Universidad de Costa Rica
Facultad de Ingeniería
Escuela de Ciencias de la Computación e Informática

CI-0117 Programación Paralela y Concurrente
Grupo 01
I Semestre

II Tarea programada: Contador de etiquetas HTML

Profesor:
Francisco Arroyo

Estudiante:
Jennifer Villalobos Fernández | B67751

12 de junio del 2020

Índice

1. Introducción

Para resolver el problema planteado es necesario un uso simultáneo de los procesos. Para tal propósito se utilizó pthreads. Tal y como lo insinúa el nombre, esta biblioteca se basa en el uso de "hilos", los cuales son considerados "procesos ligeros"que representan el segmento de código que está siedo procesado en un momento dado. El uso de hilos en un proceso hace posible ejecutar segmentos de código (que pueden o no ser los mismos) de manera concurrente, permitiendo disminuir el tiempo requerido para realizar una tarea, y sin tener que crear las estructuras de datos que se necesiten al momento de la creación de un proceso, ya que los hilos comparten ciertos elementos, como lo son las variables globales. Los hilos poseen una única identificación, así como sus variables locales y direcciones de retorno. Para solucionar este trabajo, se hicieron uso de estructuras (struct) en el proceso de retorno de cada hilo, para así facilitar la comunicación entre ellos.

2. Objetivo:

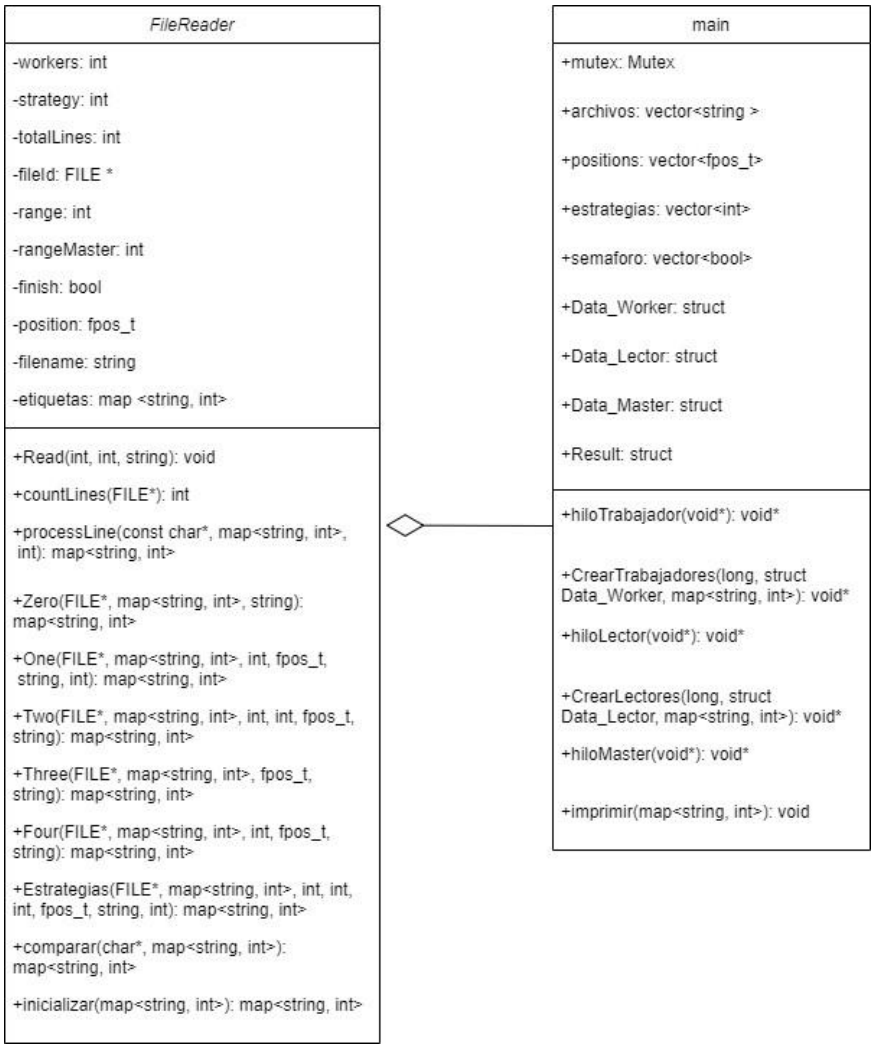
- Crear un programa que posea la funcionalidad de leer archivos y así contar las etiquetas HTML presentes en el mismo.

3. Descripción

Construir un programa en C++ para contar todas las etiquetas que aparecen en un conjunto de archivos HTML que se pasará como parámetros. El programa debe contar todas las etiquetas contenidas en los archivos indicados y mostrar un listado de esas etiquetas y su cantidad de apariciones, en orden alfabético de menor a mayor. Además, el usuario puede indicar, como parámetro también, la cantidad de trabajadores con que pretende realizar el trabajo y la estrategia de asignación de líneas a los trabajadores. Se pretende que cada archivo HTML sea procesado por un conjunto de trabajadores independientes ("contadores"), este es el parámetro que el usuario puede indicar. Cada archivo HTML tendrá entonces un programa "lector" que se encargará de crear una instancia de la clase lectora de archivos (FileReader) con su respectiva estrategia y el nombre de archivo HTML que le toca trabajar, también generar la cantidad de trabajadores indicados por el usuario para dividir el archivo en grupos de líneas y procesarlas independiente. Debe construir una clase C++ (FileReader) que sea capaz de procesar un archivo de texto HTML, con H líneas, siguiendo varias estrategias de acuerdo con la cantidad de trabajadores (t) que el usuario pretenda utilizar. Esta clase será la única encargada de manipular las lecturas del archivo y almacenar no más de 512 bytes del archivo HTML. Las estrategias son las siguientes:

- Dividir el total de líneas del archivo HTML entre los t trabajadores y entregar una porción a cada trabajador (H/t).
- Entregar al primer trabajador (0) todas las líneas cuyo resto de dividir el número de línea entre t sea 0; entregar al segundo trabajador todas las líneas cuyo resto de dividir el número de línea entre t sea 1 y así sucesivamente. Note que en estas dos estrategias la cantidad de líneas asignadas a cada trabajador es aproximadamente la misma.
- Entregar una línea del archivo a cada trabajador por demanda, cada vez que un trabajador requiere una línea le es entregada la siguiente línea disponible del archivo, el lector debe saltar a la siguiente línea. Debe sincronizar los trabajadores para que dos de ellos no reciban la misma línea y el conteo de etiquetas no sea correcto.
- Cada estudiante debe programar una estrategia adicional de su propia invención y que no replique las estrategias indicadas en este enunciado, puede utilizar la teoría revisada de OpenMP para planear su estrategia.

4. Diseño



(a) UML del programa.

5. Desarrollo

FileReader: Para la solución del problema se crea una clase (FileReader) encargada de contener todos los métodos necesarios para la realización de las tareas, tanto de los Lectores como de los trabajadores. En esta clase se encuentran los métodos: Read(), countLines(), processLine(), comparar(), Estrategias(), Zero(), One(), Two(), Three(), Four(), inicializar(), y los métodos get() y set() correspondientes a los atributos privados de la clase. Todas las etiquetas HTML que pueden aparecer en los archivos se encuentran en un mapa, con llave string (nombre de la etiqueta) y valor 0, el cual es el contador de cuántas veces apareció esa etiqueta en el archivo.

El método inicializar() toma el mapa entrante y lo "llenaçon las etiquetas HTML posibles. Así que el método Read(), encargado de inicializar todos los datos necesarios para el "maestro-Lector", lo primero

que realiza es inicializar su propio mapa de etiquetas (el cual se irá actualizando según lo que retorne cada trabajador). Dentro de Read(), luego de inicializar el mapa, se inicializa la cantidad de trabajadores, así como la estrategia a utilizar para leer el archivo. Si la estrategia es 0 (estrategia por defecto: un solo trabajador realiza todo el trabajo) se declara la cantidad de trabajadores como 1. Si la estrategia no es ninguna de las cuatro definidas, es decir, el usuario ingresó otro número, la estrategia cero (por defecto) será la que se ejecutará en su lugar. Luego se abrirá el archivo y se contarán las lineas totales del mismo. Si el total de líneas es divisibles entre la cantidad de trabajadores, el rango de todos los hilos trabajadores será el mismo. En caso contrario, habrá un rangoMaster"que almacenará su rango más el resto. Este rangoMaster será asignado al hilo cero (trabajador con id 0). Por último, se toma la posición del archivo y se guarda en el atributo privado posición. Esto mediante los comandos fgetpos() y fsetpos().

En el método Estrategias() se encuentran las cuatro disponibles. Cada una realiza su trabajo de distinta manera, sin embargo, todas llaman a getline() para obtener la linea que les corresponde, con esa línea ejecutan processLine(), método encargado de separar las palabras del archivo y .encontrar"las etiquetas (palabras encerradas en <>) mediante el uso de expresiones regulares con regex y strtok para la separación. Cuando se obtiene una posible etiqueta, esta se pasa al método comparar(), el cual recorre todo el mapa ya inicializado de etiquetas compárandolas con la palabra entrante. Si ocurre una coincidencia, se aumenta el contador (valor) de la etiqueta respectiva. El mapa se retorna y se actualiza. Este mapa actualizado retorna al método que tomó la línea y la envió a procesar.

Ahora, con las estrategias: la estrategia por defecto, la número cero, corresponde al método Zero(). En este se envía como parámetro el archivo por leer. El método lo procesa hasta llegar al final del mismo y declara que culminó su trabajo, cambiando la variable privada finish por true. Por último retorna el mapa de etiquetas actualizado.

La estrategia número uno consiste en dividir el total de lineas del archivo entre la cantidad de trabajadores. Esta se encuentra en el método One(). Este método toma el archivo ingresado como parámetro y procesa solo una línea, la cual se encuentra en la posicion ingresada también como parámetro. Se posiciona el archivo en su respectivo lugar con el comando fsetpos(). Luego de procesar la línea correspondiente, se cambia la posicion del archivo por la que quedó luego de esa lectura, esta se obtiene mediante fgetpos() y se notifica llamando al método setPos(), el cual actualiza la variable privada posición. Esta estrategia debe repetirse hasta que el hilo trabajador llegue a su respectivo rango. Este rango también es ingresado como parámetro. La cantidad de veces que llama al método se contabilizan por medio del hilo trabajador, este posee un contador de las veces que ha culminado un llamado, es decir, que ha leído una línea. Cuando este contador (que se pasa como parámetro) sea igual al rango, el método declara que finalizó su tarea, cambiando el valor de la variable privada finish a true. Este método también culmina si se ha llegado al final del archivo. Se retorna el mapa de etiquetas y finaliza.

La estrategia número dos consiste en dividir en grupos no contiguos de lineas ($linea \ \% trabajadores == 0$). Esta lee solo las líneas que le correspondan según su id. Actualiza la posición del archivo y cuando se alcance el final de este declara su finish como true. Retorna el mapa actualizado.

La estrategia número tres consiste en entregar las líneas por demanda. Por lo tanto, cada vez que el método sea invocado (no hay ningún hilo ocupando esa línea) este toma la posición actual del archivo, lee un línea, actualiza la posición y retorna el mapa actualizado. En caso de que se llegue al final del archivo, el método declara la variable finish como true.

La última estrategia, la cuatro, es la inventada. Esta es similar a la número 1, al dividir el total de lineas del archivo entre la cantidad de trabajadores, se le otorga un rango.equitativo de líneas a cada trabajador, según el total de líneas del archivo. Sin embargo, estos leen una línea a la vez, pero la acción se repite consecutivamente según ese rango. Por ejemplo, si el rango es 5, cada trabajador lee una línea a la vez, pero realizaría esta tarea 5 veces seguidas, dentro de su Lock(). De esta manera, cada hilo trabajador toma su rango (enviado como parámetro) y ejecuta un ciclo de ese tamaño, leyendo una línea a la vez. Al finalizar, actualiza la posición del archivo en donde quedó, declara su trabajo como finalizado y retorna el mapa de etiquetas actualizado.

Con respecto al funcionamiento del main: los hilos. En esta clase se encuentran cuatro estructuras, una llamada Data Worker, que son todos los datos necesarios que se le deben enviar a cada hilo trabajador, desde su maestroLector. Otra es Data Lector, que almacena los datos necesarios para la creación de cada

maestroLector, es decir, para cada archivo que ingrese se creará uno. Data Master guarda los parámetros que ingresa el usuario y es lo que se le pasa al hilo Maestro para que realice sus funciones. Y la última, Result, que corresponde a la estructura que almacena el mapa de etiquetas HTML actualizado, por cada trabajador y por cada maestroLector.

Estas son utilizadas en los siguientes métodos: el main se encarga de crear solo un hilo maestro, el que controla y crea a los maestros Lector. A este hilo maestro se le pasa como parámetro una estructura Data Master, con los datos ingresados. El hilo Maestro primero verifica si los argumentos ingresados son los correctos, en caso contrario indica al usuario como debe ingresarlos. Luego de la verificación toma la cantidad de trabajadores y se la declara a una estructura Data Lector (ya que aunque sean distintos archivos, este dato es el mismo para todos). Toma el argumento de las estrategias, las separa y las ingresa a un vector global de estrategias. Cada casilla de ese vector, es decir, cada índice, corresponde al id de cada Lector. En otras palabras, el Lector con id 1, tiene su estrategia de lectura en el index 1 del vector de estrategias. Lo mismo sucede con los archivos, luego de terminar de ingresar las estrategias en sus lugares respectivos, el hilo maestro continúa leyendo los demás argumentos ingresados: el (o los) archivos. Cada que lee un archivo, lo asocia a un vector de archivos (en realidad, un vector de strings, ya que almacena el nombre para luego poder abrirlo). De esta manera, por el momento hay dos vectores globales: uno de archvios y otro de estrategias. El Lector con id 1, leerá el archivo de index 1 en el vector archivos, y lo hará con la estrategia indicada en el index 1 del vector estrategias, así sucesivamente con todos los archivos y estrategias ingresadas.

Cuando el hilo master termina de inicializar los datos, llama al método Crear Lectores(), con la cantidad de archivos ingresados por el usuario. Este método le otorga una identificación al Lector, la agrega a la estructura Data Lector ya creada por hilo Master, y la pasa como parámetro para la creación del hilo Lector. El hilo Lector a su vez, crea una estructura Data Worker, en la cual ingresa los datos necesarios para el (o los) hilo Trabajador. Estos datos los obtiene al inicializarse con el método Read(), y con los parámetros enviados por hilo Master. Al completar los datos de la estructura, obtiene la posición inicial del archivo mediante getPos(), y el dato lo ingresa al vector global posiciones, que al igual que los anteriores, posee la posición actual del archivo x, en el index x. Además, pone en false a ese index x del vctor global Semáforo. Luego, llama al método Crear Trabajadores(), con la cantidad que hilo Master le envió.

Crear Trabajadores() realiza las mismas acciones que Crear Lectores(), le otorga una identificación a cada trabajador, agregándola a la estructura Data Worker ingresada como parámetro por el Lector. Al crearse cada hilo se realiza la última función: se llama a hilo Trabajador. Cada trabajador obtiene como parámetro una estructura Data Worker con los datos necesarios para llamar al método Estrategias(), pero ¿cómo se controla si hay otro trabajador ingresadndo a esa posición del archivo y tomando esa línea? por medio del vector global semáforo, que anteriormente fue inicializado en false por el Lector. Este vector tiene en el index correspondiente a cada Lector un bool, si está en false significa que nadie ha tomado esa posición. Si está en true, significa que ya hay un trabajador realizando cierta estrategia en una línea del archivo. El hilo trabajador va a preguntar si esa variable está ocupada o no, si lo está, espera, y vuelve a preguntar. Si está libre, la declara ocupada y reliza la estrategia correspondiente. Al finalizar, actualiza el vector de posiciones, pone en desocupada a la línea y consulta si su variable privada finish es true, si es false entonces vuelve a realizar el proceso, hasta que esta sea verdadera y signifique que su trabajo terminó.

Cuando el hilo trabajador culmina su tarea, actualiza el mapa de estrategias que posee, lo ingresa en la estructura Result, y se lo retorna al hilo Lector que lo invocó. En el join de los hilos Trabajadores, se actualiza el mapa de estrategias del Lector cada que algún hilo termina. Cuando todos terminen su tarea, el mapa (en una estructra Result) es retornado al hilo Lector que los invocó. A su vez, este hilo Lector actualiza su propio mapa de estrategias y lo retorna dentro de un struct Result al hilo Master. Cada Lector realiza esta acción, así que en el join de los Lectores, se actualiza el mapa cada que uno de ellos finaliza. Cuando todos los Lectores terminen, se le retorna el mapa actualizado al hilo Master. Este lo retorna con su join y el main se encarga de llamar al método imprimir() con el mapa resultante. Este mapa posee la acumulación de todas las etiquetas encontradas en todos los archivos.

6. Manual de usuario

Requerimientos de Software

- **Sistema Operativo:** Linux.
- **Arquitectura:** 64 bits.
- **Ambiente:** Code::Blocks o Terminal.

Compilación

Para compilar el programa puede usarse el Makefile, con solo llamar al comando make en terminal. También se puede utilizar `g++`, así como `lpthread` en la siguiente sentencia:

```
$ g++ FileReader.cpp main.cpp Mutex.cpp -lpthread -o nombre_ejecutable
```

Especificación de las funciones del programa

Para ejecutar el programa, se debe insertar el nombre del ejecutable, la cantidad de trabajadores, la estrategia deseada para cada archivo (si son varios archivos, insertarlas pegadas) y los archivos por leer. El comando completo sería de esta manera:

```
$ ./nombre_ejecutable <cantidad_trabajadores> <estrategias(pegadas)> <archivo.html> <archivo.html> ...
```

7. Casos de Prueba

Pruebas FileReader: Estas pruebas verifican el funcionamiento del programa FileReader.
El código utilizado corresponde a:

```
#include "FileReader.h"
#include <map>
#include <stdio.h>
#include <string.h>
#include <string>
#include <regex>
#include <iostream>
#include <fstream>


/*
*Metodo encargado de inicializar los datos necesitados por el Lector.
*/
void FileReader::Read(int trabajadores, int estrategia, std::string archivo)
{
    fpos_t inicio;
    this->etiquetas = inicializar(this->etiquetas);


    this->filename = archivo;
    const char * name = filename.c_str();
    this->fileId = fopen( name, "r" );

    if ( NULL == this->fileId )
    {
        perror( "El archivo no se pudo abrir.\n");
        exit( 2 );
    }
    //Se toma la posicion inicial del archivo.
    fgetpos(this->fileId, &inicio);


    //Actualizacion de las variables privadas.
    this->workers = trabajadores;
    this->strategy = estrategia;

    if(this->strategy == 0)
    {
        this->workers = 1;
    }
    else if(this->strategy > 4)
    {
        printf("No existe una estrategia con ese numero. Se utiliza la estrategia por
            default: 0.\n");
        this->strategy = 0;
        this->workers = 1;
    }

    this->totalLines = countLines(this->fileId);
    printf("Total lines: %d\n", this->totalLines);

```

```
51
52      //Si el total de lineas es divisible entre la cantidad de trabajadores.
53      if(this->totalLines%this->workers == 0)
54      {
55          this->range = this->totalLines/this->workers;
56          this->rangeMaster = this->totalLines/this->workers;
57      }
58      //En caso contrario, se le asigna a algun hilo el rangeMaster, que consiste
59      //en el rango mas el "sobro" de la division.
60      else
61      {
62          this->range = this->totalLines/this->workers;
63          this->rangeMaster = (this->totalLines-(range*(this->workers-1)));
64      }
65
66
67      //Se posiciona al inicio del archivo.
68      //fsetpos(this->fileId, &this->position);
69      setPos(inicio);
70
71
72      fclose( this->fileId );
73
74  }
75
76
77  /*
78  *Metodo encargado de contar las lineas del archivo entrante
79  */
80  int FileReader::countLines(FILE * archivo)
81  {
82      int cant_lineas = 0;
83      int chars;
84      size_t size;
85      char * line;
86      size = 512;
87
88      line = (char *) calloc( 512, 1 );
89
90      do
91      {
92          chars = getline( & line, & size, archivo );
93          if( chars > 0 )
94          {
95              cant_lineas++;
96          }
97      }
98      while ( chars > 0 );
99
100     free(line);
101     return cant_lineas;
102 }
103
104 /*
105 *Metodo encargado de procesar una linea del archivo.
106 */
```

```
107 std::map<std::string, int> FileReader::processLine( const char * line, std::map<std::
        string, int> etiquetas)
108 {
109     char * token;
110     std::regex texto( ">[^<]*<" );
111     std::string ecci;
112 
113     ecci = regex_replace( line, texto, "> <" );
114 //   std::cout << "ecci = " << ecci << std::endl;
115 
116     token = strtok( (char * ) line, "< >\t\n" );
117 
118     while ( NULL != token )
119     {
120         etiquetas = comparar(token, etiquetas); //Se compara la palabra obtenida con
                las etiquetas guardadas
121         token = strtok( NULL, "< >\t\n");
122     }
123 
124     return etiquetas;
125 }
126 
127 
128 /*
129 *Metodo encargado de comparar la palabra entrante con cada una de las
130 *etiquetas guardadas en el mapa.
131 */
132 std::map<std::string, int> FileReader::comparar(char * token, std::map<std::string, int
        > etiquetas)
133 {
134 
135     std::map<std::string, int>:: iterator it = etiquetas.begin();
136     std::string key;
137 
138     while(it != etiquetas.end())
139     {
140         key = it->first;
141 
142         //En caso de que la palabra sea una etiqueta, se aumenta el contador de esta
143         if((key.compare(token)) == 0)
144         {
145             etiquetas[key]++;
146         }
147 
148         it++;
149     }
150     return etiquetas;
151 }
152 
153 
154 /*
155 *Metodo que almacena todas las estrategias de lectura.
156 */
157 std::map<std::string, int> FileReader::Estrategias(FILE * archivo, std::map<std::string
        , int> etiquetas, int rango, int id, int estrategia, std::string filename, fpos_t
        posicion, int contador)
```

```
158 {
159     switch(estrategia)
160     {
161     case 0:
162         etiquetas = Zero(archivo, etiquetas, filename);
163         break;
164     case 1:
165         etiquetas = One(archivo, etiquetas, rango, posicion, filename, contador);
166         break;
167     case 2:
168         etiquetas = Two(archivo, etiquetas, rango, id, posicion, filename);
169         break;
170     case 3:
171         etiquetas = Three(archivo, etiquetas, posicion, filename);
172         break;
173     case 4:
174         etiquetas = Four(archivo, etiquetas, rango, posicion, filename);
175     }
176 
177     return etiquetas;
178 }
179 
180 
181 /*
182 *Estrategia por defecto: todo el trabajo lo
183 *realiza un solo trabajador.
184 */
185 std::map<std::string, int> FileReader::Zero(FILE * archivo, std::map<std::string, int>
        etiquetas, std::string filename)
186 {
187     int chars;
188     size_t size;
189     char * line = NULL;
190     size = 512;
191     line = (char *) calloc( 512, 1 );
192 
193     const char * name = filename.c_str();
194     FILE * fichero = fopen(name, "r");
195 
196     do
197     {
198         chars = getline( & line, & size, fichero );
199 
200         if( chars > 0 )
201         {
202             etiquetas = processLine( line, etiquetas);  //Se procesa cada linea del
                archivo, los contadores del mapa se actualizan
203         }
204     }
205     while ( chars > 0 );
206 
207     setFinish();
208     free(line);
209     fclose(fichero);
210 
211     return etiquetas;
```

```
212 }
213
214
215 /*
216 *Estrategia #1: dividir el total de lineas del archivo
217 *entre la cantidad de trabajadores.
218 */
219 std::map<std::string, int> FileReader::One(FILE * archivo, std::map<std::string, int>
        etiquetas, int rango, fpos_t posicion, std::string filename, int contador)
220 {
221     int chars;
222     size_t size;
223     char * line = NULL;
224     fpos_t nueva;
225
226     size = 512;
227     line = (char *) calloc( 512, 1 );
228
229     const char * name = filename.c_str();
230     FILE * fichero = fopen(name, "r");
231
232     fsetpos(fichero, &posicion);
233
234     if(contador == rango) //Si ya hizo su bloque correspondiente
235     {
236         setFinish();
237     }
238     else
239     {
240         chars = getline( & line, & size, fichero );
241
242         if( chars > 0 )
243         {
244             etiquetas = processLine( line, etiquetas);  //Se procesa cada linea del
                    archivo, los contadores del mapa se actualizan
245             fgetpos(fichero, &nueva);
246             setPos(nueva);
247         }
248         else
249         {
250             setFinish();
251         }
252     }
253
254
255     free(line);
256     fclose(fichero);
257
258     return etiquetas;
259 }
260
261
262 /*
263 *Estrategia #2: dividir en grupos no contiguos de lineas
264 *linea%trabajadores == 0.
265 */
```

```
std::map<std::string, int> FileReader::Two(FILE * archivo, std::map<std::string, int>
    etiquetas, int rango, int id, fpos_t posicion, std::string filename)
{
    int chars;
    size_t size;
    int num_linea = 0;
    int resultado;
    char * line;
    fpos_t nueva;

    //Para evitar errores al dividir por cero (id del primer trabajador)
    int identificacion = id+1;

    size = 512;
    line = (char *) calloc( 512, 1 );

    const char * name = filename.c_str();
    FILE * fichero = fopen(name, "r");

    fsetpos(fichero, &posicion);


    resultado = num_linea % identificacion;

    do
    {
        chars = getline( & line, & size, fichero );

        if( (chars > 0) && (resultado == 0) )  //Si el residuo da 0, esa linea le
            pertenece.
        {
            etiquetas = processLine( line, etiquetas);  //Se procesa cada linea del
                archivo, los contadores del mapa se actualizan
        }
        num_linea++;
        resultado = num_linea % identificacion;
    }
    while ( (chars > 0) );


    fgetpos(fichero, &nueva);
    setPos(nueva);
    setFinish();

    fclose(fichero);
    free(line);
    return etiquetas;
}


/*
*Estrategia #3: entregar las lineas por demanda.
*/
std::map<std::string, int> FileReader::Three(FILE * archivo, std::map<std::string, int>
     etiquetas, fpos_t posicion, std::string filename)
{
```

```
318         int chars;
319         size_t size;
320         char * line;
321         fpos_t nueva;
322 
323         size = 512;
324         line = (char *) calloc( 512, 1 );
325 
326         const char * name = filename.c_str();
327         FILE * fichero = fopen(name, "r");
328 
329         fsetpos(fichero, &posicion);
330 
331         chars = getline( & line, & size, fichero );
332 
333         //Se procesa solo una linea del archivo y los contadores del mapa se actualizan.
334         if( chars > 0 )
335         {
336             etiquetas = processLine( line, etiquetas);
337         }
338         else
339         {
340             setFinish();
341         }
342 
343         fgetpos(fichero, &nueva);
344         setPos(nueva);
345 
346         free(line);
347         fclose(fichero);
348         return etiquetas;
349     }
350 
351 
352     /*
353     *Estrategia #4: inventada.
354     *
355     */
356     std::map<std::string, int> FileReader::Four(FILE * archivo, std::map<std::string, int>
        etiquetas, int rango, fpos_t posicion, std::string filename)
357     {
358         int chars;
359         size_t size;
360         char * line = NULL;
361         int contador = 0; //Almacena las lineas que lleva leidas hasta el momento.
362         fpos_t nueva;
363 
364         size = 512;
365         line = (char *) calloc( 512, 1 );
366 
367         const char * name = filename.c_str();
368         FILE * fichero = fopen(name, "r");
369 
370         fsetpos(fichero, &posicion);
371 
372 
```

```
373         do
374         {
375             chars = getline( & line, & size, fichero );
376 
377             if( chars > 0 )
378             {
379                 etiquetas = processLine( line, etiquetas);  //Se procesa cada linea del
                        archivo, los contadores del mapa se actualizan
380             }
381             contador++;
382         }
383         while ( contador < rango );  //Si todavia no llega al rango correspondiente,
                continua. (chars != -1) &&
384 
385 
386         fgetpos(fichero, &nueva);
387         setPos(nueva);
388         setFinish();
389 
390         free(line);
391         fclose(fichero);
392 
393         return etiquetas;
394     }
395 
396 
397     /*
398     *Metodo encargado de inicializar y retornar el mapa con las etiquetas
399     *HTML mas probables.
400     */
401     std::map<std::string, int> FileReader::inicializar(std::map<std::string, int> etiquetas
            )
402     {
403 
404         //std::map<std::string, int> etiquetas;
405 
406         //Todas las etiquetas
407     }
408 
409     /*
410     *Metodos get() y set()
411     */
```

Los resultados obtenidos con las distintas estrategias se pueden verificar en las siguientes capturas:

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
jennifer@jennifer-VirtualBox:~/Documentos/C++/FileReader$ ./file 1 0 ejemplo.html
Crea hilo master.
Crea Lector 0.
Crea trabajador 0.
Trabajador 0 finaliza.
Lector 0 finaliza.
Hilo Master acumula los resultados.
Lectura completada.
Etiqueta: !DOCTYPE. Cantidad de veces encontrada: 1.
Etiqueta: /body. Cantidad de veces encontrada: 1.
Etiqueta: /head. Cantidad de veces encontrada: 1.
Etiqueta: /html. Cantidad de veces encontrada: 1.
Etiqueta: /title. Cantidad de veces encontrada: 1.
Etiqueta: body. Cantidad de veces encontrada: 1.
Etiqueta: head. Cantidad de veces encontrada: 1.
Etiqueta: html. Cantidad de veces encontrada: 2.
Etiqueta: title. Cantidad de veces encontrada: 1.
Tiempo transcurrido: 23907.000000.
jennifer@jennifer-VirtualBox:~/Documentos/C++/FileReader$
```

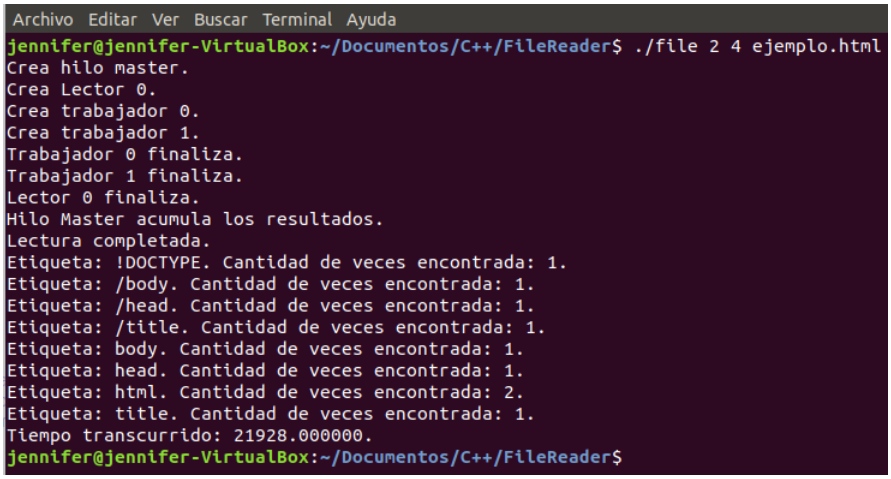
(b) Salida del programa con 1 trabajador, estrategia 0 y un archivo HTML.

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
jennifer@jennifer-VirtualBox:~/Documentos/C++/FileReader$ ./file 3 1 ejemplo.html
Crea hilo master.
Crea Lector 0.
Crea trabajador 0.
Crea trabajador 1.
Crea trabajador 2.
Trabajador 0 finaliza.
Trabajador 1 finaliza.
Trabajador 2 finaliza.
Lector 0 finaliza.
Hilo Master acumula los resultados.
Lectura completada.
Etiqueta: !DOCTYPE. Cantidad de veces encontrada: 1.
Etiqueta: /body. Cantidad de veces encontrada: 1.
Etiqueta: /head. Cantidad de veces encontrada: 1.
Etiqueta: /html. Cantidad de veces encontrada: 1.
Etiqueta: /title. Cantidad de veces encontrada: 1.
Etiqueta: body. Cantidad de veces encontrada: 1.
Etiqueta: head. Cantidad de veces encontrada: 1.
Etiqueta: html. Cantidad de veces encontrada: 2.
Etiqueta: title. Cantidad de veces encontrada: 1.
Tiempo transcurrido: 35332.000000.
jennifer@jennifer-VirtualBox:~/Documentos/C++/FileReader$
```

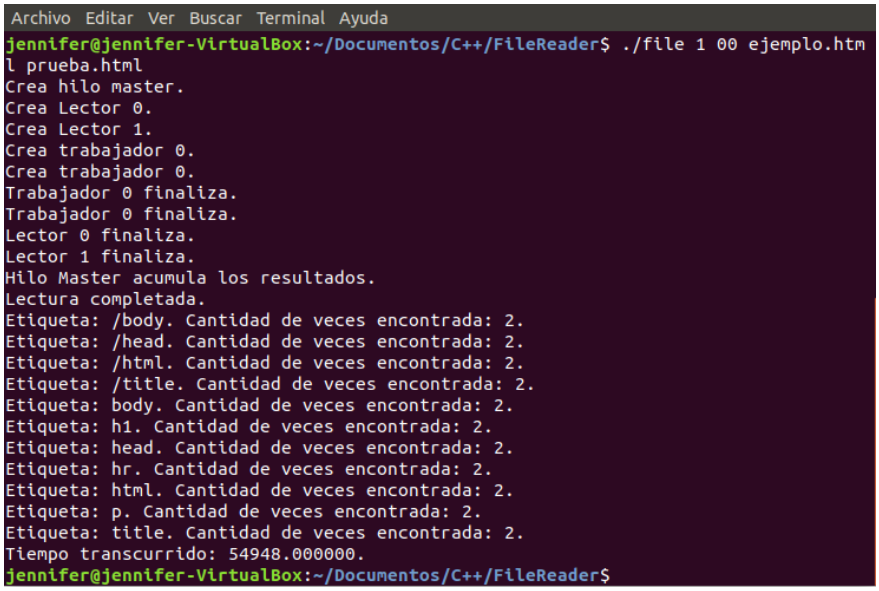
(c) Salida del programa con 3 trabajadores, estrategia 1 y un archivo HTML.

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
jennifer@jennifer-VirtualBox:~/Documentos/C++/FileReader$ ./file 2 2 ejemplo.html
Crea hilo master.
Crea Lector 0.
Crea trabajador 0.
Crea trabajador 1.
Trabajador 0 finaliza.
Trabajador 1 finaliza.
Lector 0 finaliza.
Hilo Master acumula los resultados.
Lectura completada.
Etiqueta: !DOCTYPE. Cantidad de veces encontrada: 1.
Etiqueta: /head. Cantidad de veces encontrada: 1.
Etiqueta: /html. Cantidad de veces encontrada: 1.
Etiqueta: head. Cantidad de veces encontrada: 1.
Etiqueta: html. Cantidad de veces encontrada: 1.
Tiempo transcurrido: 21160.000000.
jennifer@jennifer-VirtualBox:~/Documentos/C++/FileReader$
```

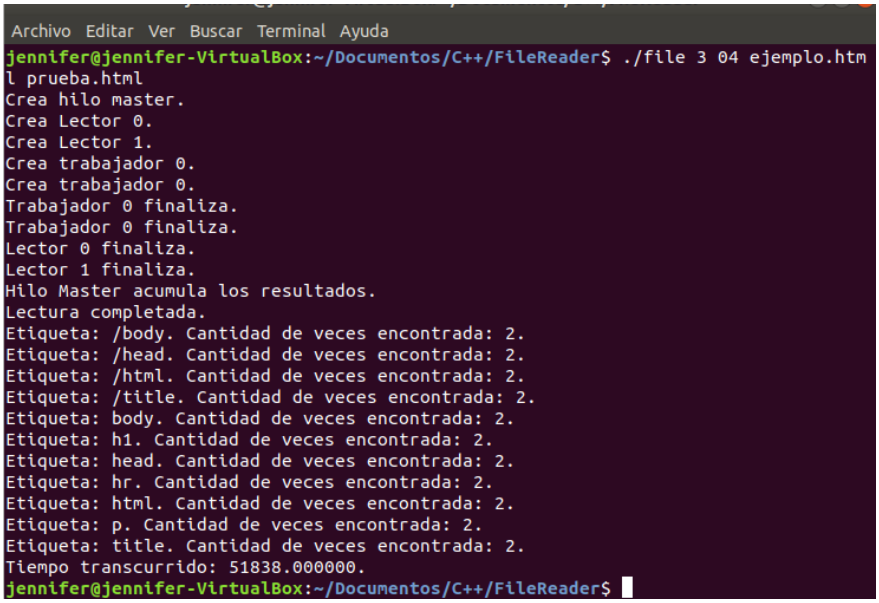
(d) Salida del programa con 2 trabajadores, estrategia 2 y un archivo HTML.



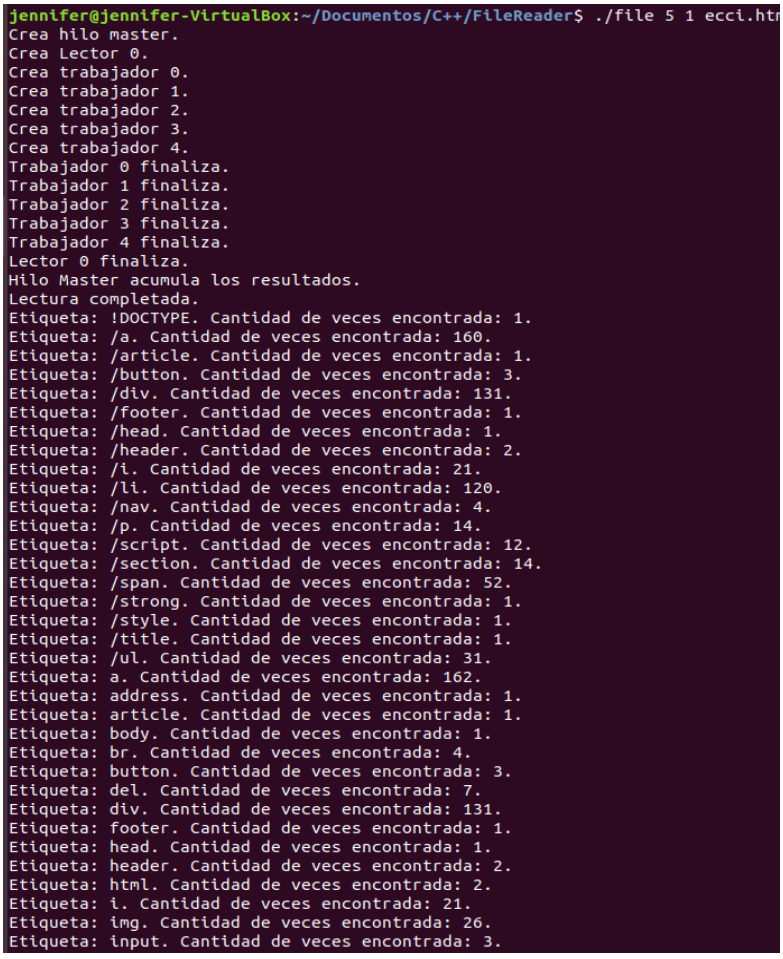
(e) Salida del programa con 2 trabajadores, estrategia 4 y un archivo HTML.



(f) Salida del programa con 1 trabajador, estrategia 0 para el primer archivo y 0 para el segundo.



(g) Salida del programa con 3 trabajadores, estrategia 0 para el primer archivo y 4 para el segundo.



(h) Salida del programa con 5 trabajadores, estrategia 1 y archivo provisto por el profesor: ecci.html.

Control de errores:

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
jennifer@jennifer-VirtualBox:~/Documentos/C++/FileReader$ ./file 5 4 a.html
Crea hilo master.
Crea Lector 0.
El archivo no se pudo abrir.
: No such file or directory
jennifer@jennifer-VirtualBox:~/Documentos/C++/FileReader$
```

(i) Salida del programa cuando se ingresa un archivo inexistente.

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
jennifer@jennifer-VirtualBox:~/Documentos/C++/FileReader$ ./file 5 8 prueba.html
Crea hilo master.
Crea Lector 0.
No existe una estrategia con ese numero. Se utiliza la estrategia por default: 0.
Crea trabajador 0.
Trabajador 0 finaliza.
Lector 0 finaliza.
Hilo Master acumula los resultados.
Lectura completada.
Etiqueta: /body. Cantidad de veces encontrada: 1.
Etiqueta: /head. Cantidad de veces encontrada: 1.
Etiqueta: /html. Cantidad de veces encontrada: 1.
Etiqueta: /title. Cantidad de veces encontrada: 1.
Etiqueta: body. Cantidad de veces encontrada: 1.
Etiqueta: h1. Cantidad de veces encontrada: 1.
Etiqueta: head. Cantidad de veces encontrada: 1.
Etiqueta: hr. Cantidad de veces encontrada: 1.
Etiqueta: html. Cantidad de veces encontrada: 1.
Etiqueta: p. Cantidad de veces encontrada: 1.
Etiqueta: title. Cantidad de veces encontrada: 1.
Tiempo transcurrido: 28710.000000.
jennifer@jennifer-VirtualBox:~/Documentos/C++/FileReader$
```

(j) Salida del programa cuando se ingresa un número de estrategia inexistente.

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
jennifer@jennifer-VirtualBox:~/Documentos/C++/FileReader$ ./file 5 prueba.html
Crea hilo master.
Por favor ingrese: <cantidad de trabajadores> <estrategia(s) a utilizar (pegadas)>
<archivo(s) a utilizar>.
 Estrategias:
 0 - Un solo trabajador realiza la tarea.
 1 - Total de lineas/cantidad de trabajadores.
 2 - Grupos no contiguos de lineas.
 3 - Entregar las lineas por demanda.
 3 - Leer lineas por bloques.
jennifer@jennifer-VirtualBox:~/Documentos/C++/FileReader$
```

(k) Salida del programa cuando se ingresan menos argumentos de los esperados.