Implementazione di HashTable Thread-safe in C

Report per l'esame di Progettazione di Sistemi Operativi

Giulio Barabino

185415
Corso di Laurea Magistrale in Ingegneria Informatica
260553@studenti.unimore.it

1 Introduzione

Le hash table sono strutture dati astratte, la cui prima apparizione risale già al 1953, che nel corso degli anni hanno visto diverse evoluzioni in tutti i suoi aspetti. L'adozione è cresciuta sempre più perché l'efficienza di questa struttura dati è indipendente dal volume dei dati da memorizzare, a differenza delle alternative disponibili. Molteplici sono i linguaggi di programmazione attuali che forniscono la propria implementazione per l'utilizzo da parte dell'utente.

Al contempo i sistemi operativi si sono evoluti dai primitivi sistemi sequenziali e batch fino ai sistemi multiprogrammati. La multiprogrammazione ha consentito una riduzione dei tempi morti, e quindi una migliore efficienza, grazie ad un maggior utilizzo delle risorse. La programmazione concorrente sfrutta la multiprogrammazione dei sistemi operativi dando all'utente l'impressione di avere un sistema di calcolo parallelo in cui i processi vengono eseguiti contemporaneamente. Ciò necessità, però, di un'attenta gestione nell'accesso alle risorse comuni per evitare le cosiddette corse critiche. Da qui la necessità dei costrutti di sincronizzazione e mutua esclusione.

Quanto fatto in questa tesi rappresenta un tentativo di coniugare la programmazione concorrente, con i suoi costrutti di sincronizzazione, con l'efficienza delle hashtable.

2 Conoscenze

In questo capitolo verranno introdotte le conoscenze necessarie per affrontare i capitoli successivi.

2.1 Hashtable

Le hashtable sono *Abstract Data Type* che utilizzano array associativi per salvare coppie di **chiavi-valori**. Per fare ciò la hashtable utilizza la funzione di hash sulla chiave fornita per calcolare la posizione in cui salvare l'elemento.

$$hash(chiave) \longrightarrow i$$

Alla posizione dell'array individuata dall'indice, anche detta **bucket** o **nodo**, verrà salvato il valore fornito.

$$array[i] = valore$$

In seguito, per recuperare il valore basterà fornire alla funzione di hash la medesima chiave. Il vantaggio della hashtable, come è facile notare, è che la dimensione della stessa non influenza il tempo medio di ricerca, a differenza di una normale ricerca binaria o lineare la cui complessità cresce all'aumentare del numero di valori salvati.

Vi è però una criticità: qualora la dimensione della hashtable non sia sufficiente gli elementi si sovrapporranno, causando così **collisioni**. Una collisione avviene quando a due elementi è assegnato lo stesso bucket. Questo accade perché la funzione di hash, benché fornisca in output hash differenti, deve effettuare il **wrapping** degli stessi. Ciò significa che limita l'hash alla dimensione della hashtable per evitare posizioni nell'array che vadano oltre alla dimensione dello stesso. Risulta quindi evidente l'importanza della dimensione della hashtable, in particolare del **load factor**. Esprimiamo il load factor come:

$$loadfactor = n/k$$

Dove:

- n è il numero di bucket occupati
- k è la dimensione della hashtable

Nelle implementazione della hashtable, il load factor è un elemento tenuto costantemente sotto controllo per decidere quando **ridimensionare** la hashtable. Il ridimensionamento può essere eseguito per aumentarne le dimensioni o per diminuirle.

2.2 Risoluzione collisioni

Le principali tecniche di risoluzione delle collisioni sono due:

2.2.1 Separate Chaining

Questa tecnica prevede, in caso di collisioni, di salvare sia il valore entrante che quello già presente all'interno di una lista. All'indice della hashtable troveremo non più l'elemento bensì la locazione di memoria della lista, sulla quale dovremo poi iterare per trovare l'elemento. Questa tecnica è detta **linked list**. Al crescere delle linked list diventa computazionalmente oneroso, e quindi meno veloce, recuperare i valori. Inoltre le moderne CPU sono relativamente più lente ad attraversare puntatori dislocati per la memoria [1], motivo per il quale questo metodo non è preferibile per applicazioni con un grande numero di elementi.

2.2.2 Open Addressing

Le strategie di open addressing risolvono il problema andando a salvare il valore all'interno di un bucket alternativo. Tale bucket può essere calcolato in una moltitudine di modi: linear probing, double hashing, quadratic probing etc. La tecnica utilizzata per l'implementazione in esame è quella del **linear probing**: qualora vi sia una collisione l'algoritmo prevede di iterare a destra del bucket originariamente designato fino a trovarne uno libero. Il problema principale di questo algoritmo è il **clustering**: qualora la hashtable abbia dimensioni ridotte vi sono alte probabilità che una collisione ne generi a sua volte altre, degradando velocemente le prestazioni. Ciò è provato dal paradosso del compleanno [4], il quale conferma che all'aumentare del numero di bucket occupati la possibilità di collisioni aumenta sensibilmente. Risulta quindi imprescindibile un meccanismo di ridimensionamento della hashtable.

2.3 Tombstone

Un altro problema da affrontare quando si tratta con le hashtable di tipo open addressing, in particolare linear probing, sono le cosidette **tombstone**. La semplice eliminazione di un elemento, cioè la rimozione di quel bucket può essere adottata nel caso in cui non vi siano collisioni. Nel caso in cui siano presenti collisioni però, il meccanismo di linear probing risulta compromesso. Un esempio: supponendo che tutti e tre i gli elementi di chiave *bar*, *foo*, *baz* abbiano lo stesso hash e siano quindi stati inseriti secondo linear probing, avremo che:

0	1	2	3	4	5	6
null	*null*	bar	foo	baz	*null*	*null*

Eliminando il bucket di indice tre:

0	1	2	3	4	5	6
null	*null*	bar	***null***	baz	*null*	*null*

In questa situazione l'algoritmo, cercando l'elemento di chiave *baz*, partirà dalla seconda posizione, e non trovandolo, si sposterebbe di una posizione verso destra. Qui terminerebbe immediatamente perché in presenza di un bucket nullo, di fatto non giungendo mai all'elemento corretto in quarta posizione. Per aggirare questa problematica vengono utilizzate le tombstone, segnaposti appositi tramite i quali distinguere i bucket eliminati da quelli non ancora utilizzati.

2.4 Hash Function

La funzione di hash è una funzione non reversibile che, dato un input, fornisce l'indice dell'array, cioè il bucket, all'interno del quale salvare l'elemento. La chiave può essere di dimensione fissa o variabile e può essere essa stessa l'elemento da salvare. La funzione di hash deve possedere diverse caratteristiche:

- Data in input la stessa chiave deve fornire il medesimo output, deve cioè essere **deterministica**.
- Deve produrre un output il più **uniforme** possibile perché l'obiettivo è avere elementi il più distribuiti possibile.
- Deve essere **veloce** sfruttando al meglio l'architettura del calcolatore disponibile.

Gli algoritmi di hashing sono un parametro molto importante della hashtable e ve ne sono di molteplici tipi, dai più semplici e rudimentali a quelli più complessi e largamente adottati come FNV-1, fino a quelli a scopo crittografico.

2.5 Sincronizzazione

Il problema della sincronizzazione che si presenta è riconducibile a quello noto come **problema dei lettori-scrittori**: assumiamo infatti che gli scrittori siano coloro i quali vogliono inscrivere all'interno della hashtable, mentre i lettori siano colori i quali vogliono recuperare gli elementi dalla hashtable. Più lettori possono accedere contemporaneamente, in quanto non operano modificazioni sui dati, ma solo uno scrittore per volta è ammesso. I lettori possono acquisire il lock se non vi è nessuno scrittore che ha già acquisito il lock o se quest'ultimo è sospeso in attesa di acquisirlo. Viceversa gli scrittori acquisiranno il lock solamente se non vi sono altri lettori o scrittori che hanno correntemente acquisito il lock. Come si può vedere l'implementazione favorisce gli scrittori sui lettori, questo accade per evitare la *starvation* cioè l'attesa perpetua, degli scrittori.

3 Implementazione

Per l'implementazione è stato scelto di utilizzare la tecnica di open addressing, in particolare quella di linear probing. Prima di poter effettuare una qualunque operazione l'utente deve prima creare la hashtable. Per fare ciò si utilizza la funzione

```
create_hash_table(size_t size);
```

a cui viene passato come parametro la dimensione iniziale desiderata dall'utente. Restituisce un puntatore ad una *struct* di tipo hashtable. La struct `HashTable` presenta diversi membri:

```
typedef struct hash_table {
    Node *node;
    size_t size;
    size_t num_elements;
    int high_density;
    int low_density;
    pthread_rwlock_t lock;
} HashTable;
```

La dimensione indicata dall'utente viene salvata nel campo `size`; `num_elements` tiene traccia del numero di bucket occupati; il ridimensionamento è deciso sulla base dei parametri di `high_density` e `low_density`; `lock` assicura la sincronizzazione nell'accesso alle API; infine `node` è l'array associativo nel quale vengono effettivamente memorizzati i dati. Quest'ultimo membro è un puntatore ad una *struct* così composta

```
typedef struct node {
    char* key;
    void* element;
} Node;
```

All'interno del metodo `create_hash_table()` vengono inizializzati e allocati correttamente tutti i membri. In particolare:

```
for (i = 0; i < size; i++) {
            ht->node[i].key = NULL;
            ht->node[i].element = EMPTY;
}
```

Tutti gli `element` della struct `node` vengono inizializzati a **EMPTY** mentre le `key` a **NULL**. EMPTY è definito come:

```
#define EMPTY (void*) 0x00
```

3.1 Application Programming Interface

3.1.1 Inserimento

L'inserimento consente di inscrivere, all'interno della hashtable specificata, l'elemento in funzione della chiave fornita. La funzione presenta la seguente sintassi:

```
hash_insert(HashTable* ht, char* key, void* element);
```

La funzione ritorna 1 nel caso di successo, 0 nel caso di avvenuta collisione e -1 in tutti gli altri casi. L'operazione di insert avviene nel seguente modo:

1. Viene calcolato l'hash della chiave tramite l'apposita funzione.
2. Viene iterato sull'array `node` a partire dalla posizione indicata dall'hash.
3. Se il bucket a quella posizione non è popolato, cioè la chiave è NULL, inscrivo i valori.
4. Se il bucket a quella posizione è popolato controllo che le chiavi non combacino. In tal caso si tratterebbe di un tentativo di sovrascrittura. Se così fosse procedo ad aggiornare unicamente l'elemento.
5. In caso contrario procedo incrementando l'hash, quindi spostandomi virtualmente a destra di uno, e riprendendo dal passo 3.

L'hash viene incrementato effettuandone il wrapping:

```
hash = (hash + 1) % ht_size
```

Dove `ht_size` è la dimensione della hashtable. Questo potrebbe portare potenzialmente a ciclare all'infinito sull'array, motivo per il quale tramite la variabile `counter` viene tenuto traccia del numero di incrementi: qualora questo fosse uguale alla dimensione della hashtable il ciclo terminerebbe.

3.1.2 Ricerca

La ricerca consente, data una chiave, di trovare l'elemento corrispondente. Presenta la seguente sintassi:

```
hash_get(HashTable* ht, char* key);
```

La ricerca, come anche l'eliminazione, si avvale di una funzione denominata `find_node` per trovare il bucket desiderato. Il funzionamento è del tutto analogo a quello descritto nella fase di inserimento, l'unica differenza è la finalità della funzione: in questo caso viene restituito il bucket trovato, altrimenti NULL.

3.1.3 Eliminazione

L'eliminazione consente, data una chiave, di eliminarne l'elemento corrispondente. Presenta la seguente sintassi:

```
hash_remove(HashTable* ht, char* key);
```

Come detto precedentemente, la funzione fa uso dell'algoritmo iterativo `find_node` per trovare il nodo richiesto. Una volta trovato, è la funzione `hash_remove` che si preoccupa dell'eliminazione vera e propria. Per i motivi descritti nella sezione dedicata alle tombstone 2.3, il bucket non viene eliminato bensì vengono eseguite le seguenti operazioni:

```
found->key = NULL;
found->element = TOMBSTONE;
```

In questo modo l'algoritmo di ricerca della funzione `find_node` è in grado di distinguere tra `element` di tipo `EMPTY`, cioè bucket vuoti, e `TOMBSTONE`, cioè elementi eliminati.

3.2 *find_node*

La versione aggiornata dell'algoritmo utilizzato da `find_node` è la seguente:

```
for (; counter < ht_size; counter++, hash = (hash + 1) % ht_size) {
            // Controllo che il nodo non sia NULL
            if (node[hash].key == NULL) {
                    if (node[hash].element == EMPTY) {
                            // Se l'elemento è EMPTY allora è libero
                            // per l'assegnazione
                            return found != NULL ? found : &node[hash];
                    }
                    // Altrimenti il nodo è una TOMBSTONE, cioè un nodo
                    // rimosso in precedenza
                    if (found == NULL) {
                            // Lo salviamo per il caso in cui questa
                            // funzione venga usata da hash_insert,
                            // cioè per la ricerca di un nodo NULL
                            found = &node[hash];
                    }
                    continue;
            }

            // Qualora non sia vuoto confronto la chiave presente con
            // quella fornita
            if (strcmp(key, node[hash].key) == 0) {
                    LOG(("Trovato alla pos. %lu\n", hash));
                    return &node[hash];
            }
    }

    // Nel caso in cui il ciclo sopra termini significa che non è
    // presente l'elemento all'interno della HashTable
    return NULL;
```

Questa funzione rappresenta il cuore dell'intera hashtable e il suo funzionamento è, come descritto nelle sezioni precedenti, alla base delle funzioni API.

Utilizza un `node* found` temporaneo per salvare la posizione della prima tombstone trovata. Questa verrà restituita dalla funzione all'iterazione successiva, in quanto inscrivibile. Le tombstone sono infatti considerate alla stregua di bucket empty per gli scopi della funzione di insert, ciò consente di sovrascriverle ed evitare il proliferare delle stesse.

3.3 Ridimensionamento

Il ridimensionamento, per i motivi descritti nel capitolo due 2, è essenziale. Esso viene realizzato controllando il load factor ad ogni inserimento e ad ogni eliminazione. Se il load factor dovesse essere minore di una certa soglia, a seguito di una eliminazione, o maggiore di un'altra, a seguito di un inserimento, vengono chiamati rispettivamente le funzioni `hash_shrink` e `hash_expand`. Di seguito il controllo effettuato in capo alle funzioni di `hash_delete` e `hash_insert`:

```
if ((int) (ht->num_elements*100/ht->size) <= ht->low_density) {

            if (hash_shrink(ht)) {
                    LOG(("HashTable rimpicciolita! Nuova dimensione: "
                         "%ld\n", ht->size));
            }
    }

if ((int) (ht->num_elements*100/ht->size) >= ht->high_density) {

            if (hash_expand(ht)) {
                    LOG(("HashTable espansa! Nuova dimensione: %ld\n",
                            ht->size));
            }
    }
```

I limiti superiori e inferiori di densità della hashtable non vengono decisi dall'utente in fase di creazione ma è possibile modificare gli stessi tramite opportune funzioni:

- `hash_set_resize_high_density`
- `hash_set_resize_low_density`

dal banale funzionamento. Sono state scelte due densità nel range di valori che empiricamente hanno dimostrato le performance migliori [3], in particolare:

```
#define TABLE_MAX_LOAD 70
#define TABLE_MIN_LOAD 30
```

3.3.1 *hash_shrink*

L'algoritmo si articola in quattro passaggi:

1. Viene allocato un `node* copy` di dimensioni dimezzate.
2. Vengono inizializzati gli elementi dell'array analogamente a quanto descritto nei capitoli precedenti:

```
for (i = 0; i < half; i++) {
        copy[i].key = NULL;
        copy[i].element = EMPTY;
}
```

3. Viene iterato su ogni nodo sulla hashtable originale e, nel caso in cui il bucket non sia vuoto, viene inserito all'interno del nodo `copy`. Per fare ciò viene sfruttata una funzione operante in modo analogo a quanto descritto nella sezione relativa all'inserimento 3.1.1 con la differenza dell'assenza di lock nella stessa. La motivazione di questa scelta verrà fornita nella sezione riguardante la sincronizzazione 3.4.
4. Una volta esauriti gli elementi della hashtable originale viene assegnata all'istanza della struct hashtable `ht` il nodo `copy` come nuovo array associativo:

```
ht->node = copy;
```

Al termine della funzione viene restituito un valore booleano, *true* in caso di successo e *false* in tutti gli altri casi, così che la funzione `hash_delete` possa procedere di conseguenza.

3.3.2 *hash_expand*

Il funzionamento è il medesimo di quello di `hash_shrink` con la differenza che le dimensioni sono raddoppiate e non dimezzate. Viene effettutato un controllo sulle stesse, nello specifico viene controllato che la nuova dimensione non causi overflow:

```
doubled = ht->size * 2;
if (doubled + doubled < doubled) {
        return false;
}
```

In tal caso la funzione termina la propria esecuzione restituendo *false* alla funzione chiamante, cioè `hash_insert`.

3.4 Sincronizzazione

La sincronizzazione, secondo le modalità descritte nella sezione omonima del capitolo precedente 2.5, viene realizzata tramite un costrutto disponibile nella libreria `pthread.h` chiamato `pthread_rwlock_t`. Esclusa l'inizializzazione, vengono utilizzati due metodi per interagire con esso:

- `pthread_rwlock_rdlock` per acquisire il lock da parte dei lettori. Questa funzione viene utilizzata all'interno di `hash_get` per la ricerca dell'elemento all'interno della hashtable.

```
void* hash_get(HashTable* ht, char* key) {
    Node* found;
    size_t hash;

    // Acquisisco il lock
    rdlock(&ht->lock);
    .
    .
    .
```

- `pthread_rwlock_wrlock` per acquisire il lock da parte degli scrittori. Questa funzione viene utilizzata all'interno di `hash_insert` e `hash_delete` poiché operano modificazioni ai bucket della hashtable.

```
int hash_insert(HashTable* ht, char* key, void* element) {
    size_t hash;
    int retr;

    // Acquisisco il lock
    wrlock(&ht->lock);
    .
    .
    .
```

Come è possibile evincere dagli estratti di codice, i lock vengono acquisiti prima di realizzare qualunque altra operazione all'interno della funzione e viene rilasciato unicamente al termine delle stesse.

La difficoltà principale della sincronizzazione risiede in fase di ridimensionamento della hashtable: occorre mantenere il lock sugli elementi mentre vengono inseriti nel nuovo nodo 3.3 per evitare che altri thread scrittori chiamanti l'`hash_insert` modifichino il contenuto in corso d'opera di ridimensionamento. Per sopperire a questa problematica è stata scorporata la funzione di *insert*, come anticipato precedentemente, in:

- Una funzione di `insert` che si occupa di agire secondo l'algoritmo descritto nell'apposito sezione 3.1.1.
- L'effettiva funzione `hash_insert` che acquisisce il lock di scrittura e chiama a sua volta la funzione `insert`. Si può pensare alla funzione `hash_insert` come ad un *wrapper* della funzione di `insert`, cioè una funzione che realizza operazioni aggiuntive, come l'acquisizione del lock o il controllo della coppia *key-element*.

Così facendo è possibile chiamare direttamente la funzione interna `insert` in fase di ridimensionamento, evitando l'acquisizione e il rilascio del lock per ogni elemento. Di seguito un diagramma di flusso che chiarisce l'ordine delle operazioni:

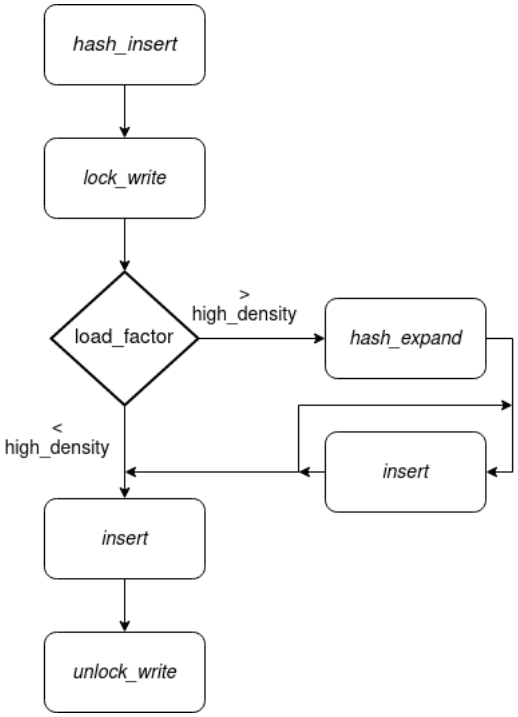

Figure 1: Flowchart Inserimento

Quanto mostrato nel diagramma di flusso è analogo anche al procedimento di eliminazione e quindi di rimpicciolimento della hashtable.

3.5 Funzione di Hash

Le funzioni di hash utilizzate sono tre:

- *FNV-1a*
- *sdbm*
- *djb2*

Le tre funzioni sono state scelte in base all'ampia adozione delle stesse e non è stato effettuato uno studio approfondito sulla bontà in quanto esulava dallo scopo di questa tesi.

3.5.1 *FNV-1a*

Questa funzione effettua lo *XOR* bitwise per ogni carattere della chiave e poi lo moltiplica con un numero primo tra quelli consigliati [2], appositamente scelto per l'architettura del calcolatore sulla quale è stato programmata l'hashtable.

```
#define FNV_OFFSET 14695981039346656037UL
#define FNV_PRIME 1099511628211UL
    for (p = key; *p; p++) {
        hash ^= (size_t)(unsigned char)(*p);
        hash *= FNV_PRIME;
    }
```

La funzione è identica alla *FNV-1* con la differenza che viene invertito l'ordine delle operazioni di XOR e moltiplicazione.

3.5.2 *sdbm*

```
    for (counter = 0; key[counter] != '\0'; counter++) {
                hash = key[counter] + (hash << 6) + (hash << 16) - hash;
    }
```

3.5.3 *djb2*

```
    size_t hash = 5381;
    for (counter = 0; key[counter] != '\0'; counter++) {
                hash = ((hash << 5) + hash) + key[counter];
    }
```

4 Dettagli

In questo capitolo viene descritto quanto tralasciato nel capitolo precedente in quanto ad implementazione perché non strettamente correlato ma degno di menzione per una migliore comprensione del codice.

4.1 Allocazioni

Finora è stato omesso volontariamente quando e in quali modalità sono allocate le variabili. Il linguaggio C prevede una gestione della memoria capillare in quanto delegata all'utente ma necessita anche di un attento uso. Vengono di seguito illustrate brevemente le scelte implementate.

- `key` ed `element` vengono allocati e copiati da quelli passati dall'utente, in fase di insert, tramite la funzione `strdup`. Quest'ultima si occupa di allocare un nuovo spazio di indirizzamento e copiarne all'interno il contenuto della stringa originale.

```
node[hash].key = strdup(key);
node[hash].element = strdup(element);
```

In caso di aggiornamento dell'`element` viene prima liberato lo spazio e successivamente allocato per l'`element` sovrascrivente.

```
free(node[hash].element);
node[hash].element = strdup(element);
```

- Analogamente le funzioni di `hash_shrink` e `hash_expand` procedono a liberare lo spazio occupato dalla hashtable originaria dopo averne copiato i dati.

```
free(ht->node[i].key);
free(ht->node[i].element);
```

- In ultima analisi la funzione `hash_delete` prima di porre `key` a NULL ed `element` a TOMBSTONE, libera lo spazio occupato in fase di inserimento dalla `strdup`.

```
free(found->key);
free(found->element);

found->key = NULL;
found->element = TOMBSTONE;
```

L'analisi tramite lo strumento *valgrind*, per il controllo dell'allocazione di memoria, rivela come non vi siano *leak* rilevati, segno di una corretta gestione della memoria.

4.2 Logging

Per agevolare lo sviluppo di questa libreria è stato implementato un basilare sistema di logging definendo una funzione `LOG(a)` da utilizzare nelle sezioni cruciali del codice. Tale funzione sfrutta semplicemente la `printf` della standard library del C, qualora il logging sia attivo, stampando a schermo. Qualora invece il logging sia disattivato redigerà tutte le `LOG` a funzioni `void` dall'effetto nullo. Di seguito il codice:

```
#if 0
    #define LOG(a) printf a
#else
    #define LOG(a) (void)0
#endif
```

Così facendo è possibile preservare le `printf` senza doverle sopprimere e al contempo non stampare a schermo ad ogni operazione di LOG.

5 Prestazioni

Al fine di testare le funzionalità della libreria realizzata e la successiva analisi delle prestazioni, sono stati creati due programmi per la **conta delle occorrenze**:

- `demo.c` per il test single-thread
- `demo-thread.c` per il test multi-thread

5.1 Programmi di test

5.1.1 *demo.c*

Questo programma prevede due parametri posizionali:

- *TABLE_SIZE*: dimensione iniziale della hashtable
- *FILE_NAME*: nome del file su cui operare

Utilizza una hashtable per memorizzare le occorrenze delle parole presenti all'interno del file fornito come parametro, il quale ha come unico requisito quello di essere già formattato prevedendo una sola parola per riga.

Per ogni parola all'interno del file viene effettuata una `hash_get` per verificare che la parola, o `key`, non sia già presente all'interno della hashtable. Se così non fosse procede con una `hash_insert` con elemento il valore numerico uno. Se, invece, dovesse essere già presente procede a salvare l'elemento, cioè il numero di occorrenze registrate, incrementarlo di uno e re-inserirlo tramite `hash_insert`.

La seconda fase del programma prevede l'eliminazione degli elementi dalla hashtable. Analogamente alla fase di inserimento, viene controllata parola per parola che essa non sia presente all'interno della hashtable tramite `hash_get`. Se così non fosse si procede all'eliminazione tramite `hash_delete`.

5.1.2 *demo-thread.c*

Questo programma prevede tre parametri posizionali:

- *TABLE_SIZE*: dimensione iniziale della hashtable
- *FILE_NAME*: nome del file su cui operare
- *N_THREADS*: il numero di thread con cui agire sul file

Il funzionamento è il medesimo del programma `demo.c`, in particolare nelle fasi di inserimento ed eliminazione. La differenza risiede nella creazione di un numero stabilito dall'utente di thread col quale suddividere il carico del file. In particolare, ad ogni thread viene assegnato un intervallo di righe all'interno del file, di cui dovrà operare conta e successiva eliminazione delle parole. Così facendo si vuole tentare di simulare una possibile applicazione multi-processo.

5.1.3 *demo.py*

Per il confronto delle prestazioni è stato realizzato anche un programma in Python che prevede lo stesso funzionamento di quello di `demo.c`. Anch'esso necessita di due parametri posizionali quali FILE_NAME e TABLE_SIZE, prevede una fase di conta delle occorrenze e una successiva fase di eliminazione. La funzione di hashtable è assolta dalla struttura dati `dict`, cioè dizionario, hashtable nativa in Python.

5.2 Esiti delle valutazioni

I test sono stati effettuati su un calcolatore con CPU i5-8250U a 8 core e 16 GB di memoria principale, su sistema operativo Linux Fedora 37.

Per misurare il tempo di esecuzione è stato utilizzato il comando nativo di UNIX/LINUX `time`. Questo comando accetta come parametro il programma di cui eseguire la misurazione dei tempi e fornisce tre risultati:

- **Real time**: il tempo reale che il processo impiega per eseguire dall'inizio alla fine. Include i tempi di attesa spesi per aspettare che altri processi terminino la propria esecuzione.

- **User time**: il tempo speso dalla CPU in *user mode* durante l'esecuzione del processo.
- **System time**: il tempo speso dalla CPU in *kernel mode* durante l'esecuzione del processo.

5.2.1 Confronto funzioni di Hash

- Il file utilizzato, `words.txt`, consta di 901428 righe.
- La dimensione iniziale della hashtable è 1000
- Il numero di thread per `demo-thread.c` è 4

Di seguito un confronto dei tempi reali sulla base di dieci iterazioni delle tre funzioni di hash precedentemente presentate.

	FNV-1a	sdbm	djb2
demo	0.478s	0.803s	0.483s
demo-thread	1.685s	2.015s	1.817s

La funzione FNV-1a risulta la migliore per quanto riguarda l'hashing sia in ambito single-thread che multi-thread. Per questo motivo è la candidata nel confronto di prestazioni con il programma Python. Già da questo primo confronto è evidente come l'applicazione multi-thread sia più lenta, in particolare fino a tre volte, della controparte single-thread.

5.2.2 Confronto con *dict*

- Il file utilizzato, `words.txt`, consta di 901428 righe.
- La dimensione iniziale della hashtable è 1000
- Il numero di thread per `demo-thread.c` è 4
- La funzione di hash utilizzata è la migliore candidata dai test precedenti, ovvero *FNV-1a*.

Di seguito un confronto dei tempi reali sulla base di dieci iterazioni:

demo	*demo-thread*	*demo.py*
0.474s	1.681s	1.415s

L'applicazione single-thread performa circa il 66% più veloce dello script Python. L'applicazione multi-thread, invece, risulta essere peggiore del 18% circa, come anche osservato nel confronto delle funzioni hash. Ciò è da imputarsi, probabilmente, a due cause principali:

- Accesso alle risorse in mutua esclusione
- L'overhead dato dalla creazione di thread appositi per l'inserimento e thread appositi per la rimozione.

Nonostante ciò, i risultati non si discostano molto dai tempi impiegati dal programma Python.

6 Conclusioni

In conclusione si può quindi affermare che i risultati ottenuti siano considerevoli sia per applicazioni single-thread che per applicazioni multi-thread utilizzanti la libreria sviluppata.

Eventuali sviluppi futuri potrebbero prevedere un meccanismo di sincronizzazione per singolo bucket invece che globale. Una sincronizzazione di questo genere comporterebbe, almeno in teoria, un incremento delle prestazioni derivante dall'aumentata concorrenza dei processi operanti. Presenterebbe però una problematica, di non facile risoluzione, in fase di ridimensionamento della hashtable.

Un'altra direzione nella quale si potrebbero apportare modifiche è quella delle funzioni di hash: integrare funzioni più complesse e con una migliore distribuzione dei valori di quelle scelte.

Nonostante gli aggiustamenti che si possano operare siano molteplici l'obiettivo che questa tesi si prefiggeva era di integrare i meccanismi di sincronizzazione all'interno delle hashtable ed esso può dirsi comunque raggiunto.

References

[1] Baptiste Wicht. C++ benchmark - std::vector vs std::list. `https://baptiste-wicht.com/posts/2012/11/cpp-benchmark-vector-vs-list.html`, 2012.

[2] Glenn Fowler, Landon Curt Noll, and Kiem-Phong Vo. The core of the fnv-1 hash. `http://www.isthe.com/chongo/tech/comp/fnv/#FNV-param`, 2023.

[3] Olumide Owolabi. Empirical studies of some hashing functions. *Information and Software Technology*, 45(2):109–112, 2003.

[4] Wikipedia contributors. Birthday problem — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Birthday_problem&oldid=1133204911`, 2023.