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
Matricola 1986183

Relatore
Prof. Giorgio Grisetti

Anno Accademico 2024/2025

ripmalloc_so
Sapienza Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Versione: 16 giugno 2025

Email dell'autore: turco.1986183@studenti.uniroma1.it

Dedicato a...

Sommario

ABS HERE

Ringraziamenti

ACK HERE

Indice

Capitolo 1

Introduzione

- Cos'è la memoria dinamica? In cosa consiste la sua gestione?
- Qual è il ruolo dell'allocatore di memoria?
- Quali sono le metriche che distinguono un buon allocatore da un allocatore inefficiente?
- Perché è importante che l'allocatore di memoria sia efficiente? In quali contesti è essenziale?
- Quali sono le diverse tipologie di allocatori di memoria?

Capitolo 2

Lavori Correlati/Basi

2.1 Object Oriented C Programming

- Perché è stato scelto C per scrivere il programma?
- Perché è importante implementare le basi delle strutture della programmazione OOP in C?

Per approfondire il tema della programmazione OOP in C è stato consultato il libro "Object-Oriented Programming With ANSI-C" del professor Axel-Tobias Schreiner. Nonostante non sia stato ritenuto di applicarne interamente gli insegnamenti per semplicità, il testo si è rivelato essere un utile riferimento teorico. La decisione di usare C piuttosto che un linguaggio che fornisce supporto diretto a questo paradigma, come C++ o C#, nasce da un'esigenza didattica di "squarciare il velo di Maya" che spesso avvolge i meccanismi alla base della programmazione orientata agli oggetti. In particolare, si è ritenuto di voler sottolineare come la gestione dell'allocazione dinamica di memoria, strettamente legata all'architettura fisica del calcolatore, sia un aspetto fondamentale della programmazione a basso livello. Colui che per la prima volta decida di approcciare C trova nel semplice uso di `malloc` e `free` le prime grandi "responsabilità" da programmatore: un obbligazione a gestire autonomamente e responsabilmente una risorsa, che porta a un livello di consapevolezza maggiore sui meccanismi interni e le routine che costituiscono i sistemi operativi.

Scegliendo di modellare consapevolmente concetti che in C++ sono automaticamente gestiti dal compilatore si acquisisce maggiore consapevolezza sui dettagli implementativi e si sottolineano importanti punti per la comprensione di nozioni quali *memory leak*, *dangling pointers*, ciclo di vita, costruttori e distruttori.

2.2 Studi teorici di base

- Da quali fonti è possibile imparare il funzionamento degli allocatori?
- Quali sono delle soluzioni didatticamente interessanti?

Poiché la memoria dinamicamente allocata è un aspetto cardine del linguaggio C e dei sistemi operativi (e di tutta la programmazione a basso livello), la letteratura didattica a riguardo è ampia e di alta qualità. Di nota per la comprensione del funzionamento e del ruolo dei gestori dinamici della memoria sono i libri "The C Programming Language" (capitolo 8.7, "Example – A Storage Allocator") di B. Kernighan e D. Ritchie e "Computer Systems – A Programmer's Perspective" (capitolo 9.9, "Dynamic Memory Allocation") di R. Bryant. Nel primo libro abbiamo un esempio pratico di implementazione di un allocatore lineare a blocchi di dimensione variabili, attraverso l'uso di una Linked List per mantenere una lista dei blocchi liberi e che, in risposta a una operazione di `free`, unisce blocchi adiacenti. Questa implementazione molto immediata funge da dimostrazione del fatto che, nelle parole degli autori: "Sebbene l'allocazione dello storage sia intrinsecamente dipendente dall'architettura fisica, il codice illustra come le dipendenze dalla macchina possano essere controllate e confinate a una parte molto piccola del programma."

Il secondo volume, a nostro avviso, definisce in modo cristallino quale sia la principale fonte del problema. Secondo Bryant, "I programmatori ingenui spesso presumono erroneamente che la memoria virtuale sia una risorsa illimitata. In realtà, la quantità totale di memoria virtuale allocata da tutti i processi di un sistema è limitata dalla quantità di spazio di swap su disco. I bravi programmatori sanno che la memoria virtuale è una risorsa finita che deve essere utilizzata in modo efficiente." Questa osservazione è più che mai rilevante in contesti come la programmazione *embedded* e *real time*, così come nei sistemi operativi. La reale criticità nel mondo dell'allocazione dinamica non consiste in un debito tecnologico, in limiti intrinseci o in euristiche inefficienti, bensì in cattive abitudini dei programmatori. Il risultato è sottovalutare l'importanza degli allocatori in contesti in cui la loro efficienza non sia strettamente indispensabile, optando piuttosto per scegliere di adoperare artefici di gestione della memoria che sacrificano spazio e prestazioni in cambio di una complessità artificiosa e che risultano essere sibillini e difficilmente applicabili al di fuori del contesto per cui sono stati concepiti.

> "Such problems may be hidden because most programmers who encounter severe issues may simply code around them using ad-hoc storage management techniques—or, as is still painfully common, by statically allocating "enough" memory for variable-sized structures. These ad-hoc approaches to memory management lead to 'brittle' software with hidden limitations (e.g., due to the use of fixed-size arrays). The impact on software clarity, flexibility, maintainability, and reliability is significant, though difficult to estimate. It should not be underestimated, however, because these hidden costs can incur major penalties in productivity—and, to put it plainly, human costs in sheer frustration, anxiety, and general suffering."

L'autore continua, definendo quattro problemi che ogni implementazione di un gestore dinamico di memoria deve risolvere. Essi sono:

- L'organizzazione dei blocchi liberi in memoria;
- La scelta del blocco corretto a seguito di una richiesta;

- Il meccanismo di *splitting* in blocchi di memoria delle dimensioni necessarie;
- Le modalità di *coalescing* di blocchi liberi per poter soddisfare richieste future.

Illuminante è stato il capitolo "Dynamic Storage Allocation" del volume primo di "The Art of Computer Programming", D. Knuth.

2.3 Riferimenti storici

- Quali sono le principali problematiche negli allocatori moderni evidenziate dalla letteratura?
- Quali sono le possibili soluzioni?

L'articolo "Dynamic Storage Allocation, A Survey and Critical Review" di P. Wilson et al. è stato preso come riferimento storico: in particolare il capitolo 4 presenta un sunto della letteratura pubblicata sull'argomento negli anni precedenti e delle soluzioni proposte per affrontare il problema, che gli autori sottolineano essere "per lo più considerato essere già risolto o irrisolvibile". Un problema che emerge in più punti della letteratura sulla allocazione dinamica di memoria consiste nelle acute differenze tra le modalità con cui generalmente vengono fatti benchmark sintetici delle capacità degli allocatori e quelle con cui negli use cases reali vengono fatte richieste da parte dell'utente. Il carico di lavoro di un allocatore che viene sottoposto a una suite di test non rispecchia le applicazioni nel mondo in cui successivamente esso viene applicato: ciò è dovuto in gran parte alla natura eterogenea delle stesse.

Gli autori continuano, sottolineando che soluzioni di gestione dinamica di memoria efficienti fanno uso di "regolarità" nel comportamento del programma. Ottimizzare per il caso più comune risulta dunque essere controproducente rispetto alle applicazioni reali.

2.4 Ispirazione per la struttura

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.

Di particolare importanza è stata l'analisi di `dlmalloc`.

Nessuno conosce le esigenze di gestione dinamica della memoria come il suo creatore, e dunque è appropriato che egli scelga e, se necessario, implementi una soluzione *ad hoc* piuttosto che affidarsi a un sistema *one size fits all*.

Capitolo 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.

3.1 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.

3.1.1 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.

Capitolo 4

Esperimenti, Casi d'Uso

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

Capitolo 5

Conclusioni

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