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: 19 giugno 2025

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

Dedicato a...

Sommario

ABS HERE

Ringraziamenti

ACK HERE

Indice

1 Introduzione **3**

2 Lavori Correlati/Basi **4**
2.1 Object Oriented C Programming: perché? 4
2.2 Letteratura scientifica sull'allocazione dinamica 4
2.3 Didattica degli allocatori 6
2.4 Ispirazione per la struttura 7

3 Implementazione **8**
3.1 L'interfaccia Allocator 8
3.2 La classe SlabAllocator 10
3.2.1 Funzionamento dello SlabAllocator 11
3.2.2 Efficienza dello SlabAllocator 12
3.3 La classe BuddyAllocator 13
3.3.1 Funzionamento del BuddyAllocator 14
3.3.2 Efficienza del BuddyAllocator 15

4 Esperimenti, Casi d'Uso **16**

5 Conclusioni **17**

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?

- 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 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

- 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 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

- 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. 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 *Slab Allocator* e del *Buddy Allocator* viste durante le lezioni del corso di Sistemi Operativi tenuto dal professor Grisetti. Tuttavia, la struttura presenta sostanziali differenze che rendono le procedure leggermente diverse, e che 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 del 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 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`).

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 allocatori 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 allocatore 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 dell'allocatore). 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 dell'allocatore 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 allocatore.

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

3.2.1 Funzionamento dello SlabAllocator

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

L'inizializzazione di un'istanza di `SlabAllocator` richiede la grandezza della slab (nei termini di Bonwick, la grandezza dell'oggetto da immagazzinare) e il numero delle stesse. La memoria richiesta viene suddivisa in blocchi. Essi sono poi organizzati in una linked list, che mantiene un pratico riferimento ai blocchi disponibili e la cui lunghezza massima è pari al numero totale di blocchi.

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 dell'allocatore, 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, viene semplicemente inserito al primo posto della lista per uso futuro. La memoria non viene reimpostata né durante la richiesta né al rilascio.

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 li rende manipolabili da

parte dell'utente, che può inavvertitamente o con intenzioni maligne corromperli scrivendo sopra di essi: in questo modo, la funzione di free non funzionerebbe più. Tuttavia, questa scelta implementativa ricalca quella che è stata adottata nella `libc` con `malloc`.

3.2.2 Efficienza dello SlabAllocator

Descriviamo ora più nel dettaglio la complessità computazionale delle operazioni compiute dall'allocatore. 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, l'allocatore slab 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 dell'allocatore la dimensione del blocco più appropriata per i propri scopi.

Operazione	Slab Allocator
Allocazione	$O(1)$
Deallocazione	$O(1)$
Ricerca blocco libero	$O(1)$
Frammentazione interna	?
Frammentazione esterna	Nulla

L'efficienza dell'allocatore slab 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 allocatori 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 ad allocatori generici (come buddy system o malloc tradizionale), l'allocatore slab 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 ricorsivamente 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. La differenza consiste 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.

3.3.1 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, lo stato di libero/occupato, e puntatori al buddy e al parent. La scelta di memorizzare esplicitamente queste relazioni, anziché calcolarle dinamicamente, semplifica il debug e la visualizzazione dello stato dell'allocatore, a scapito di un leggero overhead in memoria. Infatti conoscendo la taglia del blocco e l'indirizzo di partenza, potremmo raggiungere l'header del buddy senza bisogno di immagazzinare questa informazione nell'header. Ciononostante, per facilitare la visualizzazione dell'occupazione di memoria e dello stato dell'allocatore, si è ritenuto di salvare questa informazione, così come anche l'indirizzo del parent.

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. Questa scelta massimizza l'efficienza, poiché le dimensioni degli oggetti allocati sono fisse e note a priori. La memoria gestita dall'allocatore è riservata tramite `mmap`, garantendo allineamento e flessibilità nella gestione di grandi aree di memoria.

3.3.2 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.

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.