SAPIENZA
UNIVERSITÀ DI ROMA

Implementazione di un allocatore di memoria bare metal in C

Come ho imparato a non preoccuparmi e ad amare l'allocatore

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

Implementazione di un allocatore di memoria bare metal in C
Sapienza Università di Roma

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

Versione: 23 giugno 2025

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

Dedicato a...

Sommario

ABSTRACT 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?

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 il linguaggio **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 Letteratura scientifica sull'allocazione dinamica

L'articolo *Dynamic Storage Allocation, A Survey and Critical Review* di P. Wilson et al. è stato adottato 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 argutamente essere "per lo più considerato essere già risolto o irrisolvibile". Un punto critico che emerge infatti in più punti della letteratura riguarda le differenze tra i *benchmark* sintetici usati per valutare gli allocatori e i carichi di lavoro reali. Le suite di test, infatti, raramente riflettono le profonde correlazioni e le sistematiche interazioni tra

allocazioni e deallocazioni. La mancata comprensione di questi collegamenti causa incomprensioni e interpretazioni errate dei risultati di questi test, che sono dunque inadatti a rappresentare l'efficienza degli allocatori nel mondo reale.

Le conseguenze di questa divergenza sono immediate: l'allocazione dinamica è considerata un problema "risolto" per chi abbia abbondanti risorse computazionali a disposizione e contemporaneamente "irrisolvibile" in contesti dove vi siano importanti limitazioni temporali o spaziali. A tal proposito, lo studio *Real-Time Performance of Dynamic Memory Allocation Algorithms* di I. Puaut offre un contributo prezioso, svolgendo il pregevole lavoro di combinare test (reali e sintetici) con precisi studi analitici. Nessuna possibilità è lasciata inesplorata ed è dimostrato che, in determinate condizioni, è possibile realmente predire il comportamento degli allocatori di memoria in casi dove è essenziale che essi rispettino determinati parametri per giustificarne l'applicazione.

Le conclusioni delle esperienze di Puaut confermano le tesi di Wilson: l'inefficienza non risiede negli allocatori stessi, quanto nella mancata comprensione del loro funzionamento. Il timore nella percepita inefficienza dell'allocazione dinamica porta a scelte inappropriate. Essa presenta certamente diverse sfide, ma attraverso caute valutazioni è possibile applicarla anche laddove tradizionalmente viene preferita l'allocazione statica.

> "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."

Gli autori del survey continuano, sottolineando che soluzioni efficienti per la gestione dinamica di memoria fanno uso di "regolarità" nel comportamento del programma. Infatti, osservando come viene allocata e deallocata la memoria è possibile scegliere la corretta politica di gestione per il proprio caso d'uso. Non esiste dunque una soluzione "set and forget" e invece risulta essere appropriato dedicare risorse all'esplorazione di diverse soluzioni. Successivamente l'articolo definisce una chiara tassonomia delle principali specie di allocatori, la quale avremo modo di approfondire nel capitolo terzo.

2.3 Didattica degli allocatori

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. 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. Illuminante è stato il capitolo *Dynamic Storage Allocation* del volume primo di *The Art of Computer Programming*, di D. Knuth. Quest'ultimo volume va nel dettaglio spiegando l'analisi matematica che supporta le euristiche comunemente adottate nel progetto degli allocatori di memoria, fornendo chiari esempi e illustrazioni.

Nel libro di Kernighan e Ritchie abbiamo un esempio pratico di implementazione di un allocatore lineare a blocchi di dimensione variabili, attraverso l'uso di una *Linked List* per mantenere un indice dei blocchi liberi e che, in risposta a una operazione di *free*, unisce blocchi adiacenti. Questa implementazione descritta dagli stessi autori come "semplice e immediata" funge da dimostrazione del fatto che "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 citato, ad opera di Bryant, definisce a nostro avviso in modo cristallino quale sia la principale fonte del problema. Secondo l'autore, "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 nella progettazione di 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 di questa percezione è apparente nell'assenza di riconoscimento dell'importanza degli allocatori quando la loro efficienza non sia strettamente indispensabile. Nei contesti in cui invece essa lo sia, viene spesso scelto di adoperare artefici di gestione della memoria che evitano la componente dinamica, sacrificando spazio e prestazioni in cambio di una complessità sibillina e artificiosa, che li rende di difficile manutenzione e applicabilità al di fuori del contesto per cui sono stati concepiti.

L'autore continua definendo i quattro problemi che ogni implementazione di un gestore dinamico di memoria deve risolvere. Sottolineiamo che queste necessità si manifestano nel caso in cui si decida che l'allocatore debba essere *general use*, che sono l'oggetto di analisi in corso. In casi particolari, si può decidere di sacrificare la generalità dell'allocatore in cambio di risultati migliori. Essi sono:

1. L'organizzazione dei blocchi liberi in memoria;
2. La scelta del blocco corretto a seguito di una richiesta;
3. Il meccanismo di *splitting* in blocchi di memoria delle dimensioni necessarie;
4. Le modalità di *coalescing* di blocchi liberi per poter soddisfare richieste future.

Nel corso delle descrizioni del nostro progetto, descriveremo come li abbiamo affrontati in tutte le specifiche implementazioni, sottolineando il costo della nostra soluzione, così come i compromessi accettati.

Di particolare importanza è stata l'analisi di *dlmalloc*, l'allocatore di memoria sviluppato da Doug Lea intorno agli anni novanta del secolo scorso. Esso ha fornito le basi per *ptmalloc*, una fork modificata per essere *thread-safe* da Wolfram Gloger e che successivamente è stata adottata dalla *glibc* (*GNU C library*). Studiare questa implementazione è stato particolarmente utile in quanto rappresenta un esempio di allocatore dinamico di memoria con *chunk* di dimensioni variabili largamente adoperato e documentato. Inoltre, è stato interessante studiare come il problema dell'accesso concorrente sia stato risolto attraverso *mutex* e "arene", nonostante nella nostra implementazione non siano state integrate soluzioni per affrontare il problema del *multithreading*.

2.4 Ispirazione per la struttura

Il progetto si basa principalmente sull'implementazione dello *SlabAllocator* e *BuddyAllocator* vista durante le lezioni del corso di Sistemi Operativi tenuto dal professor Grisetti. Tuttavia, la struttura presenta sostanziali differenze, che rendono le procedure leggermente diverse. Sono esplorate più nel dettaglio nel capitolo successivo.

Sono stati di riferimento per lo sviluppo le pubblicazioni dell'utente **mtrebi**[1] e di Emery Berger, professore presso l'Università di Massachusetts Amherst[2] su Github: il primo ha fornito chiare indicazioni sul funzionamento e i compromessi tra diverse tipologie di allocatori di memoria, mentre il secondo ha offerto una preziosa analisi storica, catalogando diversi popolari algoritmi di allocazione che si sono succeduti nel corso del tempo. Ciò ha permesso di osservare l'evoluzione nel tempo delle soluzioni per l'allocazione dinamica di memoria.

[1] `https://github.com/mtrebi/memory-allocators`
[2] `https://github.com/emeryberger/Malloc-Implementations`

Capitolo 3

Implementazione

Il progetto contenuto nella repository è gestito in quattro cartelle principali. *bin* e *build* contengono i risultati del processo di compilazione, mentre il codice sorgente è contenuto in *header* e *src*. Il programma contiene anche delle basilari implementazioni delle strutture dati per esso necessarie: una semplice *double linked list* e una *bitmap*. La loro struttura è volutamente molto semplice per evitare costi di tempo aggiuntivi e non è d'interesse ai fini di questa analisi. Di ogni funzionalità viene accertato il comportamento desiderato attraverso una serie di test.

Notiamo che tutte le implementazioni descritte successivamente condividono alcune caratteristiche, quali la possibilità di soddisfare unicamente richieste di memoria di dimensioni contenute nei parametri di creazione dell'allocatore. La dimensione dell'area di memoria dinamicamente gestita infatti non cambia nell'eventualità che venga fatta un'allocazione impossibile da soddisfare. L'allocatore non reclama ulteriore memoria dal sistema operativo neppure a seguito di richieste che potrebbero essere soddisfatte se memoria fosse rilasciata ad esso. Invece in entrambi i casi viene gestito l'errore ritornando al richiedente un valore invalido per segnalare l'insuccesso.

3.1 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`).

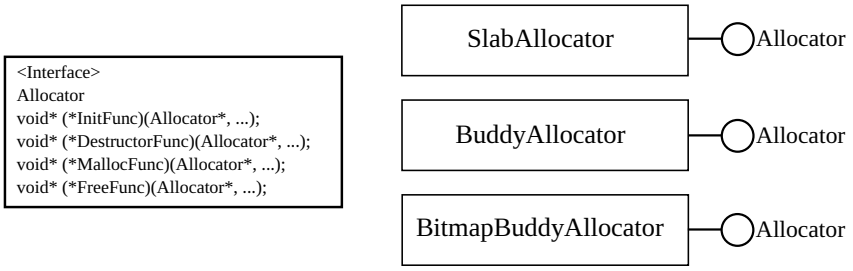

Figura 3.1. Diagramma UML dell'interfaccia Allocator e delle sue implementazioni.

Secondo la più recente specifica UML, *"Un'interfaccia è un tipo di classificatore che rappresenta una dichiarazione di un insieme di caratteristiche e obblighi pubblici che insieme costituiscono un servizio coerente. Un'interfaccia specifica un contratto; qualsiasi istanza di un classificatore che realizzi l'interfaccia deve soddisfare tale contratto."* Tutti gli allocatori devono quindi implementare metodi che abbiano signature corrispondente e che svolgano le operazioni elencate sopra.

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* è un suggerimento, 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 necessario. 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 a 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.

Tutte le classi che implementano l'interfaccia `Allocator` usano *mmap* per chiedere memoria da gestire al sistema operativo. Durante la fase di progetto, è stato valutato alternativamente di poter utilizzare la primitiva *sbrk*, fornita dalla libreria C standard, che permette di "accrescere" l'*heap* esplicitamente. Questo approccio avrebbe permesso un più granulare controllo sulla memoria, al costo di una minore flessibilità. In più, la prospettiva di usare *sbrk* avrebbe permesso di studiare come avveniva l'allocazione di memoria in tempi passati.

Si è ritenuto tuttavia di usare *mmap* per evitare complicazioni nella deallocazione (la memoria allocata attraverso *sbrk* può infatti essere deallocata solamente in modo sequenziale o si rischia di introdurre frammentazione). La struttura a cui si può accedere attraverso *sbrk* è infatti di tipo LIFO, ossia una pila di memoria. Ciò avrebbe potuto creare problemi laddove gestori fossero distrutti in ordine diverso da quello di creazione e laddove si fosse deciso di permettere l'utilizzo *multithreaded* (che al netto di possibili complicazioni impreviste potrebbe essere aggiunto con relativa facilità adoperando *mutex* per le operazioni di richiesta e rilascio di memoria).

La flag `MAP_ANONYMOUS` (anche nota come `MAP_ANON`) è stata adoperata alla chiamata di *mmap*. Essa fa sì che la memoria richiesta non sia "supportata" da alcun file. Dal manuale, "The mapping is not backed by any file; its contents are initialized to zero. The fd argument is ignored; however, some implementations require fd to be `-1`. If `MAP_ANONYMOUS` (or `MAP_ANON`) is specified, and portable applications ensure this. The offset argument should be zero. for `MAP_ANONYMOUS` in conjunction with `MAP_SHARED` added in Linux 2.4." La memoria si trova dunque nella RAM fisica e non in un file (chiaramente a meno che non sia stata posta in un file di swap).

Feature	*sbrk*	*mmap*(MAP_ANONYMOUS)
Memory Type	Heap-only	Any virtual address
Fragmentation	High (contiguous heap)	Low (independent mappings)
Deallocation	Only last block	Arbitrary (*munmap*)
File Backing	No	No (unless explicitly mapped)
Modern Usage	Legacy (brk in *malloc*)	Preferred for large allocations

3.2 La classe SlabAllocator

Lo *slab allocator* è un gestore pensato per richieste di memoria di taglia costante. La sua struttura interna lo rende particolarmente efficiente al costo di poca flessibilità. Ciò lo rende adatto quando sono necessarie solamente allocazioni di memoria di dimensione nota e fissa (ad esempio, un'istanza di una classe): il termine *slab* fa riferimento a questa "fetta" di memoria.

La prima menzione di un'implementazione di *slab allocator* viene descritta nell'articolo di Jeff Bonwick "The Slab Allocator: An Object-Caching Kernel Memory Allocator" del 1994. In esso vengono elencati i benefici di una soluzione che, rispetto a quella da noi implementata, risulta ben più complessa e strutturata. Il codice di Bonwick infatti trae beneficio non solo dalla taglia definita dei *chunk*, ma anche dalla conoscenza della struttura dei dati che verrà allocata nella memoria richiesta (dichiarata alla creazione del gestore). I blocchi liberi vengono già inizializzati come oggetti e mantengono la loro struttura alla restituzione del blocco, evitando così di dover spendere risorse per riorganizzare la memoria alla prossima richiesta. L'idea consiste nel "preservare la porzione invariante dello stato iniziale di un oggetto nell'intervallo tra gli usi, in modo che essa non debba essere distrutta e ricreata ogni volta che l'oggetto è usato."

Non scendiamo ulteriormente nei dettagli del gestore di Bonwick per semplicità, ma notiamo che per quanto possa sembrare a posteriori non significativa, l'eleganza della sua soluzione è degna di nota. L'autore dell'articolo infatti non solo definisce algoritmi efficienti e con strumenti approfonditi per il *debugging*, ma si cura di approfondire la relazione tra il suo algoritmo e le strutture del sistema operativo, in particolare con il *Translation Lookaside Buffer*, fornendo chiare evidenze dell'attenzione posta non solo nell'approccio teorico, ma anche all'applicazione pratica del suo gestore.

La specializzazione della soluzione applicata da Bonwick la rende ideale per l'utilizzo all'interno di sistemi operativi. Essi spesso gestiscono numerosi oggetti rappresentati da strutture dati di grandezza nota e fissa (*socket*, *semafori*, *file*...). La prima implementazione di questo modello è presentata nel kernel di SunOS 5.4, per poi comparire a uso interno a molti altri kernel, compreso quello di FreeBSD (v5.0) e Linux (a partire dalla versione 2.1.23), dove successivamente diventerà anche disponibile per l'uso da parte dell'utente.

Nella nostra implementazione non viene fatto *caching* della struttura interna dell'oggetto e l'utente è lasciato libero di gestire liberamente lo slab assegnato. Chiaramente, questo lo rende ordini di grandezza più lento della soluzione applicata da Bonwick. Lo scopo didattico nonostante questo è la dimostrazione di come l'efficienza dei gestori dinamici di memoria sia strettamente correlata alla compren-

sione da parte del programmatore delle richieste fatte durante il corso della vita dell'applicazione: lo *slab allocator* può essere usato al massimo delle sue potenzialità solo a seguito della profonda comprensione del succedersi delle allocazioni e rilasci di memoria.

Funzionamento dello SlabAllocator

Come stabilito precedentemente, l'utente non usa le funzioni interne per accedere alle funzionalità del gestore, ma bensì adopera gli helper qui delineati:

L'inizializzazione di un'istanza di SlabAllocator richiede la grandezza dello slab (nei termini di Bonwick, la grandezza dell'oggetto da immagazzinare) e il numero delle stesse. Dopo una serie di controlli sui parametri, la memoria richiesta viene suddivisa in blocchi. Essi sono dunque organizzati in una *linked list*, che mantiene un pratico riferimento alla memoria disponibile e la cui lunghezza massima è pari al numero totale di blocchi. Al termine dell'uso le operazioni di distruzione sono immediate: l'unica accortezza è restituire la memoria al sistema operativo con *unmap*.

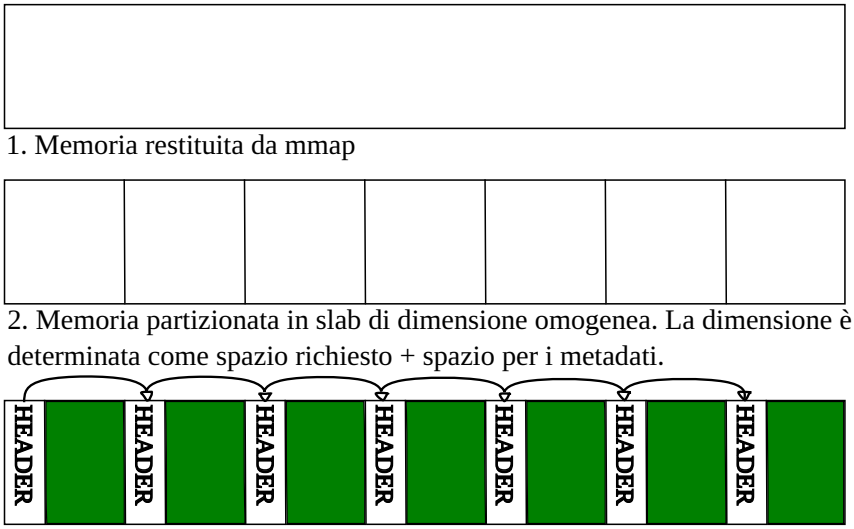

Figura 3.2. Inizializzazione dello SlabAllocator: suddivisione della memoria in blocchi di dimensione fissa e organizzazione in una lista concatenata.

Lo spazio per gestire l'appartenenza del blocco alla lista (ossia i campi di SlabNode, sottoclasse di Node) sono inseriti in cima al blocco. Ciò rende la struttura manipolabile da parte dell'utente, che può inavvertitamente o con intenzioni maligne corromperli scrivendo sopra di essi. A questa problematica sarebbe possibile porre rimedio mantenendo in memoria una struttura dati che tenga un riferimento

di tutti gli indirizzi allocati e li possa dunque verificare. Tuttavia, ciò introdurrebbe complicazioni e la scelta implementativa di "fidarsi dell'utente" ricalca quella che è stata adottata nella libc con *malloc*.

Poiché tutti i blocchi hanno la stessa dimensione, alla richiesta non è necessario stabilire quale di essi sia più opportuno allocare: la suddivisione avviene a priori durante l'inizializzazione del gestore, e la taglia dei blocchi non è modificata in nessun momento. La lista viene consultata e il blocco in testa viene estratto e restituito. Quando un blocco viene rilasciato, l'indirizzo di memoria viene controllato: se esso risulta essere corretto, viene semplicemente inserito al primo posto della lista per uso futuro. Notiamo che l'ordine della lista non rappresenta assolutamente la contiguità dei blocchi e richieste immediatamente successive possono ritornare blocchi non contigui.

Efficienza dello SlabAllocator

Descriviamo ora più nel dettaglio la complessità computazionale delle operazioni compiute dal gestore. L'allocazione ha un costo costante, così come la liberazione di un blocco, poiché in entrambi i casi viene semplicemente manipolata la testa di una *linked list* contenente i riferimenti ai blocchi liberi. I blocchi non sono in alcun modo manipolati: la loro grandezza rimane costante e questo elimina completamente i costi legati alle operazioni di divisione e unione.

Grazie alla sua struttura particolare, lo *slab allocator* non può mai presentare frammentazione esterna: poiché tutti i blocchi hanno la stessa dimensione, se è presente almeno uno slab di memoria libero, la richiesta dell'utente potrà essere esaudita e non può mai esistere memoria libera che l'utente non può chiedere di utilizzare. La frammentazione interna viene invece limitata dal programmatore, che, conoscendo le proprie necessità, può scegliere all'inizializzazione del gestore la dimensione del blocco più appropriata per i propri scopi.

Operazione	SlabAllocator
Allocazione	$O(1)$
Deallocazione	$O(1)$
Ricerca blocco libero	$O(1)$
Frammentazione interna	Determinata dal programmatore all'inizializzazione
Frammentazione esterna	Nulla

L'efficienza dello *slab allocator* dipende quindi dalla corretta scelta iniziale della dimensione dei blocchi. Tuttavia, in scenari dove le esigenze variano nel tempo (ossia si rende necessaria l'allocazione di oggetti di taglia diversa) è possibile combinare più gestori slab, ciascuno ottimizzato per una diversa dimensione. Questo approccio ibrido mantiene i vantaggi della complessità costante per le operazioni base, introducendo un trade-off legato alla gestione di più liste separate. La frammentazione interna rimane comunque controllabile, poiché limitata alla discrepanza tra la dimensione richiesta e quella dello slab più adatto.

Abbiamo già definito come, nel caso sia nota la dimensione massima necessaria per un blocco di memoria, lo *slab allocator* sia molto efficiente. Tuttavia, si potrebbero presentare situazioni in cui gli slab completamente liberi occupano memoria

inutilmente, perché ad esempio il numero di *chunk* è molto maggiore di quello degli oggetti che contemporaneamente vengono allocati. La necessità di slab del programma potrebbe variare nel corso delle operazioni da esso svolte. Per mitigare questo problema, alcune implementazioni introducono meccanismi di *reclaiming*: dopo un periodo di inattività o sotto pressione di memoria, gli slab vuoti possono essere rilasciati al sistema operativo. Questa operazione, seppur con un costo aggiuntivo (tipicamente $O(n)$ rispetto al numero di slab liberi), è compensata dalla flessibilità nel ridurre l'impronta memoria quando necessario. Per la nostra applicazione ciò non è stato ritenuto necessario.

Rispetto a gestori generici (come *buddy system* o *malloc* tradizionale), lo *slab allocator* eccelle in velocità e assenza di frammentazione esterna, ma è meno adatto a contesti con richieste eterogenee. La sua complessità spaziale è proporzionale al numero di slab preallocati, il che lo rende ideale per sistemi con risorse dedicate e pattern di allocazione prevedibili.

3.3 La classe BuddyAllocator

Il problema dell'allocazione di memoria per richieste di dimensioni variabili rimane un tema aperto e ampiamente discusso, con approcci diversi. Essi hanno nel corso del tempo suscitato dibattiti e proposte contrastanti. Sono state sviluppate numerose soluzioni, ciascuna con i propri vantaggi e limiti, ottenendo livelli di adozione e consenso variabili nell'ambito dei sistemi moderni.

I primi tentativi alla divisione dinamica dello spazio disponibile presero il nome di "sequential fits". In base alle necessità e richieste del programma in esecuzione, la memoria viene divisa in blocchi di dimensione variabile. Essi, organizzati in una o più liste concatenate, sono esplorati con costo lineare per trovare il first (il primo blocco sufficientemente grande) o best fit (il blocco più piccolo in grado di soddisfare la richiesta).

Questa soluzione, famosamente esplorata da Knuth, ha importanti difficoltà. La perdità di scalabilità per via del costo lineare è un punto critico: all'aumentare del numero di blocchi, il costo temporale della ricerca diventa proibitivo. Sebbene con dovuti accorgimenti si possano evitare un eccessivo overhead e una debilitante frammentazione, l'inefficienza della scansione lineare è un fattore limitante nei contesti ad alte prestazioni.

L'evoluzione di questo algoritmo mantiene la divisione dinamica in taglie non prestabilite, ma prova a risolvere il problema della lunghezza eccessiva: investire nell'organizzazione maggiore spazio per gestire i blocchi liberi più efficientemente permette di velocizzare la ricerca. La memoria disponibile viene suddivisa quindi in blocchi liberi, che sono però raccolti in liste diverse in base alla loro taglia. Le liste sono dunque numerose, ma di lunghezza minore, e quindi sono più facilmente esplorabili.

Al momento della richiesta, è esaminata la lista contenente i blocchi della taglia più appropriata, e laddove non via sia un blocco adeguato viene ricursivamente controllata la lista di blocchi di taglia "superiore". Il blocco eventualmente individuato è suddiviso e la memoria in eccesso (quella che non risulta necessaria per soddisfare la richiesta di memoria) è organizzata in un nuovo chunk libero che viene riposto

nella lista corretta secondo la sua grandezza. Questo meccanismo viene chiamato nell'articolo di Wilson et al. "segregated free lists".

L'allocatore buddy è descritto nella stessa pubblicazione come un "caso particolare" di questa tipologia di allocatori. Inventato da Harry Markowitz nel 1963 e pubblicato per la prima volta nell'articolo "A Fast Storage Allocator" del 1965 da Kenneth C. Knowlton, ingegnere presso Bell Telephone Laboratories, il buddy system è facile da implementare, e presenta buoni risultati se usato in risposta a richieste di taglia variabile, ma fissa e nota.

La differenza rispetto agli algoritmi che lo hanno preceduto consiste principalmente nelle politiche di splitting e coalescing. Se la metodologia descritta nel paragrafo precedente non stabilisce esplicitamente se, come o quando i blocchi liberi debbano essere riuniti e aggiunti alle free lists di grandezza maggiore, i buddy systems invece stabiliscono una chiara gerarchia che rende il procedimento più ordinato.

Quando è necessario dividere un blocco (che prende il nome di parent) per soddisfare una richiesta, esso viene diviso in parti uguali e i blocchi ottenuti diventano buddies, aventi chiaramente la stessa dimensione. Al rilascio da parte dell'utente, il blocco controlla il suo buddy e verifica se esso sia a sua volta libero. Nell'eventualità che entrambi i buddies siano contemporaneamente non riservati dall'utente, essi vengono riunificati nel blocco parent da cui derivano. Il buddy allocator rappresenta una soluzione elegante al problema della frammentazione esterna grazie alla sua struttura costituita da blocchi le cui dimensioni sono esclusivamente potenze di due. Ciò previene la formazione di aree di memoria inutilizzabili e garantisce che tutti i blocchi allocati abbiano dimensioni standardizzate.

Questa differenza consente di evitare un problema significativo che emerge quando la dimensione dei blocchi non è vincolata. In particolare, pattern di allocazione tipici – come l'alternanza di allocazioni e deallocazioni di blocchi di dimensioni diverse – causano frammentazione esterna negli allocatori che adottano sequential fits o segregated free lists. La libertà nella gestione delle dimensioni dei blocchi unita alla ricerca lineare porta alla formazione di numerose aree libere sparse e non contigue. Gli allocatori con segregated free lists, sebbene più efficienti grazie alla suddivisione in liste separate per intervalli di dimensione, non sono immuni al problema.

L'architettura del buddy system risolve radicalmente il problema della frammentazione esterna tipica degli allocatori tradizionali. La memoria libera viene infatti divisa equamente in base alle necessità reali del programma e costantemente riaggregata in blocchi ordinati e perfettamente allineati. Tuttavia, questa soluzione non è esente da compromessi. L'arrotondamento sistematico alla potenza di due superiore comporta inevitabilmente una certa quantità di frammentazione interna, particolarmente evidente quando le richieste di memoria sono solo leggermente superiori a una data potenza di due. Inoltre, la rigidità del sistema lo rende meno adatto a gestire pattern di allocazione estremamente variabili o imprevedibili.

Funzionamento del BuddyAllocator

Dalla descrizione del sistema buddy, notiamo facilmente che la struttura dati delineata corrisponde a un albero binario. Infatti, ogni nodo (blocco di memoria) tranne la radice possiede un singolo genitore e un buddy. Esso può inoltre a sua

volta essere scomposto in ulteriori due nodi liberi. Un vantaggio della struttura binaria è che il buddy corrisponde sempre con il blocco adiacente (precedente o successivo).

Ogni blocco di memoria è rappresentato da un BuddyNode, che contiene metadati come la dimensione, un'indicazione sullo stato e puntatori al buddy e al parent. La scelta di memorizzare esplicitamente queste relazioni, anziché calcolarle dinamicamente attraverso manipolazione degli indirizzi di memoria, semplifica il debug e la visualizzazione dello stato dell'allocatore: infatti, poiché sono note la taglia del blocco e l'indirizzo di partenza, gli header del buddy e del parent potrebbero essere raggiunti senza bisogno di immagazzinare esplicitamente questa informazione nell'header.

I nodi non sono salvati in una struttura ad albero, ma bensì in una serie di free lists, corrispondenti ai vari livelli dello stesso. La metodologia è ripresa dalle tecniche elencate precedentemente negli algoritmi "segregated free lists". Alla creazione, viene richiesto all'utente la grandezza dell'area di memoria da gestire e il numero massimo di livelli (alternativamente, poteva essere richiesta la grandezza del blocco di dimensione minima). L'allocatore utilizza due SlabAllocator interni: uno per gestire i BuddyNode e l'altro per le liste libere, che vengono tutte inizializzate alla creazione. Questa scelta rappresenta un chiaro luogo dove le caratteristiche dello slab allocator possano essere valorizzate, poiché le dimensioni degli oggetti allocati sono fisse e note a priori.

Efficienza del BuddyAllocator

L'operazione di allocazione cerca prima nella lista libera del livello appropriato. Se non trova blocchi disponibili, risale ai livelli superiori, dividendo i blocchi fino a raggiungere la dimensione desiderata. Questo approccio garantisce un costo $O(1)$ nel caso ideale (blocco disponibile nel livello corretto) e $O(L)$ nel caso peggiore, dove L è il numero di livelli. La fusione dei blocchi liberi avviene in tempo $O(L)$, grazie alla verifica ricorsiva dello stato del buddy.

L'uso di free lists separate per ogni livello elimina la necessità di strutture ad albero complesse, semplificando l'implementazione e riducendo l'overhead. Tuttavia, l'allocatore paga un costo in termini di memoria per i metadati aggiuntivi (puntatori a buddy e parent), che potrebbe essere evitato con un calcolo dinamico degli indirizzi dei buddy. La tecnica adoperata nel BuddyAllocator evita frammentazione esterna, ma rischia di introdurne interna.

Operazione	BuddyAllocator
Allocazione	$O(1)$ / $O(L)$
Deallocazione	$O(1)$
Ricerca blocco libero	$O(1)$ / $O(L)$
Frammentazione interna	Potenzialmente molto alta
Frammentazione esterna	Generalmente bassa

3.4 La classe BitmapBuddyAllocator

Nelle implementazioni analizzate finora, la ricerca di un blocco libero avviene tipicamente tramite l'esplorazione di liste concatenate. Se queste sono correttamente ordinate o suddivise per dimensione, la scansione può essere relativamente efficiente quando il blocco cercato è presente. Tuttavia, un problema intrinseco di questo approccio è la possibile discontiguità spaziale dei blocchi nella lista, che può essere causa di inefficienza nella gestione della cache, causando numerosi miss.

Per ovviare a questa limitazione, sono stati introdotti allocatori che utilizzano strutture dati più avanzate per memorizzare le informazioni sui blocchi liberi, migliorando così l'efficienza grazie a un utilizzo della cache più avveduto. Essi nell'articolo di Wilson prendono il nome di indexed fit. Tra le strutture usate, alberi binari bilanciati (self-balancing binary trees) e heap si sono dimostrati particolarmente efficaci; ciononostante, essi richiedono un overhead gestionale non trascurabile per mantenere l'equilibrio della struttura.

Un approccio alternativo e più semplice rispetto alle strutture dati complesse è l'utilizzo di bitmap, in cui ogni bit rappresenta lo stato, libero o occupato, di un blocco di memoria. A differenza delle liste concatenate (che richiedono dereferenziamenti di puntatori potenzialmente dispersi in memoria, con conseguenti cache miss), le bitmap permettono di verificare lo stato dei blocchi in modo più efficiente, poiché le informazioni risiedono in memoria contigua. Ciò può anche avvalersi delle istruzioni SIMD (Single Instruction, Multiple Data) e funzionalità hardware avanzate fornite dall'architettura.

Questo metodo consente una ricerca estremamente rapida di blocchi contigui liberi, migliorando significativamente la località spaziale e riducendo i problemi di caching tipici delle liste. Tuttavia, poiché la verifica della disponibilità richiede comunque l'ispezione sequenziale dei bit (seppur accelerata da ottimizzazioni hardware), la complessità computazionale rimane $O(n)$ per algoritmi come first-fit o best-fit.

Si rende dunque necessario mitigare gli effetti della scansione mediante euristiche che restringono l'area di esplorazione a intervalli predefiniti. In questo contesto, il buddy system visto nella sezione precedente riemerge come soluzione particolarmente efficace. Grazie alla sua struttura gerarchica binaria, esso permette infatti di individuare rapidamente i blocchi liberi e le relazioni tra di essi, ottimizzando sia l'allocazione che la deallocazione. In particolare, sfruttando una bitmap associativa, è possibile delimitare con precisione la zona di memoria in cui cercare i blocchi disponibili, migliorando ulteriormente l'efficienza.

Funzionamento del BitmapBuddyAllocator

Poiché il BitmapBuddyAllocator è una variante ottimizzata del classico buddy system, le operazioni che esso svolge sono simili a quelle viste precedentemente: la differenza è nella struttura dati che viene consultata. (la bitmap piuttosto delle liste concatenate). Quando viene richiesta della memoria, l'allocatore cerca nella bitmap un blocco libero della dimensione giusta. Se non lo trova, risale di livello per trovarne uno più grande e lo suddivide in due blocchi (buddy), aggiornando la

bitmap di conseguenza. Un blocco parent che sia scomposto in buddy di cui almeno uno è utilizzato è segnato a sua volta come non adoperabile per esaudire richieste.

Durante la deallocazione, il bit del blocco viene segnato come libero. Se anche il buddy è libero, i due vengono fusi e il blocco originario viene ricostruito, riducendo la frammentazione.

In breve, il BitmapBuddyAllocator unisce la flessibilità del buddy system con la velocità e compattezza delle bitmap, risultando ideale per ambienti ad alte prestazioni.

Efficienza del BitmapBuddyAllocator

Il BitmapBuddyAllocator rappresenta un compromesso ottimale tra efficienza (grazie alle ottimizzazioni bitwise), semplicità (nessuna gestione di strutture complesse come alberi bilanciati) e scalabilità (adatto a sistemi con grandi pool di memoria). Mentre il BuddyAllocator tradizionale rimane una scelta valida in contesti semplici, in quanto di più facile implementazione, il BitmapBuddyAllocator si dimostra superiore in scenari ad alte prestazioni, dove è critico il ruolo del caching. Tuttavia, la frammentazione interna rimane un problema irrisolto, rendendo questo allocatore meno adatto per carichi di lavoro con richieste di memoria estremamente variabili.

Operazione	BitmapBuddyAllocator
Allocazione	$O(1)$ / $O(L)$
Deallocazione	$O(1)$
Ricerca blocco libero	$O(1)$ / $O(L)$
Frammentazione interna	Potenzialmente molto alta
Frammentazione esterna	Generalmente bassa

Capitolo 4

Test e Performance

La letteratura descritta nel secondo capitolo giunge a una conclusione concorde sui benchmark per i gestori di memoria dinamicamente allocata: per valutare un algoritmo di allocazione, è necessario osservarne il comportamento all'interno di un contesto realistico. Ciò può avvenire solamente laddove le tracce adoperate per condurre i benchmark siano vicine alle allocazioni realmente compiute da programmi reali, che sono presi come esempio (esistono utilities che permettono di registrare le richieste, in modo da poterle usare a questo scopo).

Quando le tracce sono casualmente generate, il risultato finale ci dice ben poco sulle capacità effettive dell'allocatore. Le richieste prodotte da un algoritmo probabilistico creano un modello di comportamento, ma questo non è sufficiente: riprodurre le complesse interazioni tra allocazioni e deallocazioni di memoria è molto difficile, poiché queste ultime sono poco comprese e differiscono grandemente tra tipologie di applicazione. Il comportamento a fasi dei programmi dà vita a fenomeni di interconnessione sistematica che sono per la maggior parte ignorati.

Nel 1998, Wilson e Johnston approfondiscono i risultati del procedente paper sull'allocazione dinamica di memoria indagando il comportamento di diversi noti programmi scritti in C e C++. Nell'articolo *The Memory Fragmentation Problem: Solved?* gli autori tentano di dimostrare come la frammentazione può essere evitata laddove sia scelta con attenzione una politica di allocazione appropriata a prescindere dall'implementazione.

> "This substantially strengthens our previous results showing that the memory fragmentation problem has generally been misunderstood, and that good allocator policies can provide good memory usage for most programs. The new results indicate that for most programs, excellent allocator policies are readily available, and efficiency of implementation is the major challenge."

4.1 Test delle funzionalità

I test delle funzionalità si concentrano unicamente sulla correttezza del codice: verificano che le funzioni rispondano correttamente a parametri sbagliati o richieste inappropriate. Definendo la flag DEBUG a tempo di compilazione abbiamo accesso a maggiori informazioni sugli errori e sulle loro cause.

4.1.1 SlabAllocator

Nome del Test	Descrizione
`test_invalid_init`	Verifica che l'allocatore gestisca correttamente parametri di inizializzazione non validi (es. dimensione zero o numero massimo di slab non valido).
`test_create_destroy`	Controlla che la creazione e distruzione di uno slab avvengano correttamente, senza memory leak o errori.
`test_alloc_pattern`	Testa il comportamento dell'allocatore con un pattern di allocazioni e deallocazioni ripetute per verificare la correttezza della gestione della memoria.
`test_exhaustion`	Verifica il comportamento quando lo slab è pieno (es. ritorno di NULL o gestione degli errori quando non c'è più memoria disponibile).
`test_invalid_free`	Controlla come l'allocatore gestisce la deallocazione di puntatori non validi (es. NULL o indirizzi non allocati).

Tabella 4.1. Test funzionali per SlabAllocator

4.1.2 BuddyAllocator e BitmapBuddyAllocator

Nome del Test	Descrizione
`test_invalid_init`	Verifica che l'allocatore gestisca correttamente inizializzazioni non valide (es. dimensione zero, parametri NULL, o valori non supportati).
`test_create_destroy`	Testa la corretta creazione e distruzione di un allocatore, assicurandosi che non ci siano memory leak o corruzione dei dati.
`test_single_allocation`	Verifica che l'allocatore possa rispondere correttamente ad allocazioni invalide e gestire correttamente una singola allocazione.
`test_multiple_allocation`	Controlla il comportamento dell'allocatore quando vengono effettuate più allocazioni consecutive, assicurandosi che tutte abbiano successo e non si sovrappongano.
`test_varied_sizes`	Testa l'allocazione di blocchi di dimensioni diverse per verificare che l'allocatore gestisca correttamente richieste eterogenee.
`test_buddy_merging`	Verifica che, dopo una serie di allocazioni e deallocazioni, l'allocatore riesca a fondere correttamente i blocchi liberi adiacenti (buddy merging) per evitare frammentazione.
`test_invalid_reference`	Controlla come l'allocatore gestisce tentativi di deallocazione di riferimenti non validi (es. NULL, doppio free, o puntatori non allocati).

Tabella 4.2. Test funzionali per BuddyAllocator e BitmapBuddyAllocator

4.2 Benchmark

4.2.1 Timing e Performance

Capitolo 5

Conclusioni

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

Bibliografia

[1] A. Schreiner, *Object-Oriented Programming with ANSI-C*, 1994.

[2] B. Kernighan, D. Ritchie, *The C Programming Language (2nd Edition)*, 1988.

[3] R. Bryant, D. O'Hallaron, *Computer Systems: A Programmer's Perspective (3rd Edition)*, 2015.

[4] D. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition)*, 1997.

[5] D. Lea, *A Memory Allocator (dlmalloc)*, 1987.

[6] J. Bonwick, *The Slab Allocator: An Object-Caching Kernel Memory Allocator*, 1994.

[7] P. Wilson, M. Johnstone, M. Neely, D. Bryant, *Dynamic Storage Allocation: A Survey and Critical Review*, 1995.

[8] P. Wilson, M. Johnstone, *The Memory Fragmentation Problem: Solved?*, 1998.

[9] I. Puaut, *Real-Time Performance of Dynamic Memory Allocation Algorithms*, 2002.

[10] M. Trebi, *Memory Allocators: Implementations and Comparisons*, GitHub Repository, 2020.
`https://github.com/mtrebi/memory-allocators`

[11] E. Berger, *Malloc Implementations: Historical and Technical Analysis*, GitHub Repository, 2018.
`https://github.com/emeryberger/Malloc-Implementations`

Ringraziamenti

ACK HERE