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: 30 giugno 2025

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

Dedicato a chi rimane curioso per tutta la vita

Sommario

L'allocazione di memoria

Indice

Capitolo 1

Introduzione

La gestione dinamica della memoria è una delle principali responsabilità dei sistemi operativi moderni[1]. La memoria ospita i processi attivi e i dati correntemente elaborati dal *software* in esecuzione; in quanto essa è più veloce da accedere dei dispositivi di memoria di massa, spesso viene usata come intermediario nella comunicazione che essi hanno con i processi. Con l'avvento dei sistemi multiprogrammati, la suddivisione della memoria in partizioni dedicate a ogni *task* è diventata un aspetto principale dell'attività dei sistemi operativi: partizionare in aree contingentate la memoria in modo veloce ed efficiente (senza perdere tempo o sprecare memoria) è un compito sfidante.

Il sistema operativo si deve occupare di gestire e monitorare lo stato di ciascuna locazione di memoria fisica, regolando l'allocazione della memoria tra i processi concorrenti, e definire le politiche di assegnazione, stabilendo quali processi possano accedere alla memoria, i tempi di allocazione e la quantità di memoria disponibile per ciascuno. Durante l'allocazione, il sistema operativo determina le specifiche locazioni di memoria da assegnare e ne mantiene traccia, aggiornandone lo stato in caso di rilascio o deallocazione. Un ulteriore aspetto che richiede attenzione è la memoria di cui il sistema operativo stesso ha necessità per svolgere le sue funzioni; poiché esso viene richiamato numerose volte dai programmi in esecuzione per svolgere molteplici compiti, laddove le strutture in atto per gestire le sue necessità di memoria siano lente o abbiamo un grande sovracosto ciò potrebbe portare a risultati disastrosi per la generale fluidità del calcolatore.

Quando la grandezza del programma è nota alla compilazione e non cambia, è semplice segnalare al sistema operativo quanta memoria sarà necessaria per tutto il ciclo di vita del programma. Questa memoria prende il nome di **staticamente allocata**. L'incarico di gestione risulta dunque semplificato: non sempre però è possibile determinare a priori le necessità del programma, in quanto queste potrebbero dipendere da vari fattori che non sono noti al programmatore (e.g. *user input*). Un primo approccio, dispendioso e generalmente da evitare, consiste nell'allocare staticamente la memoria necessaria nel *worst case scenario*. In questo modo però il sistema operativo potrebbe esaurire la memoria disponibile quando invece all'interno dei programmi esista memoria non utilizzata[2]. In generale questa pratica risulta

[1] Dove per memoria si intende la *Random Access Memory*, o RAM.

impossibile da applicare in contesti dove la quantità di memoria non è abbondante.

L'alternativa risulta immediata, ma non di facile implementazione: fornire al programmatore sistemi per **allocare dinamicamente** la memoria, ossia per variare la grandezza dell'area di memoria dedicata ai dati del programma durante la sua vita, ingrandendola e restringendola in base alle necessità. In questo modo, memoria viene occupata solamente quando è necessaria e nel momento in cui non è più utile viene restituita al sistema operativo, che può assegnarla a un altro programma. L'allocazione della memoria consiste dunque nell'identificare un blocco di memoria libera di dimensione adeguata per soddisfare una richiesta. Le allocazioni di memoria vengono gestite attingendo da un'area contigua denominata *heap* (o *free store*).

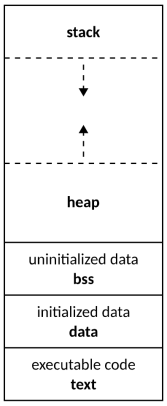

Figura 1.1. Layout semplificato della memoria di un programma.

1.1 *Memory Allocators*

In un dato istante, alcune regioni dell'*heap* risultano allocate (in uso da parte di processi o strutture dati), mentre altre rimangono libere (non allocate) e pertanto disponibili per future richieste di memoria. Tale funzionalità è implementata da un modulo specializzato del sistema operativo, noto come *Memory Allocator* (allocatore di memoria), il quale sovrintende all'assegnazione e al rilascio delle risorse di memoria. All'inizializzazione dell'allocatore viene riservata per il suo uso una quantità di memoria; esso si occupa dunque di gestirla dinamicamente, rispondendo alle necessità del programma e restituendo la memoria inutilizzata al sistema operativo. Ciò può avvenire secondo strategie e politiche ben diverse: obiettivo di questo progetto è l'esplorazione di un sottoinsieme rappresentativo di questi moduli, caratterizzati da un campione delle suddette politiche di allocazione, evidenziando nelle loro implementazioni i benefici e i punti deboli.

Linguaggi di programmazione diversi gestiscono questa problematica in due modi: l'allocatore può funzionare in modo manuale, ossia rispondendo alla chiamata esplicita di funzioni che comunicano le necessità del programma, o in modo automatico, attraverso *garbage collectors*. Questi ultimi consistono in un insieme di *routine*

[2]Out of memory (OOM) error

che determinano se esiste memoria il cui riferimento viene perso all'interno del programma e che dunque sono irraggiungibili e li aggiungono alla memoria disponibile. Esistono diversi meccanismi per effettuare questa analisi, ma nel corso della nostra analisi ci concentreremo sull'allocazione manuale di memoria, in particolare nelle modalità in cui avviene nel linguaggio di programmazione **C**.

La libC espone come interfaccia due primitive per l'allocazione di memoria: *malloc* e *free*. L'implementazione è lasciata al sistema operativo, che può decidere autonomamente quale tipologia di allocatore implementare, oppure addirittura di diversificare le strategie in base alle proprietà spaziali e temporali delle allocazioni.

Capitolo 2

Riferimenti, influenze e fonti

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 [1]. 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. [8] 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 [10] 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."

(Wilson, *Dynamic Storage Allocation, A Survey and Critical Review* [8], ch. 1.1)

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 [2] e *Computer Systems – A Programmer's Perspective* (capitolo 9.9, "Dynamic Memory Allocation") di R. Bryant [3]. Illuminante è stato il capitolo *Dynamic Storage Allocation* del volume primo di *The Art of Computer Programming*, di D. Knuth [4]. 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 [5]. 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** [12] e di Emery Berger, professore presso l'Università di Massachusetts Amherst [13] 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.

Capitolo 3

Sviluppo

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`).

```
1 // Define function pointer types
2 typedef void* (*InitFunc)(Allocator*, ...);
3 typedef void* (*DestructorFunc)(Allocator*, ...);
4 typedef void* (*MallocFunc)(Allocator*, ...);
5 typedef void* (*FreeFunc)(Allocator*, ...);
```

```
6
7  // Allocator structure
8  struct Allocator {
9      InitFunc init;
10     DestructorFunc dest;
11     MallocFunc malloc;
12     FreeFunc free;
13 };
```

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.

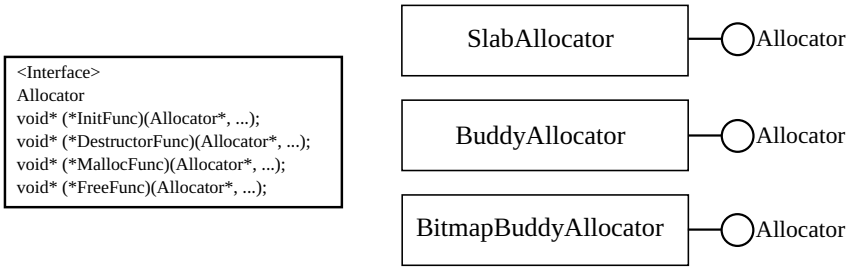

Figura 3.1. Diagramma UML dell'interfaccia Allocator e delle sue implementazioni.

Le funzioni di cui manteniamo un riferimento all'interno delle *struct* sono progettate per uso interno: infatti, gli argomenti sono passati attraverso modalità definite dalla libreria di sistema `<stdarg.h>`. In questo modo, possiamo mantenere i puntatori alle funzioni nonostante per le varie implementazioni della classe essi abbiamo *signature* diverse, dando grande libertà e flessibilità. Contemporaneamente però questa pratica costituisce un rischio, poiché le verifiche sulla correttezza del tipo e del numero degli argomenti 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 (per questo motivo, è importante che queste funzioni helper siano brevi e concise, in modo da evitare *code bloat*).

Una nota sulla keyword *inline*: è importante ricordare che essa è 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 necessario. Ulteriori test po-

trebbero 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:

- `[AllocatorClass]_create` (wrapper di `[AllocatorClass]_init`)
- `[AllocatorClass]_destroy` (wrapper di `[AllocatorClass]_dest`)
- `[AllocatorClass]_malloc` (wrapper di `[AllocatorClass]_reserve`)
- `[AllocatorClass]_free` (wrapper di `[AllocatorClass]_release`)

Il linker del linguaggio C non permette l'esistenza di funzioni non dichiarate come *static* con nome uguale, ma argomenti diversi (*polimorfismo*). Siamo dunque costretti ad anteporre la classe alla funzione per distinguerle, come vediamo sopra. Sono state esplorate soluzioni a questo problema, ma sfortunatamente introducevano livelli di complessità oppure sacrificavano a livello di *type checking*, come ad esempio avviene con l'utilizzo di *macro*, soluzione comunemente adottata. Grazie alla duplice struttura con funzioni helper e internal sarebbe possibile immagazzinare la specifica classe dell'istanza all'inizializzazione e in base ad essa adoperare la corretta funzione: tuttavia, ciò avrebbe comportato numerose complicazioni nella semplicità e facilità di interpretazione da parte dell'utente.

3.1.1 Richiesta iniziale di memoria

Tutte le classi che implementano l'interfaccia `Allocator` usano *mmap* per chiedere memoria 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 e, dal punto di vista didattico, avrebbe costituito un'importante opportunità per studiare come avveniva l'allocazione di memoria in versioni precedenti della *sbrk*. 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 o si fosse deciso di permettere l'utilizzo *multithreaded* [1].

[1] Al netto di possibili complicazioni impreviste, il supporto per l'utilizzo da parte di più entità potrebbe essere aggiunto con relativa facilità adoperando *mutex* per contingentare 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 fa riferimento a un file [2].

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 adatto quando sono necessarie unicamente 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. Esso è dunque particolarmente efficiente al costo di flessibilità ridotta.

```
struct SlabAllocator {
    Allocator base;
    char* memory_start;
    uint memory_size;
    size_t slab_size;
    size_t user_size;
    DoubleLinkedList* free_list;
    uint free_list_size;
    uint num_slabs;
};
```

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"* [6] 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."

[2] Chiaramente a meno che non sia stata posta in un file di swap

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, che gestiscono spesso 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 comprensione 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 i metodi helper sotto delineati, la cui funzione è puramente quella di "filtro" a tempo di compilazione dei parametri e di gestire in maniera appropriata il valore di ritorno delle funzioni interne

```
1 inline SlabAllocator* SlabAllocator_create(SlabAllocator* a,
     size_t slab_size, size_t n_slabs);
2 inline int SlabAllocator_destroy(SlabAllocator* a);
3 inline void* SlabAllocator_malloc(SlabAllocator* a);
4 inline int SlabAllocator_free(SlabAllocator* a, void* ptr);
```

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*.

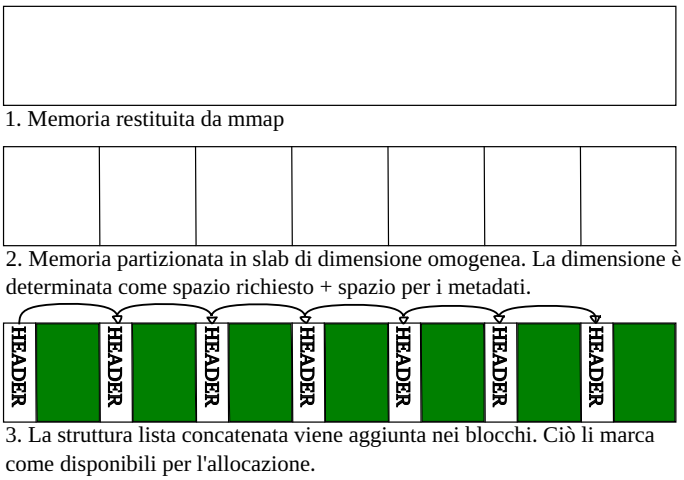

Figura 3.2. Inizializzazione dello SlabAllocator: suddivisione della memoria in blocchi di dimensione fissa e organizzazione in una lista concatenata.

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

Analizziamo più nel dettaglio la complessità computazionale delle operazioni svolte 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. Essi non sono in alcun modo manipolati a seguito di richieste: la loro grandezza rimane costante e questo elimina completamente i costi legati alle operazioni di divisione e unione.

La struttura dello *slab allocator* garantisce che, se è presente almeno uno slab di memoria disponibile, la richiesta dell'utente potrà essere soddisfatta. Tutti i blocchi hanno la stessa dimensione, che viene scelta dal programmatore in fase di inizializzazione, in base alle esigenze specifiche dell'applicazione (ad esempio, la grandezza di una istanza di un oggetto che vi deve essere immagazzinato).

Operazione	SlabAllocator
Allocazione	$O(1)$
Deallocazione	$O(1)$
Ricerca blocco libero	$O(1)$
Gestione dello spazio	Dipende dalla scelta iniziale della dimensione dei blocchi

L'efficienza dello *slab allocator* è quindi strettamente legata alla 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 discrepanza tra la dimensione richiesta e quella dello slab più adatto può essere controllata dal programmatore, che può scegliere la configurazione più efficiente per il proprio caso d'uso.

Lo *slab allocator* risulta dunque particolarmente efficace e parsimonioso. Tuttavia, possono verificarsi situazioni in cui slab liberi e inutilizzati occupino memoria inutilmente, ad esempio quando il numero di *chunk* supera quello degli oggetti effettivamente allocati. Le esigenze del programma possono infatti variare nel tempo: talvolta il numero massimo di oggetti che esistono concorrentemente può variare drasticamente. Per ridurre l'occupazione di memoria in questi casi, alcune implementazioni prevedono meccanismi di rilascio (*reclaiming*) degli slab inutilizzati: dopo un periodo di inattività o sotto pressione per memoria, una parte degli slab può essere restituita al sistema operativo. Questa operazione, che comporta un costo aggiuntivo (tipicamente $O(n)$ rispetto al numero di slab da gestire), offre però maggiore flessibilità nell'adattare l'uso della memoria alle necessità correnti. Nel nostro caso, tale funzionalità non è stata implementata.

Rispetto ad altri gestori più generici (come il *buddy system*, esplorati successivamente, o le implementazioni tradizionali di *malloc*), lo *slab allocator* eccelle in rapidità delle operazioni e prevedibilità nell'uso della memoria, risultando particolarmente adatto a sistemi con risorse dedicate e pattern di allocazione ben definiti. La quantità di memoria utilizzata dipende dal numero di slab preallocati, rendendo questa soluzione ideale quando le richieste sono omogenee e pianificabili.

3.3 La classe astratta VariableBlockAllocator

Il problema dell'allocazione di memoria per richieste di dimensioni variabili rimane un tema aperto e ampiamente discusso. Diversi approcci alla sua soluzione sono stati discussi nel tempo, suscitando dibattiti e proposte contrastanti. Sono state

sviluppate numerose alternative, 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 un'unica lista concatenata, 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). Se la dimensione del blocco selezionato è maggiore della quantità di memoria necessaria, esso viene partizionato e la memoria in eccesso diventa a sua volta un blocco disponibile.

Adoperare queste o altre politiche di scelta del blocco tuttavia può comportare la possibilità concreta che, alla richiesta dell'utente, per via della suddivisione in atto dovuta alle allocazioni precedenti, non vi siano blocchi di memoria **contigua** disponibili per soddisfarla, anche se la quantità di memoria frammentata (o *sparse*) potrebbe esaudire la richiesta. Questa problematica prende il nome di **frammentazione esterna**.

```
1 struct VariableBlockAllocator {
2     Allocator base;
3     size_t internal_fragmentation;
4     size_t sparse_free_memory;
5 };
```

La classe astratta *VariableBlockAllocator* implementa l'interfaccia *Allocator*, a cui aggiunge anche campi per monitorare la frammentazione esterna (`size_t sparse_free_memory`) e interna (`size_t internal_fragmentation`), il cui significato sarà esplicitato nella sezione successiva. Quando viene allocata memoria, viene aggiornata la quantità di byte liberi gestiti dall'allocatore in blocchi separati. Se una richiesta non può essere soddisfatta dall'allocatore, vengono controllati questi valori per stabilire se la frammentazione abbia causato il fallimento.

L'implementazione dei *"sequential fits"*, famosamente esplorata da Knuth, presenta importanti limitazioni. 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*[3] e una debilitante frammentazione esterna, l'inefficienza della scansione lineare è il fattore che principalmente ne impedisce l'applicazione 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 della lista: investire nell'organizzazione maggiore spazio, gestendo i blocchi liberi più efficientemente, permette di velocizzare la ricerca del blocco corretto. La memoria disponibile viene suddivisa sempre in blocchi liberi, che sono però raccolti in base alla loro taglia in liste diversificate. La struttura è semplice e simile a quella del metodo visto precedentemente, tuttavia grazie alla lunghezza minore delle singolari liste, esse sono più rapidamente esplorabili.

[3] Knuth stesso ne ha proposto uno che successivamente è diventato molto popolare: il concetto di *boundary tag*, ossia salvare le informazioni al termine dell'area allocata per facilitare la riunione dei blocchi.

Al momento della richiesta, è esaminata la lista contenente i blocchi della taglia più appropriata, e laddove non vi sia un blocco adeguato vengono ricorsivamente controllate le liste di livello "superiore", contenente blocchi di dimensione maggiore. Il blocco eventualmente individuato è suddiviso e la memoria in eccesso è organizzata in un nuovo *chunk* libero, riposto nella lista corretta secondo la sua grandezza. Questo meccanismo viene chiamato nell'articolo di Wilson et al. *"segregated free lists"*.

3.4 La classe BuddyAllocator

Il *Buddy Allocator* è descritto nella stessa pubblicazione come un "caso particolare" di quest'ultima tipologia di allocatori. Inventato da Harry Markowitz nel 1963 e pubblicato per la prima volta nell'articolo *"A Fast Storage Allocator"* [7] 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 generalmente nota, che vengono ripetute numerose volte.

La differenza rispetto agli algoritmi che lo precedono consiste principalmente nelle politiche di *splitting* e *coalescing*. Mentre le metodologie viste finora non stabiliscono esplicitamente regole che l'allocatore debba seguire nel dividere i blocchi liberi per soddisfare le richieste, i *buddy systems* invece stabiliscono una chiara gerarchia che rende il procedimento più ordinato. I blocchi infatti, come approfondiremo successivamente, sono partizionati sempre in metà uguali e riuniti solo quando entrambe le metà sono contemporaneamente libere.

Questa differenza consente di evitare un problema significativo che emerge quando la dimensione dei blocchi non è vincolata. In particolare, modelli di allocazione tipici, come l'alternanza di richieste e rilasci di blocchi di dimensioni diverse, causano frammentazione esterna negli allocatori che adottano metodi come i *sequential fits*. La libertà nella gestione delle dimensioni dei blocchi unita alla ricerca lineare porta alla formazione di numerose aree libere sparse. 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 *Buddy Allocator* rappresentano una possibile soluzione, ma non senza introdurre alcuni compromessi.

Funzionamento del BuddyAllocator

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*. Le informazioni su di essi potrebbero essere raggiunte, note la taglia del blocco e l'indirizzo di partenza, senza bisogno di immagazzinarle esplicitamente nell'header: tuttavia, la scelta di memorizzare queste relazioni, anziché calcolarle dinamicamente attraverso manipolazione degli indirizzi di memoria, semplifica il debug e la visualizzazione dello stato dell'allocatore.

```
1 typedef struct BuddyNode {
2     Node node;
3     char *data;
4     size_t size; // Size of this block (including header)
```

```
    size_t requested_size; // Requested size (for logging)
    int level; // Level in the buddy system
    int is_free; // Whether this block is free
    struct BuddyNode* buddy; // Pointer to buddy block
    struct BuddyNode* parent; // Pointer to parent block
} BuddyNode;
```

I nodi sono salvati in una serie di free lists, corrispondenti ai vari livelli di un albero binario. 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.

```
typedef struct BuddyAllocator {
    VariableBlockAllocator base; // Base allocator interface
    void* memory_start; // Start of managed memory
    size_t memory_size; // Total size of managed memory
    size_t min_block_size; // Minimum block size (power of 2)
    int num_levels; // Number of levels in the system
    SlabAllocator list_allocator;
    SlabAllocator node_allocator;
    DoubleLinkedList** free_lists;  // Array of free lists
       for each level (in mmap)
} BuddyAllocator;
```

Quando è necessario partizionare un blocco 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. Sia l'operazione di divisione dei blocchi che quella di ricongiungimento sono svolte in modo ricorsivo, esplorando tutti i livelli dell'albero fino a che la richiesta non sia stata esaudita o sia stato accertato che non vi siano blocchi liberi per farlo.

```
// inline has been omitted for brevity
BuddyAllocator* BuddyAllocator_create(BuddyAllocator* a,
   size_t memory_size, int num_levels);
int BuddyAllocator_destroy(BuddyAllocator* a);
void* BuddyAllocator_malloc(BuddyAllocator* a, size_t size);
int BuddyAllocator_free(BuddyAllocator* a, void* ptr);
```

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

Efficienza del BuddyAllocator

L'operazione di allocazione esplora per prima la lista libera del livello più appropriato (*best fit*). 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 talvolta complesse ad albero, semplificando l'implementazione e riducendo il costo. 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.

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 criticità. 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. La rigidità del sistema lo rende dunque meno adatto a gestire pattern di allocazione estremamente variabili o imprevedibili. La differenza tra la memoria richiesta e quella ottenuta viene registrata nel campo `size_t internal_fragmentation`.

Ad esempio, supponiamo di avere blocchi disponibili di dimensione A, B, C e D, dove ogni lettera rappresenta una potenza di due crescente (ad esempio, $A = 8$, $B = 16$, $C = 32$, $D = 64$). Se viene richiesta una quantità di memoria pari a $B + 1$, il sistema non potrà allocare un blocco di dimensione B, ma dovrà assegnare il blocco successivo più grande, ovvero C. In questo modo, una richiesta di poco superiore a B comporta l'allocazione di un blocco di dimensione doppia, lasciando inutilizzata una parte significativa della memoria allocata.

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

3.5 La classe BitmapBuddyAllocator

Nelle implementazioni analizzate finora, la ricerca di un blocco libero avviene 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. 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 costo 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*: questa struttura dati, nota anche come *bit array* o *bit field*, permette di immagazzinare informazioni in modo denso e compatto. Nella gestione del *BitmapBuddyAllocator*, è fatto uso di una *bitmap* dove ogni bit rappresenta lo stato, libero o occupato, del corrispondente 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; tuttavia la problematica della scansione lineare della *bitmap* ricalca la criticità dei metodi *sequential* e *segregated fit*.

Si rende dunque necessario applicare 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

La memoria necessaria per gestire la bitmap viene inserita all'interno dell'area assegnata all'allocatore dall'operazione di *mmap*.

```
typedef struct {
    VariableBlockAllocator base;
    char* memory_start; // Managed memory area
    int memory_size; // Size of managed memory
    int num_levels; // Number of levels in the hierarchy
    int min_block_size; // Minimum allocation size
    Bitmap bitmap; // Bitmap tracking block status
} BitmapBuddyAllocator;
```

Poiché il *BitmapBuddyAllocator* è una variante 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 che le liste concatenate). Quando viene richiesta della memoria, l'allocatore cerca in `bitmap` un blocco libero della dimensione giusta. Se non lo trova, risale di livello per trovarne uno più grande e lo suddivide nei due *buddy* di dimensione uguale, 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, in quanto parte di esso è allocata.

```
// inline has been omitted for brevity
BitmapBuddyAllocator*
    BitmapBuddyAllocator_create(BitmapBuddyAllocator* a,
    size_t memory_size, int num_levels);
int BitmapBuddyAllocator_destroy(BitmapBuddyAllocator* a);
void* BitmapBuddyAllocator_malloc(BitmapBuddyAllocator* a,
    size_t size);
int BitmapBuddyAllocator_free(BitmapBuddyAllocator* a, void*
    ptr);
```

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.

```
static int levelIdx(size_t idx) {
    return (int)floor(log2(idx+1));
}

static int buddyIdx(int idx) {
    if (idx == 0) return -1;
    // formula: even goes to idx - 1, odd goes to idx + 1
    if (idx % 2 == 0) {
        return idx - 1; // even index, buddy is left
    } else {
        return idx + 1; // odd index, buddy is right
    }
}

static int parentIdx(int idx) {
    if(idx == 0 ) return -1;
    return (idx - 1) / 2;
}

static int firstIdx(int level) {
    return (1 << level) - 1; // 2^(level-1)
}
```

Efficienza del BitmapBuddyAllocator

Il *BitmapBuddyAllocator* rappresenta un compromesso ottimale tra efficienza (grazie alle ottimizzazioni *bitwise*), semplicità (nessuna gestione di strutture complesse come *self-balancing trees*) e scalabilità (adatto a sistemi con grandi *memory pool*). 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*.

Questo metodo consente una ricerca estremamente rapida di blocchi contigui liberi, migliorando significativamente la località spaziale e riducendo i problemi di

caching tipici delle *linked list*. 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*. Tuttavia, la *internal fragmentation* 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?* [9] 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."
>
> [...]
>
> "If these results hold up to further study with additional programs, we arrive at the conclusion that the fragmentation problem is a problem of recognizing that good allocation policies already exist, and have inexpensive implementations. For most programs, the problem of simple overheads is more significant than the problem of fragmentation itself."

Il programma è stato sottoposto anche ad analisi attraverso *valgrind*, il popolare programma di *debugging* e *memory analysis*, in particolare adoperando i *tool memcheck, massif* e *cachegrind*. Questi strumenti sono stati fondamentali per comprendere in primo luogo se il programma accedesse alla memoria come previsto, e secondariamente per comprendere dove si trovassero le ostruzioni e i punti critici che causavano perdita di efficienza.

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

I test per lo SlabAllocator sono generalmente semplici in natura: poiché la grandezza della richiesta non varia e la lista da gestire è singola, gli unici punti di difficoltà sono la creazione con parametri invalidi o il rilascio di puntatori incorretti.

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

Gli allocatori che accettano richieste a taglia variabile presentano una complessità maggiore rispetto a quelli a taglia fissa, poiché devono gestire dinamicamente la suddivisione e la fusione dei blocchi di memoria per rispondere a richieste di dimensioni differenti. Questo comporta una maggiore probabilità di introdurre errori nella gestione della memoria come la perdita di riferimenti o la mancata fusione dei blocchi liberi (*buddy merging*). I test svolgono un ruolo fondamentale nell'evidenziare anomalie e comportamenti indesiderati

Oltre ai test elencati, è utile prevedere casi limite e scenari di stress, come sequenze di allocazioni e deallocazioni ripetute con dimensioni variabili, tentativi di allocazione che saturano la memoria disponibile, e la verifica della corretta gestione di errori (ad esempio, *double free* o allocazioni fuori dai limiti consentiti).

4.2 Benchmark

Poiché la nostra analisi si concentra principalmente sulla correttezza dell'implementazione, i test sopra delineati forniscono una prima conferma del nostro operato. Tuttavia, l'importanza della *performance* dell'allocatore in diversi casi applicativi

risulta essenziale per poter veramente trarre delle conclusioni a riguardo. Nel corso delle ricerche, l'articolo *"Designing a Trace Format for Heap Allocation Events"* [11] di T.Chilimbi et al ha fornito un'interessante base teorica per il parser qui descritto. Tuttavia, il nostro vuole essere unicamente un prototipo di funzionamento al fine di svolgere analisi più profonde, e sicuramente fornisce ben poche delle capacità descritte nell'articolo o altrimenti rese disponibili da altri strumenti simili applicati nel campo.[1]

Per facilitare dunque l'analisi del comportamento degli allocatori in presenza di pattern complessi di allocazione e deallocazione, è stata introdotta una funzionalità che consente di definire facilmente le sequenze di richieste di memoria tramite file di configurazione esterni, senza dover modificare o ricompilare il codice sorgente. In questo modo è possibile variare rapidamente i benchmark e testare diversi scenari. Il codice sorgente di questa funzionalità si trova nei moduli `benchmark` e `parse`.

Ciò è reso possibile dall'implementazione da parte di ogni allocatore dell'interfaccia `<Allocator>`. Poiché abbiamo a nostra disposizione i puntatori alle funzioni necessarie per svolgere tutte le operazioni, possiamo standardizzare il funzionamento delle classi dividendole in due categorie: allocatori a taglia fissa, che permettono di richiedere blocchi di grandezza prestabilita (e.g. `SlabAllocator`) e allocatori a taglia variabile, ossia le classi "figlie" di `VariableBlockAllocator`, le quali invece lasciano all'utente la scelta della dimensione dell'area di memoria richiesta (ossia `BuddyAllocator` e `BitmapBuddyAllocator`, nella nostra implementazione).

```
1  // template per allocatore a dimensione fissa
2  void *FixedSizeAllocator_malloc(FixedSizeAllocator *a);
3  void *FixedSizeAllocator_free(FixedSizeAllocator *a);
4  // template per allocatore a dimensione variabile
5  void *VariableSizeAllocator_malloc(VariableSizeAllocator
       *a, size_t size);
6  void *VariableSizeAllocator_free(VariableSizeAllocator *a)
```

Il programma cerca all'interno della cartella `./benchmarks` file che abbiano l'estensione `.alloc`. Modificando il testo al loro interno si possono definire la tipologia di allocatore, i suoi parametri di inizializzazione e la sequenza di `malloc`/`free` da provare. I comandi sono terminati dal carattere "a capo" (\n) e le componenti sono divise da una virgola. Comandi che sono preceduti dal carattere percentuale (%) sono considerati commenti e ignorati.

Vediamo ora come descrivere la classe di allocatore da sottoporre al benchmark e le sue caratteristiche. La prima riga non preceduta da un segno percentuale deve avere necessariamente la seguente struttura: `i,<allocator class>` dove la classe può essere `slab`, `buddy` o `bitmap`. Il secondo comando informa il programma sui parametri di creazione: `p,<param1>,<param2>,...`, il numero e il tipo dei quali varia in base alla classe scelta precedentemente.

[1] Alcuni di questi che menzioniamo per dovere di cronaca sono `perf`, `LTTng` e `ETW`.

Allocatore	Dimensione	Parametro 1	Parametro 2
Slab	Fissa	`slab_size`	`num_slabs`
Buddy	Variabile	`memory_size`	`max_levels`
Bitmap	Variabile	`memory_size`	`max_levels`

Tabella 4.3. Parametri di inizializzazione per ciascuna classe di allocatore

Per allocare e liberare memoria, l'istruzione deve iniziare rispettivamente con "a" o "f". Il benchmark per tenere traccia delle allocazioni usa un array avente lunghezza pari al numero massimo possibile di frammenti di memoria. Il comando `a,<index>` alloca memoria in una posizione specifica dell'array, mentre `f,<index>` la libera. Nel caso di allocatore a richiesta variabile, dopo l'index dell'array va specificato il numero di byte da allocare (con struttura `a,<index>,<size>`).

È fondamentale non allocare più volte sullo stesso indice senza prima liberarlo, altrimenti si perderà il riferimento all'area di memoria e non sarà più possibile gestirla correttamente[2]. Il programma esegue la sequenza di *request* e *release*, informa l'utente di eventuali errori e stampa a schermo informazioni sul tempo richiesto: `elapsed_seconds`, `user_seconds` e `kernel_seconds`. In più, produce un file di `.log` che contiene informazioni sulle singole istruzioni (in particolare, per i `VariableBlockAllocator`, inserisce anche lo stato della frammentazione interna ed esterna dell'allocatore a ogni istruzione). L'analisi di queste informazioni fornisce alcune metriche aggiuntive sul comportamento dell'allocatore e sulla sua *performance*.

4.2.1 Analisi dei pattern di allocazione

La letteratura individua tre principali pattern di utilizzo della memoria che ricorrono nella maggior parte dei programmi: rampe, picchi e plateau[3]. In particolare, Wilson et al. evidenziano come sia spesso possibile riconoscere, nel ciclo di vita di un programma, macrosequenze di allocazione e deallocazione che seguono questi modelli: inoltre, all'interno di tali pattern principali, possono emergere sottostrutture più piccole che ripetono dinamiche simili, riflettendo la complessità e la varietà dei comportamenti reali di gestione della memoria.

- **Rampe**: molti programmi accumulano determinate strutture dati in modo monotono. Ciò può essere dovuto al fatto che conservano un registro degli eventi o perché la strategia di risoluzione del problema richiede la costruzione di una rappresentazione di grandi dimensioni, dopo la quale è possibile trovare rapidamente una soluzione.
  Esempio: parser di log che accumulano eventi in memoria, compilatori che costruiscono un AST, o applicazioni che caricano progressivamente dati da una sorgente esterna.

[2]Il programma di benchmarking non esegue istruzioni che porterebbero alla perdita di un indirizzo e avverte se il benchmark termina senza deallocare tutti gli indirizzi.

[3]Altri *pattern* dell'uso di memoria possono essere incontrati, ma sono meno comuni: allo stesso modo, variazioni o combinazioni dei modelli qui citati sono spesso presenti.

- **Picchi**: molti programmi utilizzano la memoria in modo discontinuo, creando strutture dati relativamente grandi che vengono utilizzate per la durata di una particolare fase e poi la maggior parte o tutte le strutture dati viene scartata. Si noti che le strutture di dati "sopravvissute" sono probabilmente di tipo diverso, perché rappresentano i risultati di una fase, rispetto a valori intermedi che possono essere rappresentati in modo diverso. (Un picco è come una rampa, ma di durata più breve).
  Esempio: algoritmi di sorting che allocano buffer temporanei, elaborazione di immagini o video dove ogni frame richiede molta memoria temporanea, oppure fasi di calcolo numerico in cui vengono creati e poi distrutti grandi array temporanei.

- **Plateau**: molti programmi costruiscono rapidamente strutture dati e poi le utilizzano per lunghi periodi (spesso per quasi tutto il tempo di esecuzione del programma).
  Esempio: server web che mantengono strutture dati per la gestione delle connessioni attive, database in-memory che caricano dati all'avvio e li mantengono residenti, o giochi che caricano la mappa di gioco all'inizio e la usano per tutta la sessione.

Utilizzando lo strumento da noi creato, siamo in grado di simulare questi pattern e verificare il comportamento degli allocatori in loro presenza. Nel considerare i risultati ottenuti ricordiamo tuttavia che l'applicazione del *profiling* comporta ovviamente un costo non trascurabile: i risultati successivamente riportati sono chiaramente influenzati dalle operazioni di *logging* svolte dal programma. Sono state fatte numerose scelte nel tentativo di evitare costi eccessivi, ma sfortunatamente è impossibile analizzare gli allocatori così a fondo senza introdurre una misura di *overhead*.

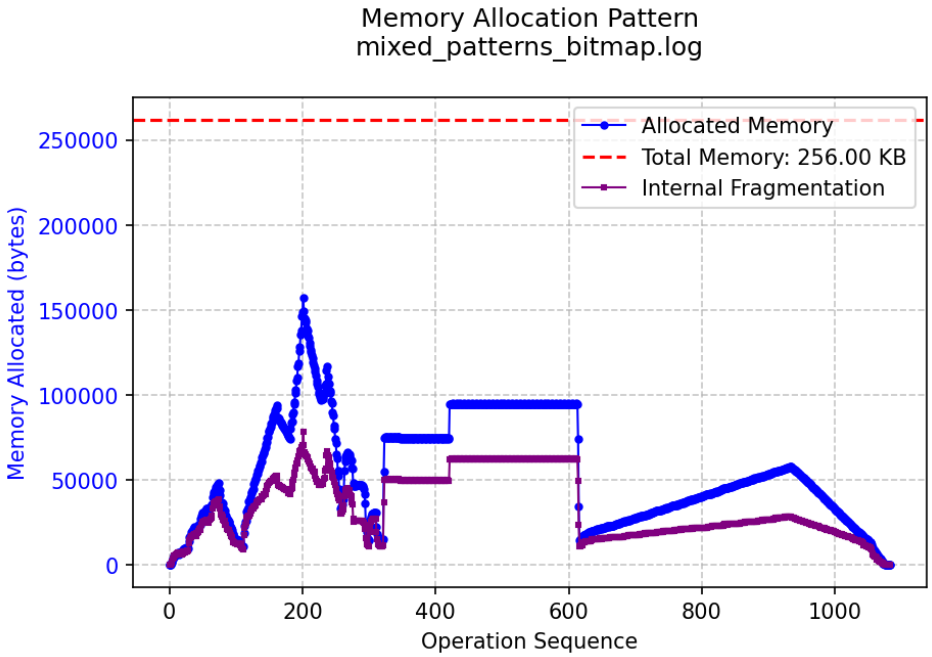

Figura 4.1. Comportamento del BitmapBuddyAllocator su un benchmark contenente in ordine, *peaks, plateaus* e *ramps*.

4.2.2 Allocazioni sfavorevoli: frammentazione interna

Come abbiamo menzionato precedentemente nella nostra analisi dell'implementazione degli allocatori che implementano un *buddy system*, la possibilità di frammentazione interna molto elevata al punto da essere debilitante non è da trascurarsi. Adoperando lo strumento da noi creato per simulare un caso estremo, possiamo facilmente osservare dai grafici come determinate combinazioni di parametri di inizializzazione e richieste mal formulate possano rendere l'allocatore molto inefficace nel gestire la memoria a sua disposizione. Le linee rosse sul grafico rappresentano richieste che non è stato possibile soddisfare per via di frammentazione interna o esterna.

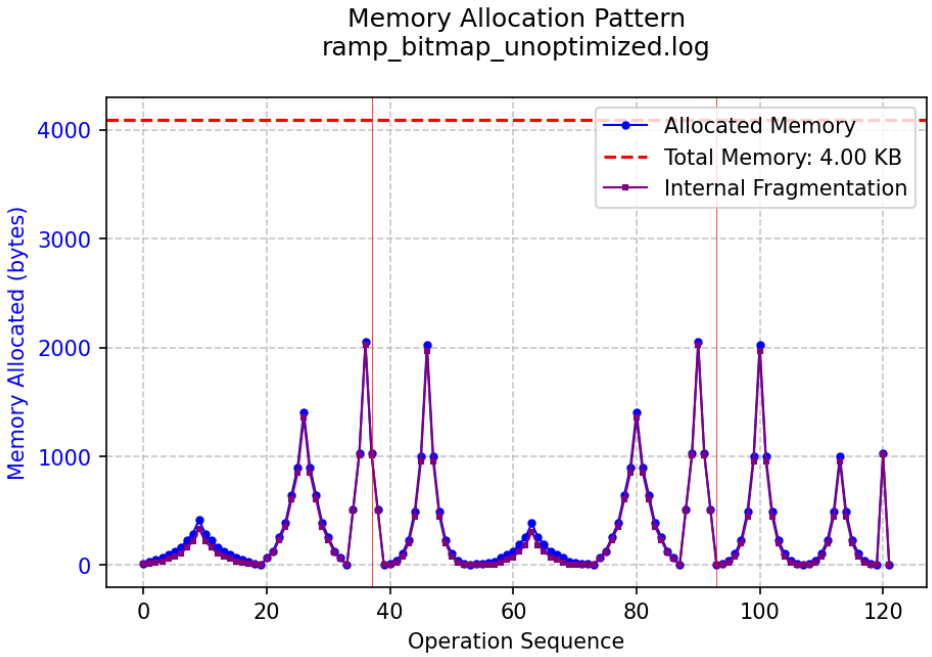

Figura 4.2. Sequenza di istruzioni che causano nel BuddyAllocator grande frammentazione interna. Notiamo che la quantità di memoria inutilizzabile è pari a quella allocata.

Chiaramente, il caso riportato sopra risulta essere manipolato per esasperare il nostro punto. Non vi è alcun dubbio che sarebbe ben difficile incontrare in natura un pattern così sfortunato: tuttavia, bastano una serie di allocazioni mal concepite per rendere, soprattutto in corrispondenza di picchi di utilizzo, una cospicua parte della memoria inutilizzabile e pertanto riteniamo sia importante che il programmatore avveduto faccia uso di strumenti simili per comprendere il comportamento del proprio allocatore e, se necessario, apportare cambiamenti che lo rendano più adatto al proprio caso d'uso. Di seguito riportiamo altri due esempi di pattern in cui tuttavia è stata presa cura di minimizzare la frammentazione diminuendo le taglie degli oggetti allocati oppure, più semplicemente, aumentando di poco la dimensione dell'area di memoria che l'allocatore amministra.

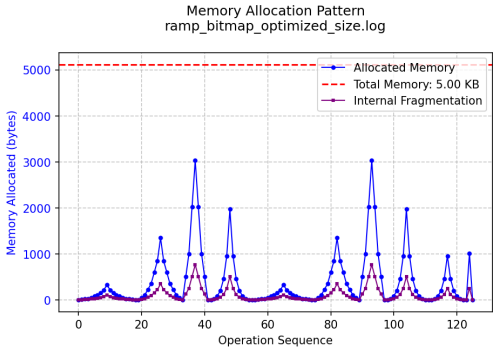

Figura 4.3. Con richieste di dimensione leggermente minore, la frammentazione diventa irrilevante.

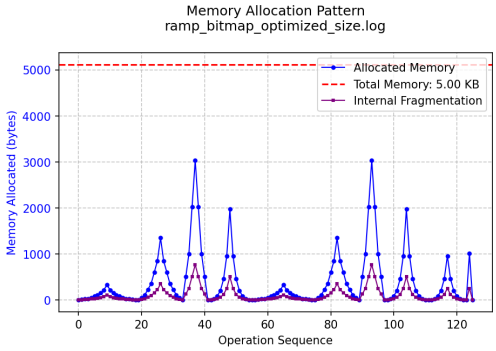

Figura 4.4. Allocando relativamente poca memoria in più, la frammentazione diminuisce sostanzialmente.

4.2.3 LinkedList vs. Bitmap

Sebbene esse siano quasi uguali in termini di frammentazione interna ed esterna (la politica di allocazione è difatti la medesima), la caratteristica che distigue il `BuddyAllocator` e il `BitmapBuddyAllocator` è la struttura dati adoperata per mantenere i riferimenti ai blocchi liberi, in quanto l'implementazione del primo adopera liste concatenate per rappresentare ogni livello. Questa scelta è stata fatta in quanto in ogni momento in ogni lista può essere presente un numero di elementi altamente variabile: la natura delle *LinkedList* le rende quindi particolarmente adatte a svolgere questo ruolo. Ciononostante, nella prima iterazione questo allocatore è andato incontro a una problematica, di cui svolgiamo un'analisi nella sezione successiva.

Utilizzando lo strumento `cachegrind` e gli altri *tool* compresi in *valgrind*, abbiamo potuto svolgere un analisi più approfondita del comportamento del nostro programma, in particolare rilevando le differenze tra le due implementazioni del *buddy system*. Il comportamento della cache dipende dalle caratteristiche del calcolatore. I test eseguiti di seguito sono stati svolti su un sistema avente le seguenti caratteristiche:

```
desc: I1 cache:    32768 B, 64 B, 8-way associative
desc: D1 cache:    32768 B, 64 B, 8-way associative
desc: LL cache:    6291456 B, 64 B, 12-way associative
```

Infine, i dati chiaramenti risentono della presenza delle infrastrutture di controllo e logging. Chiaramente gli accessi in memoria sono molti di più di quelli che avverrebbero in un'applicazione reale perché il programma sta salvando informazioni sullo stato. Poiché però lo "svantaggio" dato dalle risorse aggiuntive di cui sopra si applica in maniera uniforme a tutti i test, riteniamo non sia fuorviante condurre un'analisi relativa e comparata.

Prevenzione dei *Double free*

Un'analisi preliminare delle prestazioni del `BuddyAllocator` mostra che risulta significativamente più lento rispetto all'alternativa basata su bitmap. Questo dato

conferma le aspettative teoriche di una maggiore complessità, ma l'entità della differenza osservata è superiore a quanto previsto e suggerisce la presenza di ulteriori fattori che penalizzano l'efficienza dell'implementazione.

```
i,buddy
p,16276,10
I refs:        13,289,525
I1  misses:         2,910
LLi misses:         2,459
I1  miss rate:       0.02%
LLi miss rate:       0.02%

D refs:         8,944,880  (7,197,682 rd   + 1,747,198 wr)
D1  misses:     1,229,603  (1,217,474 rd   +    12,129 wr)
LLd misses:         6,545  (    1,608 rd   +     4,937 wr)
D1  miss rate:       13.7% (     16.9%     +       0.7%  )
LLd miss rate:        0.1% (      0.0%     +       0.3%  )

LL refs:        1,232,513  (1,220,384 rd   +    12,129 wr)
LL misses:          9,004  (    4,067 rd   +     4,937 wr)
LL miss rate:         0.0% (      0.0%     +       0.3%  )
```

- **I refs**: Fetch istruzioni; **I1/LLi misses**: Mancati accessi cache istruzioni (primo/ultimo livello); **I1/LLi miss rate**: Percentuale miss cache istruzioni.
- **D refs**: Accessi dati (letture/scritture); **D1/LLd misses**: Mancati accessi cache dati (primo/ultimo livello, letture/scritture); **D1/LLd miss rate**: Percentuale miss cache dati.
- **LL refs**: Accessi totali cache ultimo livello; **LL misses**: Mancati accessi totali; **LL miss rate**: Percentuale miss cache ultimo livello.

Come vediamo facilmente, il programma così formulato ha un altissimo numero di istruzioni che accedono alla memoria, e da un'analisi più approfondita scopriamo che ciò avviene per la maggior parte proprio all'interno della funzione `list_find`. Questa funzione è chiamata unicamente all'interno dello `SlabAllocator`, in particolare come parte delle verifiche di correttezza del programma.

Funzione	Ir (%)	Dr (%)	Dw (%)
`list_find`	9,633,626 (72.49%)	6,018,037 (83.61%)	1,205,430 (68.99%)

Tabella 4.4. Profilazione della funzione `list_find` in `double_linked_list.c`: Ir = *Istruction, read*, Dr = *Data, read*, Dw = *Data, write*.

All'interno del `BuddyAllocator` sono presenti due `SlabAllocator` per uso interno. Uno di essi in particolare, `node_allocator`, si occupa di gestire la memoria necessaria per mantenere i `BuddyNode`: essi contengono le informazioni sui *buddies* e sono quindi indispensabili. Quando la memoria viene rilasciata nel `BuddyAllocator`, se due *buddies* sono liberi e pertanto riunificati, i `BuddyNodes` contenenti le informazioni su di essi vengono a loro volta riconsegnati alla *free list* dello `SlabAllocator`.

Quando esso riprende possesso di uno *slab*, si assicura che non sia stato precedentemente rilasciato (evitando l'errore noto come *double free*). Questa verifica può essere fatta attraverso la scansione della lista contenente tutti gli slab liberi, che è molto costosa poiché il costo è lineare al numero degli stessi.

La nostra implementazione, per mantenere le informazioni necessarie per costruire i log, finisce per esacerbare il problema: il numero di slab nella lista è dato da 2^{L+14}, dove L rappresenta il numero di livelli dell'allocatore, in quanto, partizionando la memoria, dobbiamo tenere un riferimento non solo ai *buddies*, ma anche ai genitori. In fase di sviluppo abbiamo menzionato che sarebbe possibile ottenere le informazioni del genitore dinamicamente, ma avevamo scelto di fare altrimenti per semplificare la raccolta di informazioni. Il numero di istruzioni che accedono alla memoria diventa quindi altissimo: l'efficienza ne risente a causa dell'elevato numero di *data cache miss*.

```
1  // Check if node is already in free list using list_find
2  if (list_find(slab->free_list, &slab_node->node)) {
3      #ifdef DEBUG
4      printf(RED "ERROR: Failed to free: slab node already in
           free list!\n" RESET);
5      #endif
6      return NULL;
7  }
```

Per evitare la scansione lineare, la soluzione applicata è l'utilizzo di una flag all'interno di `SlabNode`, `bool in_free_list`, che ci permette di verificare più rapidamente se il blocco sia stato già liberato e quindi aggiunto alla `free list`. Questo check tuttavia è molto meno sicuro, e se il dato venisse corrotto permetterebbe il *double free* di uno slab, che potrebbe dunque essere restituito due volte come risultato di una richiesta di memoria.

Efficienza della cache

A seguito dell'ottimizzazione vista precedentemente, il comportamento dell'implementazione di `BuddyAllocator` che adopera le *linked list* produce risultati molto migliori. Il numero di istruzioni è ridotto significativamente, così come la quantità di *data reference* e conseguentemente dei cache miss. Viene tuttavia pagato un caro prezzo in termini di memoria da allocare per gestire la struttura ad albero: in particolare, un allocatore caratterizzato da un numero elevato di livelli richiede che sia adoperata moltissima memoria per gestire lo `SlabAllocator` al suo interno. Questo tuttavia si traduce in risultati molto positivi da punto di vista dell'efficenza temporale delle operazioni di allocazione e deallocazione, che avvengono molto velocemente. Verifichiamo *throughput*, in base alla sequenza scelta, nell'ordine delle centinaia di migliaia di operazioni al secondo.

Contemporaneamente però, alle preoccupazioni sulla quantità di memoria necessaria all'allocatore se ne aggiungono altre sulla non località della cache. La memoria gestita e quella usata a scopi organizzativi sono diverse, e quindi non abbiamo nessuna assicurazione sul comportamento del buffer: questo comporta maggiore imprevedibilità nella previsione delle *performance* dell'allocatore. Ricordiamo infatti che

[4] Uno per ogni nodo di un albero binario completo.

il nostro benchmark non accede alla memoria allocata: pertanto il quadro a nostra disposizione non è completo e sicuramente in casi d'uso reali l'efficienza della cache che si trovi allo stesso tempo a gestire le richieste di accesso ai dati da parte dell'utente si comporterebbe in modo assai meno efficiente, in quanto verrebbero anche caricate nella memoria temporanea linee di cache in ordine per noi imprevedibile.

```
i,buddy
p,16276,10

I refs:        3,652,823
I1  misses:        2,913
LLi misses:        2,463
I1  miss rate:      0.08%
LLi miss rate:      0.07%

D refs:        1,719,901  (1,178,874 rd   + 541,027 wr)
D1  misses:        7,915  (    2,430 rd   +   5,485 wr)
LLd misses:        6,545  (    1,608 rd   +   4,937 wr)
D1  miss rate:       0.5% (      0.2%     +     1.0%  )
LLd miss rate:       0.4% (      0.1%     +     0.9%  )

LL refs:          10,828  (    5,343 rd   +   5,485 wr)
LL misses:         9,008  (    4,071 rd   +   4,937 wr)
LL miss rate:        0.2% (      0.1%     +     0.9%  )
```

In contrasto con i risultati ottenuti per l'implementazione originale del *buddy system*, la classe `BitmapBuddyAllocator` presenta risultati ben diversi. Il numero delle istruzioni ritorna ad essere molto elevato: ciò è dovuto alla natura ricorsiva delle funzioni di manipolazione della bitmap, che deve essere aggiornata quando cambia lo stato. Non escludiamo che sia possibile, forse dedicando più risorse per mantenere lo stato della memoria gestita, evitare questo numero così alto. Dal punto di vista della rapidità, il programma ne risulta sicuramente rallentato: tuttavia, la località della cache è molto migliore. Il numero di miss è irrisorio e sappiamo anche che la quantità di memoria necessaria per immagazzinare le informazioni è minore. La bitmap infatti è compatta e può essere più facilmente immagazzinata nella memoria intermedia.

```
i,bitmap
p,16276,10

I refs:        14,221,610
I1  misses:         2,790
LLi misses:         2,460
I1  miss rate:       0.02%
LLi miss rate:       0.02%

D refs:         7,089,558  (4,813,930 rd   + 2,275,628 wr)
D1  misses:         4,567  (    2,269 rd   +     2,298 wr)
LLd misses:         3,405  (    1,616 rd   +     1,789 wr)
D1  miss rate:        0.1% (      0.0%     +       0.1%  )
LLd miss rate:        0.0% (      0.0%     +       0.1%  )

```

```
16 LL refs:            7,357  (    5,059 rd   +     2,298 wr)
17 LL misses:          5,865  (    4,076 rd   +     1,789 wr)
18 LL miss rate:         0.0% (      0.0%     +       0.1%  )
```

I risultati ottenuti ci forniscono le seguenti linee guida:

Se il tempo è critico e la quantità di memoria non è un fattore, BuddyAllocator fornisce un throughput ben maggiore e, con alcune ottimizzazioni ulteriori, potrebbe risultare la soluzione migliore per buddy system dalla scarsa profondità, in quanto il numero dei nodi cresce con il suo quadrato. Ad esempio, in un sistema operativo o in un hypervisor dove si devono gestire moltissime allocazioni e deallocazioni al secondo (come nel caso di un server di rete o un sistema real-time), l'elevata velocità di risposta del BuddyAllocator è cruciale e giustifica il maggiore uso di memoria.

Nei casi dove la memoria scarseggia ed è essenziale sfruttare al massimo l'efficacia della cache, il BitmapBuddyAllocator splende per località e semplicità. Le sue caratteristiche lo rendono più adatto quando è necessario avere performance stabili e prevedibili anche sotto carico realistico. In ambienti embedded o su dispositivi IoT dove la RAM è limitata (es. microcontrollori con pochi KB/MB di memoria), oppure in sistemi mobile o edge computing, l'uso efficiente della cache e la ridotta impronta della bitmap sono essenziali per garantire affidabilità e reattività.

Considerazioni sulla thread-safety Se si richiedesse di integrare l'implementazione degli allocatori in modo che essi siano *thread-safe*, i fattori da considerare aumenterebbero ulteriormente. La gestione concorrente delle liste concatenate nel `BuddyAllocator` richiederebbe l'uso di meccanismi di sincronizzazione più complessi (ad esempio, mutex o lock per ogni lista), aumentando il rischio di contese e rallentando le operazioni in presenza di molti thread. Al contrario, la struttura compatta e lineare della bitmap nel `BitmapBuddyAllocator` si presta meglio a implementazioni thread-safe, poiché è più semplice proteggere sezioni critiche e, in alcuni casi, è possibile sfruttare operazioni atomiche su bit per ridurre l'overhead della sincronizzazione. Pertanto, in scenari multithreaded, il `BitmapBuddyAllocator` risulta generalmente preferibile per la maggiore scalabilità e semplicità nella gestione della concorrenza.

4.2.4 Esempi di file `.alloc`

```
% Tipo di allocatore (Slab)
i,slab
% Parametri: slab_size=64, num_slabs=16
p,64,16
% Alloca un blocco nell'indice 0
a,0
% Alloca un blocco nell'indice 1
a,1
% Libera il blocco nell'indice 0
f,0
```

```
% Benchmark per allocatore variabile
i,buddy
% memory_size=1024, max_levels=5
p,1024,5
% Alloca 256 byte nell'indice 0
a,0,256
% Alloca 128 byte nell'indice 1
a,1,128
% Libera l'indice 0
f,0
% Alloca 64 byte nell'indice 2
a,2,64
```

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] K. C. Knowlton, *A Fast Storage Allocator*, Communications of the ACM, Vol. 8, No. 10, pp. 623–625, 1965.

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

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

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

[11] T. Chilimbi, R. Jones, B. Zorn, *Designing a Trace Format for Heap Allocation Events*, 2000.
`trishulc@microsoft.com`, `R.E.Jones@ukc.ac.uk`, `zorn@microsoft.com`

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

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

Ringraziamenti

ACK HERE