Notes Software Engineering

Brendon Mendicino

March 10, 2023

Contents

1 Memoria

La memoria si compone si suddivide in:

- registri;
- cache
- memoria primaria;
- memoria secondaria;

Uno principali problemi di accesso alla memoria è quello di assicurarne la sua protezione, ovvero evitare che un programma in memoria riesca ad accedere alle zone di memoria di altri programmi. Una volta fatto il bootstrap del SO, sia esso che i programmi verranno caricati in memoria, per evetire che ogni processo abbia una vista al di fuori del suo scope, si possono usare dei controlli a livello della CPU, esistono dei registri chimati **base** e **limit**, che attraverso dei meccanismi, riescono ad effettuare la protezione della memoria. Esiste anche un altro problema che è quello **relocation**, consiste nel come mappare gli indirizzi delle varie istruzioni di salto e dei vari puntari una volta che il programma viene caricato in memoria, se si utilizzaro degli indirizzi statici (creati solo in fase di compilazione e prendondo come riferimento di inizio del programma l'indirizzo 0), una volta che questo viene caricato in memoria le istrizioni non punterebbero più alle label corrispondenti ma sempre nello stesso punto, dove ad esempio si trova un altro programma. Per questo motivo quando si utilizzano valori di registri vengono sommati al base register, mentre le boundry del programma in memoria vengono salvate nel limit. Binding ... Si fa:

- **in compilazione**: fatto quando molto semplice, ad esempio quando esistono solo due programmi;
- **in fase di load**: viene fatta la rilocazione durante il caricamento in memoria;
- **in esecuzione**: viene fatto il binding degli indirizzi in modo dinamico;

Per risolver questo problema ci si affida all'hardware, che è incaricato di fare la traduzione: l'indirizzo rimane lo stesso (logico) all'interno del processore, prima di arrivare all'address bus viene tradotto in indirizzo fisico, esiste dunque una dicotomia di indirizzo logico e fisico, in questo modo quando si scrive un programma, l'indirizzo parte sempre da 0. Esistono due tipi di indirizzamento che sono di tipo logico, dove l'intervallo degli indirizzi utilizzabili è logico, ed un indirizzamento fisico, dove il range è limitato dalla memoria del sistema. Per effettuare questa traduzione da indirizzo fisico a indirizzo logico e viceversa si utilizza una **MMU** (Memory Management Unit). Il modo più facile per realizzare una MMU, è quello di usare un **relocation register**, ovvero un registro che contiene il valore da aggiungere un indirizzo logico per fare

un indirizzo fisico, il modello più semplice di MMU è fatto da un sommatore ed un comparatore.

Per aumentare le prestazioni ed usare la memoria in modo più efficente si possono usare delle tecniche dinamiche.

- Si parla di **dynamic loading** quando, un programma viene caricato in memoria principale in modo frammentato, utilizzando solo i componenti che effettivamento vengono chiamati;
- Si parla di **dynamic linking** quando i file che contengono le funzioni che devono essere linkate (come le librerie standard) non vengono inserite all'interno dell'eseguibile, ma gli indirizzi vengono risolti in modo dinamico durante l'esecuzione;
- Il **link statico** è quando si crea un eseguibile con tutte le funzioni dentro, di fatto il loader carica tutto quando in memoria.

Il **dynamic loading** vuol dire che una routine non è caricata finché non è necessaria, questo può essere fatto quando il programmatore ne è consapevole, infatti il processo di load non è trasparente:

```
void myPrinf(**args) {
  static int loaded = 0;
  if (!loaded) {
    load("printf");
    loaded = 1;
  }
  printf(args);
}
```

Il **linker-assisted DL** si usa una chiamata fasulla che prima chiama la load, usando una **stub**.

Le **shared libraries** sono in grado di condividere le risorse, infatti se più processi utilizzano la stessa funzione essa viene messa a disposizione e per ogni nuova chiamata non ci sarà bisogno di chiamre una load.

Come si alloca memoria per un programma (immagine)? La più semplice è l'allocazoine contigua, dove si vede la RAM come due partizioni, una per il SO una per i processi (indirizzi più alti), per caricare un processo si parte da un indirizzo di inzio ed un indirizzo di fine, la MMU vista prima funziona solo con i casi di allocazione contigua (basilare). L'**allocazione contigua con paritizione variabile**: quando ci sono più buchi ci sono delle politiche differnti per inserire nuovi programmi:

- **first-fit**: il primo che si trova;
- **best-fit**: il buco con la dimensione più piccola;
- **worst-fit**: il buco can li dimesnione più grande;

La frammentazione è sparsità dei buchi all'interno della memoria. Si dice **esterna** perché è al di fuori dei processi, si dice **interna** quando è interna al processo, ovvero che ha più memoria di quello che serve. La frammentazione ha bisongo di **compattazione**, il SO sposta i pezzi e poi si riparte, partitione della memoria in zona libera e zona occuputa, per fare una **deframmentazoine** (o compattazione) voul di re creare solo due partizioni (parte processi e zona libera), per effettuare la compattazione bisogna che i processi si possano spostare, inoltre un processo non può essere spostato se in quel momento il processo sta effettuando delle operazioni di IO, una soluzione per il problema dell'IO è il **latch job** in cui quella parte non può essere spostata, oppure si utilizza un **buffer IO del kernel**, si fa IO solo in buffer del sistema operativo.

Con **backing store** si definisce uno spazio della memoria secondaria in cui non si salvano programmi ma vengono immagazzinati dei dati che altrimenti dovrebbero andare in RAM.

La **Paginazione** risolve i problemi della allocazione continua, rimuovendo l'allocazione contigua al suo posto si utilizzano della pagine, che sono l'unità minima di allocazione, questo risolve il problema della frammentazione.

Definition 1.1 – Pagine

Le partizioni della memoria logica.

Definition 1.2 – Frame

Le partizioni della memoria fisica.

Definition 1.3 – Blocco

...

Tipicamente la loro dimensione è un multiplo di due, sia pagine che frame hanno la stessa dimensione, da qualche parte andranno salvate le informazioni di mapping tra pagine e frame, si utilizza un **frame table**, dove ogni riga corrisponde una mappatura. Anche utilizzando la paginazione si ha frammentazione interna, l'ultima pagina soffrirà di frammentazione.

Una indirizzo generato dall CPU si divide in:

- **numero di pagine (p)**:
- **numro di offset (d)**:
- **numero di frame (f)**:

Un indirizzo è composto da: (p-d, d)

Example 1.1

Se si hanno frame piccoli diminuisce la frammentazione ma aumentano anche le righe della tabella.

Nelle righe della tabelle si aggiungono dei bit in più:

- parte di protezione, specifica che la parte di codice non può essere scritta;
- **modify bit**: pagina modifica;
- **page present/absent**:

Come si implementa una page table? La page table si trova in memoria RAM, per sapere dove si trova si utilizzano due rigistri: **page table base register** e **base table lenght register**, infatti la base table si trova in memoria contigua. Per velocizzare questa operazione si utilizza che TLB che è più piccola e si trova direttamente sulla CPU, si usa la **Translation Look-aside Buffer** (TLB), un tipo di memoria in cui si accede per contenuto. Si aggiunge anche un altra informazione **ASID** in cui viene salvata l'infromazione del processo a cui la pagina appartiene, se non è presente questa informazione si i processi si contentendono la TLB. Quando avviene un **TLB miss** se reinserisce nella TLB e poi si ritenta, utilizzando una politica scelta.

Lo schema finale è la combinazione di TLB e page table.

Il **Tempo di accesso effettivo** (EAT) in memoria è il tempo che mi costa accedere alla RAM:

$$EAT = h \cdot M + (1 - h) \cdot 2M$$

- h = TLB hit ratio;
- M = Memory access time;

La protezione è implementata associando un bit di protezione per un frame specifico, e quindi per la rispettiva pagina, grazie a questo bit si possono definire parti ci codice che indica che il frame è in solo scrittura. Anche un altro bit associato è **validity** che indica quando una pagina è valida, ovvero quando c'è ed è associata ad un frame, o quando una pagina non è valida che vuol dire che la pagina non esiste o che non è associata ad frame, se si cerca di accedere una pagina non valida viene lanciata una **trap**.

La tabella della pagina permette di **condividere le pagine** (anche se in realtà sono i frame), se più processi hanno delle pagine che sono comuni vengono condivise da entrambi i processi.

La page table è una struttura dati del kernel:

Example 1.2

HP: spazio logico di indirizzamento di 32 bit, una pagina di 4KB (2^{12}), la page table avrà un milione di entry ($2^{32}/2^{12}$), ogni entry della page table sono 4bytes, quindi una pagetable è grande 4MB. Adesso è possibile avare:

- **page table contigua**:
  - dimensione critica (troppo grande):
    - page table gerarchica;
    - hash page table;
    - inverted page table;

La **la page table gerarchica**, la page table viene divisa in blocchi più piccoli non contigui, se usa una page table di livello superiore che porte alle page table di livello inferiore, questi blocchi devono rimanere contigui, anche se diventano molto piccoli. Il page number viene diviso in due parti, una per la tabella outer ed una per la tabella inner (può avere anche più di due livelli), quando si va su 64 bit e la outer dienta molto grande si può usare solo una parte dell'indirizzamento se l'eseguibili è molto piccolo.

La **hashed page table** permette di creare una tabella di hash, con una funzione direttamente implementata in hardware che dato p in input ritorna f. Vanno implementate delle liste di collisione, e quindi vieni immagazzinato che p che è la chiave di accesso. Usando una tabella di hash si potrebbe pensare anche di usare una page table condivisa tra tutti i processi, ma a quel punto va inserito anche l'ID del processo.

La **inverted page table** è una tabella condivisa tra tutti i processi, la sua grandezza viene dimensionata sulla grandezza della RAM e non sulla grandezza dei processi, in cui ogni frame fisico è associata una pagina, in questo caso le entry che corrisponde ad un frame ritorna la pagina a cui è associto, il problema che si vuole la traduzione da pagine ad entry e non il contrario, la prima strategia che si può utilizzare è la scansione lineare. Per rendere l'accesso più veloce viene messa una tabella di hash prima di arrivare alla tabella invertita.

1.1 Swapping

Si suppunga che ad un certo punto si volgia far partire un processo e che la memoria sia piena, cosa si può fare? Una soluzione è lo **swapping**, ovvere viene rimesso sul backing store (in modo temporaneo) che in linux è la partizione di swap, e viene caricato il nuovo processo, grazie allo swapping si possono avere processi che hanno più memoria occupata di quanto la RAM sia grande.

Example 1.3

Qunato costa il **context switch**? Memoria piena e viene fatto partire un programma, dunque deve essere portato un processo nel backing store. 100MB da fare la swap con velocità di trasferimento di 50MB/sec.

- swap out = 2000ms
- più swap in dello stessa grandezza
- il tempo totole è di 4s

Se il porcesso sta fendo IO non si può fare lo swap out, una soluzione è non spostare un processo che sta facendo IO e spanane un altro, oppure usare un buffer del kernel.

Sui telefoni non esiste lo swaping, vengono in auito le **lifecycle dei contesti**. Il vantaggio di fare swapping è l'utilizzo del **paging swapping**, dove il swap in/out divento page in/out, dove vengono trasferite solo le pagine e non l'intero processo.

1.2 Esercizi

Example 1.4

- memoria fisica da 512MB
- allocazione minima 64B
- 128MB al SO
- tabella dei processi contiene: contiene indirizzo iniziale e size
- allocazione worst fit
- partizioni libere sono contenute in lista ordinata con dimensione descrescente
-

Si supponga ...

Example 1.5

Si descrivano brevemente

- tlb: 0.9 hit ratio
- page table a due livelli

- indirizzo in tre parti, 10 11 11 bit
-

Si fa una outer con cui si utillizzano 10bit, ed una inner con 11bit. Per calcolare il numero di inner ci sono due strade, il numero di byte preciso che contengono 100MB oppure si prende la prima potenza di 2 che contiene 100MB. Il resto delle righe delle outer overanno l'invalid bit settato ad 1.

2 Memoria Virtuale

Con **memoria virtuale** si indica che delle pagine esistono solo virtualemente ma non fisicamente, in questo modo si vuole supportare uno spazio di indirizzamento più grande dello spazio di indirizzamento fisico. Si parte dalla premessa che *un programma non bisogna di essere tutto in memoria per essere esguito*, si pensa ad un programma parzialmente caricato in memoria, dunque le mamoria di un programma non è più vincolato dalla grandezza della RAM, in questo caso si parla di **memoria virtuale**. Avere questa memoria virtuale porta una serie di vantaggi che volocizzano tutte le operazioni di IO tra la RAM e la memoria secondaria. La conseguenza è che viene introdotta la **demand paging**, ovvero che un pagina viene caricata solo quando è richiesta. Il **demand paging** è molto simile allo swapping, infatti il caso limite è il **lazy swapper** dove inizialmente non si ha nessun frame caricato, e ogni richiesta viene preso un frame.

3 Rust

Rust è un linguaggio di programmazione moderno, che non permette di avere undefined behaveiur e soprottuto è un linguaggio **statically strongly typed**, inlotre è capace di inferire i tipi di dati in modo molto potente. Rust fornisce delle *astrazioni a costo nullo*, ed è anche *interoperabile* con C. La memoria in rust è gestita dal programmatore solo in parte, il motivo è che il complilatore forza il programmatore a scrivere del codice senza errori, ed il compilatore si occupa di effettuare la gestione della memoria. Esempio di **dangling pointer** in C.

```c
typedef struct Dummy { int a; int b; } Dummy;

void foo(void) {
  Dummy *ptr = (Dummy *)malloc(sizeof(Dummy));
  Dumyy *alias = ptr;
  ptr->a = 2048;
  free(ptr);
  alias->b = 25;  // WARNING!!
  free(alias);
}
```

Grazie alle feature di rust si possono prevenire:

- **dangling pointers**
- **doppi rilasci**
- **corse critiche**
- **buffer overflow**
- **iteratori invalidi**
- **overflow aritmetici**

Inoltre rust propone delle convenzioni per aiutare il programmatore ad avere del codice ideomatico.

Rust parte dall'assunzoine che un valore può essere **posseduto** da una singola variabile, se la variabile esce al di fuori del suo scope il valore viene deallocato. Quando si fanno delle assegnazioni l'**ownership** del valore viene trasferito alla variabile alla quale è stata fatto l'assegnazione, inoltre è anche possibile **dare in presetito** (**borrow**) un valore; l'insieme di queste idee sono alla base della sicurezza che fornisce Rust. I puntatori sono tutti controllati in fase di compilazione, oltre all'**ownership** ogni puntatore ha un **tempo di vita** (**lifetime**), tutti gli accessi al di fuori della lifetime sono negati dal compilatore, in alcuni casi non può essere fatto l'infering della lifetime e questo andrà specificato esplicitamente usando la notazione `<'...>`. La **sicurezza dei thread** è incorporata nel sistema dei tipi e anch'essi hanno una lifetime. Rust inoltre non ha stati nascosti, come in java con le eccezioni.

I due strumenti per gestire l'ambiente di sviluppo in rust sono:

- `$ rustup`: installer and updater di rust;
- `$ rustc`: compilatore;
- `$ cargo`: gestione dei progetti e delle dipendenze;

I comandi di base di `cargo` sono:

- `$ cargo new project-name`
- `$ cargo new --lib library-name`
- `$ cargo build`
- `$ cargo run`

Terminologia in rust:

- **crate**: unità di compilazione, crea un eseguibile o una libreria;
- **create root**: radice del progetto e solitamente contiene il `main.rs`, oppure `lib.rs` in caso di librerie;
- **module**: il progetto è organizzato in *moduli*, se si crea un modulo come cartella il file .rs andrà nominato `mod.rs`, all'interno possono essere presenti moduli interni;
- **package**:

Come in kotlin i blocchi condizionali hanno dei valori di ritorno. In rust non esiste l'eriditarità, però esistono i **trait**, che sono simili alle interfacce, offrendo la possibilità di poter implementare dei metodi, ad esempio alcuni di questi sono `Display` e `Debug`, ogni tipo può implementare più tratti.

I tipi elementari di rust sono:

- numerici: `i8, i16, i32, i64, i128, isize` (valore nativo del processore).
- numerici senza segno: `u8, ...`
- naturali: `f32, f64`
- logici: `bool`
- caratteri: `char` (32 bit, rappresentazione **Unicode**)
- unit: `()` rappresenta una tupla di 0 elementi, corrisponde a `void` in C/C++

Per rappresentare le stringhe invece si use la codifica `utf-8`, che codifica i caratteri unicode in blocchi di 8 bit, la codifica utilizza la maggior parte dei primi 8 bit come caratteri standard, se si vogliono usare simboli più strani si combina un byte con il successivo, se il carattere è ancora più raro si utilizzano più byte, fino ad arrivare a 4.

Una **tupla** è una struttura che permette di contenere più tipi di valori, per accedere alla posizione corrispondente si utilizza la notazoine `<variabile>.<numero>`.

Rust ha un meccanismo al suo interno per rappresentare i puntatori, infatti quando viene creato un valore in una funzione si può decidere di passare l'ownership alla funzione chiamante, che poi ha la responsabilità di liberla oppure di passare ancora l'ownership, in rust esistono tre tipi di puntatori:

- **reference**;
- **box**;
- **native pointer**;

L'utilizzo dei puntatori nativi (che sono gli stessi del C), è possibile solo in un blocco `unsafe { ... }`.

Le **reference** sono puntatori senza possesso e possono essere assegnati con `&`:

```
let r1 = &v;
let r1 = &mut v;
```

In questo caso `r1` borrows il valore `v`, per acceder al valore si utilizza la notazione per **derefenziare**: `*r1`. Quando si crea un puntatore che `&` si può un creare un puntatore in sola lettura, si vuole anche scrivere all'indirizzo del puntatore si deve usere la keyword mut: `&mut v`, l'accesso in scrittura è *esclusivo*, ovvero può essere assegnato ad una sola variabile e quando viene asseganto in modo mutabile nussun altro può utilizzarlo, infatti è *mutuamente esclusivo*.

```
fn main() {
  let mut i = 32;

  let r = &i;     // r is of type "int ref";
  println!("{}", *r);

  i = i+1;        // ERROR: i was borrwed
  println!("{}", *r);
}
```

```
fn main() {
  let mut i = 32;

  let r = &mut i;
  println!("{}", i);  // ERROR: i was mutably borrowed

```

```
  *r = *r+1;
  println!("{}", *r);
}
```

In questo caso la variabile `i` non è accessibile in alcun modo per tutta l'esistenza di `r`.

In altre situazioni c'è l'esigenza di allocare un dato, per prolungare la sua vita all'infuori della funzione in cui viene creato, l'equivalente della `malloc` in rust è `Box<T>` che alloca un parte di memoria presa dall'*heap*, per creare un valore che che si vuole venga conservato, si passa a `Box` il valore che voglio si venga conservato:

```
let b = Box::new(v);
```

Rust è in grado di fare l'infer del tipo di `v` e riesce ad allocare lo spazio necessario, quando `b` non sarà più visibile il `Box` verrà liberato dalla memoria.

```
fn useBox() {
  let i = 4;                      // i si trova nello stack
  let mub b = Box::new((5, 2));   // *b si trova nell'heap, b si trova nello stack

  (*b).1 = 7;

  println!("{:?}", *b);   // (5, 7)
  println!("{:?}", b);    // (5, 7)
}
```

Quando `println!` si trova un puntatore lo dereferenzia automaticamente. Quando si arriva alla fine della funzione `a` e `b` vengono rimossi dallo stack, dato che il valore a cui puntava `b` non possiede un owner in vista quel valore viene rimosso dell'heap.

```
fn makeBox(a: i32) -> Box<(i32,i32)> {
  let r = Box::new((a, 1));
  return r;
}

fn main() {
  let b = makeBox(5);
  let c = b.0 + b.1;
}
```

In questo caso il valore l'ownership del valore puntato da `r` passa la sua ownership a `b` che si trova nel `main`, quando il `main` termina e la vita di `b` finise il valore puntato viene anch'esso liberato.

I **Puntatori nativi** sono `*const T` e `*mut T`, il `*` indica un puntatore nativo e possono essere utilizzati solo nei blocchi `unsafe ...` .

Rust supporta nativamente anche gli **array**, in rust gli array sono composti da dati omogenei allocati contiguamente nello stack. Si possono inizializzare nel seguente modo:

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
let b = [0; 5];     // array lungo 5, inizializzato a 0
```

In rust gli array hanno la conoscenza di quanto sono lunghi. Se si cerca di accedere ad indice fouri dal range di utilizzo rust fa `panic`, se `panic` viene lanciato in nel thread principale il programma termina, se viene lanciato in un altro thread viene terminato il thread. Per accedere all'array si utilizzano gli **Slice** (invece di utilizzare i puntatori), che vengono creati come riferimento ad una porzione di un array:

```
let a = [1, 2, 3, 4];
let s1: &[i32]  = &a; // s1 contiene 1, 2, 3, 4
let s2 = &a[0..2]     // s2 contiene 1, 2
let s3 = &a[2..]      // s3 contitne 3, 4
```

Per questa sua natura viene detto **fat pointer**, perchè oltre a contenere il puntatore contiene anche la lunghezza. Come nel caso degli array se si va al di fuori del range viene generato un `panic`.

Il problema degli array è che quando vengono creati con una dimensione, questa rimane fino alla del loro lifetime, un array variabile può essere creato con `Vec<T>` che rappresenta una sequenza di oggetti di tipo `T`, al contrario degli array gli oggetti presenti in `Vec` sono allocati nell'heap.

```
fn useVec() {
  let mut v: Vec<i32> = Vec::new();

  v.push(2);
  v.push(4);

  let s = &mut v;
  s[1] = 8;
}
```

Quando viene inizializzato `Vec` nello stack vengono inseriti 3 parametri: pointer, size, capacity. Appena viene fatto push viene allocata della memoria nell'heap, e i parametri size e capacity vengono aggiornati. Se vuole creare uno slice `s`, nello stack viene messo un fat pointer, ovvere il puntatore all'heap ed il size corrente. Quando `Vec<T>` viene inizialmente inizializzato, nel campo puntatore viene inserito l'allinemento che il tipo `T` dovrà avere in memoria.

Lo stesso discorso fatto per i `Vec` vale per gli oggetti di tipo `String`. Quando si crea una stringa tra le virgolette viene creato un `&str` e non una `String`, questo dato andrà inserito nella sezione del programma dei *dati inizializzati*, al contrario delle `String` queste non modificabili, se si vuole utilizzare una stringa si utilizza `"...".as_str()`, che ritorna uan stringa allocata nell'heap.

```
fn main() {
  let hello: &str = "hello";

  let mut s = String::new();    // il valore (futuro) puntato da s viene allocato
    nell'heap

  s.push_str(hello);            // l'heap viene inizializzato una capacita' ed un
    size uguale al size della stringa puntata da "hello"
```

```
  s.push_str(" world!");
}
```

In rust le keyword `let` e `let mut` sono delle istruzioni, mentre un blocco racchiuso tra `...` è un espressione e quindi ha un valore di ritorno, anche il costrutto `if` è un espressione, **il valore viene ritornato a patto che l'ultima istruzione del blocco non abbia un punto e virgola**. Si può asseganre un etichetta con `'<nome-etichetta>:` ad un espressione per poter specificare il ritorno:

```
fn main() {
  'outer: loop {
    'inner: loop {
      //...
      continue 'outer;
    }
  }
}
```

Per leggere gli argomenti passati da linea di comando si utilizza:

```
fn main() {
  let args: Vec<String> = args().skip(1).collect();

}
```

Si può utilizzare la libreria `clap` per fare il parsing degli argomenti. per farlo si può andare nel file `.toml` ed inserire la dipendenza.

Per fare IO da console si utilizza `stdin`:

```
fn main() {
  let mut s = String::new();

  if io::stdin().read_line(&mut s).is_ok() {
    println!(..., s.trim());
  } else {
    println!("Failed!");
  }

  // Oppure...
  io::stdin().read_line(&mut s).unwrap();
  println!(..., s.trim());
}
```

3.1 Ownership

Il concetto dell'ownership in rust permette al **borrow checker** di controllare l'utilizzo corretto dei puntatori, in rust *un valore può essere posseduto da una e una sola variabile*, il fallimento di questa condizione comporta ad un fallimento della compilazione. Possere un valore vuol dire *occuparsi del suo rilascio*. Se una variabile mutabile viene

riasseganta, ed il valore precedente implementava il tratto `Drop` esso viene prima liberato e poi avviene l'assegnazione. Quando si fa un assegnazione da una variabile all'altra: `a = b`, il valore posseduto da `b` viene spostato in `a`, questa operazione viene detta **movimento**, ed il valore non è più accessibile da `b`, dal punto di vista fisico il movimento è una *copia*.

```
fn main() {
  let mut s1 = "hello".to_string();
  println!("s1: {}", s1);

  let s2 = s1;
  println!("s2: {}", s2);

  // s1 non e' piu' accessibile
}
```

Il movimento è il comportamento standard in rust, se però un tipo implementa il tratto `Copy` quando avviene un'essegnazione entrabe le variabili possono continuare ad accedere al valore, però i tratti `Copy` e `Drop` sono mutuamente esclusivi. In generale i tipi copiabili sono i: numeri, booleani, le reference `&`, ..., mentre i `&mut` non sono copiabili. Il tratto `Copy` può essere implementato solo se il tipo implementa il tratto `Clone`, che permette di fare una clonazione dell'oggetto **in profodità**, inoltre la clonazione è implementata dal compilatore, mentre copia e movimento sono gestite dal compilatore e veine implementata con `memcpy()`.

```
fn main() {
  let mut s1 = "hi".to_string();
  let s2 = s1.clone();  // viene creato nello stack una nuova variabile e nell'
    heap un nuovo valore
  s1.push('!');

  println!("s1: {}", s1); // hi!
  println!("s2: {}", s2); // hi
}
```

In rust il comportamento di base è il movimento, mentre in C l'unico comportamento è quello della copia, mentre in C++ si può copiare e si può muovere ma questo va scritto esplicitamente.

```
struct Test(i32);

impl Drop for Test {
  fn drop(&mut self) {
    println!("Dropping Test({}) @ {:p}", self.0, self);
  }
}

fn alfa(t: Test) {
  println!("Invoking alfa() with Test({}) @ {:p}", t.0, &t);
}

```

```
fn main() {
  let t = Test(12);

  alfa(t);

  println!("Ending...");
}
```

Quello che succede in questo caso è che il metodo di `drop()` viene chiamato scritto prima di `Ending...`, il motivo è che `t` viene mossa all'interno della funzione `alfa()`. Quando non si vuole passare la proprietà del valore ad una funzione si può decidere di **prestarglielo**, se si vuole fare ciò l'argomento della funzione deve essere una reference.

```
fn alfa(t: &Test) {
  println!("Invoking alfa() with Test({}) @ {:p}", t.0, t);
}

fn main() {
  let t = Test(12);

  alfa(&t);

  println!("Ending...");
}
```

Se si vuole modificare il valore all'interno della funzione si cambiare la firma in: `fn alfa(t: &mut Test)`.

In rust i riferimenti permettono il prestito dei valori di una variabile in solo scrittura, mentri i riferimenti mutabili permettano lettura e scrittura, ma se viene prestato la variabile originale non può leggere. A differenza del tipo di dato che si maneggia, si possono avere 3 tipi di puntatori: **reference**, **fat pointer**, **double pointer**. Quando il compilare ha tutte le informazioni sul tipo di dato ho bisogno di una reference semplice, se tuttavia non è in grado di inferire tutti i dati ricorre ad un *fat pointer* (come nelle slice), che contiene il puntatore all'oggetto e la dimensione dell'oggetto. Nel caso di un tipo che contiene delle implementazioni, viene usato un double pointer, dove il primo punta all'oggetto, ed il secondo punta alla **vtable**, questa è una tabella che contiene i puntatori alle funzioni che vengono implementati per un tratto, ed esiste una vtable per ogni tratto implementato.

```
let f: File = File::create("text.txt");
let r3: &dyn Write = &f;   // double pointer
```

Il borrow checker garantisce che tutti gli accessi ad un riferimento avvengano solo all'interno di una **lifetime**, in alcuni casi il complilatore non è in grado di fare l'infering della lifetime, per assegnare una lifetime ad un riferimento si utilizza la sintassi:

```
&'a Type
```

L'unico nome riservato è `&'static Type`, questo vuol dire che la lifetime di quella reference sopravvive per l'intera durata del programma.

3.2 Slice

Uno slice è una vista su una sequenza contigua di elementi, gli slice in queanto riferimento non possiedono il valore, oltre ad essere dei fat pointer.