TEC | Tecnológico de Costa Rica

Escuela de Ingeniería en Computación
Ingeniería en Computación
IC6600 - Principios de Sistemas Operativos

Proyecto 2

Simulación de Algoritmos de Paginación

Gerald Calderón
gecalderon@estudiantec.cr
2023125197

Óscar Obando
osobando@estudiantec.cr
2023091684

Samuel Zúñiga
sazuniga@estudiantec.cr
2023029693

San José, Costa Rica
Octubre 2025

Índice general

Capítulo 1

Introducción

La gestión de memoria en una computadora es el proceso de asignar memoria a los programas que la solicitan, esta es una tarea de gran importancia para el funcionamiento correcto de la computadora y de cada proceso(Eduardo, s.f.). En las computadoras modernas, varios procesos pueden correr concurrentemente, usualmente cada proceso tiene su propia memoria que necesita para su funcionamiento, es por esto que es necesario gestionar la memoria, para que todos los procesos tengan la memoria que necesita. La memoria en una computadora está dividida en unidades llamadas páginas, a los procesos se les suele asignar varias de estas. En ocasiones, la cantidad de procesos abiertos pueden ser numerosos, lo que puede llevar a que no haya suficiente memoria para todos estos procesos. Es por esto que existe la memoria secundaria, esta no suele ser tan rápida pero tiene mucho espacio, lo que le da un buen lugar como apoyo para la memoria principal. El papel de la memoria secundaria usualmente lo tiene el disco duro.

La memoria secundaria es muy útil, pero tiene un problema, las páginas guardadas en el disco duro no pueden ser usadas directamente debido a la lentitud de acceso. Es preferible que las páginas solo se usen en memoria, por lo que es necesario reemplazar una página en memoria por la que se va a usar que está actualmente en disco duro. En un sistema operativo, la paginación ocurre cuando una página en memoria principal debe ser remplazada por otra en memoria secundaria (páginas guardadas en el disco duro) (Mansi, 2024). Gracias a la paginación es posible tener una cantidad muy grande de procesos que se ejecutan al mismo tiempo en la computadora. Es por esto que existen los algoritmos de paginación, ya que la operación de reemplazar las páginas es bastante cara y por lo tanto se quiere hacer esto lo menos posible. Los algoritmos de paginación son importantes, ya que dependiendo de la calidad del algoritmo que se use se va a necesitar reemplazar más o menos páginas. En este proyecto se solicitó crear una simulación para comparar algoritmos de paginación, esto para aprender sobre las características de cada uno y para poder apreciar la diferencia que tiene un buen algoritmo de paginación en contra de uno malo.

En la simulación hay dos MMU, las cuales toman un archivo que contiene

intrucciones que son parte de un proceso, una MMU ejecuta el algoritmo óptimo de paginación cuando es necesario y la otra ejecuta otro algoritmo seleccionado por el usuario para poder comparar dicho algoritmo con la situación ideal. Los algoritmos implementados en esta simulación para comparar sus ejecuciones son: Óptimo, FIFO, Second Chance, LRU, MRU y Random. Las instrucciones soportadas por la simulación son: "new(pid, size)", reserva solicita un puntero de cierto tamaño para un proceso; "use(ptr)", usa un puntero ya existente; "delete(ptr)", borra un puntero existente y "kill(pid)", finaliza un proceso y borra la memoria reservada de todos sus punteros.

Capítulo 2

Detalles de la implementación

2.1. Información relevante a los algorimtmos

Antes de iniciar con la explicación de cada algoritmo, es importante dar algunos detalles sobre la implementación del sistema. La simulación está implementada en C++, para aprovechar sus capacidades de programación orientada a objetos, se decidió crear las siguientes clases:

- Page: la clase que representa a cada página.
- MMU: la clase manejadora de cada algoritmo y de ejecutar las instrucciones, es una clase que contiene la función abstracta "paging()".
- MMU concretas: hay una por cada algoritmo e implementan su propia versión de "paging()".
- Parser: contiene dos instancias de MMU (La correspondiente al algoritmo óptimo y la otra que contiene el algoritmo seleccionado por el usuario), esta se encarga de leer el archivo de intrucciones dado como entrada y enviar dichas instrucciones a cada MMU, esta clase las mantiene sincronizadas.

La memoria es representada como un array de enteros dentro de cada instancia de MMU. Dichos enteros son los ID de la página correspondiente. El disco es representado como un mapa de enteros y páginas, siendo los enteros los ID de la página. Todas las páginas estan guardadas en disco y que la memoria solo tiene los ID, es decir que la información real de la página se encuentra en el disco. Esto claramente no es fiel a la vida real, ya que la página existe físicamente en memoria y es físicamente movida de disco a memoria y viceversa. Aún así, se decidió implementarlo de esta forma por simplicidad, ya que para uso de la simulación funciona perfectamente. De esta forma no hay necesidad de manejar intercambiar los valores entre disco y memoria.

El algoritmo de paginación es ejecutado cuando se solicita cargar una nueva página a memoria, esto implica que cada algoritmo se ejecuta solamente en el uso de las instrucciones "use" y "new", ya que para "delete" y "kill" solo es necesario eliminar la página, no cargarla. En el momento que se solicita cargar una página la MMU:

1. Se revisa si la página ya está cargada, de ser así, se toma como HIT y no se ejecuta el algoritmo.
2. Se revisa si hay espacio en memoria, de ser así, también se toma como HIT y no se ejecuta el algoritmo.
3. Si no ocurre ninguno de los anteriores, se toma como FAULT y se ejecuta el algoritmo.

2.2. Algoritmos de Paginación

2.2.1. Óptimo

El algoritmo óptimo de paginación es aquel que tiene la capacidad de ver el futuro, es decir, sabe cuales páginas serán utilizadas durante toda la ejecución de la máquina para poder decidir a cual mandar a memoria virtual (GeeksForGeeks, 2025a). Su funcionamiento consiste en buscar dentro de las páginas cargadas en memoria real aquella que nunca se va a volver a utilizar o la que se va a utilizar más tarde para reemplazarla por la que se requiera cargar en dicho momento.

Para su implementación al leer el archivo de instrucciones a utilizar se llama a la función "parse_for_optimal" (ver figura 2.1) que detecta cada aparición de "use" o "new" para registrar las páginas en una lista que contendrá el id de las páginas que son utilizadas en dichas instrucciones. Una vez se lee todo el archivo dicha lista de páginas es pasada al constructor de la MMU concreta que contiene el algoritmo óptimo como implementación de la función "paging" (ver figura 2.2).

2.2.2. FIFO y Second Chance

FIFO

El algoritmo FIFO es (junto a Random) el algoritmo más simple impementado para manejar paginación en el programa. Este algoritmo por sus siglas significa "First in First Out", como su nombre lo indica, este consiste en sacar la página que lleva más tiempo en memoria para dar espacio a las demás (GeeksForGeeks, 2025a).

Para implementar esto, se decidió utilizar dos variables "to_remove_index" (el índice de la página que se va a remover) y "time_loaded" (el tiempo que la página lleva cargada). Por defecto ambas se setean en -1, este valor es reescrito

```
void parse_for_optimal() {
  int i;
  int ptr_size, n_pages, pid, ptr, page_to_load;
  ops.pos = 0;
  while (ops.content[ops.pos] != '\0') {
    i = 0;
    while ((c = ops.content[ops.pos++]) != '(') {
      token[i++] = c;
    }
    token[i] = '\0';
    if (strcmp(token, "new") == 0) {
      pid = parse_num();
      ptr_size = parse_num();
      n_pages = ((ptr_size % PAGE_SIZE) == 0 ? ptr_size / PAGE_SIZE
                                             : ptr_size / PAGE_SIZE + 1);
      std::vector<int> pages_for_ptr;
      for (int p = 0; p < n_pages; ++p) {
        pages.push_back(page_id);
        pages_for_ptr.push_back(page_id);
        page_id++;
      }
      ptr_to_pages[ptr_id] = pages_for_ptr;
      ptr_id++;
    } else if (strcmp(token, "use") == 0) {
      ptr = parse_num();
      for (int page : ptr_to_pages[ptr]) {
        pages.push_back(page);
      }
    }
    while ((c = ops.content[ops.pos++]) != '\n' && c != '\0');
  }
  ops.pos = 0;
}
```

Figura 2.1: Imagen del parser para el algoritmo óptimo

en el momento que se ejecuta la lógica del algoritmo. Dentro de un for loop se revisan todas las páginas dentro de memoria, si la cantidad de tiempo que esa página es mayor a la cantidad guardada en "time_loaded" se actualiza el índice y dicha variable. De esta forma, una vez termina el for loop, se tiene el índice de la página que lleva más tiempo cargada, una vez hecho esto simplemente se retorna este índice. Este comportamiento se puede ver en el código mostrado en la Figura 2.3.

Second Chance

El algoritmo Second Chance es una pequeña mejora del algoritmo FIFO, este busca tomar en cuenta información de los usos pasados (GeeksForGeeks, 2025b). Este algoritmo utiliza un bit para marcar las páginas cada vez que ocurre un HIT en esta. A la hora de reemplazar una página, esta trata de seleccionar la que lleva más tiempo dentro de memoria, pero si está marcada con el bit encendido, descarta esta página, expira el bit agregando un 0, y luego elige la siguiente que lleva más tiempo, repitiendo el proceso si este también tiene el bit encendido.

Para implementar este algoritmo se copia la memoria en un vector, luego se busca la página más reciente dentro del vector. Si la página más reciente no

```
int paging() {
  int to_remove_index = -1;
  int farthest_use = -1;
  for (int i = 0; i < MEMORY_SIZE; ++i) {
    int next_use = -1;

    // se busca cuando se usara mas reciente
    for (int j = current_page + 1; j < futureRequests.size(); ++j) {
      if (futureRequests[j] == memory[i]) {
        next_use = j;
        break;
      }
    }
    if (next_use == -1)
      return i;

    // si el siguiente uso de esta pagina es mas lejano que el mas lejano
    // actual se selecciona este para usarlo
    if (next_use > farthest_use) {
      farthest_use = next_use;
      to_remove_index = i;
    }
  }
  return to_remove_index;
 }
};
```

Figura 2.2: Imagen del algoritmo óptimo

tiene el bit encendido, simplemente se retorna su índice en memoria. Si dicha página está marcada, se expira la marca y se saca la página del vector para no seleccionarla. Luego se repite el proceso para seleccionar otra página. Si se que todas las páginas estaban marcadas, se busca la más reciente dentro de memoria. El código que implementa esta lógica se puede ver en la Figura 2.4.

2.2.3. LRU y MRU

Los algoritmos de paginación MRU(Most Recently Used) y LRU(Least Recently Used) funcionan de forma similar. Ambos utilizan una marca de tiempo que se inserta en las páginas cuando se cargan en RAM y cuando se utilizan. La página que se escoge para ser reemplazada es la que fue utilizada más recientemente o menos reciente, dependiendo de que algoritmo se utilice (GeeksForGeeks, 2025a).

La implementación del MRU es simple, se trata de un algoritmo que recorre todo el arreglo de memoria para obtener que tan recientemente se usó cada página, y a través de una comparación encuentra la más utilizada recientemente (ver figura 2.5). La implementación del LRU es exactamente igual pero la comparación cambia para encontrar la página menos utilizada recientemente (ver figura 2.6).

```
int paging() {
  int to_remove_index = -1;
  int time_loaded = -1;
  for (int i = 0; i < MEMORY_SIZE; i++) {
    Page page = disk[memory[i]];
    if (page.load_t > time_loaded) {
      time_loaded = page.load_t;
      to_remove_index = i;
    }
  }
  return to_remove_index;
}
```

Figura 2.3: Ilustración de el algoritmo FIFO

2.2.4. Random

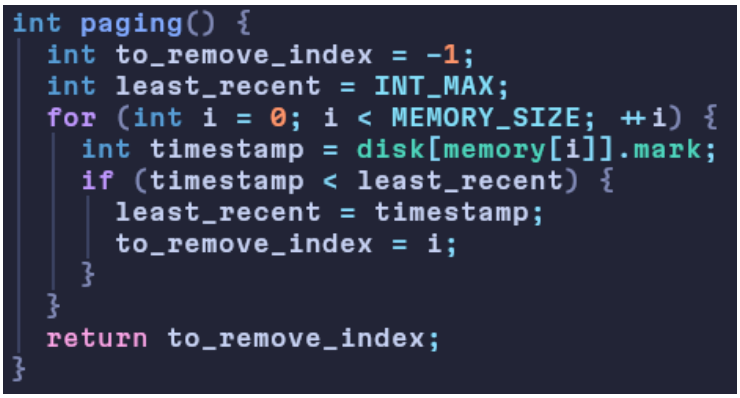
El algoritmo de paginación Random o Aleatorio reemplaza una página en memoria escogida de forma aleatoria, debido a esto su implementación es trivial y se limite a conocer sobre como se consiguen números aleatorios en el lenguaje escogido para implementarlo (ver figura 2.7). En está implementación se hace usó de una semilla para poder replicar el comportamiento del algoritmo, está semilla se configura justo antes de crear el objeto Random_MMU.

```cpp
int paging() {
  int to_remove_index = -1;
  std::vector<int> vec;
  // se copia la memoria en un vector
  for (int i = 0; i < MEMORY_SIZE; ++i)
    vec.push_back(memory[i]);
  while (true) {
    // si el tamaño es 0, todas las marcas se expiraron
    // Se vuelve a copiar para encontrar el mas viejo
    if (vec.size() == 0) {
      for (int i = 0; i < MEMORY_SIZE; ++i)
        vec.push_back(memory[i]);
    }
    int time_loaded = -1;
    Page *page;
    // se encuentra la mas vieja de los del vector
    for (int i = 0; i < vec.size(); ++i) {
      page = &(disk[vec[i]]);
      if (page->load_t > time_loaded) {
        time_loaded = page->load_t;
        to_remove_index = i;
      }
    }
    page = &(disk[vec[to_remove_index]]);
    // si hay marca se saca del vector y se le da una nueva oportunidad
    if (page->mark) {
      vec.erase(vec.begin() + to_remove_index);
      to_remove_index = -1;
      page->mark = 0; // se expira
      continue;
    }
    // si no hay marca encontramos el indice de memory y lo devolvemos
    int victim = vec[to_remove_index];
    for (int i = 0; i < MEMORY_SIZE; ++i) {
      if (memory[i] == victim)
        return i;
    }
  }
}
```

Figura 2.4: Ilustración de el algoritmo Second Chance

```cpp
int paging() {
  int to_remove_index = -1;
  int most_recent = -1;
  for (int i = 0; i < MEMORY_SIZE; ++i) {
    int timestamp = disk[memory[i]].mark;
    if (timestamp > most_recent) {
      most_recent = timestamp;
      to_remove_index = i;
    }
  }
  return to_remove_index;
}
```

Figura 2.5: Imagen del algoritmo MRU



```
int paging() {
  int to_remove_index = -1;
  int least_recent = INT_MAX;
  for (int i = 0; i < MEMORY_SIZE; ++i) {
    int timestamp = disk[memory[i]].mark;
    if (timestamp < least_recent) {
      least_recent = timestamp;
      to_remove_index = i;
    }
  }
  return to_remove_index;
}
```

Figura 2.6: Imagen del algoritmo LRU

```
int paging() { return rand() % MEMORY_SIZE; }
```

Figura 2.7: Imagen del algoritmo Random

Capítulo 3

Instrucciones

3.1. Cómo instalar el programa

1. Descargue el código fuente del programa. Puede hacerlo de las dos siguientes formas:
   - Dirigirse al repositorio de GitHub mediante su navegador a través del siguiente link: `https://github.com/Andres2950/PSO_PagingSimulator.git`
   - Instalarlo directamente con el comando `wget https://github.com/Andres2950/PSO_PagingSimulator/archive/refs/heads/main.zip`
2. Descomprima el archivo .zip descargado utilizando el comando `unzip`.
3. Al extraer el archivo podrá observar la estructura de organización similar a la figura 3.1.
4. Ejecute el archivo `install.sh`, asegúrese de qué tenga permisos de ejecución, puede utilizar el comando `chmod +x install.sh` en caso de que no los posea y luego ejecute de la siguiente forma `./install.sh`.
   Este archivo se hará cargo de la instalación del compilador *g++* necesario para compilar el código fuente. Además hará la instalación del paquete *cmake* para la ejecución del archivo "CMakeLists.txt", dicho archivo posee las instrucciones de compilación de *SDL* para resolver todas las dependencias.
5. Note que la instalación de dichos paquetes requiere permisos de usuarios root, al ejecutar el archivo `install.sh` este se volverá a ejecutar con dichos permisos, para esto solicitará la contraseña del usuario root para tener dichos permisos de ejecución (la contraseña es totalmente invisible para el programa) y así poder descargar los paquetes.
6. La compilación de los archivos se realiza sin permisos root.

3.2. Cómo utilizar el programa

Una vez terminado el procedimiento de instalar el programa puede utilizar el comando `./build/app/app` en la carpeta del proyecto descargada para ejecutar el programa de simulación.

3.2.1. Ventana principal

En la figura 3.2 se pueden observar las funciones iniciales del programa. Se detalla a continuación cada punto.

1. Semilla para aleatorización del algortimo random y generación del archivo para las instrucciones en caso de que se quiera crear uno nuevo. Esta semilla se puede ingresar manualmente para replicar una simulación anterior o mediante el botón **Aleatorio** se genera una semilla semi-aleatoria.
2. Algoritmo para ser simulado y comparado con el óptimo.
3. Generación del archivo de instruciones. Presenta una ventana para el ingreso de la cantidad de procesos y operaciones a realizar (note que el minímo de operaciones son la cantidad de procesos puesto que se le deben hacer kill).
4. Carga un archivo de instrucciones guardado anteriormente.
5. Abre la ventana de simulación con los algoritmos y instrucciones cargadas.

3.2.2. Ventana de simulación

En la figura 3.3 se pueden observar las funciones de la ventana de simulación. Se detalla a continuación cada punto.

1. Inicia la simulación de los algoritmos de paginación.
2. Volver a la pantalla principal para modificar los parámetros.
3. Barra para ajustar el delay de las operaciones, útil para seguir la ejecución del programa y analizar su comportamiento.

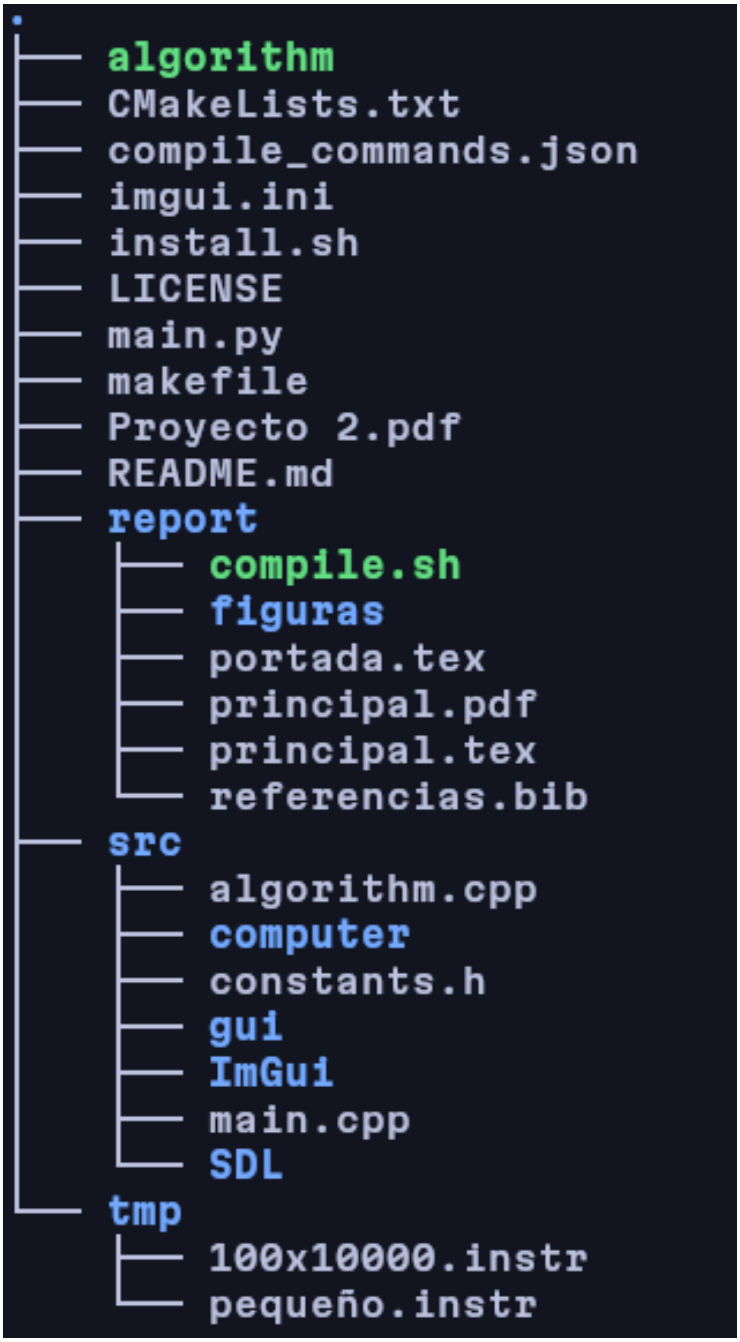


Figura 3.1: Imagen de la estructura del programa

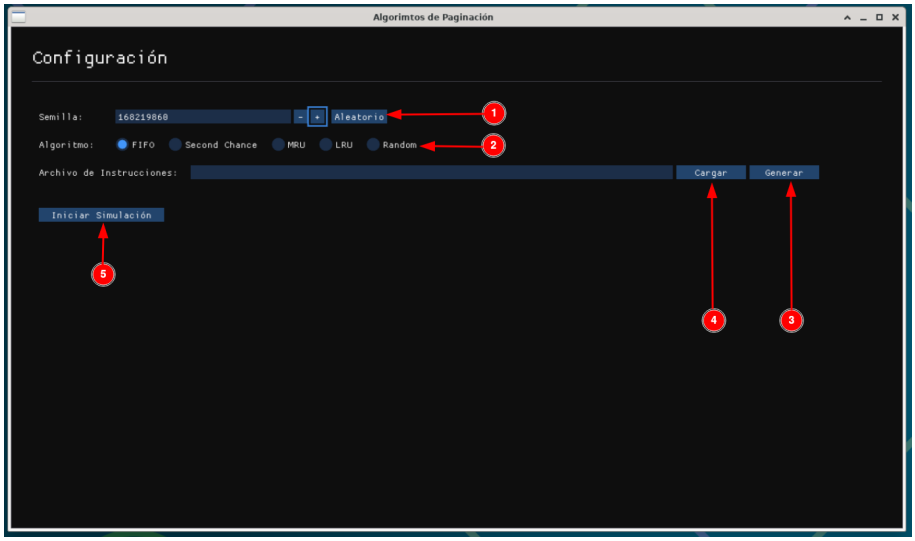


Figura 3.2: Imagen de la ventana principal del programa

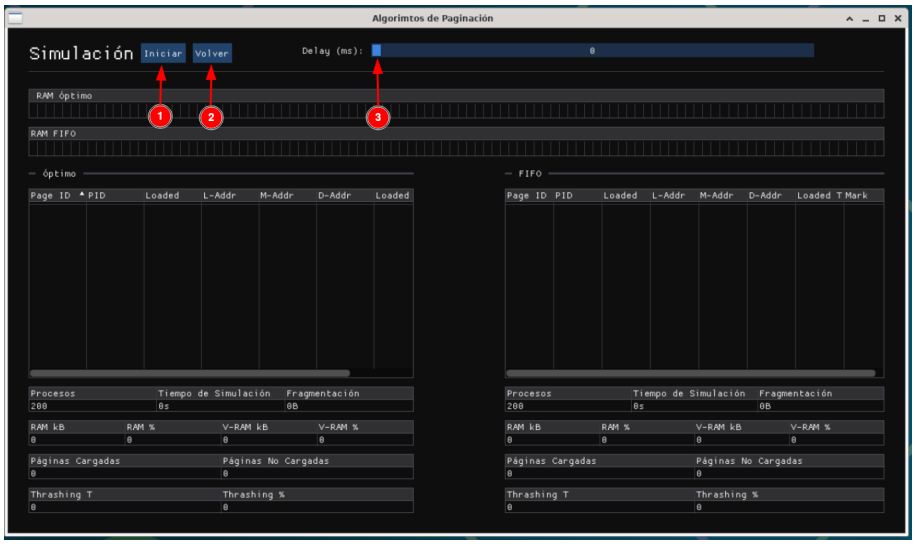


Figura 3.3: Imagen de la ventana de simulación del programa

Capítulo 4

Conclusiones

El thrashing llega al 90 % rápidamente con tan solo 500 instrucciones y sin importar que algoritmo de paginación se utilice. Con 5000 instrucciones el thrashing alcanza un 97 % con el algoritmo óptimo y un 98 % y 99 % con los demás algoritmos. En base a esto podemos concluir que el thrashing aumenta en relación a la cantidad de operaciones ejecutadas sin importar el algoritmo que se utilice.

El algoritmo óptimo puede llegar a ser peor que otros algoritmos. En las pruebas tuvo un peor desempeño que FIFO y el algoritmo aleatorio cuando se utilizaron listas de instrucciones con una densidad de deletes muy alta. Se concluye que esto sucede porque el algoritmo óptimo no toma en cuenta el uso de deletes en el programa y tampoco tiene una estrategia para beneficiarse de los posibles huecos que estos generen.

El algoritmo MRU fue el que peor desempeño tuvo, especialmente con listas de instrucciones grandes (1000-5000 instrucciones), siendo incluso peor que el algoritmo aleatorio. En base a esta información podemos concluir que los algoritmos que no cumplen con el principio de localidad temporal tienen un mal desempeño cuando ejecutan listas de instrucciones aleatorias.

Referencias

Eduardo, B. (s.f.). *Gestión de memoria en computadoras.* Página web. Descargado de https://www.scribd.com/document/610256331/Gestion-de-memoria-S-O

GeeksForGeeks. (2025a, 10 de Septiembre). *Page replacement algorithms in operating systems.* Página web. Descargado de https://www.geeksforgeeks.org/operating-systems/page-replacement-algorithms-in-operating-systems/

GeeksForGeeks. (2025b, 13 de Septiembre). *Second chance (or clock) page replacement policy.* Página web. Descargado de https://www.geeksforgeeks.org/operating-systems/second-chance-or-clock-page-replacement-policy/

Mansi, B. (2024, 9 de Abril). *Page replacement in os.* Página web. Descargado de https://www.scaler.com/topics/operating-system/page-replacement-algorithm/