Proyecto I - MPointers 2.0

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería en Computadores
Algoritmos y Estructuras de Datos II (CE2203)
I Semestre 2024
Estudiantes.: Eduardo José Canessa Quesada & Luis Felipe Chaves Mena

---

1. Introduccion

El manejo manual de la memoria es una técnica fundamental en la programación que permite a los desarrolladores no solo optimizar el uso de recursos, sino también tener un control más preciso sobre la gestión de variables dentro de un programa. Sin embargo, comprender y aplicar correctamente estas técnicas puede resultar complejo, razón por la cual surge este proyecto. En este trabajo, se ha desarrollado un sistema de administración de memoria con el objetivo de mejorar la comprensión de su funcionamiento y aplicación en la programación. Para ello, se implementará un módulo denominado **Memory Manager**, encargado de gestionar la asignación y liberación de memoria de manera eficiente. Sobre este, se construirá una biblioteca llamada **MPointers**, que proporcionará una interfaz para facilitar el uso de los métodos desarrollados en **Memory Manager**. Finalmente, se realizará una aplicación práctica utilizando listas enlazadas simples, demostrando la efectividad del sistema en escenarios concretos. Este proyecto no solo busca optimizar el manejo de memoria, sino también servir como una herramienta educativa para aquellos interesados en profundizar en esta área esencial de la programación.

Índice

1. Introduccion **1**

2. Breve descripción del problema **2**
2.1. Memory Manager . . . 2
2.2. MPointers . . . 2
2.3. Lista enlazada simple . . . 2

3. Descripción de la solución **2**
3.1. Solución General . . . 2
3.2. Solución de Memory Manager . . . 3
3.2.1. Union de Bloque y Manager . . . 3
3.2.2. Comando Create y Desfragmentación . . . 3
3.2.3. Comando Set . . . 3
3.2.4. Comando Get . . . 4
3.2.5. Comandos de RefCount . . . 4
3.2.6. Garbage Collector . . . 4
3.2.7. DumpFiles . . . 4
3.3. Solución MPointers . . . 4

4. Diseño General **5**

5. Repositorio **5**

6. Recursos **5**

2. Breve descripción del problema

El problema de forma general se divide en 3 categorías, implementación de **Memory Manager**, **MPointers** y la lista enlazada simple.

2.1. Memory Manager

El problema general con esta sección inicia con la creación de un servidor y un cliente, donde se sabe que cliente será utilizado por **MPointers**. Este servidor recibe comandos que se procesan para poder crear, guardar, modificar y enviar datos guardados en memoria, generando así administración de memoria.

2.2. MPointers

La implementación de biblioteca, como método constructor, sobrecarga de operadores * y &, componen unos de los problemas relacionados con **MPointers**, a esto hay que agregar la conexión apropiada con el cliente que ayudara para procesar comandos.

2.3. Lista enlazada simple

El único problema en relación con la aplicación de manejo de memoria es que esta funcione adecuadamente, ya que el código de una lista enlazada simple es relatviamente fácil, por lo tanto si tanto **MPointers** como **Memory Manager** funcionan adecuadamente, la lista enlazada no será mayor problema.

3. Descripción de la solución

Se presenta solución en 4 aspectos: la creación correcta tanto de **MPointers** como **Memory Manager**, luego la conexión adecuada de ambos y, por último, la creación e implemantación de la lista enlazada simple como ejemplo en el funcionamiento de la biblioteca.

3.1. Solución General

MPointers implementa un sistema de gestión de memoria distribuida mediante una arquitectura cliente-servidor. El componente central es el **Memory Manager**, un servidor que reserva y administra un bloque único de memoria, procesando peticiones mediante sockets para crear, leer y modificar datos, mientras mantiene un conteo de referencias para liberación automática mediante un garbage collector. Paralelamente, la biblioteca **MPointers** ofrece a los clientes una abstracción de puntero inteligente que sobrecarga operadores tradicionales (*, =, &) para interactuar con la memoria remota de forma transparente.

El sistema incorpora un mecanismo de **Dump Files** como parte esencial para el monitoreo y depuración de la memoria administrada. Cada vez que el **Memory Manager** procesa una operación que modifica el estado de la memoria, genera automáticamente un archivo de volcado en la carpeta especificada durante su inicialización. Estos archivos, nombrados con

marcas de tiempo precisas, registran el contenido actual de la memoria, incluyendo detalles como identificadores de bloques, tipos de datos, valores almacenados y conteos de referencia.

El proyecto incluye la implementación de listas enlazadas, las cuales preteden ejemplificar el uso de la biblioteca, al ilustrar el manejo activo de la clase.

3.2. Solución de Memory Manager

Para la solución de esta sección, se inicia con la implementación de sockets y un programa cliente y servidor. El servidor debe recibir 3 parámetros: puerto de sockets, memoria en **MB** por reservar y la carpeta de los **DumpFiles** que se crearán al realizar cambios en el estado de memoria. Luego, una vez se tenga una conexión estable cliente-servidor, se crea una clase **Manager** que se encarga de procesar comandos del servidor y aplicar cambios en memoria.

De forma más precisa, **Manager** utiliza una estructura que se llamó **Bloque**, esta consiste de 6 elementos que se presentan en 1.

Tipo de datos	Nombre	Uso
void*	frstPtr	Apunta al primer espacio de memoria del bloque
void* 4	lastPtr 5	Apunta al ultimo espacio de memoria del bloque
string	type	Indica el tipo de dato que se guardara
int	refCount	lleva un conteo de referencias a este bloque
bool	alreadyAssigned	Indica si ha sido modificado al menos una vez el bloque
int	id	Numero de identificacion (ID)

Cuadro 1: Tipos de datos de Bloque y sus usos.

3.2.1. Union de Bloque y Manager

Una vez comprendida la composición de **Bloque**, se hace el procesador de comandos, donde **Manager** se encarga de ejecutar diferentes funciones dependiendo del comando. Apenas se llama al objeto **Manager**, se crea un vector que contiene todos los objetos **Bloque** que hayan sido creados; este inicia como nulo y, a su vez, se crea un vector que contiene arreglos de 2 punteros enteros. Estos indicarán, al borrar un bloque de memoria, los espacios libres para disminuir la fragmentación.

3.2.2. Comando Create y Desfragmentación

En el caso de la función de creación de espacio de memoria, se crea una estructura **Bloque** que contenga todos los datos necesarios. Luego, se consulta el estado del vector de bloques, ya que si no hay elementos, se toma como el primero, asignando el puntero del bloque de memoria inicial como el primer puntero, siempre que el espacio designado por el comando sea suficiente en la memoria reservada. En caso contrario, es decir, si ya existen bloques guardados en memoria, primero se verifica si el tamaño de creación solicitado existe en el vector de espacios vacíos. Si es el caso, se coloca el bloque en el espacio libre; si no, simplemente se asigna el primer puntero del bloque con el mayor valor, siempre que el espacio sea suficiente. De lo contrario, no se realiza la asignación.

3.2.3. Comando Set

Una vez que el método de creación está listo, se implementa el método de asignación, donde se solicita la ID y el valor por asignar. Se realiza una búsqueda en el vector de bloques en busca del elemento que contiene la ID. Luego, se verifica que el valor por asignar tenga un

tamaño menor o igual al bloque, ya que si es mayor, esto generará problemas. Finalmente, se procede con la asignación.

3.2.4. Comando Get

En el caso de la obtención de datos, solo se itera sobre el vector de bloques y se retorna el valor del puntero inicial.

3.2.5. Comandos de RefCount

En lo que respecta a los cambios en el número de referencias, solo se busca la estructura con la ID y se aumenta o disminuye el valor correspondiente.

3.2.6. Garbage Collector

Por último, en relación con **Memory Manager**, el **Garbage Collector** es una función que se ejecuta cada 10 segundos. Su funcionamiento consiste en iterar sobre el vector de bloques y revisar dos aspectos: el número de referencias y el estado de asignación (bool). Solo se eliminan los bloques que hayan sido asignados y que tengan cero referencias. Para la eliminación, se emplean métodos para quitar el elemento del vector de bloques, pero antes se crea un arreglo con las posiciones de memoria libres y se añaden al vector de espacios libres.

3.2.7. DumpFiles

Cabe recalcar que todas estas funciones generan **DumpFiles**, excepto la función de obtención de datos, ya que esta no genera cambios en la memoria. Por otro lado, las demás funciones sí lo hacen.

3.3. Solución MPointers

MPointers es una biblioteca de punteros inteligentes diseñada para la gestión remota de memoria mediante un **Memory Manager**, utilizando comunicación a través de sockets. Estos punteros actúan como intermediarios entre las aplicaciones cliente y un servicio centralizado de memoria, delegando las operaciones críticas de reserva, acceso y liberación de memoria.

La biblioteca implementa una interfaz que replica la sintaxis de punteros tradicionales en C++ mediante la sobrecarga de operadores:

- `operator*` para desreferenciación, mediante acceso al valor remoto.
- `operator=` para asignación de valores, o el incremento de referencias internas de un bloque a través de la asignación de su identificador en otro puntero.
- `operator&` para obtención de identificadores únicos, los cuales son manejados de forma interna por la clase.

La comunicación con el servidor se realiza mediante sockets, con un sistema de mensajes estructurados que siguen un protocolo claro, permitiendo operaciones como la creación de bloques de memoria, la consulta de valores y su modificación. Además, el sistema incorpora la gestión automática de referencias en el servidor, liberando los bloques de memoria cuando ningún MPointer los referencia, lo que previene fugas de memoria en el entorno distribuido.

Cabe destacar la implementación de mutexes para garantizar la seguridad en el entorno multihilos en cualquier procedimiento que implique una conexión con el **Memory Manager**.

4. Diseño General

Se presenta en 1 el diagrama UML del proyecto.

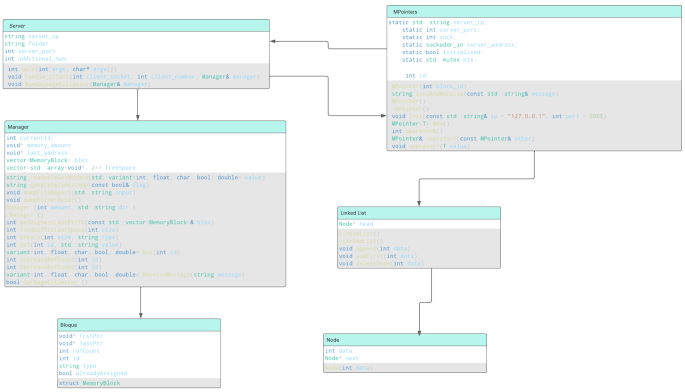

Figura 1: Diseño general con diagramas UML

5. Repositorio

El repositorio puede verse en el siguiente link: Repositorio.

6. Recursos

Se utilizó como recurso adicional la web Lucid, la cual sirvió para realizar el diagrama UML presentado.