Notes Software Engineering

Brendon Mendicino

June 13, 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, sia il SO che i programmi verranno caricati in memoria, per evitare che ogni processo abbia una vista al di fuori del suo scope, si possono usare dei controlli a livello della CPU, esistono dei registri chiamati **base** e **limit**, che attraverso dei meccanismi riescono a effettuare la protezione della memoria. Esiste anche un altro problema che è quello della **relocation**, che consiste nel come mappare gli indirizzi delle varie istruzioni di salto e dei vari puntatori una volta che il programma viene caricato in memoria. Se si utilizzassero degli indirizzi statici (creati solo in fase di compilazione prendendo come riferimento d'inizio del programma l'indirizzo 0), una volta che il programma viene caricato in memoria le istruzioni punterebbero sempre allo stesso indirizzo, ma i programmi non partono tutti dall'indirizzo 0, sorge dunque il probelema di come gestire le istruzioni che fanno riferimento ad altre parti del programma. Immagginiamo che gli indirizzi vengano creati mentre il programma viene caricato in memoria, cosa succede se quel programma viene mosso in un altra parte della memoria? I rifermenti degli indirizzi sarebbero tutti sbagliati o addirittura potrebbero puntare a pezzi di altri programmi in esecuzione. Per questo motivo quando si utilizzano valori di indirizzi, essi vengono sommati al **base register**, mentre le boundry del programma in memoria vengono salvate nel **limit register**, grazie a questa tecnica **hardware** è possibile scrivere i programmi come se gli indirizzi del programma partissero da 0.

1.1 Indirizzamento

Gli indirizzi vengono rappresentati in modo diverso a differenza della fase di vita di un programma: durante la compilazione vengono considerati come simbolici, durante la compilazione agli indirizzi è fatto il **bind** ad un indirizzo relativo (base) in modo da poter essere rilocato (questa somma è fatto dall'hardware), durante il linking o durante il loading. Il **binding** si può fare:

- **in compilazione**: fatto quando molto semplice come nei sistemi embedded, 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 e del programma, prima di arrivare all'address bus viene tradotto in indirizzo fisico, esiste dunque una dicotomia tra indirizzo logico e fisico, in questo modo quando si scrive un programma, l'indirizzo parte sempre da 0. Esistono due tipi d'indirizzamento che sono di tipo logico, dove l'intervallo degli indirizzi utilizzabili è logico, e 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 e un comparatore. Il **relocation register** va a sostituire il **base register**.

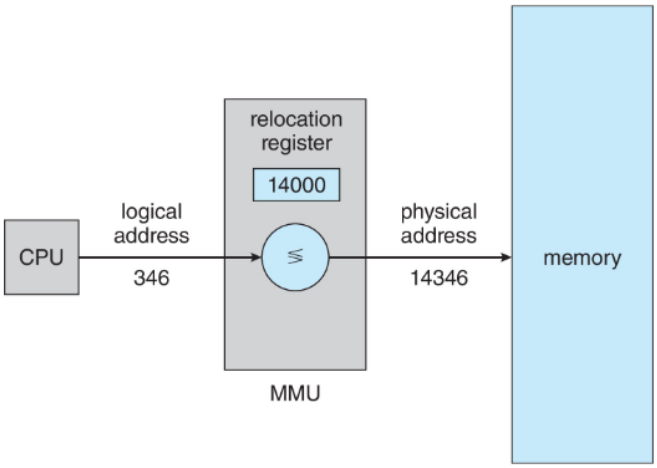


Figure 1: Basic MMU

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** usa una chiamata fasulla che prima chiama la load della funzione linkata dinamicamento e poi la invoca, questi piccoli pezzi di codice vengono detti **stub**.

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

1.2 Allocazione in Memoria

Come si alloca memoria per un programma (immagine)? La più semplice è l'allocazione contigua, dove si vede la RAM separata in due partizioni, una per il SO e una per i processi (indirizzi più bassi), per caricare un processo si parte da un indirizzo d'inizio e un indirizzo di fine, la MMU vista prima funziona solo con i casi di allocazione contigua (basilare). Una tecnica più efficente è l'**allocazione contigua con partizione variabile**, che consiste in: quando ci sono più buchi ci sono delle politiche differenti per inserire nuovi programmi:

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

La **frammentazione** è definita come la *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 bisogno di **compattazione**, il SO sposta i pezzi e poi si riparte, partiziona la memoria in zona libera e zona occupata, per fare una **deframmentazione** (o compattazione) vuol dire 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 sta effettuando delle operazioni di IO, una soluzione per il problema dell'IO è il **latch job** in cui solo la parte che sta facendo IO non può essere spostata, oppure si utilizza un **buffer IO del kernel**, si fa IO solo in buffer del sistema operativo.

Definition 1.1 – Backing Store

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 contigua, al suo posto si utilizzano della **pagine** che sono l'unità minima di allocazione e di trasferimento, questo risolve il problema della frammentazione.

Definition 1.2 – Pagine

Le partizioni della memoria logica.

Definition 1.3 – Frame

Le partizioni della memoria fisica.

Definition 1.4 – Blocco

Le partizioni della memoria secondaria.

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 una **frame table**, dove ogni riga corrisponde a una mappatura. Anche utilizzando la paginazione si ha frammentazione interna.

Un indirizzo generato dall CPU si divide in:

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

Un indirizzo è composto da: $(\underbrace{p-d}_{\text{parte logica da sostituire}} , \underbrace{d}_{\text{accesso interno al frame}})$

Example 1.1

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

Nelle righe della tabella si aggiungono dei bit in più per rappresentare delle informazioni aggiuntive rispetto al frame:

- **parte di protezione**: specifica che la parte di codice non può essere scritta;
- **modify bit**: pagina modifica;
- **page present/absent**: pagina presente in memoria;
- **page referenced**;
- **caching disabled**;

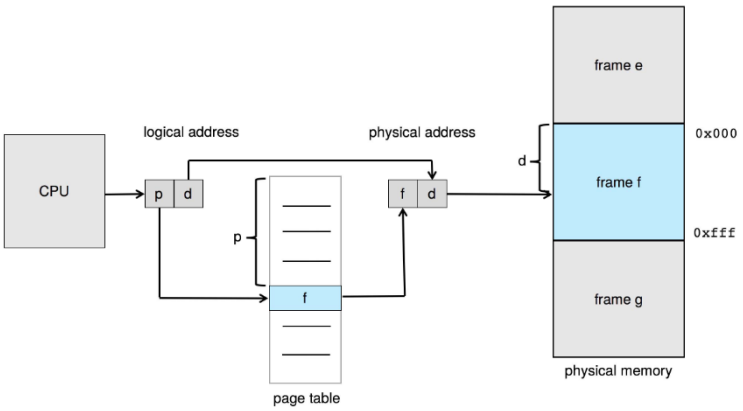


Figure 2: Page Table

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 può spostare questa tabella all'interno della CPU (le operazioni di accesso alla memoria sono costose), si usa la **Translation Look-aside Buffer** (TLB), un tipo di memoria in cui si accede per contenuto (memoria associativa). 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 i processi si contendono la TLB. Quando avviene un **TLB miss** si reinserisce la pagina nella TLB e poi si ritenta, utilizzando una politica scelta. Anche usando una TLB si mantiene comunque la page table in RAM in caso di miss, in modo da recuperare il frame e inserirlo nella TLB.

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;

Nella seconda parte dell'equazione l'accesso alla memoria è raddoppiato perché si deve prima recupare il frame dalla page table e poi si può accedere alla memoria.

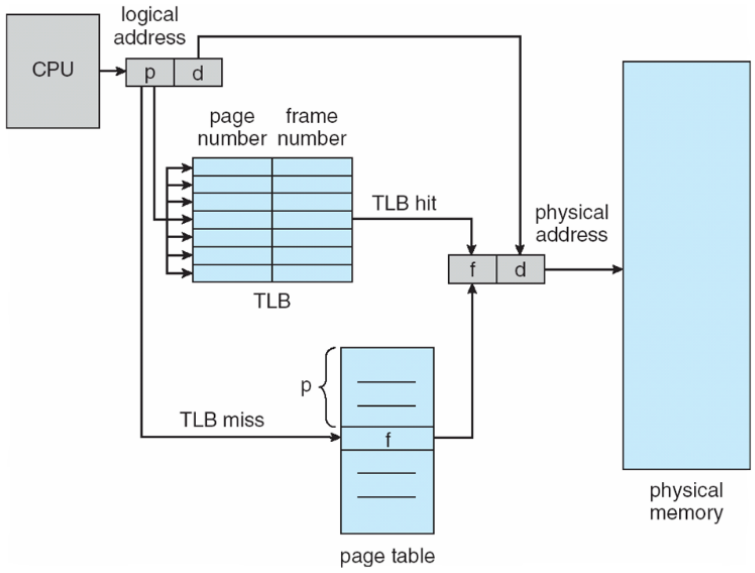


Figure 3: Paginazione Con TLB

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 di codice che indica che il frame è in solo scrittura. Anche un altro bit associato è il **validity** che indica quando un frame è valido, ovvero quando c'è ed è associata a un indirizzo in RAM, o quando non è valida che vuol dire che il frame non esiste o che non è associato a un indirizzo, se si cerca di accedere un frame non valido viene lanciata una **trap**.

La tabella della pagine 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 4 byte, quindi una page table è grande 4MB. Adesso è possibile avare:

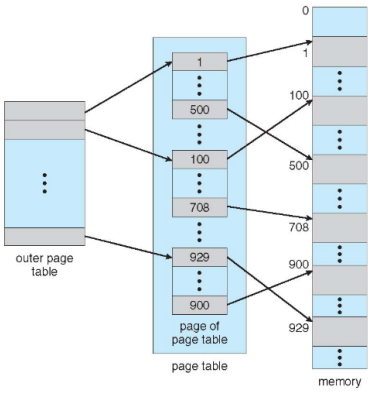
- **page table contigua**:

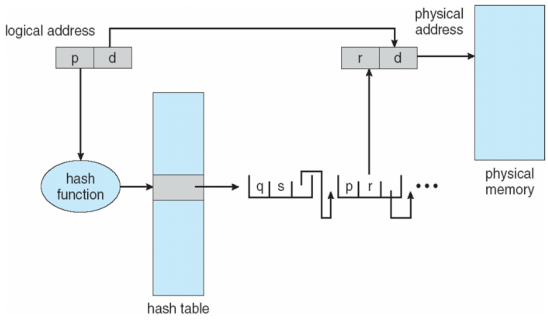
Se la dimensione è critica (troppo grande):

- page table gerarchica;

- hash page table;
- inverted page table;

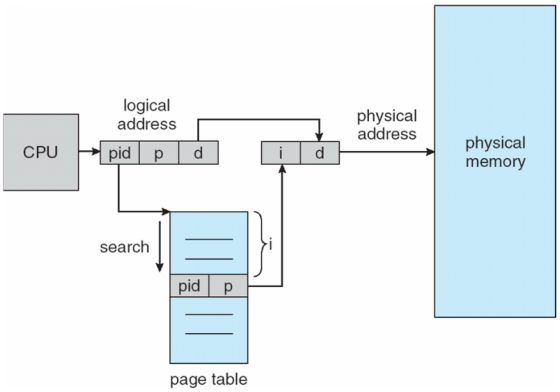
La **la page table gerarchica**, la page table viene divisa in blocchi più piccoli non contigui, si usa una page table di livello superiore che porta alle page table di livello inferiore, questi blocchi devono rimanere contigui, anche se diventano molto piccoli. Il page number viene diviso in due parti, 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 diventa molto grande si può usare solo una parte dell'indirizzamento se l'eseguibile è molto piccolo. In Linux la gestione della page table viene fatta con tre livelli, che vengono chiamati rispettivamente: **Page Gobal Directory** (`pgd_t`), **Page Midlle Directory** (`pmd_t`), **Page Table Entry** (`pte_t`).





La **hashed page table** permette di creare una tabella di hash, con una funzione di hash direttamente implementata in hardware che dato p in input ritorna l'hash a cui è associato f. Vanno implementate delle liste di collisione, e quindi vieni immagazzinato che p che è la chiave di accesso e come suo valore f. Usando una tabella di hash si potrebbe pensare anche di usare una page table condivisa tra tutti i processi, ma a quel punto andrà inserito l'ID del processo all'interno della chiave primaria, perché in molti sistemi gli indirizzi virtuali dei processi possono essere sovrapposti.

La **inverted page table** è una tabella condivisa tra tutti i processi, la sua grandezza viene dimensionata rispetto alla grandezza della RAM e non sulla grandezza dei singoli processi, all'interno ogni frame fisico è associato a una pagina. In questo caso l'indice della tabella corrisponde all'indirizzo del frame e la entry contine il PID e la pagina, il problema è che si vuole la traduzione da pagine a frame e non il contrario infatti non si ha nessuna informazione su dove si trovi la entry che contiene la pagina ed il PID, la prima strategia che si può utilizzare è la scansione lineare, ma è molto inefficente, per rendere l'accesso più veloce viene messa una tabella di hash prima di arrivare alla tabella invertita (implementate in hardware solitamente), garantendo cosí accessi molto veloci.



1.3 Swapping

Si supponga che a un certo punto si voglia far partire un processo e che la memoria sia piena, cosa si può fare? Una soluzione è lo **swapping**, ovvero la memoria associata a quel processo viene messa nel backing store (in modo temporaneo), che in linux è la partizione di swap, e viene caricato il nuovo processo al suo posto, grazie allo swapping si possono avere più processi attivi nel sistema occupando più memoria della grandezza dela RAM.

Example 1.3

Quanto 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 processo sta facendo IO non si può fare lo swap out, una soluzione è non spostare un processo che sta facendo IO ma spostarne un altro, oppure usare un buffer del kernel.

Sui telefoni non esiste lo swaping, vengono in auito le **lifecycle delle Activity e dei Fragment**. Il vantaggio di fare swapping è l'utilizzo del **paging swapping**, dove lo swap in/out diventa page in/out, che consiste nel trasferire solo le pagine e

non l'intero processo.

1.4 Esercizi

Esericizi:

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 virtualmente ma non fisicamente, in questo modo si vuole supportare uno spazio d'indirizzamento più grande dello spazio d'indirizzamento fisico. Si parte dalla premessa che *un programma non bisogna di essere tutto in memoria per essere eseguito*, si pensi a un programma parzialmente caricato in memoria che contiene solo i componenti necessari a lui in quel momento, in questo caso la grandezza di un programma non è più vincolato dalla grandezza della RAM, si parla allora di **memoria virtuale**. Avere questa memoria virtuale porta una serie di vantaggi che velocizzano 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 a ogni richiesta viene preso un frame.

Quasi tutto il meccanismo si basa sul *validity bit*, infatti quando si cerca di accedere una pagina che non è valida il SO lancia una **trap**, i casi sono due: il frame non esiste proprio, oppure il frame c'è ma deve essere recuperato, infatti il programma viene comunque caricato in memoria (backing store), perché lo spazio è tanto e in questo modo è più veloce recuperare le informazioni.

Il **page fault**, è una *trap* che viene scatenata quando si cerca di accedere a una pagina con invalid bit, il processo viene bloccato e si passa in modalità kernel, il sistema operativo agisce in due modi:

- **invalid refernce** $\implies$ abort;
- **non in memoria** $\implies$ viene recuperato il frame;

Quando il frame viene portato in memoria dal disco, la page table viene aggiornata e viene fatto ripartire il processo. Un caso estremo di esecuzione è quello di avere nessuna pagina disponibile e poi caricarle in modo incrementale.

Quanto costa una page fault?

1. Trap al sistema operativo, vengono salvati tutti i registri e lo stato del processo;
2. gestisci l'interrupt;
3. l'OS deve verificare che quel frame esiste;
4. recupera il frame facendo IO;
5. quando si fa il trasferimento, l'OS viene fatto uscire dal processore e un altro processo inizia a girarare;
6. viene tirata un eccezione al termine dell'IO;

7. viene fatto lo swapping;
8. il validity bit viene settato a valido;
9. si riparte dall'istruzione che ha causato il fault;

L'IO è la parte più onerosa, per calcolare l'EAT:

- p = probabilità page fault
- f = page fault overhead;
- s_{in} = swap page out;
- s_{out} = swap page in;
- M = memory access

$$\boxed{EAT = p \cdot (f + s_{out} + s_{in}) + (1 - p) \cdot M}$$

Example 2.1

- memory access time = $200ns$;
- avg page-fault service time = $8ms$;

$$EAT = (1 - p) \cdot 200ns + p \cdot 8ms = 200ns + p \cdot 7999800ns$$

Il page fault è molto oneroso, anche con p molto piccoli i tempi sono comunque molto lenti. L'approccio migliore è ottimizzare il tempo del service e ottimizzare p.

2.1 Copy-On-Write

Il **Copy-on-write** è una tecnica usata per velocizzare le oparioni di clonazione, in cui i frame sono inizalmente condivisi da due processi e si genera una copia di un frame solo quando uno dei due decidono di scrivere su un frame, questo approccio viene utilizzato quando viene chiamata una `fork()`, infatti la pagine dei due processi non vengono copiate ma sono condivise aumentando di molto le prestazioni, in Linux per realizzare il COW viene settato il bit di scrittura della pagina (`VM_WRITE`) a falso, in questo modo quando uno dei due cerca di scrivere la TLB fará un page fault (perché quella parte di memoria è stata dichiarata non scrivibile), durante l'interrupt viene capito che la pagina è un **cow mapping** (grazie ad un altro bit detto `VM_MAYWRITE` che specifica se la pagina può diventare scrivibile), e poi la copia.

2.2 Page Replacement

Che succede se non ci sono frame disponibili? Un processo che richiede una nuova pagina, **rimpiazza** (**page replacement**) una pagina non in uso che viene salvata sul disco, è possibile rimpiazzare una pagina dello stesso processo o anche di altri processi. Questa caratteristica va aggiunta nella funzione di gestione del page fault all'interno del sistema operativo, in questo caso deve essere aggiunto il **modify/dirty bit** per gestire se delle pagine vengono modificate, se una pagina scelta ha il modify bit settato significa che il frame ha un valore diverso rispetto a quello sul disco, in questo caso prima di rimpiazzarla deve essere copiata, altrimenti i suoi cambiamenti verranno persi. Se non esiste un frame libero si deve scegliere un **victim frame** da rimpiazzare, quindi oltre ai tempi per portare una pagina in memoria si deve anche portare una pagina su disco, questo processo viene detto **Page replacement**.

2.3 Algoritmi di Page Replacement

(il processo per il momento ha un numero fisso di frame), per misusare le prestazioni degli algoritmi di page replacement utilizziamo una **string reference**, è semplicemente una sequenza di numeri di pagine che vengono richieste in sequenza da un processo. Viene definita la **page fault frequency**:

Definition 2.1 – Page Fault Frequency

$$f(A,m) = \sum_{\forall w} p(w)\frac{F(A,m,w)}{len(w)}$$

- A = algoritmo;
- w = string reference;
- p = probability;
- m = number of availably page frames;
- F = number of page fault generated;

Gli algoritmi proposti per il rimpiazzo sono:

- **FIFO**: si mantiene un lista di pagine e se quendo il processo richiede una pagina e non si trova posto si utilizza una semplice tecnica FIFO, esiste però un problema descritto dalla **Anomalia di Belady** che per quanto controintuitivo dice che aumentando il numero di frame disponibili aumenta la frequenza dei page fault.
- **Optimal Algorithm**: un algoritmo ottimale sarebbe quello di rimpiazzare la pagine sono usate il minor numero, per mettere in pratica questo algoritmo si

dovrebbe conoscere il futuro, questo algoritmo viene solo usatato per misurare le performance di un altro algoritmo, essendo questo il caso ottimale.

- **Least Recently Used (LRU)**: viene ripiazzata la pagina usata per ultima. In questo caso per ogni pagina esiste un tempo associato alla pagina, questo è molto oneroso dal punto di vista hardware, infatti non viene adottata.

- **LRU algorithm**: ogni pagina ha un contatore (che tiene traccia del tempo), per ogni page fault si deve fare una ricerca del minimo. Esiste anche implementazione utilizzando uno stack, dove si tiene una lista doppiamente linkata dei numeri di pagina, dove ogni volta che si refernzia una pagina si mette in cima allo stack. Entrambi gli algoritmi anche con dell'hardware dedicato rimangono molto lenti.

- **Second-chance algorithm** esiste poi una famiglia di algoritmi che cerca di approssimare il LRU con performance molto migliori e vengono detti **LRU Approximation Algorithm**, in questo algoritmo ogni frame ha un **reference bit**, inizialmente è zero e quendo si fa una reference alla pagina viene messo a 1, esiste un momento nel passato in cui i reference bit sono stati azzerati, la victim viene presa nel gruppo dei frame con reference bit a 0, si usa una fifo, quando si incontra una pagina con 1 viene settato a 0, se si incontra uno 0 allora viene preso il victim.

- **Enhanced Second-Chance Algorithm**: si utilizza una combinazione di reference e modify per scegliere il frame:

  – (0, 0): ...

  – ...

- **Counting Algorithm**:

  – **Least Frequently Used (LFU):**

  – **Most Frequently Used (MFU):**

Esercizi:

Example 2.2
...

- **Page-Buffering Algorithm**: La vittima viene messa nel *free pool*, ..., il SO potrebbe dare il controllo all'applicazione per la gestione della memoria.

Allocazione fissa:

- si da a tutti la stessa misura (equal);

- si da di più a chi ha più bisogno (proporzionale): ad ogni processo si danno i frame in maniera proporzionanta.

Global vs Local allocation: un processo può scegliere una frame vittima solo tra i suoi anche tra quelli degli altri? **Locale** visto nell'esempio di sopra. **Globale**, esista una lista di free frame, si fa in modo che questa lista non arrivi mai a 0, l'idea di massima è che superata una soglia si iniziano a scegliere delle vittime, fino ad arrivare di nuovo ad un altra solgia, che superata permette il reinserimento dei frame.

... NUMA ...

Le prestazioni migliori si ottengono quando un processo gira su un unico processore.

Quando il sistema inizia a saturare la memoria in modo aggressivo, il SO ha due possibilità: bloccare il sistema ed aspettare che un processo termini, oppure inziare a rimuovere frame, il **thrashing** è tenere il processore sempre attivo con dei processi.

Come si possono realizzare pochi page fault in modo statistico? Per minimizzare il numero di page fault si sfrutta il principio di località, basandosi su questo concetto si 'e sviluppato il modello di *working set* (l'insieme di pagine su cui si lavora), si definisce una finestra temporale che in cui si definisce Δ (finestra temporale e il range di pagine a cui si fanno accesso), si suppone che ci sono 10 mila accessi in memoria, WSS_i (dimensione del set di pagine a cui si è fatto accesso, del processo P_i) = numero totale di pagine in cui si è fatto riferimento nei Δ precedenti. Ci aspettiamo:

- se Δ troppo piccolo: non si riesce a contenere l'intero programma;
- Δ troppo grande si tengono troppe pagine in memoria;

Il **working set model**, dato un intervallo Δ si contano gli id delle pagine, se una pagina non è presente nell'intervallo viene rimosso dal *working set*, il difetto è che si deve generare un'istruzione per buttare via una pagina quando esce dal working set e soprattutto si deve tenere traccia di chi deve essere buttatto fuori. Questo non si può fare in modo esatto, si usa questo principio: si tiene nel *recidence set* un po più del working set, si suppongo che un timer interrompa ogni $\Delta/2$, ogni ogni pagina si tengono 2 bit di riferimento (ref bit 1, ref bit 2), uno dei due viene settato automaticamente dall'HW, ad ogni intervallo per ogni pagina a cui si fa accesso viene settato il ref bit ad 1, il ref bit che viene selezionato viene scambiato ad ogni intervallo, quando si cambia intervallo si può decidere di rimuovere le pagine che hanno $(0, 0)$, questo approccio si può estendere con più ref bit.

Il problema del working set è che non guarda quante volte si fa accesso ad una pagina, si stabilisce una **frequenza di page-fault**, se il numero di page fault è alto allora si allarga la window, se il numero di page fault è basso allora la window si restringe. Funzionamento dell'algoritmo:

- si attiva l'algoritmo solo quando c'è un page-fault;

- quando si attiva si misura il tempo intercorso rispetto all'ultimo page fault τ, per vedere se si sta andando bene o male;
- se τ è minore di una costante c si prende un nuovo frame e lo si aggiunge al *resident set*;
- se τ è maggiore di c si rimuove dal *resident set* tutte le pagine con reference bit a zero e si settano i reference bit a zero della pagine rimanenti;

Normalmente il working set dovrebbe coincidere con i monti nel queale il SO ha delle spike di utilizzo dei frame, dovuti a vari motivi.

Il kernel è allocato in modo contiguo, il motivo è che la maggior parte delle strutture dati presenti al suo interno sono allocate in modo contiguo, ad esempio la **page table**.

Esempio di due tipi di allocazione:

- **Buddy System**: L'idea è che va bene l'allocazione contigua va bene ma l'allocazione può essere solo una potenza di 2, questo causa un sacco di frammentazione interna.
- **Slab Allocator**: L'idea è allocare per pagina ma si prendono in modo contiguo. In linux c'è una struttura di slab allocation.

Per diminuire il numero di page fault, si può allocare un po di pagine iniziali ed essere bravi ad indovinarle, questa tecnica viene detta **prepaging**.

Quanto deve essere grande una pagina? Per diminuire la frammentazione si vorrebbere pagine piccole, per diminuire la dimensione della page table si volgiono pagine grandi. Un altro parametro è dato dalle operazioni di IO, infatti una volta che si legge dal disco (molto lento), tanto vale prendere tanta roba. La grandezza prende in considerazione anche la *località* dei dati. Se si guarda l'efficienza della TLB avere poche entry è molto meglio (pagine grandi). Solitamente le pagine hanno una grandezza compresa tra 2^{12} e 2^{22} (la grandezza è sempre un multiplo di 2).

Il **TLB reach**, è la percentuale di spazio che la TLB vede sulla memoria (quantità di memoria vista dalla TLB)

$$TLBreach = TLBsize \cdot PageSize$$

La struttura dei programmi ha anche un effetto sui page fault.

Example 2.3

Si supponga di voler azzerare una matrice in un doppio loop annidato.

```
for (int i = 0; i < 128; i++)
  for (int j = 0; j < 128; j++)
    matrx[i][j] = 0;
```

```

```

```
for (int j = 0; j < 128; i++)
  for (int i = 0; i < 128; i++)
    matrx[i][j] = 0;

```

Supponendo che una riga stia all'interno di una singola pagina, il primo programma causa 128 page faults, mentre il secondo causa 128×128 page fault.

2.4 Esercizi

Gestione della memoria 3.

3 Mass-Storage System

Una delle teconologie più utilizzata è quella delgi HDD, i dati più sensibili sono:

- rotazioni al secondo;
- **transer rate**;
- **positioning time** (random access);
-

I dischi sono divisi in:

- **settori**: un certo numero di bit
- **traccie**: insieme di settori
- **cilindri**: traccie dello stesso raggio

Example 3.1

Si deve trasferire un blocco da $4KB$ su un disco a $7200RPM$, ...

Dal punto di vista logico un disco viene visto come un vettore monodimensionale di blocchi, dovesolitamente i blocchi hanno la dimensione delle pagine. I dischi sono attaccati al sistema tramite dei BUS.

Il sistema operativo è respponsabile di gestire le sue risorse hardware come l'HDD, in questo caso c'è bisgono di fare dello scheduling, i tipi di politica possono essere:

FIFO

SCAN: simile al funzionemanento di un ascenscore, quando si va in una direzione si servono tutti quelli lungo il commino e cosí in discesa.

C-SCAN: simile a SCAN, ma si servono tutti quanto solo in salita (o discesa) e poi si ritorna velocmente all'inizio, il problema di SCAN e C-SCAN è che si potrebbe venire a creare della starvation.

NVM (Non-Volatile Memory): non esiste lo scheduling

Gli errori su questi tipi di memoria vengono gestiti attraverso degli algoritmi di *error detection* o *error correction*,

3.1 RAID Structure

RAID (Redundant Array of Inexpensive Disks), è una struttura che ha come abbiettivo l'aumento del tempo medio tra due fallimenti (dove la ridondanza permette di perdere i dati in caso di fallimento di un singolo disco), ...

Esistono dei diversi livelli di RAID:

- RAID 0: **non-riduntant striping**
- RAID 1: **mirroring**, si sicrive su entrambi i dischi (uno o più)
- RAID 4: **block-interleaved parity**
- RAID 5: **block-interleaved distributed parity**

Example 3.2

Dischi mirrored falliscono in modo indipendente, i dischi hanno 100000 ore,

4 File-System Interface

Un file sono di fatto uno spazio di indirizzamento continuo, può essere interpretato come una sequenza di byte, come un file testule, come un eseguibile, ..., questo dal punto di vista *logico*. Un file ha una serie di attributi che vengono usati per gestrilo, ogni sistema operativo ha la sue caratteristiche ed il modo di gestirlo. Con i file è possibile: crearli, leggerli, scriverli, seek, rimuoverli, operirli, chiuderli.

Dal punto di vista del SO si ha bisogno di una o più **tabelle dei file aperti**, un **puntatore al file** (dipende da contesto), un **conteggio del numero di aperture del file**, un **locazione sul disco**, **diritti di accesso**. Se si vogliono gestire dei file condivisi tra più processi i FileSystem mettono a disposizione dei lock interni al file, questi possono essere **esclusivi** (in scrittura) o **condivisi** (in lettura).

La struttura di un file può essere un *record*:

- linee
- a lunghezza fissa
- a lunghezza variabile

un *struttura complessa* (solitamente si utilizza uno standard):

- struttura formattata
- struttura binaria

I metodi di accesso ad un file possono essere:

- **sequenziale**
- **ad accesso diretto**

Esistono anche degli altri metodi **gerarchici** di accesso, questo viene fatto inserendo degli indici all'interno di un file, questo permette una ricerca efficente che poi contiene la vera informazione all'interno del file.

I **direttori** sono delle strutture dati che contengono informazioni sui file, attraverso cui è possibile accedere ai file. Tutte queste infromazioni sono contenute nei dischi. Solitamente i **dischi** sono partizionati, in ... I file-system sono contentenuti in dei **volumi**, i volumi possono essere contenuti in una partizione o in più partizioni.

I direttori sono di fatto delle tabelle di ricerca, le operazioni che possono essere fatte sono: ricerca, creazione, rimozione, listare, rinominare, attraversamento del file-system. Il direttori devono essere:

- **efficenti**
- **nominazione**

Le strutture possono essere degli alberi, quanod si inziano a creare dei link si veonog acreare dei loop, esistono due soluzioni: si possono staccare dei link solo dalle foglie, c'è un sitema di garbage collection.

Quando si hanno più file-system su uno stesso sistema, si può fare il mounting del file-system, questo permette di creare un link tra un file-system e quello oche verrà montato.

I file-system in linux hanno dei permessi di accesso, questi sono gestiti grazione ai **GroupIDs** ed agli **UserIDs**.

5 File-System Implementation

La struttura

La struttura che contiene le informazioni dei file sarà il **File Control Block** (FCB)

Un file-system è organizzato in:

- device
- IO control
- basic file-system: non vede la struttura del disco, gestisce eventuali buffer o cache, vede solo i blocchi
- basic file organization:
-

...

L'**allocazione linkata** si fa quando un blocco viene linkata al prossimo, dove il puntatore al prossimo blocco viene messo all'interno del blocco, il problema di questa allocazione è che quando si vuole accedere in modo diretto ad un file si devono percorrere comunque tutti i precedenti, il motivo è che non abbiamo il puntatore.

Per ovviare a questa limitazione è stato inventata la **FAT32** (File Allocation Table), che consiste sostanzialmente nello spostare tutti i puntatori in una tabella, in questo modo i tempi rimangono sempre lineari ma sono molto più efficenti che non leggere ogni volta un blocco dal disco.

Il metodo **indicizzato** prende un allocazione contigua di spazio dove all'indice (del vettore) corrisponde il puntatore al blocco, in questo modo i tempi di accesso diventano $O(1)$ (il comportamento è simile alla *page table*), nel File Control Block il primo puntatore sarà il puntatore all'indice. Se un blocco indice non basta a contere tutti gli indirizzi, le soluzioni sono:

- **indicizzazione a due livelli**: dove ci sono dei blocci di indici (linkati tra di loro) che puntano ad i blocchi indice.

Il problema sorge quando in un sistema devono coesister sia file piccoli che file grandi, per questo motivo in Linux si usa una rappresentazione **INode**, dove nel FCB i primi 10 nodi sono all'interno, poi si trova un **signle indirect blocks**, che punta un blocco indice di primo livello, poi ci sono un **doulbe** ed un **triple**, che sono dei blocchi indice di secondo e terzo livello rispettivamente.

Come si gestisce lo spazio libero sul disco? Si può usare la **free list** oppure una **bitmap**.

Buffer Cache. ...

Come si fa fare una **recovery** del sistema quando c'è un crash? Si può utilizzare un **logging strutturato** detto **journaling**, dove si tengono dei log per ricustruire la storia della azioni.

6 I/O Systems

In molti casi i dispositivi perifierici vengono pilotati attraverso dei registri, per quanto riguarda il processare tutto quello che lui deve fare è quello di scrivere e leggere certi indirizzi di memoria che poi venogno mappati.

Ogni volta che si deve scrivere o leggere ci sono diverse modalità , una di queste è il **polling** in cui si legge in modo busy waiting il busy bit, che indica che se il dispotivo è in lavoro, e continuiamo a leggerlo fin quando non è a zero. L'alternativa al polling sono gli **interrupt**, ..., un ciclo di IO è:un device driver (all'interno di una system call, soprattutto un modulo separato del kernel) fa partire un operazione di IO che ando delle istruzione all'IO controller, nel frattempo fa check e si mette in attesa che arrivi l'interrupt, nel frattempo il dispositivo di IO fa le sue operazioni e poi mando l'interrupt al, quando la CPU riceve l'interrupt capisce da quale periferico arraiva e fa partire il corrispettivo interrupt handler.

Quando si fanno delle operazioni di IO nella maggior parte dei casi si tratta di prendere dei datai dalla memoria secondario e e mtterli in RAM, queste operzionei sono molto costose soprattuto se la CPU controlla le operazione , è per questo che sono stati inventati i DMA controller, che il suo unnnnico lavore è quello di prendere il controllo dei BUS ap posto della CPU e non fa altro che legger in ciclo dalla memoria dei dati ed immagazzinarli in un buffer della RAM, infatti il DAM prende in input l'indirizzo di sorgente e d i destinazione, il numero di bytes, modalità lettura o scrittura. Il DAM si dice che fa del **cycle stealing**, ovvero per finche non finisce la sua operazione prende il controllo del bus in modo ciclico per far si di lasciare un po di spazio alla CPU per fare delle operazioni col BUS, una volta che il DMA finsice poi segnala la CPU attraverso un interrupt.

L'OS come vede i device?

- stream di caratteri o blocchi
- sequenziole o accesso diretto
- sincrono o asincrono
- condivisibile tra piu processi o no
- ...

L'IO può essere blocante ovvero il processo che pilota viene bloccato fino al termine dell'operazione, mentre l'IO non bloccante non aspetta, ovvero si partire l'operazione e poi tutti casi vanno gestiti dopo, l'IO asincrono non aspetta la terminazione subito ma magari si aspetta dopo.

Il kernel deve gestire lo schedulingg dei dispotivi grantendo una certa fairness, inoltre deve fornire dei buffer per por garantire certe operazioni, un'impleentazione è il duoble buffer, questo viene usato quando il processo user è molto più veloce

del dispotivo, infatti una volta scritto nel buffer si deve aspettare che il dispotivo lo legga, se infatti si cerca di nuovo di scirvere si va a sovrascrivere il buffer prima che il dispotivo l'abbia letto, per questo motivo si può usere un altro buffer in cui se il primo è occupato si può occupare il secondo, se invece il dispotivo funziona a **burst di dati** si portrebbe implementare un coda fifo che nel mommento della lettura viene svuotata tutta.

7 OS161

7.1 Configurazioni del Kernel

Ogni versione del kernel corrisponde a un file di configurazione tutto in maiuscolo, la prima versione è `DUMBVM` e tutti i file di configurazione si trovano in `$HOME/kern/conf`.

Ogni volta che si aggiunge un nuovo modulo al kernel, questo va abilitato, per abilitare nuovi moduli si va in `conf.kern` e si aggiunge

```
defoption hello
optfile hello main/hello.c
```

dove `hello.c` è il nome del nuovo modulo. `defoptions` *`name`* crea un file `opt-`*`name`*`.h` che è un file `.h` di configurazione, esso contiene una define per specificare se un opzione è abilitata o meno.

```
/* Automatically generated; do not edit */
#ifndef _OPT_HELLO_H_
#define _OPT_HELLO_H_
#define OPT_HELLO 1
#endif /* _OPT_HELLO_H_ */
```

Questo serve nei file del kernel per abilitare l'aggancio di altri moduli che vengono aggiunti.

Per compilare i nuovi moduli si può creare un nuovo file di configurazione nella cartella `conf`, solitamente è meglio partire da uno già esistente come `DUMBVM`, farne una copia e poi aggiungere i nuovi file opzionali. Una volta creato il nuovo file di configurazione, ad esempio `HELLO`, si esegue il comando `./config HELLO` per creare la configurazione compilata della versione del kernel, si dovrà poi andare in `kern/compile/HELLO` e poi eseguire

```
$ bmake depend
$ bmake
$ bmake install
```

7.2 Memoria

Nella versione base (`DUMNBVM`) di OS1616 l'allocazione è allocazione contigua sia per i processi che per il kernel, inoltre questo allocatore non rilascia la memoria. Il file `dumbvm.c` è l'allocatore del mips, `vm.c` è un allocatore opzionale ma è vuoto.

`ram_stealmem` nel file `ram.c`, perdende della pagine dalla ram e le alloca.

L'allocatore è diviso in allocatore User e allocatore Kernel.

Il kernel è basato su mips (32 bit) quindi ha 4GB di memeria logica, è diviso in:

- `kuseg:  (0x00000000, 0x80000000) [user]`
- `kseg0:  (0x80000000, 0xa0000000) [kernel]`

- `kseg1: (0xa0000000, 0xc0000000) [kernel]`
- `kseg2: (0xc0000000, 0xffffffff) [kernel]`

Dall'indirizzo si capisce se il programma è kernel o user, i processi user non possono vedere lo spazio del kernel mentre il kernel può vedere lo spazio user.

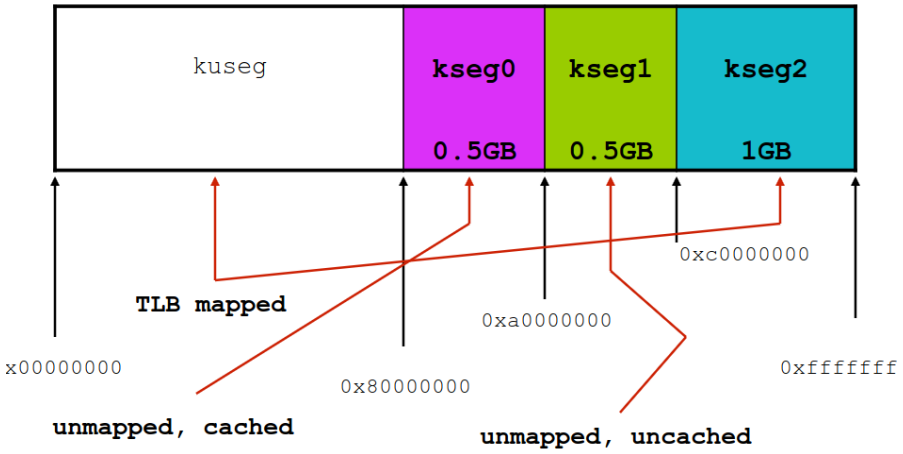


Figure 4: Mips Virtual Address Space

Il kernel utilizza `kseg0` e `kseg1` per TLB e cache. Se si fa accesso alla memoria `kuseg` e `kseg2` i loro indirizzi sono mappati sulla TLB, negli altri spazi non si intende usare la TLB e quindi sono *invisibili* alla TLB. Il `kseg1` è una memoria **uncached** e potrà essere usata per parlare con i dispositivi di IO, il motivo è che questi dispositivi non funzionano bene con le cache. Quando il kernel viene caricato la memoria la prima funzione chiamata è `ram_bootstrap`, partizionando la memoria in:

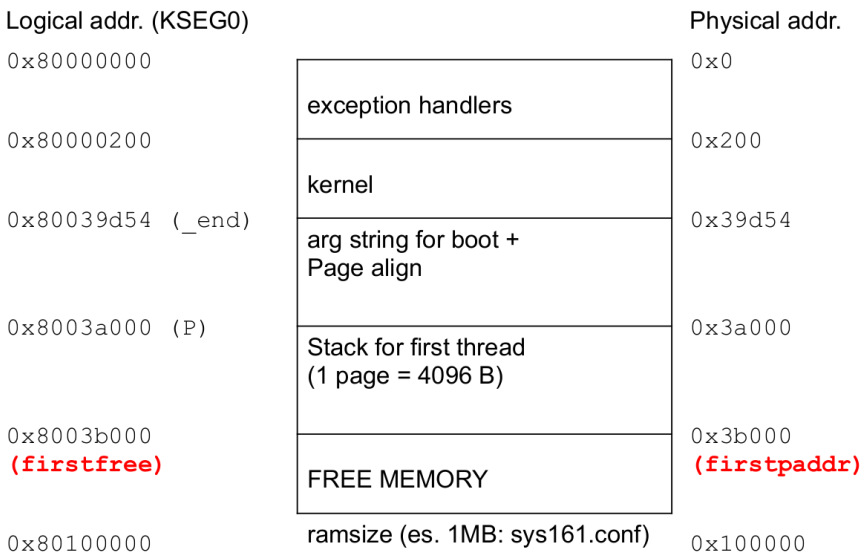


Figure 5: Kernel Loader Initial

Una volta che viene fatto il bootstrap, si hanno a dipsosizione il `firstfree` (primo indirizzo di ram libera, logico), `firstpfree` (primo indirizzo fisico) e `lastpaddr` (ultimo indirizzo fisico).

Quando si vuole allocare della memoria, si usa `ram_stealmem` a cui gli vanno passete il numero di pagine.

```
// kern/arch/mips/vm/ram.c
paddr_t ram_stealmem(unsigned long npages) {
  paddr_t paddr;
  size_t size = npages * PAGE_SIZE;
  if (firstpaddr + size > lastpaddr) {
    return 0;
  }
  paddr = firstpaddr;
  firstpaddr += size;
  return paddr;
}
```

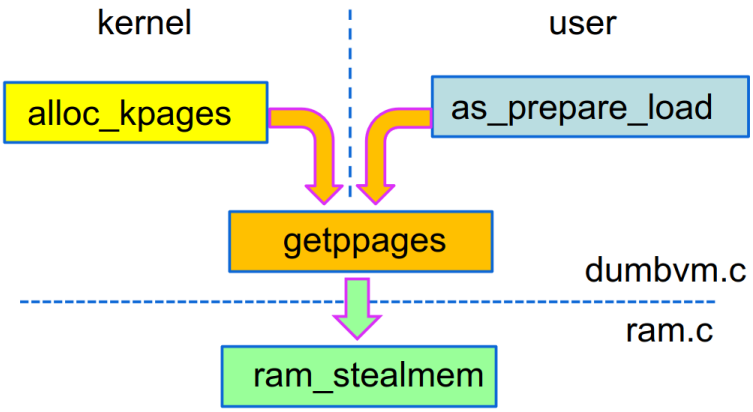


Figure 6: Interfaccia Dumbvm

Le funzioni di free devono essere implementate.

Se si volesse gestire la paginazione, si dovrebbe scindere i l'allocatore per kernel e user, infatti l'user dovrebbe vedere delle pagine mentre il kernel della memoria contigua. Per ora la soluzione proposta è:

- allocazione contigua (per pagine) comune;
- l'allocatore in bumbvm: ...
- bitmap implementata come char array, dove ogni posizione rappresenta se la pagina è libera o presa;
-

7.3 Overview

In os161 si parla di thread di kernel e poi di processi utenti (in futuro un thread di kernel si sgancia dal kernel e diventa processo user), ogni thread ha us suo **context** di esecuzione, in cui sono contenuti dei dati e il suo stack. Gli user thread sono generati da un altro processo thread, ad esempio i *POSIX pthread* permettono di creare dei nuovi thread.

La differenza tra un processo ed un thread è: un processo ha uno spazio di indirizzamento in cui sono contenute le istruzioni ed i dati, poi esistono due aree che sono lo stack e lo heap, il processo arriva da un file eseguibile (tutto o in parte). Un processo può essere *single-threaded* o *multi-threaded*, tutti i dati globali sono condivisi e visibili dai thread, ogni thread ha privato il suo **context** (registri, stack, program counter).

In os161 il contesto di kernel thread, è: stack, registri, dati. I due modi di implementare la creazione dei nuovi thread è diviso in:

- i thread sono gestiti dalla libreria;
- i thread sono gestiti dal kernel;

Questo è dovuto al fatto che: il kernl vede solo dei processi (al posto dei thread) e da come è fatto lo scheduler dei thread. In os161 un thread è fatto da:

```
/* see kern/include/thread.h */
struct thread {
  char *t_name; /* Name of this thread */
  const char *t_wchan_name; /* Name of wait channel, if sleeping */
  threadstate_t t_state; /* State this thread is in */
  /* Thread subsystem internal fields. */
  struct thread_machdep t_machdep;
  struct threadlistnode t_listnode;
  void *t_stack; /* Kernel-level stack */
  struct switchframe *t_context; /* Saved register context (on stack) */
  struct cpu *t_cpu; /* CPU thread runs on */
  struct proc *t_proc; /* Process thread belongs to */
  ...
};
```

Esistono 3 funzioni per gestire dei thread:

- `thread_fork`: crea un nuovo thread;
- `thread_exit`: termina il thread;
- `thread_yield`: sospende l'esecuzione del thread;

Quando viene creato un nuovo thread viene fatto un context switch. Come viene fatto il context switch? Si utilizza `switchframe_switch`, il codice scritto è in assembler MIPS perché vanno salvati anche i registri e questo non è possibile farlo in C.

Quando viene creato un nuovo user thread, viene generato un nuovo stack per il processo user, lo stack kernel non viene rimosso ma rimane in memoria, quando il processo farà una chiamata ad una systema call e passarà in *modalità kernel* e userà lo stack kernel che ha lasciato indietro.

I processi sono rappresentati dalla `struct proc`, questa struct ha tutte le informazioni relative al processo, un campo in esso contenuto (`struct addrspace`) contiene al suo interno che definisce lo spazio d'indirizzamento del processo, se ci fosse paginazione dovrebbe contenere la tabella della pagine (non è il caso in `DUMBVM`). Nella versione base di OS161 non ha un puntatore a ogni thread che ha (ha solo il numero di thread), mentre i thread hanno un puntatore al processo padre. Un processo user non esegue istruzioni solo in modalità user ma richiede del lavoro in modalità kernel, usando le system call.

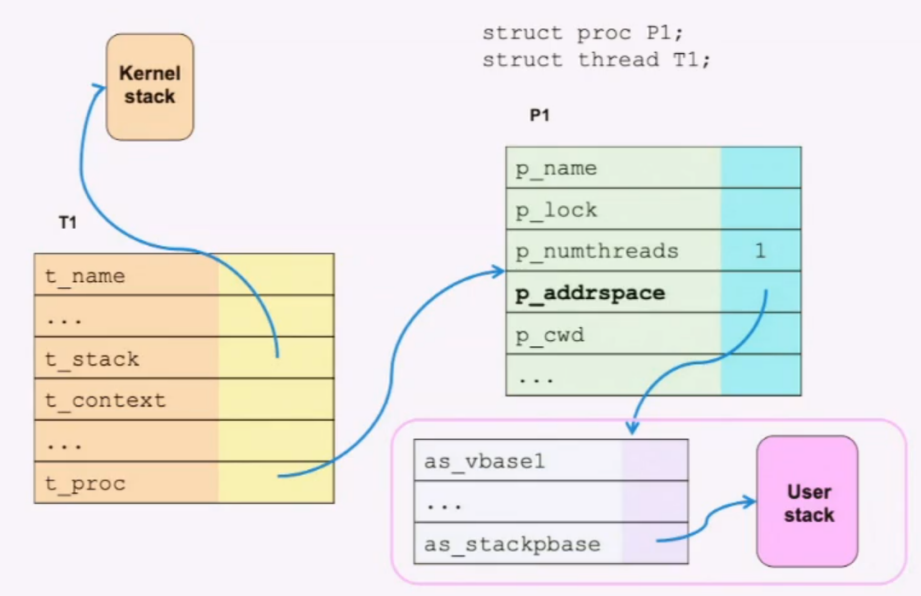


Figure 7: Process Stack

Per creare un programma user da os161 si fa

```
p <elf_file> {<args>}
```

quello che avviene è:

1. si chiama

   ```
   proc_create_runprogram
   ```

   che crea lo spazio d'indirizzamento del processo

2. legge il file ELF per caricarlo in memoria

3. si passa da kernel thread user thread

... La `runprogram` al suo interno fa partire la `enter_new_process`, che si appoggia ad una strutturea siimila allo `switchframe` che si chiama `trapframe`, questa struttura server per fare un *cambio di contesto* (context switch), la `enter` crea un nuovo processo come se questo processe stesse ritornando da una trap, infatti il `trapframe` è proprio la struttura in cui vengono salvati i dati alla chiamata di una trap.

7.4 Systemcall

Le **systemcall** sono un modo per i processi di utenti di far eseguire al kernel delle operazioni.

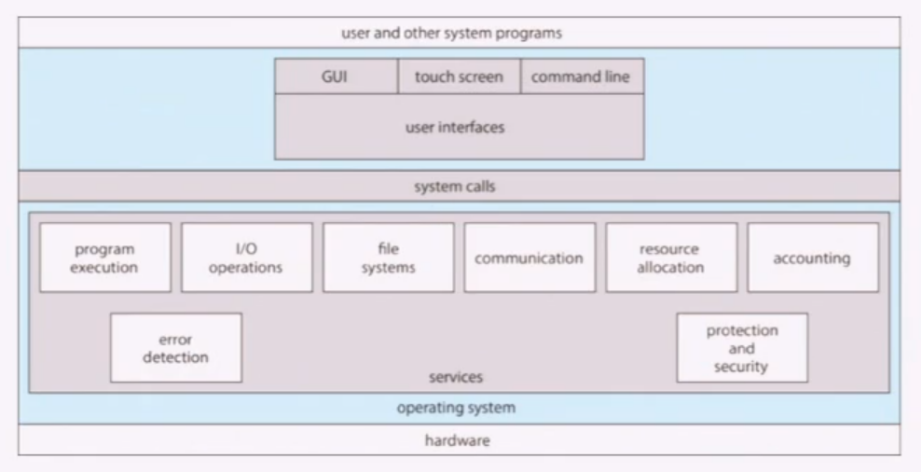


Figure 8: Os Overview

In os161 quando viene invocato una system call i dati vengono passati nel `trapframe` (come in linux), uno user program chiamerà la system call con il suo numero associato e inserisce i dati necessari.

Di system call ce ne sono di vario genere, tra le più importanti ci sono: sincronizzazione, file system, tempo, ... In MIPS esiste un solo handler per gestire gli interrupt, le eccezioni e le system calls, questo è fatto perchè la differenza sarà gestita in hardware, il syscall handler prende un numero ad attraverso uno switch decide che tipo di trap è stata scatenata (hardware o software). Nel processo utente il `trapframe` ed il `contextframe` si trovano entrami nello stack del processo.

Esistono due tipi di programmi che interagiscono che sono i processi utenti ed il kernel, come già visto i processi utenti usano indirizzi virtuali, ma il kernel ha bisogno di utilizzare indirizzi virtuali? La possibilità sono:

- Il kernel è in memoria fisica: la CPU è a consoscenza della modalità di operazione, quindi quando è il kernel ad operare viene spenta la traduzione da indirizzo logico a fisico.
- Il kernel si trova in un spazio di indirizzamento virtuale separato: in questi casi non ci sarà paginazione del suo spazio di indirizzamento
-

In linux tutti gli indirizzi sono separati, infatti si potrebbero avere due indirizzi identici che in due contesti diversi puntano a memoria diversa, in linux questo non piace, perché il kernel può metter mano all'interno dei processi, per questo motivo esistono indirizzi utente e indirizzi kernel.

Nel MIPS la traduzione logico/fisica avviene nella TLB, che contiene: numero di pagina virtuale, numero di frame fisico, falgs, ID indirizzo (non usato in OS161).

In `dumbvm` l'allocazione delle pagine è contigua, vengono definiti dei campi per l'`addrspace` di un processo e sono:

```
\struct addrspace {
#if OPT_DUMBVM
  vaddr_t as_vbase1; /* base virtual address of code segment */
  paddr_t as_pbase1; /* base physical address of code segment */
  size_t as_npages1; /* size (in pages) of code segment */
  vaddr_t as_vbase2; /* base virtual address of data segment */
  paddr_t as_pbase2; /* base physical address of data segment */
  size_t as_npages2; /* size (in pages) of data segment */
  paddr_t as_stackpbase; /* base physical address of stack */
#else
/* Put stuff here for your VM system */
#endif
};
```

In questo modo basta conoscere il `base` ed il numero di pagine che il processo possiede. In questo modo si evita di avere una page table per tenere traccia delle pagine.

Se non è presente la entry nella TLB viene generato un `page fault`.

7.5 Sincronizzazione

I problemi delle programmazione concorrente sono: deadlock, starvation, race conditions. Si chiama **sezione critica** un pezzo di codice che accede a della memoria condivisa che può essere eseguito da un solo processo. Le sezioni critiche possono essere garantite da: mutua esclusione, progresso, bounded wait. Si possono usare due approcci che garantiscono la sincronizzazione, questo è discriminato dal fatto che il sistema sia **preemptive** o **non-preemptive**, ovvero se le primitive di sincronizzazione possono essere bloccate o se non possono essere bloccate (in questo caso in kernel mode non esistono problemi di race conditions).

Example 7.1 – Algoritmo di Peterson

Supponiamo che `load` e `sotre` sia atomiche, e che siano due variabili condivise tra due processi `turn` (dice a chi tocca) e `flag[2]` (indica se il processo 1 o 2 è pronto ad entrare nella sezione critica),

```
/* Processo Pi, l'altro e' Pj */
while (true)
{
  flag[i] = true;
  turn = j;
  while (flag[j] && turn == j);

  /* Critical section */

  flag[i] = false;
}
```

> si può dimostrare che se lettura e scrittura sono atomiche questo algoritmo funziona e la mutua esclusione è garantita. Il problema è che nelle architetture moderne questo non è vero, perché non è detto che le letture e scritture funzionino in modo atomico, soprattutto nei processori multicore e con pipeline.

A causa di questi problemi si sono inventati dei metodi appositi per effettuare la sincronizzazione, i **lock**. I modi di acquisire un lock sono:

- spin o busy-waiting
- lasciare il processore

Per realizzare queste primitive l'hardware mette a disposizione **test and set** e **compare and swap** in modo atomico. La **test and set** fa

```
bool test_and set(bool *lock)
{
  bool prev = *lock;
  *lock = true;
  return prev;
}
```

mentre la **compare and swap**

```
itn compare_and_swap(int *value, int expected, int new_value)
{
  int temp = *value;

  if (*value == expected)
    *value = new_value;

  return temp;
}
```

entrambe sarebbero le realizzazioni in C, ma queste operazioni sono realizzate in hardware. Insieme a queste due istruzioni esistono anche le **variabili atomiche**, che permettono operazioni atomiche su di loro.

In OS161 esistono gli **spinlock** ovvero dei lock con busy-waiting, vengono usati quando la critical section non è molto complicata, vengono implementati con `test-and-test-and-set`, che è un leggera variante del `test-and-set` dove lo spinlock viene prima testata in modo non atomico e poi in modo atomico

```
void spinlock_acquire(struct spinlock *splk)
{
  // ...
  while (1)
  {
    if (spinlock_data_get(&splk->splk_lock) != 0)
      continue;
    if (spinlock_data_testandset(&splk->splk_lock) != 0)
```

```
      continue;

    break;
  }
  // ...
}
```

questo viene fatto per motivi di performance, infatti fare il `testandset` inficia sulla performance dell'intero sistema.

Esistono poi anche i **semafori**, che sono più potenti dei lock, ne esistono di counting e di binary. Le primitive sono dette `wait` (`P`) e `signal` (`V`), dove `wait` decrementa il semaforo se maggiore di 0 e `signal` incrementa il semaforo. I semafori solitamente non vengono implementati con del busy-waiting, per questo motivo i semfori al loro interno contengono una lista di attesa, quando la `wait` fallisce si inserisce un nuovo processo nella lista. Quando sarà fatta la `signal` si dovrà decidere quale processo svegliare. Un modo di rendere un operazione atomica è quello di disabilitare l'handler di un interrupt, questo va bene in un single core, ma in un multicore questo non si può fare.

Un semaforo contiene al suo interno una lista di processi, ovvero i processi in attesa. In os161 esistono `thread_sleep` e `thread_wakeup`, la differenza tra `sleep` e `yield` è che dopo `sleep` il thread è stato di blocco mentre con `yield` il processo è ready.

I `lock` sono degli oggetti simili a dei semafori binari, i `lock` però sono adatti solo alla mutua esclusione, il vincolo è che che fa il lock-acquire deve fare anche il lock-realease.

Esistono dei casi (**wait on codition**) in cui data una condizione basata su più variabili, si manda una `signal` per fare un'operazione su queste variabili

```
/* shared state vars with some initial value */
int x,y,z;
/* mutual exclusion for shared vars */
struct lock *mylock = lock_create("Mutex");
/* semaphore to wait if necessary */
struct semaphore *no_go = sem_create("MySem", 0);
compute_a_thing
{
  lock_aquire(mylock); /* lock out others */
  /* compute new x, y, z */
  x = f1(x); y = f2(y); z = f3(z);
  if (x != 0 || (y <= 0 && z <= 0)) V(no_go);
    lock_release(mylock); /* enable others */
}

//...
use_a_thing
{
  lock_aquire(mylock); /* lock out others */
  if(x == 0 && (y > 0 || z > 0))
```

```
    P(no_go);
  /* Now either x is non-zero or y and z are
  non-positive. In this state, it is safe to run
  "work" on x,y,z, which may also change them */
  work(x,y,z);
  lock_release(mylock); /* enable others */
}
```

In questo caso se `use_a_thing` arriva per primo al lock ci sarà un deadlock. Una soluzione è

```
use_a_thing
{
  lock_aquire(mylock); /* lock out others */
  while (x == 0 && (y > 0 || z > 0))
  {
    lock_release(mylock); /* no deadlock */
    P(no_go);
    lock_aquire(mylock); /* lock for next test */
  }
  /* Now either x is non-zero or y and z are
  non-positive. In this state, it is safe to run
  "work" on x,y,z, which may also change them */
  work(x,y,z);
  lock_release(mylock); /* enable others */
}
```

Per questo motivo esistono le **condition variable**. I **Monitor** sono dei costrutti di alti livello che contengono delle code di attesa, ..., che mettono tutto insieme.

La differenza tra signal e condition variable è che quando la signal viene segnalata, se un semaforo arriva dopo può comunque prenderla, la condition varaible quando segnala se nessuno prende il segnale allora viene perso, se un semaforo arriva dopo non vede la signal.

Esistono anche i **wait channel**, in OS161 quando un

8 File System

In OD161 esiste il `struct vnode` che è il simile ad un iNode, ma rimane più alto livello, può essere considerato come un File Control Block. In OS161 esistono i due tipi di file system, fily system emulato (`umufs`, sfrutta il file system della macchina su cui os161 gira),simple fyle sytem (`sfs`)

8.1 Esame

9 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 mut 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` e 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());
}
```

9.1 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 entrambe 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.

9.2 Clap

`clap` è una libreria di rust che permette di fare il parsing degli argomenti da linea di comando, una implementazione possibile è la seguente:

```rust
use clap::Parser;

/// Simple program to greet a person
#[derive(Parser, Debug)]
#[command(version, long_about = None)]
struct Args {
  /// Name of the person to greet
  #[arg(short, long)]
  name: String,
  /// Number of times to greet
  #[arg(short, long, default_value_t = 1)]
  count: u8,
}

fn main() {
  let args = Args::parse();
  for _ in 0..args.count {
    println!("Hello {}!", args.name)
  }
}
```

9.3 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.

10 Tipi Composti

In rust si possono creare delle `struct` che è un tipo *prodotto*, ovvero che la sua grandezza è il prodotto cartesiano dei domini dei parametri contenuti al suo interno. Per convenzione si usa il *CamelCase* per i nomi delle `struct`. Le `struct` possono essere di due tipi: `named struct {}` dove ogni campo ha un nome, `tuple struct ()` dove i campi non hanno nome e vegono invocati attraverso un numero come posizione. In rust quando viene dichiarata una `struct` i campi possono essere riordinati per permettere una maggiore efficienza nell'utilizzo della memoria, a differenza di C/C++ che dispongono i parametri in memoria secondo l'ordine di scrittura, per questo motivo rust permette di utilizzare l'attributo `#[repr(...)]` che modifica la rappresentazione in memoria dei parametri.

```
#[repr(C)]
struct Test {
  a: i8,
  b: i32,
  c: i8,
};
```

In questo caso la struttura verrà disposta in memoria come in C.

10.1 Visibilità

In rust si possono definire dei moduli con la keyword `mod`, questo si può fare sia all'interno del file `main` che in altri file.

```
mod example {
  struct S2 {
    a: i32,
    b: bool,
  }
}

fn main() {
  let s2 = S2 { a: 0, b: true };
  // ERRORE: tutto cio' che sta in un modulo e' privato
}
```

Se si vuole utilizzare un modulo esso deve essere importato, se si vuole utilizzare la `struct S2` bisogna renderla pubblica, infatti qualsiasi cosa all'interno di un modulo una visibilità privata di default (ovvero che è visibile solo all'interno del modulo).

```
mod example {
  pub struct S2 {
    pub a: i32,
    pub b: bool,
  }
}

```

```
use example::S2;

fn main() {
  let s2 = S2 { a: 0, b: true };
  // Adesso compila
}
```

Si possono implementare anche delle funzioni per `S2`:

```
mod example {
  pub struct S2 {
    a: i32,
    b: bool,
  }

  impl S2 {
    pub fn new(a: i32, b: bool) -> Self {
      Self { a, b }
    }
  }
}

use example::S2;

fn main() {
  let s2 = S2::new(0, true);
}
```

Nei blocchi `impl` se si vuole fare riferimento all'istanza della `struct` si utilizza la keyword `self`. Le funzioni che hanno come primo parametro `self` diventano dei **metodi** e vengono dette **funzioni associate**.

In rust non è possible fare l'*overloading* dei metodi, da convenzione si usa `fn with_attributes(...)` (e.g. `String::with_capacity(10)`).

Il paradigma **RAII** (Resource Acquisition Is Initialization) voul dire: le risorse vengono incapsulate in un oggetto che possiede solo un *costruttore* ed un *distruttore*, in rust un concetto in cui viene implementato RAII è la struct `Box`. Ad esempio su C++ se può creare una classe che viene allocato nello stack che incapsula un oggetto e lo libera quando viene rimossa dallo stack.

```
class Car {};

void a_function() {
  Car *ptr = new Car;
  function_that_can_throw();
  if (!check()) return;
  delete ptr;
}
```

In questo caso non esiste garanzia che `ptr` venga liberato dalla memoria, grazie al paradigma RAII si può creare una classe wrapper che libererà l'oggetto in modo

automatico.

```
class CarManager {
  Car *car;
public:
  CarManager() {
    this->car = new Car;
  }
  ~CarManager() {
    delete this->car;
  }
}

class Car {};

void a_function() {
  CarManager car;
  function_that_can_throw();
  if (!check()) return;
}
```

C++ ha proprio un modo per gestire queste casistiche già implementato nella libreria standard, si chiama `unique_pointer`

```
void a_function() {
  std::unique_ptr<Car> car = std::make_unique<Car>();
  function_that_can_throw();
  if (!check()) return;
}
```

In Rust tutto questa gestione delle memoria è fatta in automatico grazie al Borrowing System e soprattutto **senza overhead**.

10.2 Enum

All'interno di valori `enum` si possono associare dei valori oppure anche dei campi.

```
enum HttpResponse {
  Ok = 200,
  NotFound = 404,
  InternalError = 500,
}

enum HttpResponse {
  Ok,
  NotFound(String),
  InternalError {
    desc: String,
    data: usize,
  }
}
```

In rust i tipi `enum` sono dei tipi somma, ovvero che il loro dominio è la somma di tutti i suoi campi, gli `enum` sono i tipi più simili alle `union` di C, in rust però i tipi `enum` contengono anche un `tag` per poterli distinguere. Grazie agli `enum` è possibile fare del parrtern matching con `match` e con `if let`:

```
enum Shape {
  Square { s: f64 },
  Circle { r: f64 },
  Rectangle { w: f64, h: f64 },
}

fn compute_area(shape: Shape) -> f64 {
  match shape {
    // Destructory assignment
    Square { s } => s*s,
    Circle { r } => r*r*3.14,
    Rectangle { w, h } => w*h,
  }
}

fn process(shape: Shape) {
  // Destructory assignment
  if let Shape::Square { s } = shape {
    println!("Square side: {}", s);
  }
}
```

La **destrutturazione** può essere utilizzata anche nelle operazioni semplici.

```
let Point { x, y } = point;
println!("{}{}", x, y);
```

Quando si fa destructuring è possibile prendere i valori interni agli oggetti come riferimento, anteponendo la keyword `ref`, il motivo è che usando questi construtti, se la `struct` non implementa `Copy`, il valore viene consumato dal blocco.

10.3 Monad

In rust non esiste `null`, l'assenza di valori o gli errori vengono gestiti con: `Option<T>` che contiene `Some(t)` o `None` e `Result<T,E>` che contiene `Ok(T)` o `Err(E)`. Sia `Option` che `Result` sono implementati come `enum`.

10.4 Polimorfismo

Il polimorfismo è la capacità di associare comportamenti comuni a un insieme di tipi differenti. In C++ esiste l'*ereditarietà multipla*, che permette di fare il polimorfismo.

```
#include <iostream>

class Alfa {
```

```
  int i;
public:
  virtual int m() { return 1; }
  virtual unsigned size() { return sizeof(*this); }
};

class Beta: pulbic Alfa {
  double d;
public:
  int m() { return 2; }
  unsigned size() { return sizeof(*this); }
};

int main() {
  Alfa* ptr1 = new Alfa;
  Beta* ptr2 = new Beta;

  std::cout << ptr1->m() << std::endl;  // 1
  std::cout << ptr2->m() << std::endl;  // 2

  std::cout << sizeof(*ptr1) << std::endl;  // 4
  std::cout << sizeof(*ptr2) << std::endl;  // 16

  delete ptr1;
  delete ptr2;
}

// Senza virtual
int main2() {
  Alfa* ptr1 = new Alfa;
  Alfa* ptr2 = new Beta;

  std::cout << ptr1->m() << std::endl;  // 1
  std::cout << ptr2->m() << std::endl;  // 1

  std::cout << sizeof(*ptr1) << std::endl;  // 4
  std::cout << sizeof(*ptr2) << std::endl;  // 4

  delete ptr1;
  delete ptr2;
}

// Con virtual
int main2() {
  Alfa* ptr1 = new Alfa;
  Alfa* ptr2 = new Beta;

  std::cout << ptr1->m() << std::endl;  // 1
  std::cout << ptr2->m() << std::endl;  // 2

  std::cout << sizeof(*ptr1) << std::endl;  // 16
```

```
55   std::cout << sizeof(*ptr2) << std::endl;  // 16
56   std::cout << ptr1->size() << std::endl;  // 16: "8vtable + 4int + 4padd"
57   std::cout << ptr2->size() << std::endl;  // 24: "8vtable + 4int + 4padd + 8
       double"
58
59   delete ptr1;
60   delete ptr2;
61 }
```

Quando in una classe C++ appare `virtual`, gli oggetti della classe acquisiscono un campo in più detto `vtable`, che risolve il metodo chiamato dall'oggetto, che possono puntare alle funzioni di `Alfa` o di `Beta`.

In rust **non esiste l'ereditarietà**, però in rust i **Tratti** sono per rust come le **Interfaccie** per Java. *Un tratto esprime la capacità di eseguire una certa funzione*, as esempio: `std::io::Write`, `std::iter::Iterator`, ... In rust dunque non esiste l'overhead come in C++ causato dal polimorfismo, il motivo è che il compilatore risolve tutti i riferimenti in modo statico, l'unico momento in cui quel costo si presenta è quando si creano dei riferimenti dinamici (`&dyn SomeTrait`). Per implementare un tratto in rust si scrive:

```
1  trait T1 {
2    fn num() -> i32;
3    fn return_self() -> Self;
4  }
5
6  struct OtherType {};
7  impl T1 for OtherType {
8    fn num() -> i32 { 2 }
9    fn return_self() -> Self { OtherType }
10 }
```

Quando si crea un tratto si può definire un **Tipo Associato**, che sono identici ai **Tipi Generici**, ma in alcuni casi migliorano la laggibilitá.

```
1  trait T2 {
2    type AssociatedType;
3    fn f(arg: Self::AssociatedType);
4  }
5
6  struct OtherType {};
7  impl T2 for OtherType {
8    type AssociatedType = &str;
9    fn f(arg: Self::AssociatedType) {}
10 }
```

Example 10.1

Implementiamo la struct Contains con 2 tipi generici.

```
1 struct Container(i32, i32);
```

```

trait Contains<A, B> {
  fn contains(&self, _: &A, _: &B) -> bool; // Explicitly requires `A` and `B
    `.
  fn first(&self) -> i32; // Doesn't explicitly require `A` or `B`.
  fn last(&self) -> i32;  // Doesn't explicitly require `A` or `B`.
}

impl Contains<i32, i32> for Container {
  // True if the numbers stored are equal.
  fn contains(&self, number_1: &i32, number_2: &i32) -> bool {
      (&self.0 == number_1) && (&self.1 == number_2)
  }

  // Grab the first number.
  fn first(&self) -> i32 { self.0 }

  // Grab the last number.
  fn last(&self) -> i32 { self.1 }
}

// `C` contains `A` and `B`. In light of that, having to express `A` and
// `B` again is a nuisance.
fn difference<A, B, C>(container: &C) -> i32 where
  C: Contains<A, B> {
  container.last() - container.first()
}

```

Se addesso invece usiamo un Tipo Associato non è necessario specificare il tipo della struct.

```
struct Container(i32, i32);

// A trait which checks if 2 items are stored inside of container.
// Also retrieves first or last value.
trait Contains {
  // Define generic types here which methods will be able to utilize.
  type A;
  type B;

  fn contains(&self, _: &Self::A, _: &Self::B) -> bool;
  fn first(&self) -> i32;
  fn last(&self) -> i32;
}

impl Contains for Container {
  // Specify what types `A` and `B` are. If the `input` type
  // is `Container(i32, i32)`, the `output` types are determined
  // as `i32` and `i32`.
```

```
  type A = i32;
  type B = i32;

  // `&Self::A` and `&Self::B` are also valid here.
  fn contains(&self, number_1: &i32, number_2: &i32) -> bool {
    (&self.0 == number_1) && (&self.1 == number_2)
  }
  // Grab the first number.
  fn first(&self) -> i32 { self.0 }

  // Grab the last number.
  fn last(&self) -> i32 { self.1 }
}

fn difference<C: Contains>(container: &C) -> i32 {
  container.last() - container.first()
}

```

Rust permette un forma di dipendenza tra tratti e vengono detti **sub-trait** e **super-trait**:

```
trait Super {
  fn f(&self) { println!("Super"); }
  fn g(&self) {}
}

trait Sub: Super {
  fn f(&self) { println!("Sub"); }
  fn h(&self) {}
}

struct SomeType;
impl Super for SomeType;
impl Sub for SomeType;

fn main() {
  let s = SomeType;
  s.f();  // ERRORE: chiamata ambigua
  <SomeType as Super>::f(&s);
  <SomeType as Sub>::f(&s);
}
```

Esistono due tratti per poter fare dei confronti tra tipi su rust, che sono `Eq` e `PartialEq`. Un caso particolare di confronto sono i floating point. Quando si vogliono gestire dei confronti di ordine, ovvero quando un tipo è maggiore/minore, si utilizza il tratto `PartialOrd`. Quando si usa `PartialX` l'uguaglianza non è riflessiva mentre mentre senza `Partial` la comparazione è riflessiva.

Grazie ai tratti si possono ottere dei risultati simili a qualli che in C++ era

l'operation overloading, infatti è possibile definire le operazioni di addizione, molitiplicazione, accesso con indice (dove l'indice è un tipo generico), ecc... Alcuni tratti molto caratteristici sono `Derfef` e `DerefMut`, da questo tratto deriva una serie di classi di Rust che vengono dette **Smart Pointers** che ci permettono di trattarli come dei puntatori senza che siano ne un puntatore e ne un riferimento, ma che restituisce un riferimento, si fa accesso alla variabile usando `*t`, infatti il tratto `Box` è proprio uno smart pointer, il tipo al loro interno è solo movibile e non copiabile.

Per effettuare le conversioni tra tipi esistono anche i tratti `From` e `Into`. Quando non si ha `Into` è possibile inferirlo da `From`. Esistono anche `TryFrom` e `TryInto` che restituiscono un `Result`.

10.5 Funzione Generiche

È possibile definire una funzione che implementa un *tipo generico* che al momento non implementa un tipo specifico.

```
fn max<T>(t1: T, t2: T) -> T where T: Ord {
  if t1 < t2 { t2 } else { t1 }
}
```

Quando si utilizza una funzione generica per ogni tipo nuovo che viene passato alla funzione, viene generata dal compilatore una nuova versione della funzione specializzata per il tipo passato, questo processo viene detto **monomorfizzazione**, perché si passa da *polimorfismo* a *monomorfismo*. I tipi generici si possono usare anche all'interno delle struct.

```
struct MyS<T> where T: SomeTrait {
  a: T,
}

impl<T> MyS<T> where T: SomeTrait {
  fn a(b: T) -> Self {
  }
}
```

Con i tipi generici si possono aggiungere dei vincoli sui tipi generici, per farlo esistono due sintassi: si specificano all'iterno di `<>` oppure specificandoli con la keyword `where`.

Example 10.2 – Dynamic vs Generic

```
fn dynamic(w: &mut dyn Write) { ... }
fn generic<T>(w: &mut T) where T: Write { ... }

```

La differenza tra `dynamic` e `generic` è che: `dynamic` viene compilata una sola volta, ma viene passato un **fat pointer** e per accedere ai metodi di `w` si dovrà passare prima dalla v-table, mentre in `generic` viene passato solo il puntatore semplice però il codice viene duplicato, causato dalla monomorfizzazione.

10.6 Lifetimes

In Rust ogni riferimento ha tempo di vita, il compilatore è in grado d'inferire il tempo di vita dei riferimenti nella maggior parte dei casi, quando questo non è possibile deve essere scritti esplicitamente, la sintassi per farlo è: `fn test<'a>(number: &'a i32)`, dove le lifetimes sono precedute da un apice e vengono dalle lettere dell'alfabeto in modo ordinato.

Example 10.3 – Uso delle Lifetimes

Si preda come esempio il caso in cui un funzione che prende in input due riferimenti e ne ritorna un'altro:

```
fn max<'a>(num1: &'a i32, num2: &'a i32) -> &'a i32 {
  if num1 > num2 {
    return num1;
  } else {
    return num2;
  }
}

```

In questo caso le lifetimes coincidono, proviamo ad usarla:

```
fn main() {                 // lifetimes:
  let n = 10;               // n+
  let res;                  //  |
  {                         //  |
    let m = 19;             //  | m+ 'a+
    res = max(&n, *m);      //  |  |   |
  }                         //  |  |   |
  println!("max: {}", res); //  |
  // ERRORE-----------^     //  |
}                           //  |

```

In questo caso il codice non compilerà perché quando si utilizzano delle lifetimes si assume come intervallo **l'interseizione delle lifetimes** e quindi viene presa la più piccola, in questo casa `'a` concide con il tempo di vita di `b`.

10.7 Closures

Una **funzione di ordine superiore** è detta tale se prende in input a da in output una funzione. In rust è possibile assegnare ad una variabile una funzione is suo tipo sarà `fn(T1, T2, ..., Tn) -> U`. Quando una variabile ha un tipo `fn` questa sarà legata ad una funzione classica, se invece si usano le *lambda* esistono altri tre tipi:

- `FnOnce`: struttura che può essere invocata una sola volta.

- `FnMut`: la struttura può essere chiamata più volte e a ogni chiamata può alterare il suo stato.
- `Fn`: la struttura può essere chiamata più volte ma non può mutare il suo stato.

Il corpo della funzione lambda può fare riferimento alla variabili locali nello scope in cui sono state create, in questo caso la lambda acquisisce uno stato e le variabili vengono dette **variabili libere**, in questo caso si dice che le viabili vengono **catturate**. La cattura può avvenire per **riferimento**, e quindi la variabile sarà accessibile solo nel suo tempo di vita, oppure la cattura può avvenire per **movimento**, dove la variabile viene mossa all'interno della funzione lambda e quindi ne prende il possesso.

```
let number = 10;
let with_reference = |...| { number ... };
let with_movement = move |...| { number ... };
```

Il tipo di una lambda è inferito in compilazione a differenza degli utilizzi, oppure si può specificare:

```
fn generator<F>(base: &str) -> F
where F: impl FnMut() -> String {
  let mut v = 0;
  return move || {
    v += 1;
    format!("{}{}", base, v)
  };
}
```

11 Errori ed Eccezioni

In Rust non esistono le eccezioni ma usa invece `Result<T, E>` e `Option<T>`. In Java quando si lancia un eccezione si ritorna al più basso contesto di gestione di eccezioni, dove si va alle differenti `catch` fino a trovare un match, in C++ le eccezioni funzionano quasi in modo identico

```
int f2() {
  if (...)
    throw std::logic_error("err");
  return 1;
}
int f1() {
  try {
    return f2();
  }
  catch (std::locic_error e) {
    return -1;
  }
}
```

in C++, alla radice di programma c'è un `try catch` in grado di catchare tutte le eccezioni non gestite. Quando viene scritto un blocco `try`, nello stack viene inserito un nuovo **exception context**, che avrà il riferimento a quello precedente, il corrente invece si trova in un registro dalla CPU, in questo modo quando viene lanciata un eccezione in modo ricorsivo si controllano tutti gli exception context fino a trovare un match. A differenza di Java in C++ si può lanciare qualsiasi valore invece di classi `throwable`, quando viene lanciata un eccezione il pattern RAII permette di pulire lo stack prima di lasciare la funzione.

In Rust non esistono le eccezioni e adotta un approccio funzionale con `Result` e `Option`, il tipo `Result` è una **Monade** che incapsula il valore o l'errore. In Rust si può terminare l'interno processo con `exit` oppure si può terminare il thread corrente con `panic!`. Gestire gli errori è tedioso, allora Rust mette a disposione l'operatore `?` che in caso di successo ritorna `Ok` altrimenti esce dalla funzione e ritorna `Err`. Per gestire più tipi di errori come valori di ritorno si può astrarre l'errore a uno generico usando come valore di ritorno

```
Result<(), Box<dyn error::Error>>
```

poi si può anche generare un nostro errore, per fare questo si implementa il tratto `Error`, il problema è che si devono implementare una serie di tratti, che sono richiesti da `Error`: `Display, Debug, From`. Un crete chiamato `thiserror`, permette di implentare direttamente il tratto `Error` per ogni tipo usando la macro `#[derive]`

```
#[derive(Error, Debug)]
enum SumFileError {
  #[error("IO error {0}")]
  Io(#[from] io::Error)

  #[error("Parse error {0}")]
  Io(#[from] ParseIntError)
}
```

Un altro create è `anyhow`, che gestisce gli errori in modo ideomatico.

12 Iteratori

Un iteratore è una struttura dati che ha un suo stato, che premette di esplorare i dati contenuti all'interno di un altra struttura. Rust usa il tratto `Iterator`, che ha un solo metodo da implementare

```
trait Iterator {
  type Item;
  fn next(&mut self) -> Option(Self::Item);
}
```

poi esistono 3 metodi che servono a trasformare un oggetto in un iteratore: `iter(), into_iter(), iter_mut()`.

Un **adattatore** è un metodo che ogni iteratore offre che permette di trasformare l'iteratore originale in uno diverso, fino ad arrivare ad un consumatore finale

```
let sum = list
  .iter()
  .filter(|x| x.len() < 4)
  .map(|x| x.into::<i32>())
  .sum();
```

alcuni adattori sono: `map`, `take_while`, `skip`, `skip_while`, `peekable`, `fuse`, `rev`, `...` I **consumatori** invece consumano un adattatore.

13 Collezioni di Dati

Le collezioni mettono a disposizione: `new()`, `len()`, `clear()`, `is_empy()`, `iter()`.

`VecDeque` è una coda in cui si può inserire/rimuovere in modo costante in testa e in coda, in Rust viene implementato come buffer circolare.

`Mappe`, in Rust sono presnti due tipi di mappe che sono `HashMap` e `BTreeMap`, il vantaggio sul tratto delle chiavi è che possibile fare delle operazioni di ricerca e di sostituzione una sola volta, questo grazie al tratto `Entry<'a,K,V>`

```
let mut animals = HashMap::<&str, u32>::new();

animals.entry("dog")
  .and_modify(|e| *e += 1)
  .or_insert(1);
```

`BinaryHeap<T>` è una collezione di elementi con una priorità associata, infatti `T` deve implentare il tratto `Ord`, `peek()` mi permette di vedere l'elemento più grande in tempo unitario.

14 IO

Nei sistemi operativi i file sono stati implementati in modo gerarchico per ragioni storiche, anche se sarebbe molto più effincente rappresentare tutto con una mappa e la rappresentazione gerarchica solo come un tag. Esiste anche il problema della compatibilità tra i diversi SO e le diverse system call, Rust cerca di unificare le diverse rappresentazioni nella libreria standard, per ovviare a questo esistono `Path` e `PathBuf`, la rappresentazione interna dei nomi dei file è fatta con `OsString` che permette di avere compatibilità tra i diversi SO. Per aprire un file si utilizza `std::fs::OpenOption` o `File`. La libreria mette a disposizione quattro tratti fondamentali che sono `Read`, `BufRead`, `Write`, `Seek`.

15 Smart pointer

In C e C++ è lecito creare dei puntatori e derefernziarli con `*`, in Rust cosí come avere un tratto per usare l'uguaglianza tra due struct, o l'ordine, è lecito implementare il tratto per la dereferenziazione pur non essendo un puntatore, anche questo è possibile farlo in C++ con l'operator overloading. In Rust questo quando si vuole che una struttura viva fin tanto che esiste qualcuno che la referenzia si vuole che questa struttura venga mantenuta in vita, grazie al conteggio delle referenze. Il bisogno di gestire queste casistiche in C++ si sono introdotti gli **smart pointer**, questo stesso concetto è stato portato in Rust. In Rust ad esempio è impossibile creare una struttura ciclica, perché sorge la domanda, *chi possiede chi?*

In C++ si accede a un pezzo di memoria dinamica ci si deve assicurare che quel pezzo di memoria esiste e che il possessore sia uno solo, per questo motivo esiste il tipo `std::unique_ptr<T>` (che non può essere duplicato) e `std::shared_ptr<T>` (che può essere copiato).

In Rust vengono ripresi gli stessi concetti di C++ con `Box`, `Rc` (Reference Count), `Arc` (Atomic Reference Count), ... Una struct che implementa il tratto `Deref/DerefMut`, permette di utilizzare delle struct come se fossero delle referenze

```
trait Deref {
  type Target: ?Sized;
  fn deref(&self) -> &Self::Target;
}
trait DerefMut: Deref {
  type Target: ?Sized;
  fn deref_mut(&mut self) -> &mut Self::Target;
}
```

as esempio `Box<T>` si comporta come se possedesse una reference al tipo `T`, in più quando `Box` verra droppato rilascerà i dati contenuti al suo interno allocati sull'heap. `Rc` permette invece di contare il numero di volte che il dato è referenziato, solo quando il numero di conteggi `strong` e `weak` saranno zero la struttura `T` non verrà rilasciata. Per risolvere il problema dei cilci si utilizza `Weak`, per distinguere i puntatori che tengono in vita e quelli che non lo fanno, per ovviare al fatto di puntare ad una refernza nulla, per fare fare questo `Weak` offre `upgrade()`, che permette di ritornare ad `Rc` e se il dato è ancora esistente ci permette di accedere altrimenti no.

```
let five = Rc::new(5);
let weak_five = Rc::downgrade(&five);

let strong_five = weak_five.upgrade();
assert!(string_five.is_some());

drop(strong_five);
drop(five);

assert!(weak_five.upgrade().is_none());
```

Esistono delle situazioni in cui si potrebbe volere un possesso condiviso ad un dato sullo stack, questo è un problema perchè un solo una variabile può possedere un dato come mutabile, Rust permette però di cambiare un dato anche possedendo un refernce non mutabile, il borrow checker dice che: esiste un unico riferimento mutabile, ci sono più riferimenti, in alcuni casi questa condizione è troppo restrittiva, per questo motivo esiste `Cell<T>` che permette di scambiare un riferimento con un riferimento mutabile.

16 Programmazione Concorrente

Nella programmazione concorrente il thread pricipale chiede di attivare altri thread, chiedendo aiuto al sistema operativo, che a differenza dei processi essi non sono isolati ma condividono alcune delle loro risorse, il motivo per cui si vule creare un thread e che si possono svolgere delle attività in parallelo per velocizzare la computazione. Quando il thread viene sospeso dallo scheduler il SO salva al suo interno lo stato dei registri, oltre a questo mantiene lo stato del thread (RUNNABLE, NOT RUNNABLE, ...), inoltre ha anche delle code di thread runnable e delle code di thread di not runnable, una volta che un thread cerca di fare un operazione che non si può completare immediatamente allora viene messo nella coda dei not runnable e fa partire un altro thread dalla coda dei runnable, questi vengono detti **thread nativi**, e sono specifici per SO.

I SO mettono a disposizione per i thread:

- una funzione di creazine: quando viene chiamata viene allocato uno stack per ogni thread e la sua relativa struttura dati, quando il viene creato un thread esso rilascia un **handle** che permette di interagire col thread.
- una funzione di identificazione: un SO assegna un id univoco per ogni thread (**TID**).
- una funzione di attesa (join): permette di aspettare che un thread finisca in modo bloccante.
- una funzione di cancellazione: i SO non forniscono una funzione di cancellazione dei thread, perché è troppo difficile gestire questa cosa.

Abilitando la programmazione concorrente si creano una miriade di nuovi problemi che nella programmazione sequenziale non esistono, ad esempio validità della cache.

Qunado un thread legge il contenuto di una cella di momoria può:

- il valore iniziale con cui viene inizializzato il prgramma (come ad esempio la variabili statiche che vengono create in compilazione, come le strighe);
- un valore che ha insertito il thread se la zone di memoria è in scrittura;
- un valore scritto da un altro thread;

le cache rendono il 3 caso molto problematico,

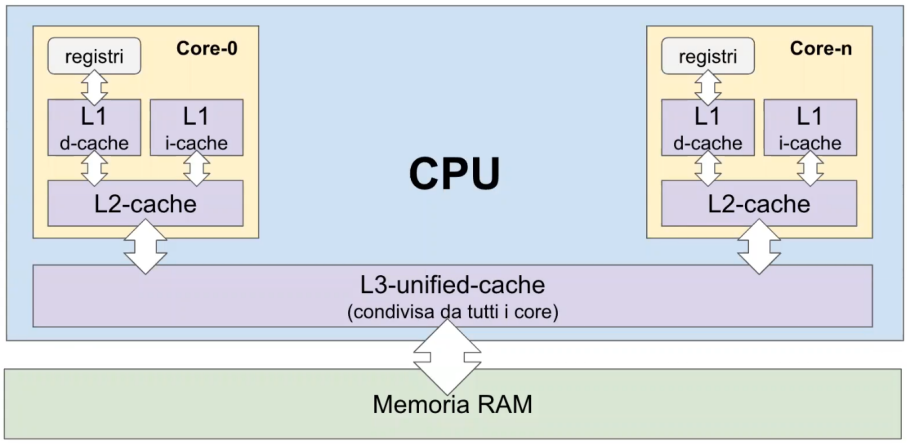


Figure 9: Modello Memoria

infatti quando si acced ad un indirizzi se un altro thread scrive nella cache possiamo travare qualsiasi cosa senza garanzia della sua correttezza, questo porta ad avere dei problemi aperti: atomicità, visibilità, ordinamento.

Atomicità: quali operazione che effettuiamo hanno degli effetti indivisibili? Ad esmempio una delle caratteristiche dei DB, questo vuol dire che un operazione viene fatta tutta in una volta, questo problema è sulla variabili globali e di istanza.

Visibilità: quando un thread fa un operazione gli altri quando possono vederlo? Questo è un problema di cache.

Ordinamento: in quale ordine appaiono le operazioni di più thread?

Per fare affronto a queste problemi i processori offrono delle soluzioni in hardware, ad esempio in x86 esistono delle operazioni **fence** che fanno un flush della cache fino alla RAM e bloccando gli altri core e invalidando le loro cache. I processari di tipo ARM usano la tecnica della **Change Propagation List**, quendo si cerca di leggere un leggere da una cella condivisa viene creata una dipendenza di lettura, la cache viene invalidata in modo essere sicuro di leggere un valore corretto, mentre in lettura viene fatto un flush, ARM offre anche delle **barrier** che garantisce una sequenza di operazioni siano fatte in modo causale.

Per sincrononizzare dei thread tra di loro si usano dei construtti appositi.

Example 16.1 – Interferenza

Il seguente programma in c++ lancia due thread in parallelo,

```
1 int a = 0;
2
3 void run() {
4   while (a >= 0) {
5     int before = a;
```

```
    a++;
    int after = a;
    if (after - before) != 1)
      std::cout << before << " -> " << after
  << "(" << (before-after) << ")" << std::endl;
  }
}

int main() {
  thread t1(run);
  thread t2(run);

  t1.join();
  t2.join();
}

```

In alcuni casi è possbile che un thread stampi una **differenza minore di zero**, il problema è l'istruzione `++` che di per se non è un operazione atomica, infatti sono 3 operazioni macchina: load, store, add. Se il thread viene sospeso mentre sta incrementando e nel frattempo il secondo scrive 10 volte il valore, il primo ha il valore vecchio e riporta la variabile `a` indietro. Questo tipo problema viene detto **Interferenza**.

Per far sia che il prgramma sia corretto, quando si accede ad una variabile per mutarne il valore in modo concorrente lo si fa sempre una alla volta,

In quasi tutti i linguaggi di programmazione sta all'utente gestire la sincronizzazione, in Rust questo non è possibile, infatti il concetto del single ownership impedisce di avere variabili mutabili condivise (**Fearless Programming**). Rust offre dei tipi di sincronizzazione nativi, come gli `Atomic` che garantiscono una lettera di un dato corretto ed una propagazione della cache fino alla RAM, i `Mutex` che forniscono due primitive: `lock` che quando viene chiamato controlla che il dato è libero e ne ritorna prendendone il possesso, altrimenti aspetta, e `unlock` che rilascia la risorsa. In Rust si crea un thread con `thread::spawn()`, dove il valore che gli si deve passare è una lambda, questa può essere una **closure**, ovvero può catturare delle variabili al suo interno, se si usa `move` le variabili che si possono catturare (prendere possesso) di tutte la variabili che implementano il tipo `Send` e tutte le reference che implementano il tratto `Sync` (la reference il lettura può essere condivisa). Quando vengono dichiarate delle nuove struct se tutti i tipi al loro interno hanno `Send` o `Sync` anche loro lo implementano di default.

Alcuni modelli di concorrenza in Rust

- **actix**: modello degli attori
- **rayon**: modello work stealing
- **crossbeam**: modello di canali generalizzati

16.1 Condivisione dello Stato

Siccome i thread vivono nello stesso ocntesto è possibile avere una vaiabile condivisa da tutti in cui è possibile sia leggere che scrivere. Per fare questo è presente un `Mutex`, esso ha due metodi: `lock()` che segna il thread da cui il lock è posseduto, `unlock()`. In C++ per implemetare i lock ad alto livello si usa `lock_guard`, che utilizza il pattern RAII, durante la creazione viene usato messo il lock e durante la distruzione viene rilasciato il lock. Questo approccio previene l'uso in modo errato dei lock, esiste però il problema dell'utilizzo inopportuno.

In Rust il dato che si vuole condividere è incapsulato dentro il `Mutex`, evitando così di potervi accedere senza l'uso del `Mutex`. Un `Mutex<T>` contiene

- **inner**: puntatore al mutex (nelle ultime versioni di Rust è contenuto direttamente all'intero della struttura)
- **poison**: ci permette di sapere se il mutex è stato rilasciato in modo appropriato, as esempio se il thread va in `panic` la stack viene svuotato ma la struttura dati potrebbe essere corrotta, è per qeusto motivo che quando si fa il `lock()` viene restituito un `Result`, proprio per gestire questo tipo di castica
- **T**: questo dato viene restituito dal metodo `lock()` all'interno di uno *smart pointer*

Un `Mutex` di suo non può essere posseduto da due thread, per questo si utilizza un `Arc`, dove il pattern tipico è

```
let shared_data = Arc::new(Mutex::new(vec![]));
let mut thread = vec![];
for i in 0..10 {
  let mut data = shared_data.clone(); // un Arc e' clonabile
  theads.push(thead::spawn(move|| {
    let mut v = data.lock().unwarap(); // v e' di tipo MutexGuard<T>
    v.push(i);
  }));  // il lock viene rilasciato con RAII
}
for t in theads { t.join().unwarap(); }
// shared_data contiene i numeri da 0 a 9
```

Il meccanismo della *single ownership* ci perette di proteggere un dato con un sigolo `Mutex` condiviso dai processi.

Un altro modo di condividere uno stato è quello di usare una reference (senza l'uso di `move`), con i thread però questo è un problema perché l'avanzamento dei due thead è diverso, questo comporta che se il chiamante termina per primo la reference esce al di fuori del suo tempo di vita, per questo motivo esiste `thead::scope(|s: std::thread::Scope| ... )`, che non fa partire il thead, ma prepare un context in cui si possono creare dei thread che si aspettano , questo viene sincronizzato in base al prametro `s` che viene passato, grazie a `scope` si possono passare tranquillamente i parametri tra i thread senza il bisogno di usare `Arc` o di fare delle `join`

```rust
let mut v = vec![1, 2, 3];
let mut x = 0;
thead::scope(|s| {
  s.spawn(|| {
    println!("length: {}", v.len());
  });
  s.spawn(|| {
    for n in &v {
      println!("{n}");
    }
    x += v[0] + v[2];
  });
});

v.push(4);
assert_eq!(x, v.len());
```

Oltre a `Mutex` esiste `RwLock` che si utilizza quando gli accessi in scrittura e lettura sono sbilanciati.

16.2 Attese Condizionate

In alcuni casi di multithread si vuole sapere se un altro thread ha fatto delle operazioni, una delle modalità per fare un check è il **polling** ovvero, mettendo in un loop se la variabile è stata settata, questa opzione è molto inefficente per la CPU, per questo i SO mettono a disposizione le **condition variable**, dove non viene implementato il busy-waiting (che è alla base del polling). Per utilizzare una condition variable va sempre accoppiata ad un mutex (quindi un dato condiviso), in Rust si ha `std::sync::Condvar`, che fornische dei metodi tra cui

```rust
pub fn wait<'a, T>(
  &self,
  gruard: MutexGuard<'a, T>
) -> LockResult<MutexGuard<'a, T>>

pub fn notify_one(&self)

pub fn notify_all(&self)
```

un esempio di utilizzo è

```rust
let pair = Arc::new((Mutex::new(false), Condvar::new()));
let pari2 = Arc::clone(&pair);

thread::spawn(move|| {
  let (lock, cv) = &*pair2;
  let mut started = lock.lock().unwrap();
  *started = true;
  // We notify the condvar that teh value has changed.
  cv.notify_one();
```

```
});

// Wait for the thread to start up.
let (lock, cv) = &*pair;
let mut started = lock.lock().unwrap();
while !*started {
  started = cv.wait(started).unwarap();
}
```

Quando viene chiamata la wait il thread corrente viene messo all'interno di una coda di attesa in cui il thread sta dormendo, quando viene chiamata la `notify_one` ne viene svegliato una, mentre can la `notify_all` vengono svegliati tutti i thread della coda che poi sono in concorrenza per riacquisire il lock. È possible che un thread in attesa possa essere sveglita senza che nessuno abbia chiamato notify, queste vengono dette **notifiche spurie** e il programmatore ne deve essere consapevole perché potrebbero avvenire in modo inpredicibile.

16.3 Condivisione di Messaggi

Nella programmazione concorrente esistone delle primitive dette **channel** che permettono di scrivere delle variabili in un channel e leggere delle variabili in un channel, questo è un approccio *leggere per sincronizzare* a differenza dei lock che hanno un approccio *sincronizzare per leggere*, un channel è come un tubo di collegamento in cui c'è un punto di entrata (in cui si può scrivere) ed un punto di uscita (da cui si può leggere), Rust mette a disposizione la funzione `std::mpsc::channel()` che ritorna una tupla `(Sender<T>, Receiver<T>)` (il canale è omogeneo, non è possibile avere tipi diversi), ripettivamente le struct implementano `send()` (che manda un dato) e `recv()` (che aspetta un dato in modo bloccante), l'implementazione che fornisce Rust è **single consumer - multiple producer**, infatti l'oggetto `Sender<T>` è clonabile.

Se per caso tutti i sender dovessero morire il receiver riceve un errore quando cerca di scrivere, così anche i sendere ricevono un errore quando quando provano a mandare un dato sul canale ed il receiver non esiste più.

I canali normalmente non hanno un limite a quante entry possono avere, in alcuni questo è buono in altri no, per questo Rust fornisce un `sync_channel<T>(buond: usize)`, che prende in input la grandezza massima del canale, se si utilizza la dimenzione 0 si un canale di tipo **rendezvous** dove ogni operazione di scrittura deve avvenire insieme ad una di lettura (entrambi sender e receiver devono essere presenti nello stesso istante per far si che l'operazione abbia successo).

Una libreria di Rust chiamata **crossbeam**, che fornisce costrutti avanzati per la sincronizzazione, come: `AtomicCell<T>`, `Injector`, `Stealer`, `Worker`, ... Per completare la descrizione dei canali **crossbeam** mette a disposizione dei canali **multiple producer - multipel consumer**, grazie a `crossbeam::channel::buonded(...), unbouned()`, che permette la distribuzione controllata del carico. Esistono anche canali senza entrata (**Fan-out**) o canali senza uscita (**Fan-in**), un esempio di **Fan-**

in/Fan-out

```rust
fn worker(id: usize, rx: Receiver<i32>, tx: Sender<String>) {
  while let Ok(value) = rx.recv() {
    tx.send(format!("W{} ({})", id, value));
  }
}

fn main() {
  let (tx_inupt, rx_input) = bounded::<i32>::(10);
  let (tx_output, rx_output) = bounded::<String>::(10);

  let mut worker_handles = Vec::new();
  for i in 0..3 {
    let rx = rx_input.clone();
    let tc = tx_output.clone();
    worker_handles.push(thread::spawn(move|| worker(i, rx, tx)));
  }

  for i in 1..=10 { tx_inptu.send(i).unwrap(); }

  drop(tx_input);
  whiel let Ok(result) = rx_output.recv() {
    println!("Received results: {}", result);
  }
  for handle in worker_handles { handle.join().unwrap(); }
}
```

in altre situazioni si hanno bisogno di **pipeline**, seguire delle operazioni una dopo l'altra, dove il prodotto intermedio viene poi terminato con un **consumer**

```rust
fn stage_one(rx: Receiver<i32>, tx: Sender<String>) {
  while let Ok(value) = rx.recv() {
    tx.send(format!("S1({})", value)).unwrap();
  }
}

fn stage_two(rx: Receiver<String>, tx: Sender<String>) {
  while let Ok(value) = rx.recv() {
    tx.send(format!("S2( {} )", value)).unwrap();
  }
}

fn main() {
  let (tx_input, rx_input) = bounded::<i32>(10);
  let (tx_stage_one, rx_stage_one) = bounded::<String>(10);
  let (tx_output, rx_output) = bounded::<String>(10);

  let stage_one_handle = thread::spawn(move ||
    stage_one(rx_input, tx_stage_one)
  );
  let stage_two_handle = thread::spawn(move ||
```

```
    stage_two(rx_stage_one, tx_output)
  );

  for i in 1..=10 { tx_input.send(i).unwrap(); }
  drop(tx_input);

  while let Ok(result) = rx_output.recv() {
    println!("Received result: {}", result);
  }

  stage_one_handle.join().unwrap();
  stage_two_handle.join().unwrap();
}
```

si poi realizzare anche modelli di mutliple consumer - multiple producer, in cui il channel è condiviso ed è unico

```
fn producer(id: usize, tx: Sender<(usize,i32)>) {
  for i in 1..=5 { tx.send((id,i)).unwrap(); }
}

fn consumer(id: usize, rx: Receiver<(usize,i32)>) {
  while let Ok((sender_id, val)) = rx.recv() {
    println!("Consumer {} received {} from {}", id, val, sender_id);
  }
}

fn main() {
  let (tx, rx) = bounded::<(usize,i32)>(10);

  let mut handles = Vec::new();
  for i in 0..3 {
    let tx = tx.clone();
    handles.push( thread::spawn(move || producer(i, tx)) );
  }
  for i in 0..2 {
    let rx = rx.clone();
    handles.push(thread::spawn(move || consumer(i, rx)));
  }
  drop(tx);
  for handle in handles { handle.join().unwrap(); }
}
```

Un altro modello molto usato è quello degli **Attori**, dove quando si vogliono mandare dei messaggi questi sono depositati in una **mailbox** (ovvero una coda FIFO), che poi l'attore (ovvero un thread) periodicamente controlla questa mailbox e ogni volta che trava un messaggio compie delle azioni, Windows ad esempio fa un uso pesante del modello degli attori all'interno della gestione delle finestre di sistema e degli eventi.

17 Processi

I processi sono degli oggetti isolati all'interno del sistema operativo, se due processi vogliono comunicare tra di loro delle strutture dati, non è possibile farlo passando direttamente la struttura, i puntatori all'interno punterebbero a zone di memoria diverse da quella dell'atro processo, un modo di farre questo è quello di **serializzare** (tramite json) o **marshaling**.

...

In Rust non si ha l'accesso diretto alla `fork` ma esiste un wrapper per poter creare direttamente un nuovo processo, la libreria ci fornisce `Command` che il implementa il pattern Builder (perchè hh molti parametri di input).

....

```rust
let mut child = Command::new("rev")
  .stdin(Stdio::piped())
  .stdout(Stdio::piped())
  .spwan()
  .expect("Failed to spawn a child");

let mut stdio = child
  .stdin
  .take()
  .expect("Failet to open stdin");

thread::spawn(move|| {
  stdin.write_all("Hello, world!".as_byte())
    .expect("Failed to write to stdin");
});

let output = child
  .wait_with_output()
  .expect("Failet to read stdou");

assert_eq!(String::from_utf8_lossy(&output.stdout), "!dlrow ,olleH");
```

In linux esistone dei problemi quando si utilizza la fork, il motivo è che se ad esempio si stavo scrivendo in un output, questo è bufferizzato, la fork fa una copy-on-write quindi se qualche buffer si trova dei dati vecchi il nuovo processo andrà a scrivere in modo concatenate a questi vecchi dati, per questo è sempre necessario **fare un flush di tutti i file aperti** (stdout, stderr inclusi).

In Rust per terminare un processo e tutti i suoi thread in modo immediato è chiamare `std::process:exit(code: i32)`, che a differenza di `panic!` non fa contrazione dello stack.

Quando vengono scambiate delle informazioni tra processi esistono due famiglie di formati: **testuali** (XML, JSON, CSV, ...), **binari** (XDR, HDF, protobuf, ...), ovviamente la rappresentazione testuale è facilmente leggibile in debug e difficile fare degli errori, mentre con quella binaria si raggiunge la massima velocità e compattezza.

I SO mettono a dispozione delle API per far comunicare i messaggi, una di queste è la **Coda dei Messaggi** che permette di mettere dei messaggi in una coda, mentre permette ad un altro processo di leggere il messaggio, consumandolo, la la tecnica utilizzata è Single Consumer Multiple Producers. In Windows si chiamano `Mailslot` (`CreateMailslot()`), in Linux esiste `fifo` (`mkfifo()`). Un altra tecnica che si può utilizzare sono le `Pipe`, questo è possibile quando i processi hanno una parentela stretta, nelle `Pipe` i messaggi non sono segmentati, questo signica che si possono leggere e scrivere un numero a piacere di bytes.

Le Pipe in Rust si ...

Si vuole relizzare un programma che

```
1 fn start_process(sender: Sender<String>, receiver: Receiver<String>)
```

Orchestrare questa gestione viene fatto per evitare che il programma si blocchi quando cerca di scrivere o leggere l'input/ouput di un altro programma, per evitare questo spreco di risorse si può utilizzare la programmazione asincrona.

18 Programmazione Asincrona

RIGUARDARE LA LEZIONE

I casi in cui la programmazione multi-thread non è ottimale sono quelli in cui la computazione richieda di ricevere informazioni da un sottosistema separato, e dove solitamente la chiamate sono bloccanti. **Si usano i thread quando si vuole elaborare in parallelo. Si usa la programmazione asincrona quando si vuole attenere in parallelo**. Il più diretto per implementatre una chiamata asincrona è quello di avere una syscall che prende in input le operazioni che deve fare e un callback che verrà chiamata quando la l'operazione è terminata. Ad esempio in JavaScript esiste l'**event loop**, che non fa altro che leggere la coda dei messaggi ed esguire le callback associate.

Anche in Rust è possibile usare la programmazione asincrona, infatti il linguaggio offre un supporto alle keyword `async`/`await`, per usare la programmazione il codice però va trasformato in una macchina a stati finita, dove ogni step che si fa nel codice rappresenta un stato all'interno della macchina.

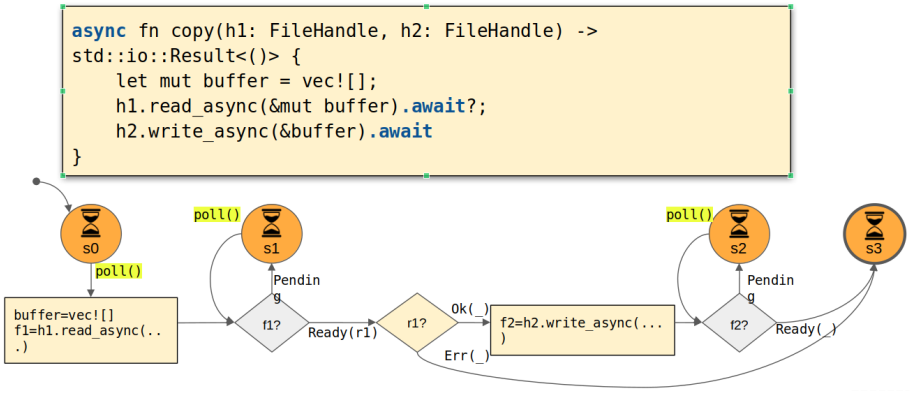


Figure 10: Macchina A Stati Finiti

Quando si usa la la programmazione asincrona, è possibbile realizzare una **chiusura** dove viene salvato la stato in cui ci traviamo della macchina che rappresenta i passi all'interno delle callback. Perché si realizza tutto questo? Si fa questo grande sforzo per non creare un **callback hell**.

Ogni volta che viene dichiarata un funzione async, il compilatore di Rust trasforma il valore di ritorno in un `Future` che ritorna il tipo originale, inoltre anche l'interno della funzione viene completamente modificato. `Future` implementa un unico metodo, `poll`

```
pub trait Future {
  type Output;

  fn poll(self: Pin<&mut Self>, cx: &mut Context)
    -> Poll<Self::Output>
}
```

`Pin` è uno smart pointer che impedisce il movimento. Context incapsula un oggetto `Waker` il quale dice al metodo `poll` quando può essere richiamato. Se si vuole che avvenga qualcosa si deve chiamare almeno una volta in metodo `poll`, infatti `Future` di per se è **inerte**, se il metodo ritorna `Pending` si aspetta e poi si richiama `poll` finché non ritorna `Ready`.

Prima di tutto Rust si crea delle `struct` che rappresentano le dipendenze per ogni stato della macchina, in questo caso per la figura 10 sono

```
struct S0 {
  h1: FileHandle,
  h2: FileHandle,
}

struct S1 {
  h2: FileHandle,
  buffer: Vec<u8>,
  f1: impl Future<Output=Result<usize>>,
}
```

```

struct S2 {
  f2: impl Future<Output=Result<usize>>,
}

struct S3 {}
```

inlotre Rust genera anche un `enum` che racchiude tutti i possibili stati

```
enum CopySM {
  s0(S0),
  s1(S1),
  s2(S2),
  s3(S3)
}

impl Future for CopySM {
  type Output = std::io::Result<()>

  fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output> {
    loop {
      match self {
        CopySM::s0(state) => { ... },
        CopySM::s1(state) => { ... },
        CopySM::s2(state) => { ... },
        CopySM::s3(state) => { ... },
      }
    }
  }
}
```

Ogni braccio contiene il seguente codice

```
CopySM::s0(state) => {
  let mut buffer = vec![];
  let f1 = state.h1.read_async(&mut buffer);
  let state = S1 {h2: state.h2, buffer, f1 };
  *self = CopySM::S1(state);
}

CopySM::s1(state) => {
  match state.f1.poll(cx) {
    Poll::Pending => return Poll::Pending,
    Poll::Ready(r1) =>
      if r1.is_ok() {
        let f2 = state.h2.write_async(&mut state.buffer);
        let state = S2{ f2 };
        *self = CopySM::s2(state);
      } else {
        *self = CopySM::s3(S3);
        return Poll::Ready(r1);
      }
  }
```

```
}

CopySM::s2(state) => {
  match state.f2.poll(cx) {
    Poll::Pending => return Poll::Pending,
    Poll::Ready(r2) =>
        *self = CopySM::s3(S3);
        return Poll::Ready(r2);
  }
}

CopySM::s3(_) => {
  panic!("poll() was invoked again after Poll::Ready has been returned");
}
```

Il codice generato per la funzione originale si limita ad una semplice inizializzazione della macchina a stati

```
fn copy(h1: FileHandle, h2: FileHandle) -> impl Future<Output = std::io::Result<()
    >> {
  CopySM::s0(
    S0 { h1, h2 }
  )
}
```

In modo nativo Rust non ha nessun modo di chimare il metodo `poll`, la gestione dell'esecuzione viene scelta dal programmatore scegliendo una libreria scritta appositamente, la scelta è qualla di associare al nostro codice un **Executor**, si possono scegliere tra tre librerie (le migliori): Tokio, smol, async-std (che fornisce una controparte asincrona per ogni funzione della libreria standard, a metà tra tokio e smol) (*"È un po come l'acqua tiepida, fa schifo a tutti" .cit Malnati*).

18.1 Tokio

Si può usare la libreira tokio per scriver del codice asincrono in Rust

```
#[tokio::main]
async fn main() {
  let task = tokio::spawn(async { println!("Hello"); });
  task.await.unwrap();
}
```

tokio fornisce anche un modo di controllore i task che venono creati, una di esse è la macro `join!` che premetter di prendere più `Future` e attendere che onguno di essi finisca, oppure `try_join!` che ritorna se una dei `Future` è fallitto oppure `Ok()` in caso tutti terminano con successo. In alcuni casi vuole ritorna prima che tutti i task finiscano, la `select!` sblocca al primo `Future` che ritorna, che permette di fare un match su quale dei `Future` ritorna

```
async fn do_stuff_async() {}
```

```
async fn more_async_work() {}

tokio::select! {
  _ = do_stuff_async() => {
    println!("do() completed first");
  }
  _ = more_async_work() => {
    println!("more() completed first")
  }
};
```

è possibile fare un pattern matching sui valori di ritorno, il primo match che trovato viene eseguito il ramo relativo, possiamo aggiungere anche un match `else` che nel caso in cui i match falliscano viene eseguita quella condizione. Esiste anche un caso in cui si vuole fare uno sleep, questo è utile quando, as esempio, si fanno delle chiamate di rete e si vuole creare un timeout per la risposta (`tokio::time::sleep(d: Duration).await`/`tokio::time::timeout(d: Duration, f: F).await`). In alcuni casi esistono dei task che devono fare delle operazioni molto lunghe, che potrebbero tenere la CPU occupata per molto tempo senza mai rilasciarla, una soluzione potrebbe essere quella di mettere delle splee in giro per il programma in grado pche dare indietro il possesso della CPU, questo però non è molto bello, per qeusto tokio offre `tokio::task::spawn_blocking(f: FnOnce() -> R)`, che wrappa la funzione e verrà sospesa ongi tanto.

In alcuni casi si deve condividere uno stato tra più task, tokio ci mette a disposizione delle primitivi simili a quelle dei thread oltre a fornire dei canali di comunicazione molti sofisticati.

tokio implementa: `Arc/Mutex`, `Barrier`, `Notify`, `RwLock`, `Semaphore`.

tokio implementa dei canali di comunicazione, uno di essi è `oneshot`, è pensato per poter permetter la comunicazione di un solo dato

```
let (tx, rx) = oneshot::channel();

tokio::spawn(async move {
  let res = some_computation().await;
  tx.send(res).unwrap();
});

let res = rx.await.unwrap();
```

i canali `mpsc` hanno bisogno di avere una dimensione, simili a quelle della libreria standard. Poi esistono i canali `broadcast` che permettono di avere più ricevitori dello stesso messaggio

```
let (tx, mut rx1) = broadcast::channel(16);
let mut rx2 = tx.subscribe();

tokio::spawn(async move {
  assert_eq!(rx1.recv().await.unwrap(), 10);
  assert_eq!(rx1.recv().await.unwrap(), 20);
```

```
});

tokio::spawn(async move {
  assert_eq!(rx2.recv().await.unwrap(), 10);
  assert_eq!(rx2.recv().await.unwrap(), 20);
});

tx.send(10).unwrap();
tx.send(20).unwrap();
```

Il receiver non è clonabilie perché è il destinatario di un puntatore mandato du un trasmettitore, se lo clonassimo il trasmettitore non sarebbe in grado di capire quanti receiver hanno fatto il subscribe.

Un altro canale è il `watch` che implementa il **pattern Observer**, (una libreria che implementa il patter Observer è RXJava)

```
#[tokio::main]
async fn main() {
  let (tx, mut rx) = watch::channel("value 0");

  for i in 0..2 {
    let mut rx = rx.clone();

    tokio::spawn(async move {
      while rx.changed().await.is_ok() {
        println!("received: {:?}", *rx.borrowed());
      }
    });
  }

  let d = Duration::from_secs(1);

  tx.send("value 1").unwrap();
  tokio::time::sleep(d).await;
  tx.send("value 2").unwrap();
  tokio::time::sleep(d).await;
}
```

Avere questo tipo di canale è molto comodo per implementare un server HTTP

```
#[tokio::main]
async fn main() {
  let listener = TcpListner::bind("127.0.0.1:8181").await.unwrap();

  loop {
    let (stream, _) = listner.accept().await.unwrap();
    tokio::spawn(handle_connection(stream));
  }
}
```

19 Esame

- Si definisca il concetto di SmartPointer, quindi si fornisca un esempio in Rust che ne evidenzi il ciclo di vita. Il concetto di smartPointer e' molto potente, infatti permette di poter trattare delle strutture dati come se puntatori, benche' essi non lo siano, in Rust questo e' possibile grazie ai tratti Derf e DerfMut che se implementati permettono alla struct che li implenta di poter ritornare una ref o una ref mutabile in caso venga chiamato l'operatore * su di essi, il concetto di smart pointer e' fortemente collegato a quello del pattern RAII (Resource Acquisition Is Initialization) che permette la gestione gestione automatica del liberamento delle risorse dallo heap.

```
struct Person {
  name: String,
  age: u32,
}

fn getPerson() -> Box<Person> {
  let  p = Box::new(Person { ... });
  return p;
}

fn main() {
  let person = getPerson();

  // Questo stampera la persona
  println!("{}{}", (*persona).name, (*persona).age);
}

```

  Box e' uno smart pointer in rust che permette di implementare questo pattern, infatti il valore che prende in new sara allaocato nello heap, il tempo di vita di Person inizia quando viene inizializzato e termina quando person esce di scope (in questo caso alla fine della funzione), e quando il drop verra chiamato su di esso, il puntatore interno a box che punta a Person nello heap verra' liberato. Questo garantisce sempre una corretta liberazione della variabili dallo heap.

- Si illustrino le differenze nel linguaggio Rust tra std::channel() e std::sync_channel(), indicando quali tipi di sincronizzazione i due meccanismi permettono. Entrambi i offrono dei meccanimi di sincronizzazione basati su canali di comunicazione che sono Multiple Producer Single Consumer, e ritornano rispettivamente (Sender Receiver) (SenderSync REceiver), infatti i sender sono clonabili mentre i receiver no, in channel() i canali sono unbounded, voul dire che ogni volta che i sender mandano dei dati ritornano subito, i receiver sono bloccati finche' non ricevono qualcosa dal canale (se un dato e' presente lo ritornano subito), mentre con con syncchannel() e' possibile craere dei canali bounded, voul dire che si crea un buffer e se questo viene riempito prima che il consumer possa legger

leggere i dati i sender che provano a fare send vengono bloccati finche' il buffer non ha spazio, si crea un canale can grandezza 0, allora viene detto rendezvous dove per poter avvenire uno scambio di informazione sia il sender che il receiver devono essere presenti.

- ```
  Dato ii seguente frammento di codice Rust (ogni linea e preceduta dal suo
      indice)
  1. struct Point {
  2. x: i16,
  3. y: i16,
  4. }
  5.
  6. enum PathCommand {
  7. Move(Point),
  8. Line(Point),
  9. Close,
  10. }
  11. let mut v = Vec::<PathCommand>::new();
  12. v.push(PathCommand::Move(Point{x:1,y:1 }));
  13. v.push(PathCommand::Line(Point{x:10, y:20}));
  14. v.push(PathCommand::C/ose);
  15. let slice = &v[..];
  Si descriva ii contenuto dello stack e dello heap al termine dell'esecuzione
       della riga
  15
  ```

  Nello heap ci saranno in modo sequenziale le strutture dati Move(Pointex:1,y:1), Line(Pointx:10, y:20), Close, dove ogni casella sara grande quanto la grandezza somma dei vari enum di PathCommand, che si compone di, un campo per il tag dell'enum, un campo e per x ed uno per y (4 byte, 2byte, 2byte, dove i 2 byte sono allineati), visto che entrambi gli enum Move e Line usano Point all'interno di una tupla, anche Close avra' la stessa grandezza benche' non ha campi, nella stack sono presenti tre valori per v, infatti v e' un double pointer nello stack che mantiene, il puntatore alla sequenza di enum, il campo size che conta quanti valori sono presenti all'interno della struttura, ed il campo di capacity che dice quanti caselle sono allocate nello heap (4 solitamente), poi sono presenti dati per slice che nello stack rappresenta un fat pointer, che continene al suo interno un compo per il puntatore al primo elemento di uno Slice (Move(...) in questo caso) ed un altro campo, size, che dice quanti elementi sono presenti nello slice, che in questo caso e' 3.

- Un paradigma frequentemente usato nei sistemi reattivi e costituito dall'astrazione detta Looper. Quando viene creato, un Looper crea una coda di oggetti generici di tipo Message ed un thread. II thread attende - senza consumare cicli di CPU - che siano presenti messaggi nella coda, Ii estrae a uno a uno nell'ordine di arrivo, e Ii elabora. II costruttore di Looper riceve due parametri, entrambi
  ```

di tipo (puntatore a) funzione: process(...) e cleanup(). La prim a e una funzione responsabile di elaborare i singoli messaggi ricevuti attraverso la coda; tale funzione accetta un unico parametro in ingresso di tipo Message e non ritorna nulla; La seconda e funzione priva di argomenti e valore di ritorno e verra invocata dal thread incapsulato nel Looper quando esso stara per terminare. Looper offre un unico metodo pubblico, thread safe, oltre a quelli di servizio, necessari per gestirne ii ciclo di vita: send(msg), che accetta come parametro un oggetto generico di tipo Message che verra inserito nella coda e successivamente estratto dal thread ed inoltrato alla funzione di elaborazione. Quando un oggetto Looper viene distrutto, occorre fare in modo che ii thread contenuto al suo interno invochi la seconda funzione passata nel costruttore e poi termini. Si implementi, utilizzando ii linguaggio Rust o C++, tale astrazione tenendo canto che i suoi metodi dovranno essere thread-safe.

```
enum WorkerMsg {
  Work(Message),
  Stop,
}
struct Looper {
  handle: JoinHandle<()>,
  worker_msg_sx: Sender<WorkderMsg>,
}

impl Looper {
    fn new<F1, F2>(process: F1, cleanup, F2) -> Self
    where F1: FnOnce(Message) -> () + Send + 'static,
    F1: FnOnce() -> () + Send + 'static, {

      let (worker_msg_sx: Sender<_>, worker_msg_rx: Receiver<_>) = mpsc::
    channel();

      let worker = thread::spawn(move || loop {
        match worker_msg_rx.recv.unwrap() {
          WorkerMsg::Stop => {
            cleanup();
            break;
          }
          WorkerMsg::Work(msg) => {
            process(msg);
          }
        }
      });

      Self {
        handle: worker,
        wroker_msg_sx,
      }
    }

```

```
35      fn send(&mut self, msg: Message) -> Result<()> {
36        self.worker_msg_sx.send(WorkerMsg::Work(msg))?;
37        Ok(())
38      }
39
40      fn stop(self) -> Result<(), Box::<dyn Error>> {
41        self.worker_msg_sx.send(WorkerMsg::Stop)?;
42        self.handle.join()?;
43        Ok(())
44      }
45  }
46
```