LRU
ÓPTIMO
CLOCK

Evaluación de una caché de metadatos para sistemas de archivos

- Jimmy Banchón
- Vanessa Revelo
- Stephany Samaniego

ESCUELA SUPERIOR POLITECNICA DEL LITORAL

Facultad de Ingeniería en Electricidad y Computación

SISTEMAS OPERATIVOS

ÍNDICE

ESCUELA SUPERIOR POLITECNICA DEL LITORAL
Facultad de Ingeniería en Electricidad y Computación

SISTEMAS OPERATIVOS

INTRODUCCIÓN

Los sistemas de archivos es una colección de archivos y directorios, que a su vez es un componente del sistema operativo. Permite administrar el uso de las memorias, ya sean estas secundarias o terciarias. Cumplen con la función de asignar espacio a los archivos, administrar el espacio libre y el acceso de datos resguardados.

Los archivos pueden leerse, escribirse y cambiar de un directorio a otro. Estos sistemas anteriormente causaban un mayor costo computacional debido a que su acceso a memoria era continuo.

Los sistemas de archivos modernos poseen una caché con la información recientemente usada, lo cual hace que lo requerido sea buscado primero en esta caché y luego en disco, haciéndolo más eficiente y rápido.

En el presente proyecto, hemos aplicado nuestros conocimientos sobre los algoritmos de reemplazo de páginas dados en clases, como son: el LRU (Least Recently Used), CLOCK (también conocido como Second-Chance) y Óptimo; conjunto con estos se utilizó un workload de 3121736 registros.

El objetivo de este proyecto es, una vez hallado los resultados de miss y hit por cada algoritmo con un determinado número de registros, realizar un análisis de rendimiento que nos pueda determinar que algoritmo es mejor en determinadas situaciones.

METODOLOGÍA

Para diseño y solución del tema propuesto, hemos utilizado diferentes métodos que serán explicados en esta sección.

En los sistemas operativos que utilizan paginación como manejo de memoria, los algoritmos de reemplazo de páginas son usados para decidir: que página deber ser removida y establecer el espacio para la página que entra.

Por otra parte, podemos liberar un espacio escribiendo en disco todo su contenido y modificando la tabla de páginas. El espacio liberado ahora puede usarse para alojar a la página por la cual ocurrió un fallo en el proceso.

LRU

El algoritmo LRU (Least Recently Used) utiliza el pasado reciente como un margen de lo que sucederá en el futuro, ya que, al ocurrir un fallo de página, se utiliza la página que no haya sido utilizada hace más tiempo.

Dentro del proceso de implementación, utiliza una tabla conocida como Hashmap, la misma que proporciona todas las operaciones para extraer los keys y los values de cada nodo. Esta aplicación ofrece un rendimiento constante de tiempo para las operaciones básicas de get y put.

Inicialmente, se tiene el BufferedReader cuyo propósito es leer el archivo y enviar cada línea a la función set de la clase LRU, como se muestra en la Imagen 1.

```
BufferedReader br = new BufferedReader(new FileReader(file));
String line;
while((line = br.readLine()) != null) {
    lr.set(line);
}
br.close();
```

Imagen 1: BufferedReader

La clase LRU, tiene un constructor, el cual inicializa las variables que son utilizadas, por ejemplo: capacity que es la capacidad de la cache, entre otras como se observa en la Imagen 2.

```
public class LRUCache {
    int capacity;
    HashMap<String, Node> map = new HashMap<String, Node>();
    Node head=null;
    Node end=null;
    int hit;
    int warmCache;
    int miss;

    public LRUCache(int capacity) {
        this.capacity = capacity;
        this.hit=0;
        this.warmCache=0;
        this.miss=0;
    }
```

Imagen 2: Constructor de LRU

El métodos set, por su parte, se encarga de verificar si el nodo enviado existe o no dentro de la caché, si lo está se procede a eliminar y colocarlo en la cima de la misma. Caso contrario, el nodo procederá a ser creado una vez que se ha verificado que la cache contenga el espacio suficiente para su ingreso, de no ser así, de igual manera eliminará el nodo que se encuentra primera en la cache e inserta el nuevo nodo creado al tope de esta.

```
public void set(String key) {

    if(map.containsKey(key)){
        Node old = map.get(key);
        remove(old);
        setHead(old);
        hit++;
    }else{
        miss++;
        Node created = new Node(key);
        if(map.size()>=capacity){
            map.remove(end.key);
            remove(end);
            setHead(created);
            warmCache++;
        }else{
            setHead(created);
        }
```

Imagen 3: Método set donde se realiza la verificacion e inserción de nodos en la caché

```
}
```

CLOCK (Second Chance)

El algoritmo de Segunda oportunidad, el cual es una modificación del algoritmo FIFO, evita deshacerse de una página de uso frecuente mediante la inspección del bit de referencia de la página más antigua. Si es 0, la página es antigua y no está siendo utilizada, se la reemplaza de manera inmediata; pero si el bit de referencia es 1, este cambia a cero y la página se la coloca al final de páginas, como si hubiera llegado en ese momento a la caché. Se prosigue con la búsqueda siguiendo la lista.

Dentro de la implementación de este algoritmo y como se puede observar en la Imagen 4, se tiene la clase Node, que contiene el key asociado a cada línea del workload, la información del nodo anterior, del nodo posterior y el valor del bit de referencia (0/1). Cabe recalcar que de igual manera se tiene en el LRU(exceptuando por el bit de referencia) , dado que como base utiliza como funcionamiento una pila, y esta es una parte esencial de los algoritmos.

```
package CLOCK;


public class Node {
    public String key;
    Node pre;
    Node next;
    int bit;

    public Node(String key){
        this.key = key;
        this.bit=0;
    }
}
```

Imagen 4: Clase Node

Así mismo esta caché contiene un Hashmap, el cual contendrá la información de cada frame de la misma y asi su capacidad nos permitirá comprobar cuando ha excedido su límite, siendo esto muy útil, ya que inicialmente los frames están vacíos.

Cómo se puede observar en la Imagen 5, también se posee información sobre el nodo Head y nodo End (cuyo funcionamiento es como Tail), dado que funciona como una pila; y de la variable hit, la cual almacena cuantas veces se produce uno.

```
public class ClockCache {
    int capacity;
    HashMap<String, Node> map = new HashMap<String, Node>();
    Node head=null;
    Node end=null;
    int hit;
    int miss;
    int warm;

    public ClockCache(int capacity) {
        this.capacity = capacity;
        this.hit=0;
        this.miss=0;
        this.warm=0;
    }
```

Imagen 5: Clase Clock Cache

Dentro de la misma clase también se tiene una función Remove, como se observa en la Imagen 6, la cual se encarga de remover un nodo de la pila abstracta, esto quiere decir que, el nodo removido desaparece de la abstracción de la pila. De igual manera en el caso en que el removido sea un head o un tail.

```
public void remove(Node n){
    if(n.pre!=null){
        n.pre.next = n.next;
    }else{
        head = n.next;
    }

    if(n.next!=null){
        n.next.pre = n.pre;
    }else{
        end = n.pre;
    }

}
```

Imagen 6: Función Remove

Y por último la función Set, es la encargada de añadir nuevas páginas a nuestra caché. Si la página ya está contenida, se procede a hacer la verificación del bit de referencia, si era 0 entonces lo setea a 1; como se observa en la Imagen 8.

```
public void set(String key) {

    if(map.containsKey(key)){
        Node old = map.get(key);
        if(old.bit==0){
            old.bit=1;
        }
        remove(old);
        setHead(old);
        hit++;
    }else{
        miss++;
        Node created = new Node(key);
        if(map.size()>=capacity){
            warm++;
            if(end.bit==0){
                map.remove(end.key);
                remove(end);
                setHead(created);
                map.put(key, created);
            }else{
                Node old = map.get(end.key);
                old.bit=0;
                remove(old);
                setHead(old);
                set(key);
            }
```

Imagen 7: Función Set

ÓPTIMO

El algoritmo Óptimo, elige la pagina en memoria que vaya a ser referencia más lejos en el futuro y es mayormente utilizado para estudios comparativos. Así mismo, es aquel que produce menos fallo de página para cualquier cantidad de registros.

Para empezar tenemos dos estructuras claves que son los Hashmap. De antemano se necesita saber información futura del workload, por lo tanto en el hashMap mapFuturo, cargamos todo el workload con su respectiva posición de entrada. En el hashmap map tendremos los frames respectivos, como se observa en la Imagen 9.

```
public class OptimoCache {
    int capacity;
    HashMap<String, Node> map = new HashMap<String, Node>();
    HashMap<Integer, String> mapFuturo = new HashMap<Integer, String>();
    int hit;

    public OptimoCache(int capacity, HashMap<Integer, String> mapFuturo) {
        this.capacity = capacity;
        this.hit=0;
        this.mapFuturo = mapFuturo;
    }
```

Imagen 8: Caché del algoritmo óptimo

En este caso, como se indica en la Imagen 10, la clase Node contiene información de la página como tal, la posición que tiene en el workload original y un valor indicándonos cuando este nodo será analizado.

```
public class Node {
    int key;
    int value; // value -1 si ya fue analizado, sino por default 0

    public Node(int key){
        this.key = key;
        this.value = 0;
    }

    public void setAnalized(){
        this.value = -1;
    }

    public void reset(){
        this.value=0;
    }
```

Imagen 9: :Clase node

La función Set, presentada en la Imagen 11, añade páginas al caché y a su vez verifica que las paginas entrantes ya existan en el frame, de ser el caso, solo se añade un hit y si no se hace una comprobación de que si el cache ha excedido su capacidad y de ser no se lo añade sin mayor problema. Pero de ser afirmativo, utiliza dos funciones:

- Find Last: trata de encontrar de todos los frames, cual se demorará más en llegar.
- Remove Last: Remueve esa página y añade la que llega.

```
public void set(String key, int pos) {

    if(map.containsKey(key)){
        hit++;
    }else{
        Node current = new Node(pos);
        if(map.size()>=capacity){
            findLast(pos);
            removeLast();
            map.put(key, current);
        }else{
            map.put(key, current);
        }
    }
}
```

Imagen 10: función Set

FIFO

El algoritmo de reemplazo más sencillo es el primero en entrar, primero en salir (FIFO). Este algoritmo asocia a cada página el instante en el que se trajo a memoria. Cuando hay que reemplazar una página, se elige la más antigua.

Este algoritmo fue el último en ser realizado, y nos basamos en la implementación del algoritmo LRU, en donde solo se modificó el comportamiento de la pila para el caso en que la página ya estaba contenida, ya que solo se genera un hit y no altera el orden de la pila.

ESCUELA SUPERIOR POLITECNICA DEL LITORAL
Facultad de Ingeniería en Electricidad y Computación
SISTEMAS OPERATIVOS

RESULTADOS

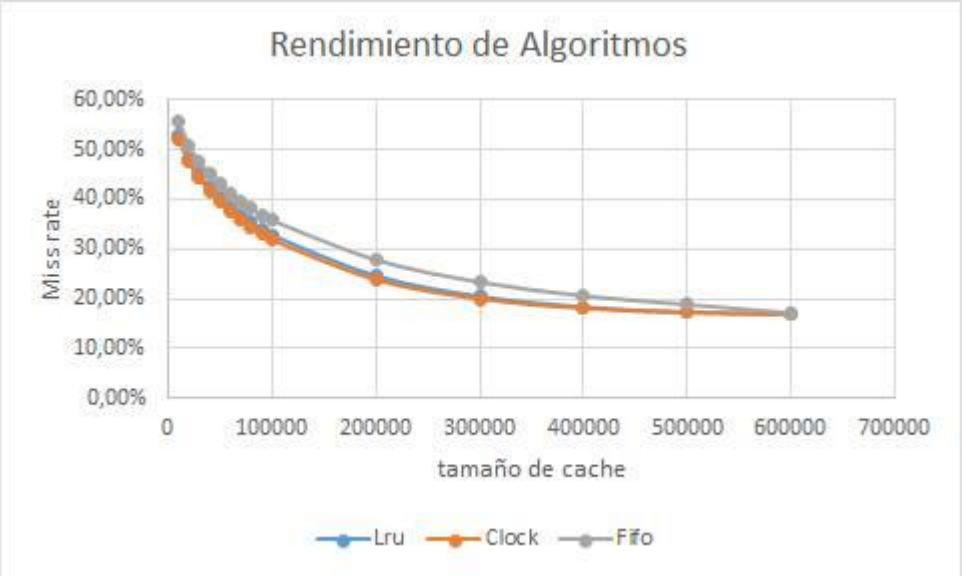

Imagen 11 : Rendimiento de algoritmos

Como se puede observar en la Imagen 11, se analizó el tamaño de la caché versus el porcentaje de miss rate de cada algoritmo implementado.

El algoritmo fifo teniendo un tamaño de cache entre 100000 y 500000 produce mayor cantidad de miss rate, con respecto al algoritmo LRU y CLOCK, pero aun cuando estos dos algoritmos estén muy cercanos entre sí, el LRU resulta ser el mejor por una pequeña diferencia de miss rate con respecto al Clock.

Cabe recalcar, que el algoritmo óptimo no está presente en la imagen superior, debido a que el tiempo de ejecución en los valores determinados de cache es muy alto y no termina de ejecutarse.

A continuación, se presentan los resultados obtenidos por consola de cada uno de los algoritmos implementas. Para este caso en particular hemos analizado con el workload.txt y tamaño de cache de 5000.

Workload.txt LRU 5000

```
C:\Users\soporte2014\Desktop\ProyectoI\ProyectoSOII\dist>java -jar ProyectoSOII.
jar workload.txt lru 5000
Miss rate:              57,12%  (2125741 misses out of 3721736.0 references)
Miss rate(warm cache):  57,06%  (2120741 misses out of 3721736.0-5000 references
```

Miss: 2125741

Hits: 1595995

Warm cache: se especifica los misses cuando la caché ya esta llena.

Miss: 2120741

Workload.txt FIFO 5000

```
C:\Users\soporte2014\Desktop\ProyectoI\ProyectoSOII\dist>java -jar ProyectoSOII.
jar workload.txt fifo 5000
Miss rate:              60,41%  (2248388 misses out of 3721736.0 references)
Miss rate(warm cache):  60,36%  (2243388 misses out of 3721736.0-5000 references
```

Miss: 2248388

Hits: 1473348

Workload CLOCK 5000

```
C:\Users\soporte2014\Desktop\ProyectoI\ProyectoSOII\dist>java -jar ProyectoSOII.
jar workload.txt clock 5000
Miss rate:              56,27%  (2094096 misses out of 3721736.0 references)
Miss rate(warm cache):  56,21%  (2089096 misses out of 3721736.0-5000 references
```

Miss: 2094096

Hits: 1627640

CONCLUSIONES

Se analizó exitosamente los algoritmos: LRU, FIFO, CLOCK, con diferentes tamaños de cache y con el workload dado. El algoritmo Óptimo tuvo algunas dificultades en su implementación, por lo cual el tiempo de ejecución no fue eficiente.

De acuerdo a los resultados analizados en la sección anterior, se concluyó que el algoritmo Clock fue el de mayor rendimiento, ya que mientras mayor tamaño de caché tenía, menos miss rates generaba. Seguido por el algoritmo LRU y por último el FIFO.

REFERENCIAS

- http://lsi.vc.ehu.es/pablogn/docencia/manuales/SO/TemasSOuJaen/MEMORIAVIRTUAL/4ReemplazodePaginas.htm
- http://algoritmosadmonmemoria.weebly.com/reemplazo-de-paginas-optimo.html
- Sistemas Operativos Modernos, Tanembaum, Segundo edición.
- http://www.programcreek.com/2013/03/leetcode-lru-cache-java/