SAPIENZA
UNIVERSITÀ DI ROMA

ripmalloc_so

How I Learned to Stop Worrying and Love the Allocator

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea Triennale in Ingegneria Informatica e Automatica

Antonio Turco
ID number 1986183

Advisor
Prof. Giorgio Grisetti

Academic Year 2024/2025

ripmalloc_so
Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Version: June 6, 2025

Author's email: turco.1986183@studenti.uniroma1.it

Dedicato a...

Abstract

HIC SUNT ABSTRACTA RE

Contents

Chapter 1

Introduzione

- Cos'è il problema affrontato?
- Perché è importante?
- Come si colloca il mio contributo?
- Dichiarazione d'intenti: che cosa voglio fare

Chapter 2

Lavori Correlati/Basi

2.1 Strumenti e letteratura

Per approfondire il tema della programmazione OOP in C è stato consultato (nonostante non sia stato ritenuto di applicarne interamente gli insegnamenti per semplicità) il libro "Object-Oriented Programming With ANSI-C" del professor Axel-Tobias Schreiner.

Il progetto è basato in primo luogo sull'implementazione dello SlabAllocator e BuddyAllocator vista durante le lezioni del corso di Sistemi Operativi tenuto dal professor Grisetti. Tuttavia, la struttura è stata rivisitata e rivista profondamente. Sono stati di riferimento i lavori degli utenti mtrebi e emeryberger, pubblicati su GitHub: il primo per aver fornito chiare indicazioni sul funzionamento e i trade-off di diverse tipologie di allocatori di memoria, il secondo per il lavoro di catalogazione storica, che ha permesso di osservare il percorso compiuto dagli allocatori nel corso del tempo.

2.2 Descrizione del sistema

Il progetto contenuto nella repository è gestito in quattro cartelle principali. `bin` e `build` sono utilizzate durante il processo di compilazione, mentre `header` e `src` contengono il codice sorgente nella sua interezza.

2.2.1 L'interfaccia Allocator

Il contratto che gli Allocatori devono seguire consiste nell'interfaccia `Allocator` (definita in `./header/allocator.h`), che stabilisce le primitive necessarie:

- l'inizializzazione (*init*);
- la distruzione (*dest*);
- l'allocazione di memoria (*reserve*);
- il rilascio di memoria per uso futuro (*release*).

Queste operazioni sono progettate per un uso interno: infatti, gli argomenti sono passati attraverso modalità definite dalla libreria di sistema `<stdarg.h>`. Ciò introduce flessibilità nella nostra implementazione delle funzioni permettendoci di gestire i parametri in modo arbitrario, ma contemporaneamente costituisce un rischio, poiché le verifiche sulla correttezza del tipo e del numero non sono fatte a *compile-time*.

Per ovviare a questo problema e permettere al nostro programma di verificare correttamente che i parametri passati siano validi, introduciamo un *buffer* tra le funzioni interne e l'utente nella forma di funzioni *helper* segnalate come *inline*. Attraverso esse, il programma mantiene la sua flessibilità internamente senza dover sacrificare in sicurezza: la correttezza dei parametri passati alla chiamata è effettuata dal compilatore e contemporaneamente la performance non è eccessivamente impattata da questo passaggio intermedio grazie alla keyword *inline*. Essa indica al compilatore di ottimizzare aggressivamente la funzione, sostituendo alla chiamata il suo corpo e per questo motivo, è importante che queste funzioni *helper* siano brevi e concise, in modo da evitare *code bloat*.

È importante ricordare che *inline* non è che un suggerimento, e non un obbligo, per il compilatore: esistono modalità per forzare questa ottimizzazione, imponendo di applicarla a tutte le chiamate, ma questo potrebbe portare nel lungo termine a una minore ottimizzazione per via della quantità di codice, che renderebbe necessari più *cache swaps* del dovuto. Ulteriori test potrebbero mostrarne l'impatto e con ciò l'importanza di lasciare che sia il compilatore a occuparsi delle ottimizzazioni, ma ciò esula dagli scopi dell'analisi.

Ogni classe che implementa l'interfaccia `Allocator` deve implementare le proprie funzioni interne, che mantengono la stessa *signature*, e le funzioni *wrapper*, che invece possono avere una *signature* diversa in base alle necessità. Per esempio, nell'allocazione di memoria per uno *SlabAllocator* (che velocemente anticipiamo poter allocare unicamente blocchi di memoria di grandezza omogenea) non sarà necessario specificare la grandezza dell'area richiesta. In più, deve fornire anche una rappresentazione grafica del suo stato ai fini di *debugging* e analisi.

Le funzioni *helper* seguono una nomenclatura più vicina a quella della *libc*, in modo da rendere l'API più intuitiva e immediata. Esse sono:

- *Allocator_create* (*wrapper* di *Allocator_init*)
- *Allocator_destroy* (*wrapper* di *Allocator_dest*)
- *Allocator_malloc* (*wrapper* di *Allocator_reserve*)
- *Allocator_free* (*wrapper* di *Allocator_release*)

Per via del *linker* del linguaggio C, siamo costretti ad anteporre al nome della funzione la classe, come vediamo sopra. Sono state esplorate soluzioni a questo problema, ma sfortunatamente introducevano livelli di complessità oppure sacrificavano a livello di *type checking*. Grazie alla duplice struttura con funzioni *helper* e *internal* sarebbe possibile realizzare in C una forma semplice di polimorfismo, ma risulta sempre necessario, al netto dell'utilizzo di *macro* (che reintrodurrebbero i problemi evidenziati precedentemente), usare nomi univoci per ogni funzione con diversa combinazione di parametri.

Chapter 3

Implementazione di ripmalloc

Il progetto contenuto nella repository è gestito in quattro cartelle principali. `bin` e `build` sono utilizzate durante il processo di compilazione, mentre `header` e `src` contengono il codice sorgente nella sua interezza.

Chapter 4

Esperimenti, Casi d'Uso

- Descrizione di una run del sistema o, se applicabile, esperimenti qualitativi.

Chapter 5

Conclusioni

Riprendi la dichiarazione d'intenti al capitolo uno e metti le spunte.