



Universidad Tecnológica Centroamericana
UNITEC

Proyecto OS2

Presentado por:

José Lobo 12241127

Presentado a:

Roman Arturo Pineda Soto

Clase y Sección:

1401 SISTEMAS OPERATIVOS II 2025Q2

Fecha de Entrega:

Tegucigalpa M.D.C.

29 de julio del 2025

Índice

Introducción	2
Instrucciones del Trabajo	3
Código.....	4
Estructura de Carpetas	4
Estructura del Programa.....	5
Page.hpp.....	5
Instruccion.hpp.....	6
Proceso.hpp.....	6
AdminMemoria.hpp.....	7
Main.cpp	8
Descripción de AdminMemoria.cpp	9
Programas Adicionales	10
Como Ejecutar	10
Resultados	11
bzip.trace	11
gcc.trace	12
Aleatorio.trace.....	13
Salida de Mapa de Memoria	14
Preguntas.....	15
Conclusión	16
Enlace al Repositorio de GitHub	17

Introducción

Los Administradores de Memoria es un área bastante importante para los sistemas operativos ya que manejar la eficiencia en que se maneja la memoria es importante. Existe mucha teoría al respecto de porque se administra la memoria de cierta manera o porque se si se necesita un administrador de memoria y no manejar toda la memoria en un solo archivo. La principal causa para utilizar memoria principal y secundaria es por los tiempos de accesos y los costos de almacenarlos.

La memoria secundaria tiene el propósito de almacenar información por un largo tiempo, en comparación con la memoria principal que su objetivo es pasar la información lo más rápido posible, sin embargo, no es necesario almacenarlo por un largo margen de tiempo.

Como tenemos bastante limitada la memoria secundaria es necesario planear de manera correcta que direcciones son las que hay que remplazar. Unas de las técnicas aplicadas es la paginación, la cual, es juntar las memorias en bloques para que estas reduzcan los page faults. Eso funciona ya que los procesos utilizan memoria muy cerca de ellos por lo cual, mientras más grande sean las paginas menos page faults van a contener. Sin embargo, eso tendrá un gran costo en otros aspectos.

Para entender los conceptos mas a fondo realizaremos un proyecto el cual simula correr un programa de este tipo para simular los remplazos de paginas en memoria. Los archivos que estaremos analizando tiene la siguiente estructura.



Instrucciones del Trabajo

1) Cargador de strings de referencia a memoria

El primer módulo consiste en el cargador de archivos de trazas de memoria, que contendrán 1,000,000 de referencias a memoria (direcciones de 32 bits) apropiadas para la arquitectura Intel x86 de 32 bits, generadas de correr de forma real el compilador GCC, y el utilitario bzip en Linux. Los archivos de traza (.trace) están disponibles en Canvas.

El archivo contiene la dirección a leer, y la operación a ejecutar R para lectura, y W para escritura. Debe notar que una escritura a página, la “ensucia”, y como tal debe ser escrita a disco antes de ser escogida como víctima.

Este módulo cargador debe mostrar gráficamente un mapa de memoria, mostrando la utilización del mismo, de la misma manera que se presentó en la clase.

2) Manejador de memoria paginada

El módulo manejador de memoria paginada servirá para investigar la efectividad relativa de las estrategias de reemplazo de páginas FIFO, LRU y OPT para algunos parámetros del sistema. El objeto es medir el número de page faults generados por cada estrategia.

Para este módulo, se debe asumir que cada proceso utiliza el espacio de direccionamiento completo, de acuerdo al string de referencia procesado por el módulo 1, con una limitación en la cantidad de memoria física (frames) de acuerdo a los parámetros que se indican a continuación.

El reporte de desempeño de la simulación de memoria paginada deberá ser ejecutado para todas las combinaciones de los siguientes parámetros: Cantidad de frames en la memoria física f: 10,50,100

El reporte de proyecto deberá incluir un análisis funcional del diseño del programa, y de sus estructuras de datos y algoritmos usados para el manejo de memoria, así como la tabla resumen mostrando el número de page faults por cada una de las combinaciones anteriormente indicadas, reemplazos realizados, escrituras a disco, y estimar un EAT para un valor de 100 ns de acceso a memoria.

Al final de la corrida, debe imprimir los resultados de la simulación. Adicionalmente, se deberá discutir y contestar las siguientes preguntas:

- 1) ¿Qué estrategia de reemplazo de páginas escogería usted y por qué? Debe considerar los resultados obtenidos y el esfuerzo que le llevó implementar cada estrategia. Discuta lo que sus resultados muestran acerca de los méritos relativos de FIFO, LRU y OPT para cada una de las diferentes combinaciones de parámetros.
- 2) ¿Qué aspectos de la administración de memoria encontró que fue más difícil implementar?
- 3) ¿Qué aspectos de la administración de la memoria encontró más fácil implementar?

Código

Estructura de Carpetas

La estructura realizada para este proyecto es la siguiente:

- **ArchivosParaTrabajar** es una carpeta dedicada para guardar los archivos con el propósito de analizar.
- **HerramientasDeAyuda** es donde se almacena esa documentación y también un programa en Python el cual genera un archivo con direcciones aleatorias para probar porque estos algoritmos funcionan mejor en la practica.
- **Bin** es solo para tener las salidas de las ejecuciones.
- **include** es donde se manejan todos los archivos hpp
- **src** es donde se manejan los archivos cpp



El archivo build.sh sirve para ejecutar el archivo. Para que el archivo funcione de manera correcta es necesario darle los permisos suficientes.

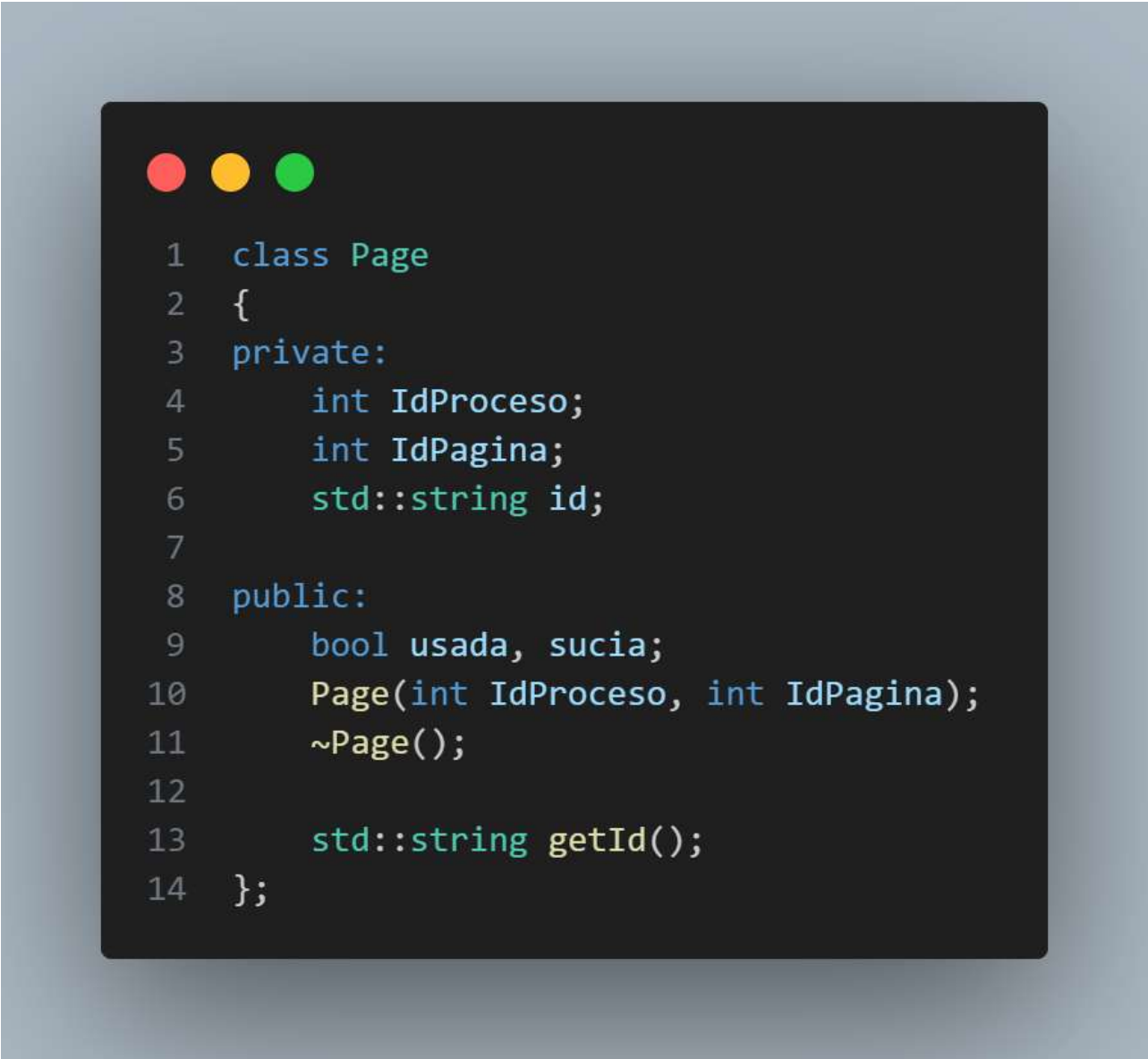
Estructura del Programa

Para comprender mejor la estructura del programa vamos a analizar cada uno de los hpp que contienen. Esa es la manera mas sencilla de comprenderlo sin irnos a bastantes sintaxis de por medio.

Iniciemos con la estructura más pequeña

Page.hpp

Page representa la unidad de pagina en el algoritmo de remplazo de algoritmo. Esa se utiliza como estructura para manejar si una pagina ha sido usada y si esta sucia. Es necesario que le pasemos un parámetro de que proceso pertenece la pagina y la id de la página.

A screenshot of a code editor with a dark background and light-colored text. At the top left, there are three colored circles (red, yellow, green) representing window control buttons. The code is written in a C++ style, defining a class named 'Page'. The code is as follows:

```
1  class Page
2  {
3  private:
4      int IdProceso;
5      int IdPagina;
6      std::string id;
7
8  public:
9      bool usada, sucia;
10     Page(int IdProceso, int IdPagina);
11     ~Page();
12
13     std::string getId();
14 };
```

Instruccion.hpp

Este se utiliza para mantener de forma estructurada la instrucción actual del archivo leído. Para ello se separa en dirección y Operación que es lo que se nos entrega en los archivos.

```
1  class Instruccion
2  {
3  private:
4      std::string entrada;
5      std::string direccionString;
6      char operacion;
7      void procesarEntrada();
8
9  public:
10     Instruccion(const std::string &linea);
11     ~Instruccion();
12     uint32_t getDireccion() const;
13     char getOperacion() const;
14     void setEntrada(const std::string &nuevaEntrada);
15     void mostrar() const;
16     uint32_t getPaginaId(uint32_t tam_paginas) const;
17 };
```

Proceso.hpp

Ese objeto se creo con el propósito de poder manejar multiples ejecuciones al mismo tiempo. Sin embargo aun no esta completo esa funcionalidad del código. Por lo cual, el proceso se vincula a uno de los archivos ejemplos los cuales esta clase se encarga de administrar los parámetros de dicho proceso.

```
1  class Proceso
2  {
3  private:
4      std::string ruta, linea;
5      int idProceso;
6      int promedioBytesPorLinea;
7      std::unique_ptr<std::ifstream> archivo;
8      std::unique_ptr<Instruccion> instruccionActual;
9      std::streamsize tamaño;
10     int estadoInstrucciones, instruccionesLeídas;
11     bool CalcularTamaño();
12
13 public:
14     Proceso(std::string ruta, int idProceso, int promedioBytesPorLinea);
15     bool abrirArchivo();
16     bool cerrarArchivo();
17     bool leerLinea();
18     bool estaAbierto();
19     bool findArchivo();
20     int getIdProceso() const;
21     int getInstruccionesRestantes() const;
22     int getEstadoInstrucciones() const;
23     const Instruccion *getInstruccionActual() const;
24     std::string getLinea() const;
25 };
```

AdminMemoria.hpp

Es la clase principal que maneja todo como que si fuera un procesador. Incluye el planificador que ofrece los 3 tipos de planificadores distintos y también ofrece cambiar el remplazo de algoritmo. Ofrece la opción de personalizar muchos otros factores como los frames, tamaño por ciclo y los tamaños de las páginas.

```
1 class AdminMemoria
2 {
3 private:
4     size_t frames;           // 10 recomendado
5     uint32_t tam_paginas;    // 4096 recomendado
6     int promedioBytesPorLinea; // 11 recomendado
7     int tamCiclos;           // infinito recomendado
8     std::vector<std::unique_ptr<Proceso>> procesos;
9     std::list<std::string> ordenPaginas;
10
11     std::vector<std::string> secuenciaReferencias;
12     bool referenciasCargadas = false;
13     int posActual = -1;
14
15     std::unordered_map<std::string, std::pair<std::shared_ptr<Page>, std::list<std::string>::iterator>> memoriaPrincipal;
16     std::unordered_map<std::string, std::shared_ptr<Page>> almacenamientoExterno;
17     std::shared_ptr<Page> paginaUsadaActualmente;
18
19     AlgoritmoDeReemplazo algoritmoDeReemplazo;
20     Planificador planificacion = Planificador::FCFS;
21
22     int ultimoProcesoEjecutado = -1;
23     int numProcesoEjecutar = -1;
24     int numPageFaults = 0;
25     int numRegistroProcesos = 0;
26     int instruccionesLeidasTotal = 0;
27
28     void busquedaPagina();
29     void remplazoFIFO();
30     void remplazoLRU();
31     void remplazoReloj();
32     void remplazoOPT();
33     void construirSecuenciaReferencias();
34
35 public:
36     AdminMemoria(int frames, int tam_paginas, int promedioBytesPorLinea, int tamCiclos);
37     ~AdminMemoria();
38     bool addProceso(std::string ruta);
39     void ciclo();
40     bool setPlanificador(Planificador planificador);
41     bool setAlgoritmo(AlgoritmoDeReemplazo algoritmo);
42     int getPageFaults();
43     double getHitRate();
44     double getEATS(int Tm, int Tf);
45 }
```


Main.cpp

La sintaxis del main.cpp se encuentra bastante sencilla solo para Realizar el análisis con un ciclo mega largo para facilitar los datos. Mediante ello se imprimen los análisis que ya esta gestionado por AdminMemoria.

```
1  int RealizarAnalisis(std::string ruta)
2  {
3      size_t nAlgoritmos = 4;
4      AlgoritmoDeReemplazo A[nAlgoritmos] = {
5          AlgoritmoDeReemplazo::FIFO,
6          AlgoritmoDeReemplazo::LRU,
7          AlgoritmoDeReemplazo::CLOCK,
8          AlgoritmoDeReemplazo::OPT
9      };
10
11     std::cout << "\nAnálisis de archivo: " << ruta << "\n";
12     std::cout << std::left
13         << std::setw(10) << "Algoritmo"
14         << std::setw(8) << "Frames"
15         << std::setw(12) << "PageFaults"
16         << std::setw(12) << "HitRate (%)"
17         << std::setw(10) << "EAT (ns)"
18         << "\n";
19
20     std::cout << std::string(58, '-') << "\n";
21
22     for (size_t i = 0; i < nAlgoritmos; i++)
23     {
24         int frames[] = {10, 50, 100};
25         for (int f : frames)
26         {
27             AdminMemoria a(f, 4096, 11, 100000000);
28             a.addProceso(ruta);
29             a.setAlgoritmo(A[i]);
30             a.ciclo();
31
32             double hitrate = a.getHitRate() * 100.0;
33             double eat = a.getEATS(100, 10000);
34
35             std::cout << std::left
36                 << std::setw(10) << toString(A[i])
37                 << std::setw(8) << f
38                 << std::setw(12) << a.getPageFaults()
39                 << std::fixed << std::setprecision(2)
40                 << std::setw(12) << hitrate
41                 << std::setw(10) << eat
42                 << "\n";
43         }
44     }
45     return 1;
46 }
47
48
49 int main()
50 {
51     RealizarAnalisis("ArchivosParaTrabajar/bzip.trace");
52     RealizarAnalisis("ArchivosParaTrabajar/gcc.trace");
53     return 0;
54 }
```

Descripción de AdminMemoria.cpp

AdminMemoria se encarga de gestionar los procesamiento de cual proceso ejecutar y realizar el cambio de paginas. Este recibe 4 parametros los cuales son



```
1 size_t frames;           // 10 recomendado
2 uint32_t tam_paginas;    // 4896 recomendado
3 int promedioBytesPorLinea; // 11 recomendado
4 int tamCiclos;           // infinito recomendado
```

Viene un planificador y algoritmo de remplazo por default sin embargo tambien contienen métodos para remplazar eso mismo. EL promedio de Bytes por línea se utiliza para realizar el calculo del tamaño del archivo porque este se abre en binario y intenta estimar cuantas instrucciones contiene el archivo. Mediante eso el planificador puede escoger cual programa ejecutar de siguiente.

También se ofrece 4 algoritmo de cambios de páginas. LRU,OPT,FIFO y CLOCK. Cada uno siguiendo su propia lógica para realizar los cambios de las páginas.

La memoria se administra por un share_ptr de paginas y un interador para ubicar las paginas de manera rápida. Tambien se registra un Almacenamiento Externo en caso de que las paginas no han sido ensuciadas. Gracias a eso podemos manejar muchos datos en orden en la creación de nuevas paginas o mover paginas de el Almacenamiento Externo al Interno.



```
1 std::unordered_map<std::string, std::pair<std::shared_ptr<Page>, std::list<std::string>::iterator>> memoriaPrincipal;
2 std::unordered_map<std::string, std::shared_ptr<Page>> almacenamientoExterno;
```

Programas Adicionales

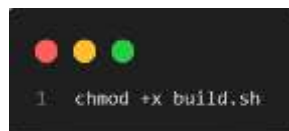
Para probar que el algoritmo solo es optimo al momento de que las ubicaciones de las memorias se encuentra cerca tambien se creo un generador de programas aleatorio utilizando Python. Este genera c instrucciones

A screenshot of a terminal window with a dark background and light-colored text. The terminal shows a Python script with 13 lines of code. The script imports the 'random' module, defines a function 'generar_trace_memoria' that takes 'filename', 'num_referencias' (default 1,000,000), and 'prob_write' (default 0.1) as arguments. The function opens the file in write mode and loops 'num_referencias' times, generating random memory addresses and operations ('W' for write, 'R' for read) based on 'prob_write'. It then calls the function with 'memoria.trace' and 'c' (1000000) and prints a confirmation message.

```
1 import random
2
3 def generar_trace_memoria(filename, num_referencias=1_000_000, prob_write=0.1):
4     with open(filename, 'w') as f:
5         for _ in range(num_referencias):
6             direccion = random.randint(0, 0xFFFFFFFF)
7             operacion = 'W' if random.random() < prob_write else 'R'
8             f.write(f"{direccion:08x} {operacion}\n")
9
10 c = 1000000
11 generar_trace_memoria("memoria.trace", c)
12 print(f"Archivo memoria.trace generado con {c} referencias.")
13
```

Como Ejecutar

Para ejecutar hay un archivo que realiza todo lo necesario. Solo necesita darle los permisos suficientes. El Comando utilizado para darle mas permisos es:

A small snippet of a terminal window showing a single command to set permissions on a file named 'build.sh'.

```
1. chmod +x build.sh
```

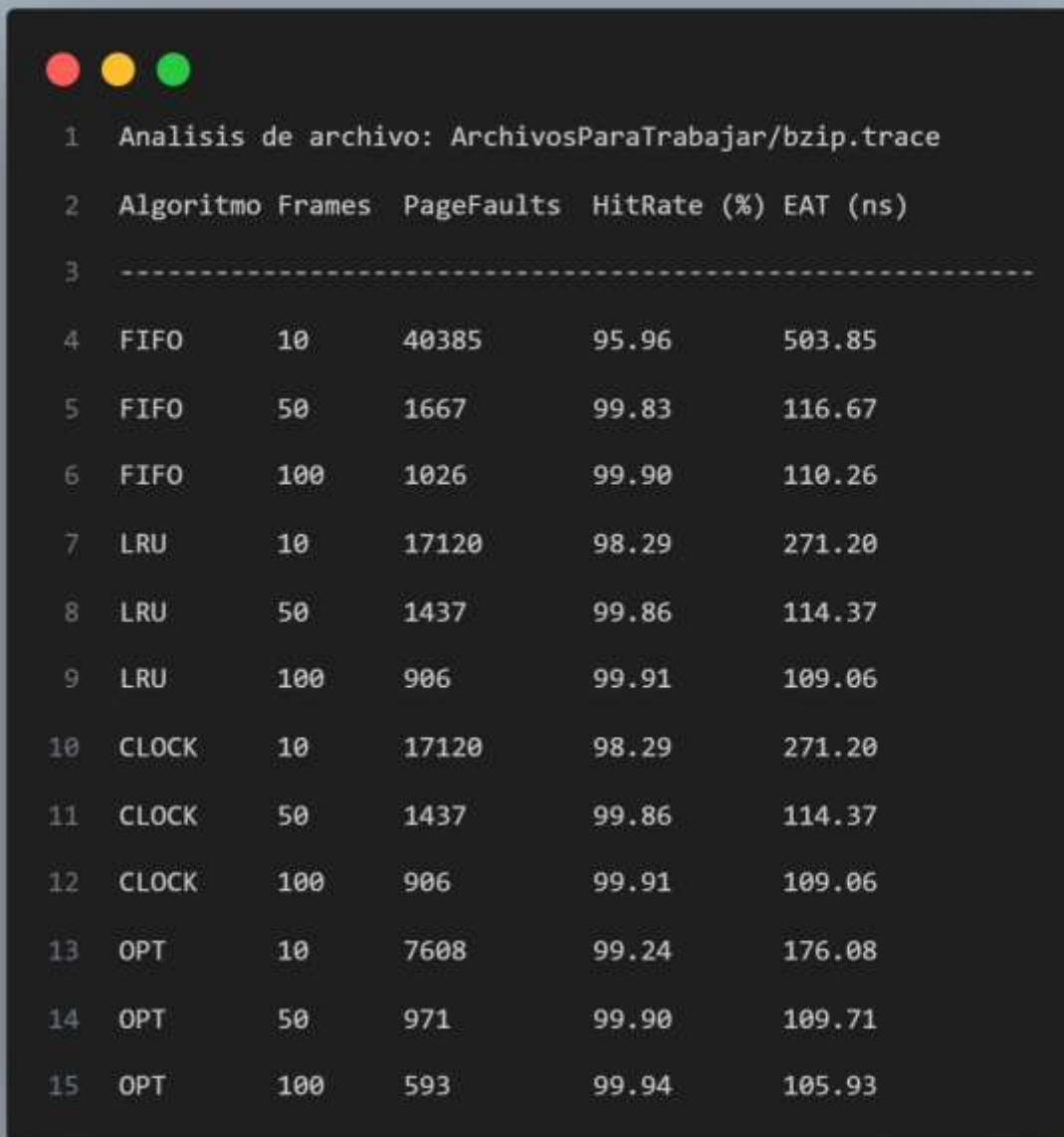
Es necesario estar en la ubicación correcta para que ese comando funcione.

Resultados

El cálculo del EATS está realizado que acceder al disco duro dura 10000ns mientras que a la ram 100ns.

bzip.trace

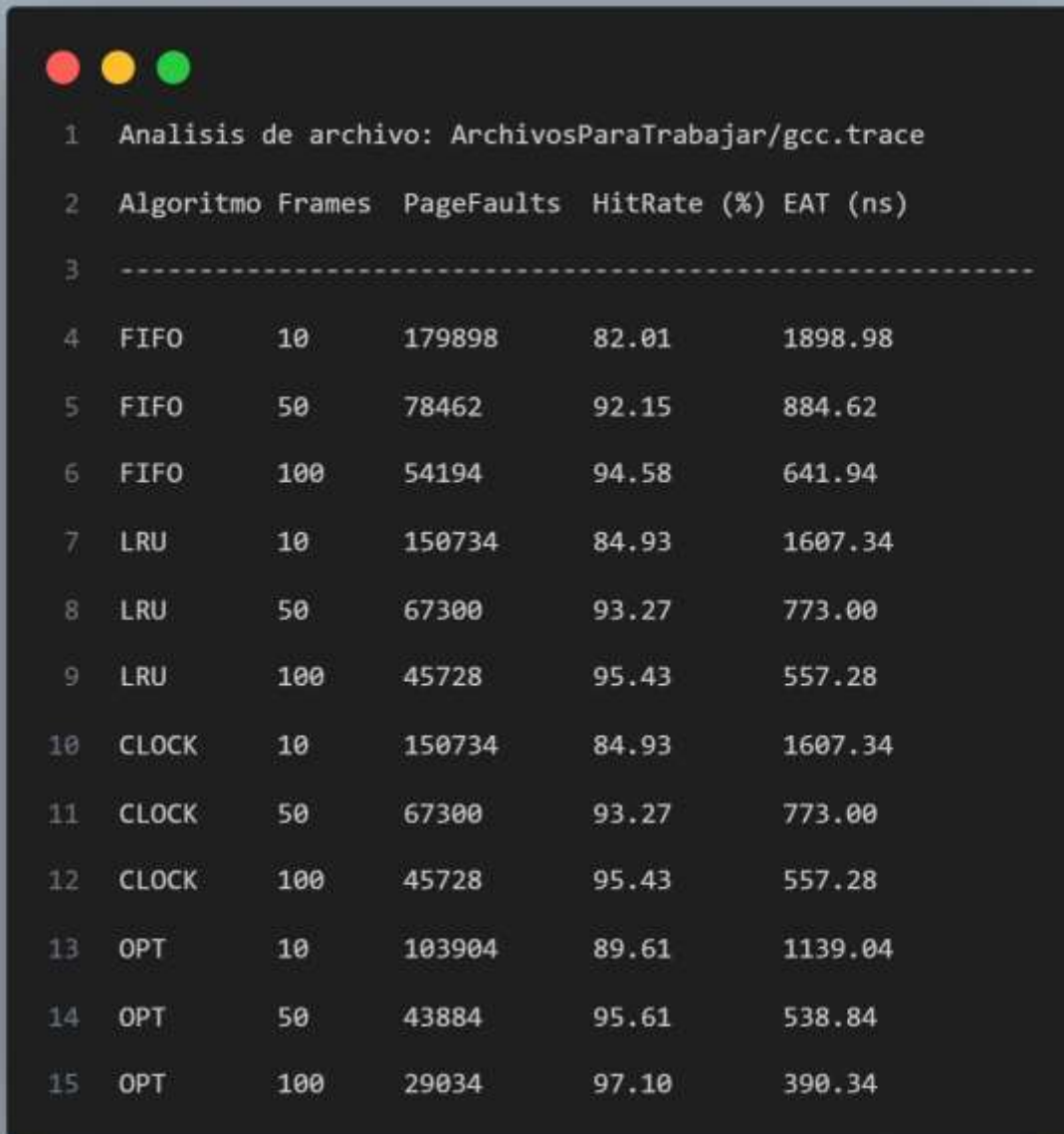
bzip es el proceso que mejor ejemplo hace a la utilización de memoria cerca. La paginación con este es bastante efectivo y aumenta bastante los hitrates.

A terminal window with a dark background and light gray text. It shows the results of an analysis for the file 'ArchivosParaTrabajar/bzip.trace'. The results are presented in a table with 5 columns: 'Algoritmo', 'Frames', 'PageFaults', 'HitRate (%)', and 'EAT (ns)'. The table lists 15 rows of data for different algorithms (FIFO, LRU, CLOCK, OPT) and frame sizes (10, 50, 100).

1	Análisis de archivo: ArchivosParaTrabajar/bzip.trace				
2	Algoritmo	Frames	PageFaults	HitRate (%)	EAT (ns)
3	-----				
4	FIFO	10	40385	95.96	503.85
5	FIFO	50	1667	99.83	116.67
6	FIFO	100	1026	99.90	110.26
7	LRU	10	17120	98.29	271.20
8	LRU	50	1437	99.86	114.37
9	LRU	100	906	99.91	109.06
10	CLOCK	10	17120	98.29	271.20
11	CLOCK	50	1437	99.86	114.37
12	CLOCK	100	906	99.91	109.06
13	OPT	10	7608	99.24	176.08
14	OPT	50	971	99.90	109.71
15	OPT	100	593	99.94	105.93

gcc.trace

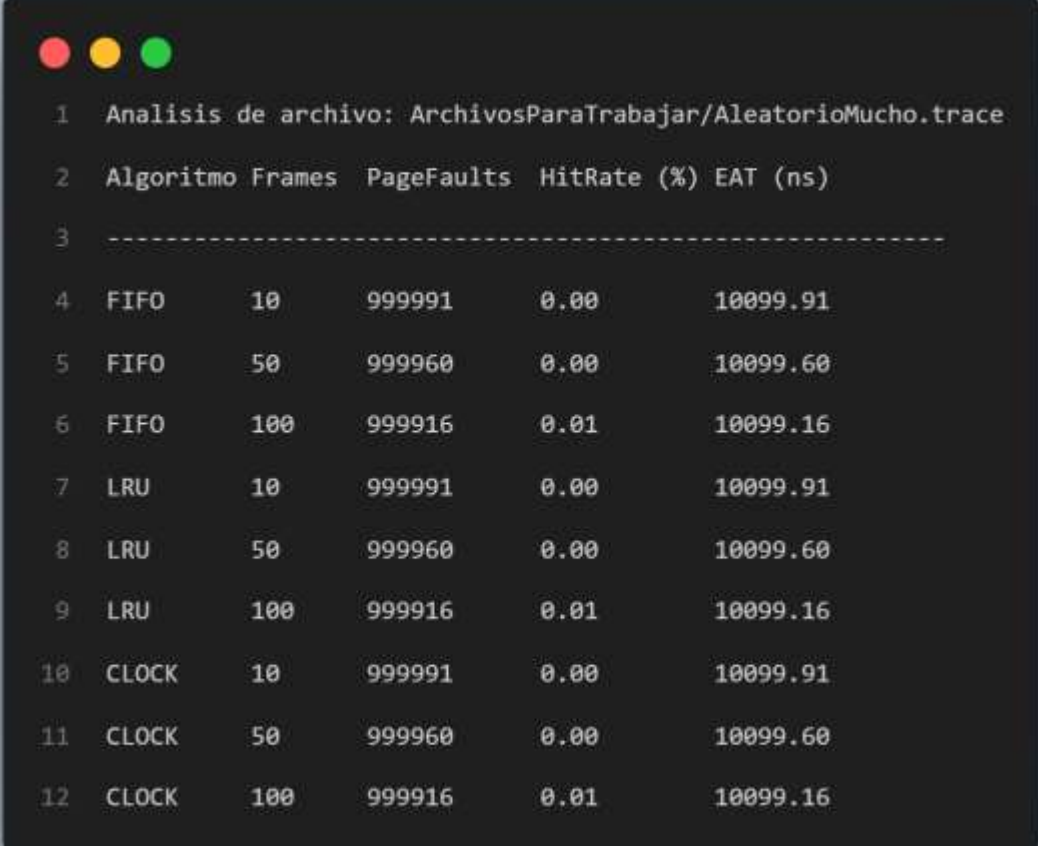
gcc tambien sigue la misma salida solo que utiliza memoria un poco mas alejadas por lo cual tiene un poco mas de fallos.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The terminal displays the output of a program analyzing 'gcc.trace'. It shows a table with 5 columns: 'Algoritmo', 'Frames', 'PageFaults', 'HitRate (%)', and 'EAT (ns)'. The table lists results for FIFO, LRU, CLOCK, and OPT algorithms with frame sizes of 10, 50, and 100. The OPT algorithm with 100 frames shows the lowest EAT value of 390.34 ns.

1	Analisis de archivo: ArchivosParaTrabajar/gcc.trace				
2	Algoritmo	Frames	PageFaults	HitRate (%)	EAT (ns)
3	-----				
4	FIFO	10	179898	82.01	1898.98
5	FIFO	50	78462	92.15	884.62
6	FIFO	100	54194	94.58	641.94
7	LRU	10	150734	84.93	1607.34
8	LRU	50	67300	93.27	773.00
9	LRU	100	45728	95.43	557.28
10	CLOCK	10	150734	84.93	1607.34
11	CLOCK	50	67300	93.27	773.00
12	CLOCK	100	45728	95.43	557.28
13	OPT	10	103904	89.61	1139.04
14	OPT	50	43884	95.61	538.84
15	OPT	100	29034	97.10	390.34

Aleatorio.trace

El aleatorio prueba la teoría de que si la memoria no estuviera cerca, la paginación seria poco efectiva. La línea de ejecución de el archivo Aleatorio se encuentra comentado porque el algoritmo simplemente tarda bastante en analizar el OPT ya que tiene una complejidad de $O(n^2)$ y $n=1000000$.



```
1 Analisis de archivo: ArchivosParaTrabajar/AleatorioMucho.trace
2 Algoritmo Frames PageFaults HitRate (%) EAT (ns)
3 -----
4 FIFO 10 999991 0.00 10099.91
5 FIFO 50 999960 0.00 10099.60
6 FIFO 100 999916 0.01 10099.16
7 LRU 10 999991 0.00 10099.91
8 LRU 50 999960 0.00 10099.60
9 LRU 100 999916 0.01 10099.16
10 CLOCK 10 999991 0.00 10099.91
11 CLOCK 50 999960 0.00 10099.60
12 CLOCK 100 999916 0.01 10099.16
```

Algoritmo	Frames	PageFaults	HitRate (%)	EAT (ns)
FIFO	10	999991	0.00	10099.91
FIFO	50	999960	0.00	10099.60
FIFO	100	999916	0.01	10099.16
LRU	10	999991	0.00	10099.91
LRU	50	999960	0.00	10099.60
LRU	100	999916	0.01	10099.16
CLOCK	10	999991	0.00	10099.91
CLOCK	50	999960	0.00	10099.60
CLOCK	100	999916	0.01	10099.16

Salida de Mapa de Memoria

Se puede observar que todos los espacios de memoria están siendo utilizados, sin embargo no todos están sucios.

```
===== MAPA DE MEMORIA =====  
C C C S C S S S C C  
S C S C S C C S C C  
C S S S C C S S C C  
S C C C C S C C S C  
S S S S C C S S C S  
S S C S S S S S S S  
C S C S S C C S S S  
S S S S S C C S S C  
C S C C C S C C C C  
C S S C C C S C S S  
=====
```

Preguntas

1) ¿Qué estrategia de reemplazo de páginas escogería usted y por qué? Debe considerar los resultados obtenidos y el esfuerzo que le llevó implementar cada estrategia. Discuta lo que sus resultados muestran acerca de los méritos relativos de FIFO, LRU y OPT para cada una de las diferentes combinaciones de parámetros.

CLOCK es el algoritmo que escogería de primera porque es la mas fácil de implementar y tenemos aproximadamente los mismos resultados que obtendríamos con LRU. Para comparar la eficiencia de cada una tomare de ejemplo los que tienen 10 marcos del bzip.

LRU: 17120

CLOCK: 17120

FIFO: 40385

OPT: 7608

Si comparamos las page faults de cada una observamos que opt es el que menos tienen. Sin embargo sabemos de antemano que OPT es imposible en la realidad porque no conocemos el futuro. Por lo cual la mejor técnica que podemos implementar es Clock.

FIFO simplemente ofrece una implementación rápida para el reemplazo de algoritmo el cual tiene la lógica de que si se utilizo ya una vez ya no se va a volver a utilizar. Sin embargo en los programas no es necesariamente real esta premisa.

No utilizo LRU porque implementarlo es mucho mas complicado, ya que hay que manejar una lista para controlar que son las ultimas direcciones que se han utilizados. Si no tendríamos la opción de utilizar CLOCK el mejor algoritmo por utilizar después de este es LRU.

2) ¿Qué aspectos de la administración de memoria encontró que fue más difícil implementar?

La mas complica fue implementar el algoritmo OPT, ya que había que analizar todo el archivo y insertarlo en memoria para luego tenerlo en un arreglo que sepamos el orden. Luego tendríamos que saber en que línea va exactamente el programa para recorrer por enfrente el arreglo y ver cuales de las paginas no se vuelven a repetir. Esa implementación no es eficiente y es bastante tardado. La complejidad de ese algoritmo es incluso $O(n^2)$ por lo cual es el mas ineficiente para calcular.

3) ¿Qué aspectos de la administración de la memoria encontró más fácil implementar?

El más fácil de implementar fue FIFO ya que solo es una cola la con limite de marco. Sin embargo, también es una solución que solo es fácil de implementar sin embargo es bastante ineficiente al momento de compararlo con otros algoritmos.

Conclusión

La realización de este proyecto fue excelente para la comprensión a fondo de cada uno de los algoritmos. La importancia de conocer a fondo los algoritmos para remplazar las paginas es importante para comprender como funcionan los sistemas operativos. Realizar un programa que simule estos mismos algoritmos es una forma excelente en que le estudiante comprenda estos mismos.

FIFO es uno de los algoritmos más fáciles de implementar, sin embargo es la solución menos optimas de las presentadas. LRU es una de las mas optimas presentadas, y CLOCK es una simulación de LRU con un poco de aleatoriedad incluida. Todas estas intentan aproximarse al OPT que no es posible de conseguir en un entorno real.

Para mejorar la eficiencia de estos algoritmos se utiliza una técnica llamada paginación, la cual divide en bloques las direcciones de memoria y mueve en conjunto todas las direcciones de dicho bloque en caso de que se encuentre una. Esta técnica es super optima porque los programas tienden a utilizar memoria cerca de las mismas memorias utilizadas. Hay un ejemplo llamado Aleatorio.trace la cual muestra exactamente porque esa técnica solo funciona por la cercanía de estas mismas.

Enlace al Repositorio de GitHub

Para acceder al código fuente de este documento está a través del siguiente enlace:

<https://github.com/JRafaelLobo/AdministradorDeMemoria.git>