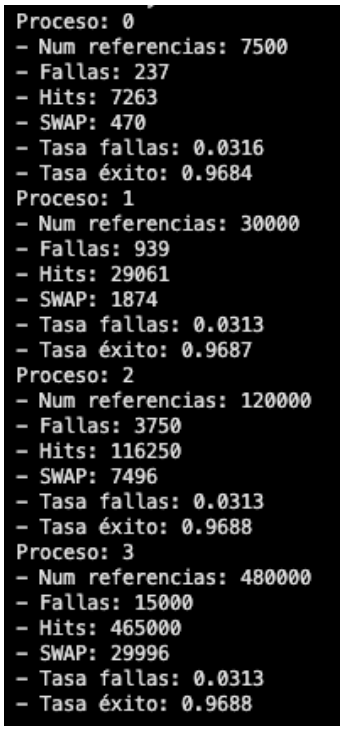

FIGURA 11. Resultados del escenario de prueba *c*

- d) Utilizando la configuración de 32 *page frames*, TP=128, $NPROC$=2 y $TAMS$=100x100:

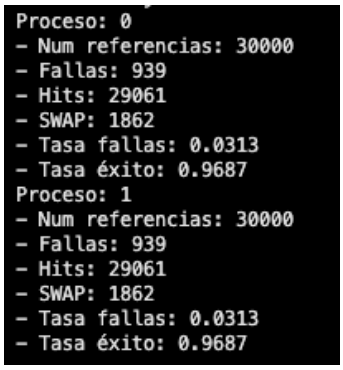

FIGURA 12. Resultados del escenario de prueba *d*

4. Ánalisis de resultados

- *¿Los resultados corresponden a lo que se esperaba?* En términos generales, podría decirse que sí, los resultados coinciden con lo que esperaba: al aumentar el tamaño de página (TP) el número total de fallas de página tiende a disminuir, y al reducir la cantidad de marcos por proceso (o aumentar el número de procesos compartiendo los mismos marcos) las fallas aumentan. Esto se observa en las figuras individuales para TP=16 hasta TP=1024 y en la gráfica comparativa (Figura 8), donde la tendencia principal es de descenso de fallas cuando TP crece. Cabe resaltar que la disminución de fallas no siempre es lineal, pues hay una zona donde aumentar TP ya no aporta mucha mejora, porque el número de páginas ya es pequeño y el bottleneck pasa a ser el número de marcos físicos (esto también se aprecia en la figura comparativa); además, en los escenarios heterogéneos (por ejemplo el caso con 50×50 y 500×500) las matrices grandes dominan el comportamiento dado que estas son las que producen la mayoría de las fallas a menos que se les asignen suficientes marcos (las figuras de los escenarios a–d ilustran estos efectos).

- *¿Cómo se afecta el tiempo de respuesta de un proceso por el número de accesos a SWAP?* Cuando un proceso tiene muchos accesos a SWAP su tiempo de respuesta aumenta de manera considerable. Esto se debe a que cada vez que ocurre un fallo de página, el sistema debe traer información desde el disco, lo cual es mucho más lento que leer directamente de la memoria principal. Además, en la simulación, cada fallo hace que el proceso pierda su turno, lo que añade aún más espera antes de continuar. En consecuencia, a mayor número de accesos a SWAP, más demorado será que el proceso termine sus operaciones, mientras que con pocos accesos su tiempo de respuesta mejora notablemente.

- *¿Sumar matrices representa un problema de localidad alta, media o baja?* La suma de matrices tiene alta localidad espacial y baja localidad temporal. Al recorrer las matrices fila por fila, los elementos consecutivos están juntos en memoria, por lo que muchos accesos tocan la misma página o páginas contiguas (buena localidad espacial). Sin embargo, cada elemento se usa casi solo una vez (se lee A y B y se escribe C) y no suele volver a usarse después, por lo que no hay mucha reutilización temporal. En conjunto, eso da un comportamiento de localidad global media: se beneficia claramente de páginas de tamaño moderado (por la localidad espacial), pero no se gana mucho por intentar explotar reutilización temporal.