Julián Oliveros Forero – 201821595

Alvaro Plata – 201820098

CASO 2 - INFRAESTRUCTRA COMPUTACIONAL

Tabla de contenido

Índice de tablas y figuras

1. Descripción de las estructuras de datos.

1.1 Estructura de datos **Páginas Por Leer**

La estructura de datos *páginasPorLeer* se encuentra definida como un arreglo de enteros como se muestra en la ecuación 1. Esta estructura contiene la secuencia de referencias a páginas del proceso, para el proyecto esta será leída del archivo que se pasa como parámetro y tendrá el total de llamados a página de un proceso. La estructura simularía los llamados a las paginas que pida la CPU a medida que va ejecutando instrucciones, es decir contendrá las páginas que estaría necesitando la CPU a medida que va ejecutando un proceso, cabe aclarar que en este caso ya se tienen todas las páginas que se llaman en un sistema real de paginación no se sabría las páginas a llamar sino que iría llamando páginas a medida que va ejecutando el proceso.

$$private\ static\ Integer[\ \ ]\ paginasPorLeer; \qquad ecuación\ 1$$

La estructura *páginasPorLeer* se utiliza en el thread 1 (TActualizarReferefencias) esta estructura no es actualizada sino que solo leída mediante un for que recorre todos los valores del arreglo de modo que el thread pueda saber cual es la página que se quiere acceder.

1.2 Estructura de datos **Contadores De Llamados A Cada Página**

La estructura *contadoresDeLlamadosACadaPágina* es utilizada para el funcionamiento del algoritmo de envejecimiento, su inicialización se ilustra en la ecuación 2, esta estructura tendrá una tamaño igual a el número de páginas del proceso y en cada casilla del arreglo se guarda el contador que lleva la cuenta aproximada de la frecuencia con que se hace referencia a cada página, por lo que en la posición 2 del arreglo se tendrá el contador de la página 3.

$$private\ static\ long[\ \ ]contadoresDeLlamadosACadaPagina; \qquad ecuación\ 2$$

La estructura simularía los marcos de pagina que podrían estar en la memoria RAM, además cada valor del arreglo haría referencia a la información del estado para cada marco de página.

Esta estructura es utilizada en 2 momentos, el primer acceso es cuando el thread 1 (TActualizarReferefencias) lee una referencia nueva (lee una página) y por ende actualiza el buffer o bit R (indica las páginas que fueron leídas), en este caso el thread 2 (TEnvejecimiento) teniendo en cuenta el buffer actualizará el contador de todas las casillas (páginas) del arreglo llamando a el método de la figura 1, este método se encarga de realizar el corrimiento a la derecha ingresando un 0 si no fue leída la página y un 1 de lo contrario.

```
//Este método realiza un corrimiento hacia la derecha a todas las páginas que no fueron llamadas, y adiciona un uno
//a la izquierda en la página que fue llamada
private void actualizarContadores(int paginaLlamada){
    for(int i = 0; i < this.contadoresDeLlamadosACadaPagina.length; i++){
        this.contadoresDeLlamadosACadaPagina[i] = this.contadoresDeLlamadosACadaPagina[i] >> 1;
    }
    this.contadoresDeLlamadosACadaPagina[paginaLlamada] += Math.pow(2,30);
}
```

Figura 1. Método Actualizar Contadores

El segundo momento en que se utiliza esta estructura es cuando ocurre un fallo de página y no se cuenta con espacio libre el la tabla de página para insertar la página, en este caso el thread 1 llamará a el método de la figura 2 dicho método buscará en todo el arreglo de *contadoresDeLlamadosACadaPágina* y buscará el contador el menor número (valor con la menor sumatoria), una vez se recorra toda la lista se devolverá la página que se procederá a sacar de la tabla de páginas que seria la menos accedida.

```java
public synchronized int darPaginaConMenosSumatoria() {
    long menorSumatoria = Long.MAX_VALUE;
    int paginaConMenorSumatoria = -1;
    for (int i = 0; i < this.contadoresDeLlamadosACadaPagina.length; i++) {
        if (this.contadoresDeLlamadosACadaPagina[i] < menorSumatoria && revisarSiLaPaginaEstaEnLaTablaDePaginas(i)) {
            menorSumatoria = this.contadoresDeLlamadosACadaPagina[i];
            paginaConMenorSumatoria = i;
        }
    }
    return paginaConMenorSumatoria;

}
```

Figura 2. Método Dar Página Con Menor Sumatoria

1.3 Estructura de datos **Buffer**

La estructura de datos *buffer* es utilizada para el funcionamiento del algoritmo de envejecimiento, su inicialización se ilustra en la ecuación 3, esta estructura tendrá una tamaño igual a el número de páginas del proceso y en cada casilla del arreglo se tendrá una valor booleano que indicará si la página fue leída o no (true si fue leída), por ejemplo si tengo 3 páginas y leí la pagina 1 el buffer tendría seria (true, false, false). La similitud con el sistema de paginación es que estructura *buffer* haría referencia a el Bit R el cual indica las páginas que fueron leídas y las que no.

$$private\ static\ Boolean[]\ buffer; \qquad ecuación\ 3$$

El buffer es accedido y modificado en varias ocasiones, En primer lugar Cuando el thread 1 realiza una lectura de una referencia este debe actualizar el *buffer* este ejecuta el métodos de la figura 3 el cual se encargar de cargar en el buffer la o las páginas que fueron leídas.

```java
public synchronized void actualizarBuffer(int paginaLlamada) {
    Arrays.fill(this.buffer, val: false);
    this.buffer[paginaLlamada] = true;
}
```

Figura 3. Método Actualizar Buffer

Esta estructura también es utilizada por el thread 2, en un primer instante usa el método de la figura 4 para saber si ejecuta o no el algoritmo de envejecimiento este

se ejecutaría si el *buffer* no esta limpio, en caso de que no este limpio el buffer se procederá a ejecutar el algoritmo de envejecimiento método de la figura 5, dicho método utilizará el buffer para realizar los respectivos corrimientos sobre cada contador de cada página. Por último se una el método de la figura 6 para limpiar el Buffer para la siguiente lectura, es decir dejar en nulo todos los valores del *buffer*.

```
public synchronized boolean elBufferEstaLimpio() { return this.buffer[0] == null; }
```

Figura 4. Método El Buffer Esta Limpio

```
public synchronized void algoritmoEnvejecimiento() {
    boolean encontreElUno = false;
    int paginaLlamada = -1;
    for (int i = 0; i < this.buffer.length && !encontreElUno; i++) {
        if (this.buffer[i]) {
            encontreElUno = true;
            paginaLlamada = i;
        }
    }
    actualizarContadores(paginaLlamada);
    limpiarBuffer();
}
```

Figura 5. Método Algoritmo Envejecimiento

```
private void limpiarBuffer() { Arrays.fill(this.buffer,  val: null); }
```

Figura 6. Método Limpiar Buffer

1.4 Estructura de datos **Tabla De Paginas**

La estructura de datos tablaDePaginas es utilizada para representar cuáles páginas del proceso estarían soportadas en memoria.Esta estructura consiste en un arreglo de enteros, donde el número de casillas corresponde a la cantidad de marcos de página disponibles, y los valores en cada casilla corresponden a las páginas del proceso que están soportadas en memoria en ese momento. Este arreglo es utilizado en varios momentos:

- Es accedido para conocer si una página determinada ya se encuentra en la tabla de página.

```
public boolean revisarSiLaPaginaEstaEnLaTablaDePaginas(int paginaPorRevisar) {
    for (Integer tablaDePagina : this.tablaDePaginas) {
        if (tablaDePagina != null && tablaDePagina == paginaPorRevisar) {
            return true;
        }
    }
    return false;
}
```

Figura 7. Método revisar Si La Pagina Esta En La Tabla De Paginas

- Es accedido para reemplazar una página que ya se encuentre en la tabla de páginas, por una nueva que no esté soportada y que haya sido referenciada, es decir, que necesite ser accedida en ese momento.

```
public void reemplazarPaginaEnTablaDePaginas(int paginaNueva, int paginaVieja){
    for(int i = 0; i < this.tablaDePaginas.length; i++){
        if(this.tablaDePaginas[i] == paginaVieja){
            this.tablaDePaginas[i] = paginaNueva;
            return;
        }
    }
}
```

Figura 8. Método reemplazar Pagina En Tabla De Paginas

- Es accedido para soportar una nueva página en la tabla de páginas, conociendo previamente que hay casillas disponibles y que no es necesario retirar una página ya presente en la tabla de páginas

```
public void agregarPaginaEnTablaDePaginas(int paginaNueva){
    this.tablaDePaginas[ocupacionTablaDePaginas]= paginaNueva;
}
```

Figura 9. Método agregar Pagina En Tabla De Paginas

2. Esquema de Sincronización

En nuestra implementación del proyecto, contamos con tres clases principales: la clase *MemoriaVirtual*, que es la clase principal y que se encarga de manejar la información de las páginas accedidas, los contadores de acceso a cada página, el buffer que informa la página que acaba de ser leída y la lista con el orden de llamado a todas las páginas. La clase *TEnvejecimiento*, que corresponde al Thread encargado de revisar el buffer que se encuentra en *MemoriaVirtual* cada milisegundo, y cuando haya una actualización en éste, llamar a la clase MemoriaVirtual para que ejecute el algoritmo de envejecimiento. La clase TActualizarReferencias, que corresponde al Thread encargado de actualizar el estado de la tabla de páginas de acuerdo con las referencias de la lista de llamados a cada página, manejar los fallos de página, y actualizar el buffer con la nueva referencia a la última página leída.

Para garantizar el cumplimiento de los requerimientos es necesaria la exclusión mutua para ciertos métodos de la clase *MemoriaVirtual*, específicamente, en los métodos que manejan las estructuras de datos que son utilizadas por ambos Threads. A continuación, enumeramos estos métodos de la clase *MemoriaVirtual* que son llamados por los Threads y la razón por la que deben ser synchronized.

- ***public synchronized void algoritmoEnvejecimiento():*** es el método encargado de acceder al buffer para identificar la página que fue referenciada y según eso actualizar los contadores, es decir, hacer los corrimientos y poner el bit más a la izquierda en 1 de la página que fue llamada. Es necesario que sea synchronized porque manipula el buffer y los contadores de llamados a cada página, que es una variable compartida por ambos Threads.

- ***public synchronized boolean elBufferEstaLimpio():*** es el método encargado de determinar si el buffer ha tenido alguna actualización. Es necesario que sea synchronized porque manipula el buffer, que es una variable compartida por ambos Threads.

-***public synchronized int darPaginaConMenosSumatoria():*** es el método encargado de determinar cuál es la página que se debe reemplazar de la tabla de páginas cuando ocurra un fallo de página. Es necesario que sea synchronized porque manipula los contadores de llamados a cada página, que es una variable compartida por ambos threads.

- ***public synchronized void actualizarBuffer(int paginaLlamada):*** es el método encargado de ingresar al buffer la referencia a la nueva página que fue llamada, para que pueda ser leído por el thread 2. Es necesario que sea synchronized porque manipula el buffer, que es una variable compartida por ambos Threads.

- public synchronized void actualizarTermineDeLeer() y darTermineDeLeer(): son los métodos encargados de consultar y actualizar la variable booleana termineDeLeer, que indica cuando el TActualizarReferencias haya terminado de recorrer la lista de llamdos a páginas. Es necesario que sea synchronized porque manipula la variable termineDeLeer que es compartida por ambos Threads.

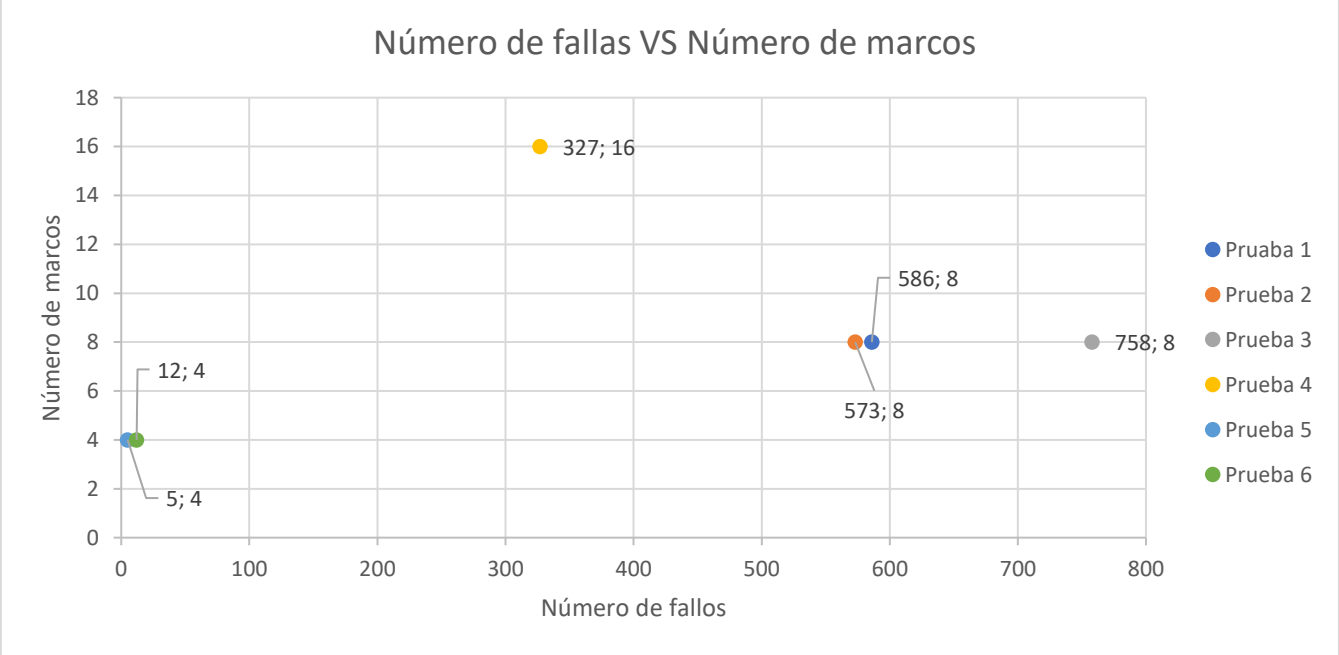
3. Tabla con las pruebas realizadas

DATOS RECOPILADOS EN LAS PRUEBAS						
Prueba	**Número de marcos de página**	**Número de páginas del proceso**	**Nivel de localidad**	**Cantidad de fallos de página esperado**	**Cantidad de fallos de página encontrados**	**Porcentaje de diferencia entre los resultados esperados y encontrados**
1	8	40	25	587	586	0,17%
2	8	60	75	577	573	0,69%
3	8	100	75	758	758	0,00%
4	16	40	25	340	327	3,98%
5	4	8	25	5	5	0,00%
6	4	16	25	12	12	0,00%

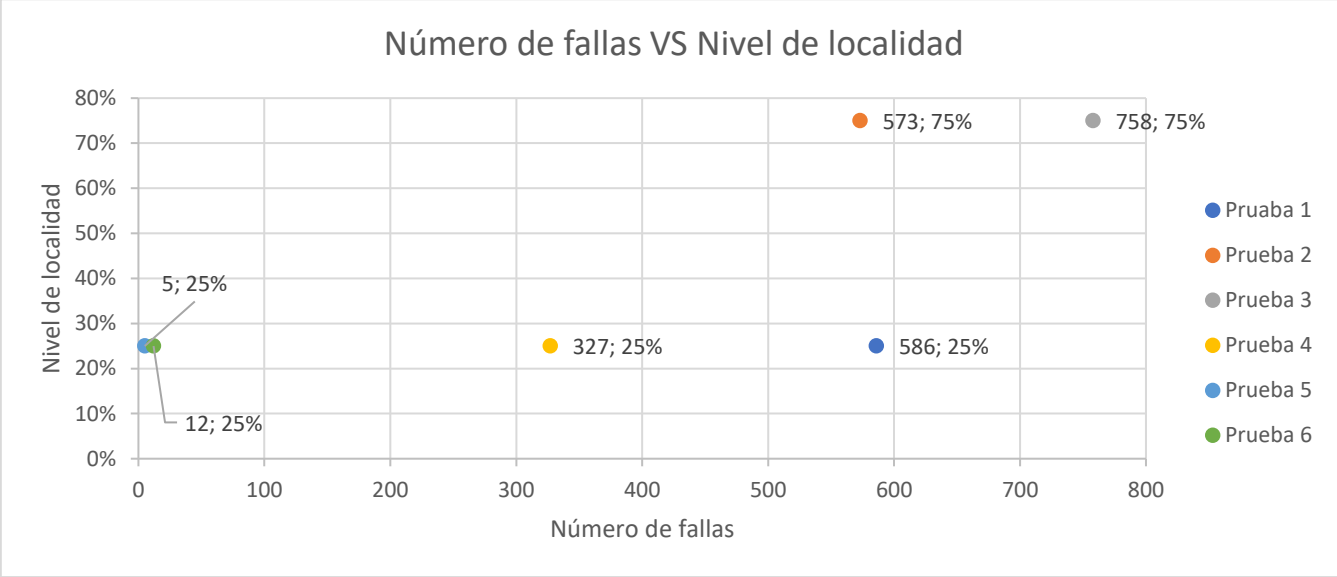
4. Gráfica del comportamiento de sistema.

A continuación, se encuentran 4 gráficas que representan los resultados de las pruebas realizadas:

- Esta gráfica compara el número de fallos de página con el número de marcos de página para cada una de las pruebas. Cada prueba se representa con un color diferente. Por simplicidad, y para poder comparar el rendimiento para los diferentes tamaños de programa, decidimos representar todos los tamaños analizados en una sola gráfica.



- Esta gráfica compara el número de fallos de página contra el nivel de localidad para cada una de las pruebas. Cada prueba se representa con un color diferente.



5. Interpretación de Resultados

A partir de las pruebas realizadas y la teoría podemos concluir principalmente que si tenemos un número de páginas constante se tiene una relación inversa entre el número de marcos de página y los fallos de página, pues a medida que se tengan más marcos de página en un programa, éste tendrá menos fallos de página. Esta conclusión tiene sentido pues si llegamos al caso de tener el mismo número páginas y de marcos de página, el número de fallos de página seria constante e igual a el número de páginas porque solo se tendrían que cargar las páginas al marco de página y no se tendrían más fallos de ahí en adelante, porque todas las páginas siempre estarían en RAM. Sin embrago tener una RAM tan grande como para que almacene todas las paginas seria extremadamente costoso.

Igualmente podemos observar que, para procesos con cantidades similares de marcos de página y de páginas del proceso, es de esperar que el número de fallos de página sea igualmente similar. Esto se debe a que cuentan con el mismo espacio disponible en memoria para soportar un número similar de instrucciones, por lo que la cantidad de veces que una página debe ser intercambiada por otra será similar.

Respecto a el principio de localidad se puede concluir teóricamente que entre mayor sea este porcentaje menores fallos de página se tendrá pues que el principio de localidad va asociado a que si yo utilizó una página es buen indicador de que la usaré pronto, por lo que se tratará de tener en memoria las páginas que sean más usadas con el fin de reducir la cantidad de fallos de página.