01 - PIATTAFORME DI ESECUZIONE

Fatta eccezione per i sistemi più elementari, l'esecuzione di un'applicazione avviene nel contesto di un sistema operativo, il quale offre un insieme di servizi, funzionalità e convenzioni che ne permettono il funzionamento, che prendono il nome di API (Application Programming Interface).
Le ABI (Application Binary Interface) definiscono invece quale formato debba avere un prodotto software per essere compatibile con un determinato sistema operativo.

Per sviluppare un programma che si interfacci direttamente con il sistema operativo, occorre accedere e conoscere le API da chiamare (file header), le strutture dati coinvolte e le convenzioni definite dal SO (strutture che descrivono lo stato del sistema ed a cui si può accedere tramite API in modo indiretto attraverso riferimenti opachi come HANDLE di winzoz o FileDescriptor in Linux).

Tutte le funzioni invocate possono anche fallire, per cui è bene anche gestire gli errori, controllando il valore di ritorno di ogni API per capire SE c'è stato un errore e QUALE errore sia.

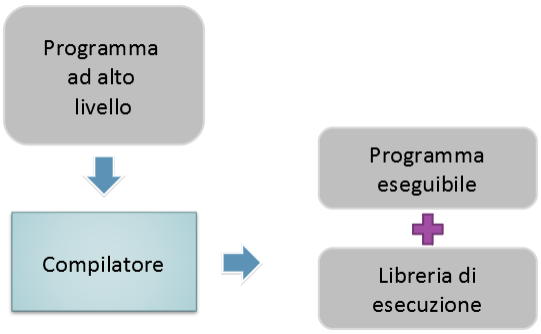
Siccome la rappresentazione dei caratteri a 8 bit è insufficiente per la maggior parte degli alfabeti, occorre anche avere un meccanismo di codifica e gestione del testo (possibili rappresentazioni: UTF-32, UTF-16, UTF-8) che diventa particolarmente complicato nel caso di rappresentazioni a lunghezza variabile, in quanto si possono avere ordinamenti differenti (es: big endian o little endian): spesso quindi si adotta una codifica a 16 bit ignorando i caratteri rari.
In Windows ci sono due tipi base: il char (8 bit) e il wchar_t (16 bit).
In Linux di default GCC codifica eventuali caratteri non-ASCII con UTF-8, che però può creare problemi (per esempio *strlen(str)* non indica più il numero di caratteri effettivamente presenti ma solo il numero di byte non nulli).

Gli standard POSIX (acronimo per Portable Operating System Interface) sono una famigli di standard per mantenere la compatibilità tra i diversi sistemi operativi. Sono delle API a basso livello e un set di comandi *shell* volti a rendere portabili gli applicativi.

02 - IL MODELLO DI ESECUZIONE

Ogni linguaggio di programmazione propone uno specifico modello di esecuzione, che non corrisponde quasi mai a quello di un dispositivo reale. E' compito del compilatore introdurre uno strato di adattamento che implementi il modello nei termini offerti dal dispositivo soggiacente.
Il programma originale quindi viene trasformato in uno nuovo dotato di un modello di esecuzione più semplice tramite la traduzione fatta dal compilatore.

Le librerie di esecuzione offrono agli applicativi meccanismi di base per il loro funzionamento e sono costituite da due tipi di funzione: alcune sono inserite in fase di compilazione per supportare l'esecuzione (come il controllo dello stack) e sono quindi invisibili al programmatore; altre offrono funzionalità standard che gestiscono opportune strutture dati ausiliarie e/o richiedendo al SO quelle non altrimenti realizzabili (malloc).



Un programma C/C++ assume di poter accedere a qualsiasi indirizzo di memoria all'interno del quale può leggere o scrivere dati o dal quale può eseguire codice macchina. Inoltre, l'insieme di istruzioni che definiscono il programma vengono eseguite, una per volta, nell'ordine indicato dal programmatore.
Il punto di partenza del flusso di esecuzione è predefinito (*main()* in C e i costruttori delle variabili globali in C++) e lo stack permette di gestire chiamate annidate tra le funzioni.

Ad ogni modo, le sole librerie di esecuzione non possono implementare tutte le astrazioni del modello di esecuzione, si ha bisogno che i programmi siano separati in modo tale che un malfunzionamento di uno non vada a toccare gli altri. Per fare ciò venne introdotto il concetto di processo, che porta con sè il contesto di un programma e vengono gestiti dal SO (non solo a livello di creazione ma anche di gestione di flusso di esecuzione).
Quando un processo viene selezionato per l'esecuzione, il SO configura il processore con le relative risorse e ripristina lo stato ad esse associato: si richiede quindi un supporto hardware per distinguere due modalità di esecuzione:

- Modalità **utente**: può eseguire un sottoinsieme delle istruzioni offerte dalla CPU ed esegue il codice associato ai diversi processi, sottostando a vincoli di accesso sia per memoria che per periferiche

- Modalità **supervisore**: crea e gestisce le strutture dati che modellano i processi, il loro spazio di indirizzamento e le altre risorse che posseggono. Accesso illimitato a tutte le funzionalità e parti (incluse periferiche, filesystem e rete)

L'innalzamento di privilegio comporta la sostituzione dello stato della CPU e delle struttture di supporto all'esecuzione tramite istruzioni sicure, per cui il costo per attraversa la barriera tra le due modalità è elevato.

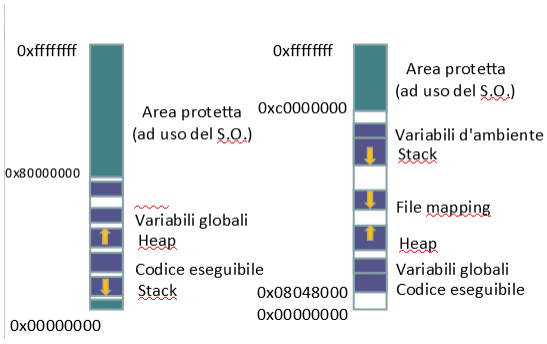
L'esecuzione di un programma comporta quindi la creazione di un nuovo processo, articolata in diverse fasi:

1. Creazione spazio di indirizzamento: insieme di locazioni di memoria accessibili tramite indirizzo virtuale. Attraverso la MMU del processore, gli indirizzi virtuali vengono tradotti in fisici, per cui si possono creare innumerevoli spazi tra loro separati. Attenzione ai *page_fault* (tentativi di accesso a indirizzi non mappati)
2. Caricamento dell'eseguibile in memoria: dopo aver inizializzato lo spazio di indirizzamento, il loader inserisce il contenuto del programma da eseguire (incluse le dipendenze). Si hanno quindi due sezioni, una per il codice e una per i dati e altre sezioni per stack e heap
3. Caricamento delle librerie: una sezione del file eseguibile è dedicata ad elencare ulteriori eseguibili necessari, che vengono mappati ricorsivamente nello spazio di indirizzamento, mentre tutti i riferimenti a variabili/funzioni delle dipendenze vengono aggiornati con gli indirizzi effettivi.
4. Avvio dell'esecuzione: quando il processo di caricamento termina, è possibile avviare l'esecuzione, tramite un'apposita funzione della libreria di esecuzione. Il comportamento di tale funzione dipende dal linguaggio di programmazione e dal SO, ma in sostanza inizializza stack, registri e strutture per gestire eccezioni; invoca costruttori e distruttori di oggetti globali e la funzione principale e infine rilascia l'intero spazio di indirizzamento (con le risorse in esso allocate) attraverso la system call *exit()*.

03 - ALLOCAZIONE DELLA MEMORIA

Quando un processo viene creato, il suo spazio di indirizzamento viene popolato con diverse aree, ciascuna dotata di propri criteri di accesso (non puoi? Arresto il processo e lancio Access Violation su Windows e Segmentation Fault su Linux).

- Codice eseguibile: contiene le istruzioni in codice macchina ed ha accesso in lettura ed esecuzione
- Costanti: accesso in sola lettura
- Variabili globali: accesso lettura/scrittura
- Stack: contiene indirizzi e valori di ritorno, parametri e variabili locali ed ha accesso lettura/scrittura
- Free store o heap: insieme di blocchi di memoria disponibili per l'allocazione dinamica e gestiti tramite funzioni di libreria (come *new, malloc, free, ...*) che li frammentano e ricompattano in base alle richieste del programma



Le variabili globali hanno un indirizzo fisico, determinato dal compilatore e dal linker e sono accessibili in ogni momento. Le variabili locali hanno un indirizzo relativo alla cima dello stack, con ciclo di vita uguale a quello del blocco in cui sono dichiarate. Le variabili dinamiche, infine, hanno un indirizzo assoluto, determinato in fase di esecuzione e accessibili solo mediante puntatori: il loro ciclo di vita è deciso dal programmatore.

In C++ viene definito il costrutto *new NomeClasse* per allocare nello heap un blocco di dimensioni opportune, invocando il costruttore della classe sul blocco e restituendo il puntatore ad esso. Per averne di più? Uso *new[] NomeClasse*!
Poichè ogni funzione di allocazione mantiene le proprie strutture dati occorre che un blocco sia rilasciato dalle funzioni duali di quelle con cui è stato creato (*delete* e *delete[]*) e se il blocco viene rilasciato con la funzione sbagliata di rischia di corrompere le strutture dati degli allocatori con conseguenze imprevedibili.

I puntatori sono uno strumento che permette di accedere qui ed ora ad un'informazione fornita da altri: per accedere ad array monodimensionali di dati, ad esempio, il compilatore trasforma gli accessi agli array in operazioni aritmetiche sui puntatori, e il programmatore perde così di vista l'effettiva dimensione della struttura dati. Occhio a non spostare il puntatore al di fuori della sua zona di competenza!
In ogni caso, è uno strumento molto potente, usato come base per implementare strutture dati composte come liste, mappe, grafi. Possibili rischi nell'utilizzo dei puntatori?

- Dangling pointer: accedo ad un indirizzo quando il corrispondente ciclo di vita è terminato. Effetto impredicibile
- Memory leakage: non rilascio la memoria non più in uso, sprecando risorse del sistema e potenzialmente saturando lo spazio di indirizzamento
- Segmentation fault: assegno ad un puntatore un indirizzo non mappato e poi lo utilizzo
- Wild pointer: non inizializzo un puntatore e lo uso, dove punta?

Il vincolo di rilascio risulta problematico per via di ambiguità nel linguaggio (dato un indirizzo non nullo, non è possibile distinguere a quale area appartenga nè se sia valido o meno), per cui ogni volta in cui si alloca un blocco dinamico se ne salva l'indirizzo in un puntatore, quella variabile diventa proprietaria del blocco ed ha la responsabilità di liberarlo.
Ma non tutti i puntatori posseggono il blocco a cui puntano (se ad un puntatore viene assegnato l'indirizzo di un'altra variabile, la proprietà è della libreria di esecuzione) ed il problema si complica se un puntatore che possiede il proprio blocco viene copiato (chi può liberarlo?)

In Linux, quando il processo viene inizializzato, viene creato un blocco di memoria a disposizione dello heap, utilizzato per le funzioni di allocazione. Occorre richiedere altra memoria in caso di problemi di dimensioni. Il sistema mantiene compatta l'area dello heap delimitandola con due puntatori interni al sistema (*start_brk* e *brk*) e, utilizzando la system call *sbrk()* posso spostare la locazione del program break della quantità indicata, aumentando o diminuendo l'area dello heap.

04 - IL LINGUAGGIO C++

Il C++ è un linguaggio progettato per garantire un'elevata espressività senza penalizzare le prestazioni. Gli obbiettivi del linguaggio sono quelli di avere astrazione a costo nullo, alta espressività (le classi definite dall'utente devono possedere lo stesso livello di espressività offerto ai tipi base) e alta sostituzione (posso usare una classe utente ovunque si possa usare un tipo base). Inoltre, offre supporto alla programmazione ad oggetti come a molti altri stili di programmazione.

Una classe C++ modella una struttura dati con supporto all'incapsulamento: può contenere variabili e funzioni legate all'istanza. Una funzione membro è simile ad un metodo Java: ha un parametro implicito ("*this*") che rappresenta l'indirizzo dell'istanza e la sua implementazione può essere inline (ossia all'interno della definizione della classe stessa) oppure separata (nella stessa unità di compilazione o in un altro modulo collegato).

Variabili e funzioni membro possono avere un diverso grado di accessibilità: per default sono *private*, ma possono essere anche *public* o *protected*.
Ad ogni oggetto corrisponde un blocco di memoria e può essere allocato in varie aree (occhio agli attributi statici, che sono sempre allocati nella memoria globale). Un costruttore ha il compito di inizializzare lo stato di un oggetto, ha il nome che coincide con quello della classe, non indica alcun tipo di ritorno ma sono possibili costruttori differenti se hanno parametri diversi. I costruttori UDT (definiti dall'utente) possono esseguire operazioni più complicate del semplice "copio i valori di ingresso nelle mie variabili di istanza" come allocare dinamicamente un vettore con l'operatore *new*.
Ogniqualvolta si alloca una variabile, ne viene invocato il costruttore (generato automaticamente se non si è specificato) mentre le variabili locali e globali sono inizializzate semplicemente dichiarandone tipo, nome e argomenti di inizializzazione.

Un oggetto può, inoltre, essere inizializzato a partire da un altro oggetto, istanza della stessa classe. Per i tipi base è easy, ma per gli UDT occorrerebbe definire un costruttore di copia, che riceve un riferimento costante all'originale di cui si vuole creare una copia e, ispezionando le singole variabili di istanza provvede a duplicarle nell'oggetto la cui creazione è in corso:

```
// esempio semplice di costruttore di copia
class CBuffer{ int size; }
CBuffer(const CBuffer& source);
CBuffer(const CBuffer& source) = delete; // non posso copiarlo!
```

Copiare un oggetto in un altro può essere un'operazione dispendiosa o potenzialmente rischiosa. Per ovviare a ciò, venne introdotto il concetto di movimento, che significa "svuotare" un oggetto che sta per essere distrutto del suo contenuto e "travasarlo" in un altro oggetto: variabili locali al termine del blocco in cui sono definite, risultati di espressioni temporanee o tutto ciò che non ha un nome e può comparire solo a destra di "=" nelle assegnazioni (RVALUE). L'oggetto originale verrà modificato, ma poichè sta per essere distrutto, questo non ha importanza a patto che ciò che rimane dell'originale possa essere distrutto senza causare danni.

```
// esempio semplice di costruttore di movimento
class CBuffer{ int size; char* ptr }
CBuffer(CBuffer&& source){
        this->size = source.size;
```

```
        this->ptr = source.ptr;
        source.ptr = NULL;
}
```

A differenza di quello di copia, il costruttore di movimento non è sempre generato automaticamente dal compilatore, ma è compito del compilatore definirlo, seppur sia il compilatore poi a effettuare il movimento quando è necessario (facendo *Obj_MoveConstructor(dest, source)* e *Obj_Destructor(source)*) e riducendo, quando possibile, il costo del passaggio per valore.

Il distruttore è una callback con il compito di rilasciare le risorse contenute in un oggetto, ed il suo nome è come quello della classe me preceduto da una ~, da non dichiarare se l'oggetto non possiede risorse esterne.

```
// esempio semplice di distruttore
class CBuffer{ int size; char* ptr; }
CBuffer(int size): size(size){ ptr = new char[size]; }
~CBuffer(){ delete[] ptr; }
```

Il complatore invoca costruttore e distruttore al procedere del ciclo di vita di un oggetto.
L'operatore *new* acquisisce dall'allocatore della libreria di esecuzione un blocco di memoria di dimensioni opportune e lo inizializza invocando l'opportuno costruttore. L'operatore *delete* esegue i compiti duali, in ordine inverso.
Sebbene il linguaggio consenta l'accesso a basso livello ai puntatori nativi e permetta di richiedere manualmente le operazioni di allocazione e rilascio, è bene non farlo, in quanto il C++ offre una serie di supporti ad alto livello (smart pointer) volti a garantire la correttezza di tale operazione.

05 - IL LINGUAGGIO C++ (2)