

Proyecto 1

Algoritmos y estructuras de datos II (CE 2103)
Instituto Tecnológico Costa Rica

Anthony Artavia Leitón
2024067097

Instituto Tecnológico de Costa Rica
Cartago, Costa Rica
a.artavia.1@estudiantec.cr

Sergio Álvarez Chanto
2024013043

Instituto Tecnológico de Costa Rica
Cartago, Costa Rica
s.alvarez.2@estudiantec.cr

En resumen- El presente documento tiene la finalidad de mostrar la solución a un problema planteado como proyecto programa por parte de la materia algoritmos y estructuras de datos dos de la carrera de Ingeniería en Computadores del Tecnológico de Costa Rica. El objetivo principal por demostrar es el diseño e implementación de clases que encapsulen el uso de punteros en C++.

El proyecto I de Algoritmos y estructuras de datos II consiste en el diseño e implementación de un Administrador de memoria en c++, consiste en dos componentes fundamentales, el administrador de memoria y una biblioteca programada por nosotros llamada MPointers, además, para la comunicación del cliente – servidor, se hace el uso del framework gRPC, fundamental para crear los servicios. La principal funcionalidad de una parte es recíproca para el ideal funcionamiento de la otra, el administrador de memoria se encarga de reservar un bloque de memoria de un tamaño especificado y lo administra, posterior, la biblioteca MPointers es como una interfaz de alto nivel para el cliente, quien especifica las operaciones a ejercer sobre la memoria reservada.

Palabras clave— reserva de memoria, gRPC, punteros, c++, biblioteca, cliente - servidor.

I. TABLA DE CONTENIDOS

I.	TABLA DE CONTENIDOS.....	1
II.	DESCRIPCIÓN DE LA SOLUCIÓN:.....	1
III.	Biblioteca MPointers:	3
IV.	PRUEBAS REALIZADAS:	3
V.	DISEÑO GENERAL, DIAGRAMA UML:.....	3
VI.	ENLACE AL REPOSITORIO DE GITHUB:.....	3
VII.	CONCLUSIONES:	3
VIII.	REFERENCIAS:	4

II. DESCRIPCIÓN DE LA SOLUCIÓN:

Primero, se realizó la planificación del ¿cómo lograr el proyecto? Para esto se dividieron las distintas responsabilidades del proyecto, de manera que el trabajo fuera equitativo y equilibrado. Las divisiones fueron: Primero lograr instalar correctamente el framework de gRPC para poder trabajar en el modelo cliente – servidor, realizar correctamente una estructura básica del proyecto subdividiendo en carpetas. Posterior a esta organización inicial, la meta fue comenzar a implicar una reserva

de memoria con la operación malloc, por consiguiente, al cerrar el server, lograr liberar esa memoria reservada inicialmente. Para la parte de las operaciones individuales se configuró el archivo CMakeLists.txt y el .proto responsable de crear los servicios, esto para luego crear las dependencias para que gRPC pudiera enviar las requests y responses correctamente.

La parte de la desfragmentación, el garbage collector y la creación de registros se implementó de forma final con tal de tener preparada una base sólida, una vez lo mencionado estuvo listo, se implementó como la parte final del proyecto.

Se procede a explicar con mayor detalle la solución de cada requerimiento:

Primero se buscó la documentación necesaria para la instalación de gRPC y sus dependencias, posteriormente gracias a los ejemplos que vienen por defecto con la instalación del framework se logró comprender como realizar el esqueleto del cliente – servidor, los pasos para montar la estructura inicial fueron: Crear los folders necesarios para organizar los distintos archivos, creación del archivo CMakeLists.txt para ejecutar los archivos de ejecución y compilación, posterior el archivo .proto con todos los servicios que requería el proyecto, por último el código simple de la conexión tanto para el cliente como para el servidor.

Única asignación de memoria por medio de malloc: Este requisito se implementó haciendo uso de una asignación del malloc a una variable que almacenaría el tamaño de la reserva en memoria.

Peticiones que es capaz de realizar memory manager, como se implementaron, alternativas consideradas, limitaciones, problemas y demás ...

1. Create(size, type):

Este requerimiento en un inicio consistía en simplemente obtener una referencia a un espacio dentro de la memoria reservada, pero era solo una manera “best case” de hacerlo, pues no se tenían en cuenta problemas futuros como la desfragmentación y espacios de memoria vacíos, luego de que estábamos avanzados en el desarrollo del proyecto teníamos varias opciones para la implementación de este requisito, una era implementar una asignación de memoria simple y luego recurrir a diseñar algún algoritmo para la desfragmentación, de manera que al asignar vagamente el espacio y posterior liberación del mismo se ejecutara una búsqueda secuencial para reordenar espacios libres, esto con el fin de lograr optimizar el espacio.

La alternativa por la cual optamos fue evitar la desfragmentación gracias al uso de una estructura conocida, llamada “Allocator” esta consiste en crear una estructura “struct” con los atributos que se necesitan para darle la característica de un bloque de memoria, tal como: id, size(tamaño que tendrá), is_free(para poder saber si está libre o si contiene datos), start(indica donde comienza en la memoria reservada, esto permite saber de dónde a dónde va esa memoria almacenada), estos bloques serán almacenados en una lista llamada “bloques_memoria”. El propósito de este “create” es lograr administrar correctamente esa lista, es por esto por lo que se

realizan distintas optimizaciones, el flujo de trabajo que realiza este método es el siguiente:

Se cuenta con:

PrimeraOptimización(Encargada de encontrar el bloque libre con el espacio lo más justo posible entre todos los almacenados en la lista para almacenar la solicitud entrante).

Segunda Optimización(Calcula el espacio no utilizado al final del bloque grande inicial, la idea es usar el espacio contiguo libre que queda al final de la memoria reservada).

TerceraOptimización(Fusión de bloques libres adyacentes, la idea es unir bloques libre contiguos para crear uno más grande).

Como se puede apreciar la PrimeraOptimización tiene sentido cuando ya hay bloques en la lista, por lo que cuando el programa se inicia la lista al estar vacía, automáticamente se pasa a la SegundaOptimización, donde se calcula el espacio libre (total – usado) y se crea un bloque con ese valor, además se mete en la lista. En un segundo uso de “Create” normalmente tampoco se usa la PrimeraOptimización pues el bloque anterior estará ocupado “is_free->false;”, entonces no hay ningún bloque en la lista que cumpla la condición, por lo que se usa la SegundaOptimización otra vez. Si se libera el primer bloque(por ejemplo, por el garbage collector), entonces ahí se volvería a empezar por PrimerOptimización pero esta vez si el espacio es suficiente para almacenar entonces sí se define como best_block y se guarda en la lista.

Algo a tener en cuenta es que la tercera opción se usará únicamente si: La PrimeraOptimización no puede ser usada, por ejemplo, al inicio del programa(que la lista está vacía) o que no haya bloques con el espacio suficiente... y además si en la SegundaOptimización el free_space calculado como “la resta del total – lo solicitado” es menor a lo que se solicita. Es en ese momento donde entra la TerceraOptimización(Intenta fusionar los espacios libres en los bloques ocupados para poder formas más espacio utilizable).

Se procede a explicar profundamente que hace cada optimización:

Primera Optimización:

Se crea una variable puntero del tipo BloquesMemoria(que es la estructura que define los bloques a almacenar) llamada best_block que inicialmente está definida como nullptr, esta será la encargada de ir buscando el mejor bloque en la lista para definir como el espacio a almacenar, posterior por medio de un for lo que se hace es iterar sobre cada bloque posiblemente almacenado en el vector bloques_memoria que está definido en el constructor, se hace una primera verificación: si el bloque que se está analizando es libre y además su espacio disponible para almacenar es mayor que lo que se ocupe almacenar entonces procedemos a la última verificación que permite definir el mejor bloque a almacenar: si aún no se ha asignado un best_block o si el espacio disponible para almacenar del bloque actual es mayor o igual al necesario entonces se define al best_block como la dirección de memoria de ese bloque en la lista.

Posterior si se encontró ese best_block, lo que hacemos es definir ese best_block como best_block->is_free = false; y además se devuelve la ID de ese bloque, finalmente se genera el dump de memoria y se devuelve el grpc::Status::OK

Segunda Optimización:

Se ejecuta si no se encontró un bloque libre adecuado. Se encarga de comprobar si el espacio libre es suficiente por medio de (espacio libre = total - solicitado),

Si hay espacio suficiente, crea un nuevo bloque al final y lo añade a la lista de bloques_memoria, esto por medio de “next_id” que marca hasta donde se ha utilizado la memoria, por lo tanto, con la creación del bloque lo asigna en la posición de “next_id”, posterior incrementa ese next_id para futuras creaciones “next_id += size_needed”. Finalmente se devuelve el id de ese bloque, se devuelven las configuraciones del response como exitosas y además se genera el dump de memoria y finalmente se devuelve el Status OK.

Tercera Optimización:

Esta optimización se usa solo si las dos optimizaciones anteriores fallaron y consiste en recorrer el vector de bloques de memoria para **fusionar pares de bloques libres contiguos**, combinando sus tamaños en un solo bloque más grande. Si el espacio del bloque resultante de la fusión es suficiente para la solicitud actual, se marca como ocupado y se asigna; de lo contrario, continúa buscando más bloques adyacentes para fusionar. Esta estrategia **solo une bloques libres que estén físicamente adyacentes** en memoria (no separados por bloques ocupados) y actúa como último recurso para reducir la fragmentación antes de declarar un fallo por memoria insuficiente.

Primero un for se encarga de recorrer cada bloque en “bloques_memoria”, luego se verifica si el bloque actual está libre, posterior se obtiene el siguiente bloque por medio del id: “auto next_id = it + 1”. Luego se comprueba que exista un bloque siguiente(que no sea el final del vector) y además que el bloque siguiente también esté libre.

Posterior a esto se da la fusión: donde se suman los tamaños de los dos bloques contiguos libres “it->size += next_it->size” y posterior para no tener duplicados se elimina ese segundo bloque(ya fusionado) del vector “bloques_memoria.erase(next_it)”

Finalmente se hace un intento de asignación: donde primero se verifica si el bloque fusionado es suficiente para la solicitud entrante “if (it->size >= size_needed)”, si fuese suficiente entonces se marca ese bloque como ocupado, se configura la respuesta como exitosa y retorna el Status como OK.

Si esto falla, entonces se configura la respuesta como fallida y se retorna.

2. Set(id, value):

El propósito de este método es almacenar en un bloque de memoria un valor en específico.

Cuando el cliente tenga el ID de un bloque creado, lo utiliza para poder almacenar un valor, enviando dicho ID y un valor a almacenar, posterior se comprueba que el bloque exista y además no esté marcado como libre. También que el tamaño de lo que se va a almacenar no sea superior al disponible en el bloque.

Luego si las validaciones permiten avanzar, lo que se hace es copiar el valor al espacio de memoria usando memcpy.

Finalmente genera el dump para el registro y retorna estados de éxito. Las alternativas para este método se situaron en como copiar el valor a ingresar por el cliente, utilizar casos switch para cada tipo de dato, utilizar serialización entre otras, pero memcpy fue la mejor. Una imitación y problema fue el encontrar la forma correcta de copiar el valor.

3. Get(id, value):

El principal funcionamiento de este método es lograr recuperar el valor almacenado de un bloque de memoria.

Primero el cliente envía el ID del bloque a consultar, el servidor busca el bloque y verifica que exista y que no esté marcado como libre. Convierte el contenido del bloque a string, retorna el valor al cliente y finalmente devuelve éxito

En este proyecto presentó la idea de un administrador de memoria en C++, donde aplicamos los conocimientos aprendimos en la

materia de algoritmos y estructuras de datos II, el cometido principal no solo fue aplicar lo aprendido, sino aprender a desarrollar soluciones consistentes a los problemas que se presentaran, esto mediante el trabajo en equipo. Gracias a la organización necesaria se logró implementar de forma correcta todos los requerimientos que se pedían.

VIII. **REFERENCIAS:**

gRPC. (n.d.). C++ quick start.

<https://grpc.io/docs/languages/cpp/quickstart/>

Programación en Español. (2021, January 10). gRPC crash course

[Video]. YouTube. <https://youtu.be/iv9ylBYgACE>

