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.

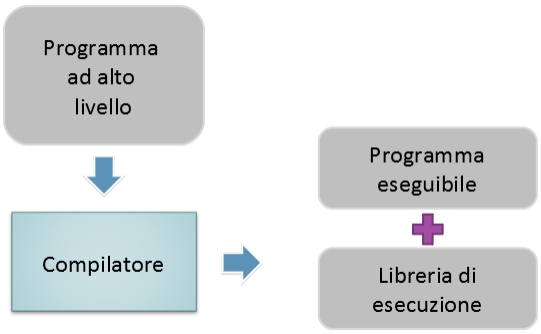
Siccome la rappresentazione dei caratteri a 8 bit è insufficiente per la maggior parte degli alfabeti, occorre anche avere un meccanismo di codifica e gestione del testo (possibili rappresentazioni: UTF-32, UTF-16, UTF-8) che diventa particolarmente complicato nel caso di rappresentazioni a lunghezza 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.

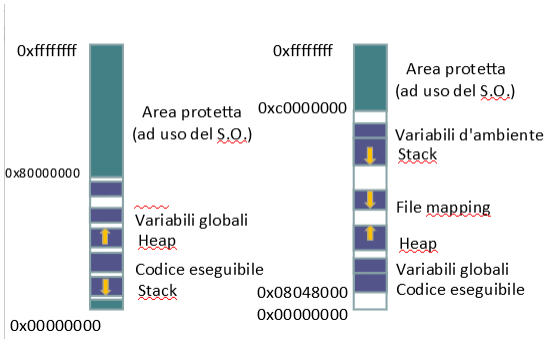
L'esecuzione di un programma comporta quindi la creazione di un nuovo processo, articolata in diverse fasi:

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.

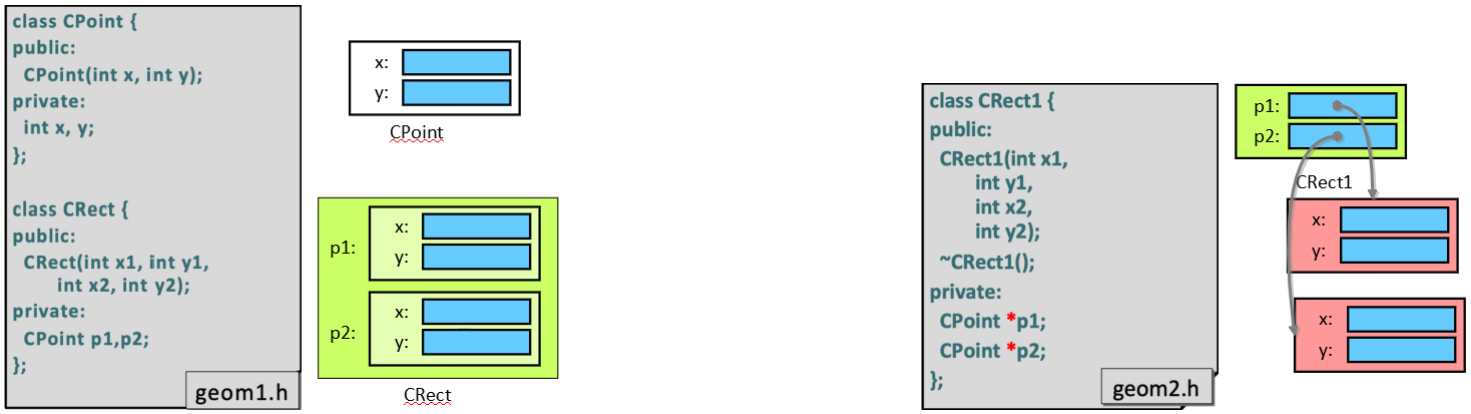
Il distruttore è una callback con il compito di rilasciare le risorse contenute in un oggetto, ed il suo nome è come quello della classe me preceduto da una ~, da non dichiarare se l'oggetto non possiede risorse esterne.

```
// 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)

Un oggetto può contenere altri oggetti come parte integrante della propria struttura dati ed è compito del costruttore inizializzare opportunamente tali oggetti.



Il linguaggio definisce più strategie per gestire il passaggio di valori da un contesto all'altro:

- Passaggio per valore: la funzione agisce su una copia della variabile passata, che può essere il risultato di un'espressione (parameter). Usato solitamente per valori la cui copia richiede un costo basso.
- Passaggio per indirizzo: viene passata una copia dell'indirizzo del dato: la funzione chiamata deve esplicitamente dereferenziarlo. L'indirizzo può essere NULL e il chiamato può modificare l'originale. La funzione non può fare assunzioni sull'indirizzo nè sulla durata del ciclo di vita del dato (type* parameter). Si usa quando si restituisce un risultato preferibilmente tramite il valore di ritorno.
- Passaggio per riferimento: viene passato un riferimento all'argomento originale (che DEVE essere una variabile). Sintatticamente sembra un passaggio per valore ma semanticamente corrisponde ad un passaggio per indirizzo (che non è mai NULL). Può essere usato per modificare l'originale e ritornarne la versione modificata, a meno che il parametro formale non sia preceduto da **const.** Più efficiente del passaggio per valore (type& parameter). Si usa per argomenti che devono poter essere modificati dalla funzione.
- Passaggio per riferimento RVALUE: ottimizzazione che permette al compilatore di MUOVERE i dati contenuti all'interno del valore per costruirne, a minore prezzo, una copia (type&& parameter). Quando occorre trasferire la proprietà di un dato ad una funzione invocata, questa riceve il dato sotto forma di riferimento RVALUE.

Talora occorre copiare il contenuto di un oggetto in un altro oggetto. Se l'oggetto di destinazione esisteva prima della copia, si parla di assegnazione, altrimenti si parla di assegnazione.
Di base, quando ad una variabile di tipo UDT viene assegnato un nuovo valore, il compilatore ricopia il contenuto di tutte le parti dell'oggetto sorgente nell'oggetto destinazione (occhio ai puntatori!) e può essere ridefinito (distruggi prima di ricostruire!)

```
CBuffer& operator=(const CBuffer& source){
    if(this != &source){
        delete[] this->ptr;
        this->ptr = null;
        this->size = source.size;
        this->ptr = new char[size];
        memcpy(this->ptr, source, size);
    }
    return *this;
}
```

L'operazione di assegnazione deve rilasciare tutte le risorse possedute per non creare leakage.

Se una classe dispone di una qualunque di queste funzioni membro, occorre implementare le altre due:

- Costruttore di copia
- Operatore di assegnazione
- Distruttore

Se ne manca uno, lo implementa il compilatore, quasi sicuramente sbagliato. In alcuni casi, la copia non è possibile, ma il movimento si (come per uno std::unique_ptr). Se i tipi di valore contenuti rappresentano una risorsa esterna, il vantaggio può essere anche maggiore (come descrittori di file, socket, connessione a base dati..).
Analogamente a quanto avviene per l'operatore di assegnazione semplice, per l'assegnazione per movimento occorre dapprima liberare le risorse esistenti e poi trasferire il contenuto dell'oggetto sorgente ricordandosi di svuotarlo.

```
CBuffer& operator=(CBuffer&& source){
    if(this != &source){
        delete[] this->ptr;
        this->size = source.size;
        this->ptr = source.ptr;
        this->ptr = null;
    }
    return *this;
}
```

Infine, *std::move(...)* è una funzione di supporto per trasformare un oggetto generico in un riferimento RVALUE, così da poterlo utilizzare nella costruzione o assegnazione per movimento: forza l'oggetto passato come parametro ad essere considerato come un riferimento RVALUE rendendolo disponibile per usi ulteriori (viene eseguito *static_cast<T&&>(t)*) e aiuta a rendere esplicita la conversione quando si vuole forzare l'uso del movimento rispetto alla copia.

06 - EREDITARIETA' E POLIMORFISMO

A volte, classi diverse presentano comportamenti simili, in quanto modellano concetti semanticamente simili. E' un buona idea non duplicare codice condiviso (sia per manutenibilità che per leggebilità). Una classe può quindi essere definita come specializzazione di una classe esistente: la classe così definita viene detta sotto-classe, mentre quella da cui deriva viene detta super-classe.

La sotto-classe specializza il comportamento della super-classe aggiungendo ulteriori variabili istanza e funzioni membro: inoltre, in C++ una classe può ereditare da una o più classi specializzando così il concetto che esse rappresentano. L'ereditarietà può essere pubblica, privata o protetta e i metodi della super-classe possono essere invocati usando l'operatore "::" preceduto dal nome della classe base.

Se la classe D estende in modo pubblico la classe B, è lecito assegnare ad una variabile "v" di tipo B* (o B&) un puntatore ad un oggetto di tipo D (D <<is_a>> B). Se la classe D ridefinisce il metodo *m()*, cosa succede quando si invoca *v->m()* (o *v.m()* se il tipo è B&)? Dipende da come è stato dichiarato il metodo nella classe base. Per default, il C++ utilizza l'implementazione definita nella classe a cui ritiene appartenga l'oggetto: nell'esempio, v ha come tipo B& per cui il compilatore invocherà la funzione contenuta nella classe B anche se, di fatto, l'oggetto a cui fa riferimento è la classe D.

Per modificare questo comportamento, occorre anteporre alla definizione del metodo la parola chiave **virtual** abilitando così il funzionamento polimorfico. Solo le funzioni denominate "virtual" sono polimorfiche mentre le funzioni virtuali astratte sono dichiarate "=0;". Una classe astratta contiene almeno una funzione virtuale astratta, mentre una classe puramente astratta contiene solo funzioni virtuali astratte (equivalente ad una interfaccia Java).
Anche i distruttori non sono polimorfici per default, ed è quindi opportuno che tutte le classi che posseggono ed espongono risorse con funzioni virtuali abbiano il distruttore dichiarato virtuale.

Per i metodi virtuali, si introduce un livello di indirezione attraverso un campo nascosto detto V-Table per cui ogni volta che viene creato un oggetto, nella memoria allocata si aggiunge un puntatore ad una tabella che contiene tante righe quanti sono i metodi virtuali dell'oggetto creato al cui interno vi è il puntatore all'effettivo metodo virtuale. Se esistono più istanze di una data classe, queste condividono la V-Table.

Una classe può ereditare da più di una classe: la sua interfaccia risulterà l'unione dei metodi contenuti nelle rispettive interfacce delle classi base uniti ai metodi propri della classe derivata.
C++ fornisce una serie di operatori per il type cast più efficienti e sicuri:

- static_cast<T>(p) : converte il valore di "p" rendendolo di tipo "T" se esiste un meccanismo di conversione disponibile. Esso viene scelto in base al tipo di "p" come noto al compilatore: nel caso di puntatori, bisogna fare attenzione. Se la conversione è illecita, il risultato non è predicibile
- dynamic_cast<T>(p) : effettua il controllo di compatibilità runtime assicurando cast sicuri tra tipi di classi. Applicato a un puntatore, ritorna 0 se il cast non è valido. Applicato a un riferimento genera un'eccezione in caso di incompatibilità
- reinterpret_cast<T>(p) : interpreta la sequenza di bit di un valore di tipo come valore di un altro tipo, autorizzando il compilatore a violare il sistema dei tipi. Utilizzato di solito per dati ritornati da systemcall o da hardware
- const_cast<T>(p) : elimina la caratteristica di costante dal suo argomento. Occhio alle variabili globali in sola lettura

07 - GESTIONE DELLE ECCEZIONI

Non tutte le operazioni hanno sempre successo, ma possono verificarsi errori e fallimenti di varia natura (sia dall'utente che dal sistema). Occorre quindi verificare se essi siano o meno presenti, gestendoli nel punto preciso in cui sono stati identificati, e questo è particolarmente semplice nel caso di errori attesi. Più complicato il discorso per quelli non attesi, più difficili da gestire in quanto possono verificarsi pressochè ovunque, rendendo il codice illeggibile nel tentativo di gestirli.

Le eccezioni sono un meccanismo che permette di trasferire il flusso di esecuzione ad un punto precedente, dove si ha la possibilità di gestirlo: si notifica al sistema la presenza di un'eccezione creando un dato qualsiasi e passandolo come valore alla parola chiave *throw*.
Il tipo del dato passato viene utilizzato per scegliere quale contromisura adottare. Se un'eccezione si verifica in un blocco *try* o in un metodo chiamato all'interno di esso, si contrae lo stack fino al blocco, eliminando tutte le variabili locali.

```
try {
    // le istruzioni eseguite qui possono lanciare un'eccezione
}
catch(out_of_range& oor){
    // contromisura per errore specifico
}
catch(exception& e){
    // contromisura generica
}
```

Le clausole *catch* effettuano un match sul tipo dell'eccezione e i blocchi sono esaminati nell'ordine in cui sono scritti (quindi, ordino da eccezione più specifica a quella più generica).
Se non c'è alcuna corrispondenza, l'eccezione rimane attiva e il programma ritorna alla funzione chiamante, ed eventualmente al chiamante del chiamante fino a trovare un blocco catch adatto.

La classe *std::exception* incapsula una stringa che viene inizializzata nel costruttore e a cui si può accedere con la funzione *what()*: serve per documentare, non per risolvere!
E' nel blocco catch che bisogna rimediare all'eccezione che si è verificata. Bisogna, innanzitutto, terminare ordinatamente il programma per poi ritentare l'esecuzione. Per terminare si può salvare lo stato e poi fare *exit()* ma è un'azione estrema. Si può ritentare l'esecuzione se il fallimento è dovuto a cause temporanee (es: è andata via la rete).
Si può registrare un messaggio usando un meccanismo opportunamente predisposto come il flusso *std::cerr* oppure *logger::log(e.what())*.

MAI scrivere un blocco catch che non esegue nessuna strategia di riallineamento.
MAI stampare un errore e poi continuare.

Il fatto che lo stack venga contratto in fase di lancio di un'eccezione è alla base di un pattern di programmazione tipico del C++ chiamato RAII (Resource Acquisition Is Initialization), il quale aiuta a garantire che le risorse acquisite alla costruzione siano liberate. Se nessuna eccezione viene lanciata, non ci sono sostanziali penalità in termini di costi: se invece viene lanciata un'eccezione, il costo è abbastanza elevato (un ordine di grandezza maggiore dell'invocazione di una funzione) ergo per gestire logiche locali meglio usare un *if*.

08 - FUNZIONI E OPERATORI

Tutti gli operatori in C++ possono essere ridefiniti, ma non è possibile definirne di nuovi. Se si ridefinisce un operatore, almeno uno degli operandi deve essere un UDT.

Gli operatori overloaded possono appartenere alla classe del loro operando di sinistra oppure essere funzioni semplici: si utilizzano operatori membro quando per la loro implementazione è necessario accedere alla parte privata/protetta di un oggetto oppure quando occorre modificare l'operando di sinistra.
Se, all'interno di una classe, è presente un metodo privo di argomenti, senza tipo di ritorno e il cui nome coincide con quello di un tipo (sia esso standard o UDT) preceduto dalla parola chiave "operator", questo viene assunto come operatore di conversione, il cui scopo è quello di trasformare l'oggetto istanza della classe nel tipo indicato.

```
class Fract{
    int num, den;
    operator float() const{
        return num * 1.f / den;
    }
}
```

Alcuni operatori non possono essere sovrascritti (es: "::", ".", "? : " ...).

In C++ è lecito inoltre salvare in una variabile il puntatore ad una funzione, dichiarando il puntatore come *<tipo_ritornato>(* var)(<argomenti>)* e si assegna un valore attraverso l'operatore =. Sia il tipo di ritorno che tutti i parametri formali della funzione devono corrispondere a quanto dichiarato nella definizione di variabile.

```
int f(int i, double d){
    // corpo della funzione
}
int(*var)(int, double);
var = f;
var = &f; // identico al precedente
```

In C++ esiste un ulteriore tipo invocabile: l'oggetto funzionale, che è un'istanza di qualsiasi classe che abbia ridefinito la funzione membro *operator()*. E' possibile includere più definizioni di questo operatore, con tipi differenti nei parametri. Un oggetto funzionale può contenere variabili membro ed il comportamento non è più matematico (a parità di input non ho lo stesso output)

```
class FC{
    int operator()(int v){
        return v * 2;
    }
}

FC fc;
int i = fc(5); // i vale 10
i = fc(2); // i vale 4
```

Spesso, l'utilizzo degli algoritmi presenti nella libreria standard richiede l'introduzione di funzioni usate una volta sola, e ciò si può alleggerire facendo uso di funzioni lambda: il tipo di ritorno deve essere specificato quando non si ha la sola istruzione di return

```
[](int num, int den) -> double {
    if(den == 0) return sd::NaN;
    return (double) num / den;
```

```
}
```

Le parentesi "[ ]" introducono la notazione lambda, e al loro interno è possibile elencare variabili locali: usando [X, Y] vengono catturate per valore (viene effettuata una copia), mentre con [&X, &Y] si ha una cattura per riferimento (cambio l'originale occhio). Con [&] catturo tutto per riferimento e tali catture sono interscambiabili (es: posso fare benissimo [X, &Y]).
Una funzione lambda che catturi dei valori viene detta "chiusura": essa racchiude al proprio interno (una copia de)i valori catturati rendendoli disponibili durante una successiva invocazione. Per default, la chiusura è immutabilem ma posso aggiungere "mutable" per poter cambiare i valori catturati.
Le funzioni lambda supportano diversi stili di programmazione migliorando la leggibilità di un programma permettendo di eliminare classi/funzioni piccole.

La classe *std::function<R(Args ...)>* permette di modellare una funzioine che riceve argomenti di tipo *Args...* e restituisce un valore di tipo R. Il meccanismo utilizzato internamente si basa sull'uso del polimorfismo (passo dalla V-Table!) .

09 - PROGRAMMAZIONE GENERICA E SMART POINTER

Un sistema di tipi stringente facilita la creazione di codice robusto identificando, in fase di compilazione, quei frammenti di codice che violano il sistema dei tipi. In certe situazioni, per non violare quest'ultimo, occorre replicare una grande quantità di codice generando problemi in fase di manutenzione e inoltre occorre garantire che eventuali modifiche ad una versione del codice vengano propagate a tutte le altre.
Il linguaggio C++ offre un supporto alla generalizzazione dei comportamenti di un blocco di codice attraverso il concetto di "template". Essi sono frammenti (funzioni, classi) parametrici, che vengono espansi in fase di compilazione, in base al loro uso effettivo. Può essere visto come un generatore di tipi, in quanto permette di implementare una varietà di funzionalità diverse adattandole ai singoli tipi di dato.

Grazie ai Template si abilita il paradigma della programmzione generica, che enfatizza la scrittura di algoritmi non legati ad un tipo specifico. Nell'istanziare un template, il compilatore introduce una forma di polimorfismo statico (più efficiente di quello dinamico con i metodi virtuali).

```
template <typename T>
const T& max(const T& t1, const T& t2){
    return ( t1 < t2 ? t2 : t1 );
}
int i = max(10, 20); // T è un intero
std::string s = max(s1, s2); // T è una stringa
max<double>(2, 3.14); // forza la scelta di T a double
```

Il tipo generico T deve supportare l'operatore "<" e il costruttore di copia.

Lo stesso principio può essere adottato nella definizione delle classi.
Il processo di compilazione viene diviso in due fasi:

1. Definizione del template: viene creato un albero sintattico astratto (AST) corrispondente al codice generico contenuto nella funzione/classe parametrica
2. Istanziazione del template: quando si trova un riferimento al template, questo viene istanziato, e l'AST viene specializzato con i dati concreti e si procede a generare il codice corrispondente

Se si crea una classe generica, occorre che TUTTA la sua definizione appaia nella stessa unità di traduzione in cui viene istanziato: per questo motivo, di implementano i template esclusivamente all'interno di file header. Ma cosa succede se alcuni tipi non posso usarli nel mio template? Posso modificare la classe fornendo la semantica richiesta (ridefinisco qualche operatore) oppure posso specializzare il template, fornendo una definizione specifica per la classe non altrimenti utilizzabile. I template aumentano quindi notevolmente le possibilità espressive ma occhio alle violazioni sul tipo utilizzato (errori di ccompilazione difficili da interpretare).

La ridefinizione degli operatori permette di costruire oggetti che "sembrano" puntatori ma che hanno ulteriori caratteristiche (come garanzia di inizializzazione e rilascio e conteggio dei riferimenti). il generico *smart_ptr<MyClass>* si occupa di chiamare *delete* sul puntatore dell'oggetto quando si esce dal blocco evitando memory leakage anche in caso di eccezioni (molto importante!).

A chi appartiene la memoria puntata da una smart pointer? Se l'oggetto referenziato viene usato da un unico utilizzatore, può essere eliminato al termine delle operazioni ma se viene utilizzato da più persone è necessario definire chi possa eliminare la risorsa: o l'ultimo a conoscere l'oggetto o un garbage collector.
Sono disponibili diversi template per la gestione automatica della memoria:

- std::unique_ptr<BaseType> : incapsula il puntatore a un oggetto modellandone il possesso esclusivo. Non può essere copiato nè assegnato ma trasferito usando *std::move()*. Quando viene distrutto, il blocco è rilasciato. Esistono due versioni: per la gestione di un singolo oggetto di classe T e per la gestione di un array. Utilizzato per garantire la distruzione di un oggetto, per acquisire o passare la proprietà di un oggetto con ciclo di vita dinamico tra funzioni e gestire in modo sicuro oggetti polimorfici

- std::weak_ptr<BaseType> : mantiene al proprio interno un riferimento "debole" a un blocco custodito in uno *shared_ptr* modellando il possesso temporaneo di un blocco in memoria. Si acquisisce temporaneamente l'accesso tramite il metodo *lock()*

- std::shared_ptr<BaseType> : Mantiene la proprietà condivisa a un blocco di memoria referenziato da un puntatore nativo. Quando tutti sono stati ditrutti, il bloco viene rilasciato (con la delete). Un oggetto di questo tipo può anche non contenere alcun puntatore valido. Il distruttore decrementa il conteggio dei riferimenti e libera la memoria solo quando esso raggiunge 0.
  Si ha una funzione di supporto chiamata *make_shared<BaseType>(params..)* che crea sullo heap un'istanza della classe BaseType usando i parametri passati per il ostruttore e ne incapsula il puntatore in uno *shared_ptr* .

  Ognuno mantiene due puntatori: uno all'oggetto e l'altro a un blocco di controllo, allocato dinamicamente e contenente distruttore, allocatore, contatore di *shared_ptr* (per la proprietà) e contatore di *weak_ptr* (riferimento).

  Data una classe D che eredita in modo pubblico da B, un puntatore ad una classe derivata D è anche puntatore alla classe B, è possibile dichiarare uno smart pointer di classe B e inizializzarlo con un oggetto di classe D. Il conteggio dei riferimenti garantisce il rilascio della memoria in modo deterministico (non ho riferimenti? libero) ma in alcuni casi non funziona, ad esempio se si forma un ciclo di dipendenze, che si evita appunto ricorrendo a *weak_ptr<BaseType>*

10 - LIBRERIE C++

E' possibile utilizzare le funzioni della libreria stdio.h, ma il C++ fornisce un'alternativa come *istream* per l'input e *ostream* per l'output e *iostream* per entrambe. Per i file? *ifstream* e *ofstream.*
Per le operazioni di output su video si usa *std::cout* (possibile usare width per ampiezza del campo e precision per la precisione). I manipolatori sono particolari oggetti che possono essere inviati in scrittura o in lettura ad un flusso (definiti in "iomanip") per manipolare il formato.
Si hanno anche *std::cin, std::cerr* e *std::clog* per le operazioni primarie di I/O. Tali operazioni non sollevano eccezioni, per cui il programmatore deve testare esplicitamente il risultato di ogni operazione effettuata.

La Standard Template Library è basata sulla programmazione generica con l'uso di template. Fornisce tre insiemi di componenti riutilizzabili perchè parametrici:

- Container: sequenziali (organizzano linearmente una collezione di oggetti: array, vector, deque, list, forward_list), associativi (contenitori non lineari: set, multiset, map, multimap, unordered_set, unordered_multiset, unordered_map, unordered_multimap) e container adapter (contenitori sequenziali specializzati come stack, queue, priority_queue) e non supportano gli iteratori.

  Nonostante ognuno sia specializzato per uno specifico uso, i containter hanno molto in comune: hanno un allocatore che lavora in background, la creazione e cancellazione sono uguali, dimensione, accesso, swap, compare.

- Algoritmi generici: la STL fornisce un gran numero di algoritmi per lavorare sui container e i loro elementi. Gli algoritmi sono implementati come funzioni template e sono indipendenti dal tipo degli elementi nel container. In <numeric> si trovano algoritmi numerici mentre in <algorithm> sono raccolti algoritmi per ricerca, partizionamento, ordinamento, merge, operazioni insiemistiche, operazioni su heap....

  Gli algoritmi non modificanti permettono di applicare funzioni agli elementi di un container senza modificarne la struttura mentre quelli modificanti la possono cambiare

- Iteratori: sono la "colla" tra i container e gli algoritmi della STL, supportano le operazioni e sono una generalizzazione dei puntatori (sebbene abbiano gli stessi problemi). L'iteratore più semplice è un puntatore ad un blocco di oggetti: si delimita il blocco con due puntatori (inizio e fine). Esistono 3 tipi di iteratori, che dipendono dal container utilizzato: forward iterator (per container associativi non ordinati), bidirectional iterator (per container associativi ordinati) e random access iterator

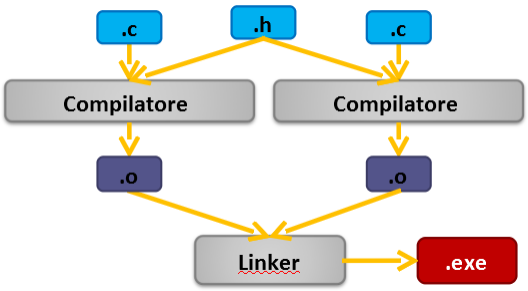
La maggior parte degli algoritmi della STL accetta come parametro una delle execution policy definite in <execution>. Le execution policy permettono l'esecuzione sequenziale, parallela o parallela con vettorizzazione, ed è responsabilità del programmatore evitare data race e deadlock.

11 - USO DI LIBRERIE

Il programma è suddiviso in un insieme di file sorgenti che sono compilati separatamente. La compilazione produce moduli oggetto, che vengono uniti grazie al linker in un eseguibile finale.

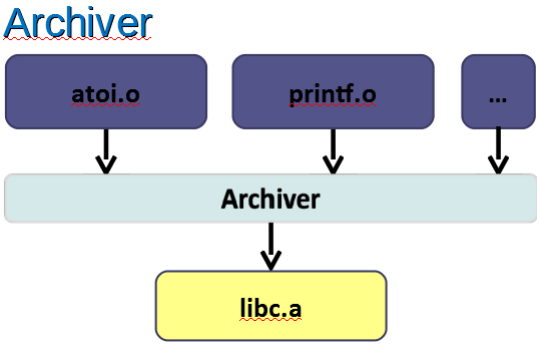
Le librerie sono un insieme di moduli oggetto archiviati in un unico file, e permettono di radunare funzioni e strutture dati pertinenti ad uno stesso dominio, riducendo i tempi di compilazione e favorendo l'incapsulamento. Una libreria contiene:

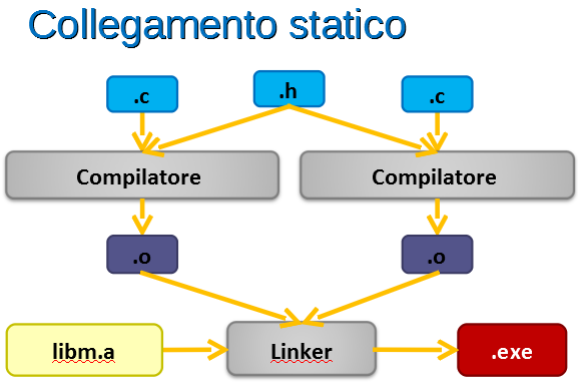
- Funzioni e variabili esportate (accessibili ai programmi che la utilizzano)
- Funzioni e variabili private (accessibili solo alle altre funzioni della libreria)
- Costanti e altre risorse



L'uso di una libreria richiede due fasi: identificazione dei moduli necessari e loro caricamento in memoria, e aggiornamento degli indirizzi per puntare correttamente ai moduli caricati. Queste due operazioni possono essere fatte in fasi differenti.

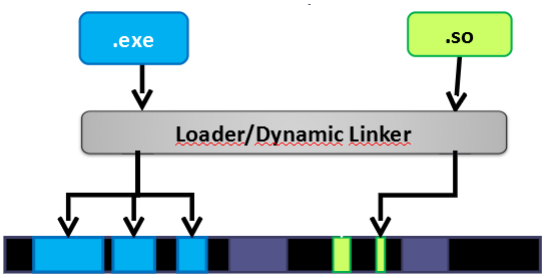
Le librerie a collegamento statico contengono funzionalità collegate staticamente al codice binario cliente in fase di compilazione. Un libreria statica è un file archivio che contiene un insieme di file object, mentre l'impacchettamento viene fatto dall'archiver. Il linker identifica in quali moduli della libreria si trovano le funzioni richiamate nel programma, carica nell'eseguibile solo quelli necessari e, terminata la fase di caricamento, i moduli della libreria statica risultano annegati nel codice binario originale.
Vantaggi: il codice delle librerie è certamente presente nell'eseguibile e non ci sono dubbi sulla versione della libreria adottata.





Svantaggi: gli stessi contenuti sono presenti in processi differenti e l'uso delle librerie statiche comporta una riduzione della modularità del codice. In Linux GCC offre come archiver il tool ar, mentre Windows hanno estensione .lib .

Con le librerie a collegamento dinamico il file eseguibile non contiene i moduli della libreria, ma vengono caricati successivamente nello spazio di indirizzamento del processo. All'atto della creazione del processo, il loader mappa nello spazio di indirizzamento



tutte le librerie condivise. In Linux il dynamic linker è il programma Id.so, che mappa le librerie nello spazio di memoria del processo e aggiorna la tabella di simboli per ogni libreria caricata. In Windows il dynamic linker fa parte del kernel stesso.
E' possibile controllare esplicitamente il caricamento delle librerie condivise: per esempio Linux espone le Dynamic Loading API *dlopen(), dlsym(), dlerror(), dlclose()*.

In windows abbiamo moduli binari in formato Portable Executable con estensione ".dll": possono contenere funzioni, variabili globali, costanti, risorse. La funzione LoadLibrary(...) carica la libreria indicata e la mappa nello spazio di indirizzamento del processo restituendo la handle del modulo caricato.
La DLL può specificare un punto di ingresso opzionale chiamato quando un processo o un thread mappano e/o rilasciano la DLL nel proprio spazio di indirizzamento. Normalmente, le variabili di una stessa DLL non sono condivise tra processi diversi, ma ognuno ne contiene una copia indipendente e codice e risorse sono condivisi in sola lettura. Le variabili globali esportate da una DLL sono memorizzate all'interno del processo che la utilizza.
Per poter esportare ed importare funzioni e struttura dati: usare direttive chiave (come *dllexport* o *dllimport*) oppure creare un file module definition (.def).
Le direttive chiave servono in fase di compilazione per fare in modo che il compilatore possa modificare il nome della funzione esportata per tenere conto del numero e tipo dei suoi parametri (non voglio? usare "extern").
Il file module definition è più o meno così:

```
// SampleDLL.def
LIBRARY "sampleDLL"

EXPORTS
    HelloWorld
```

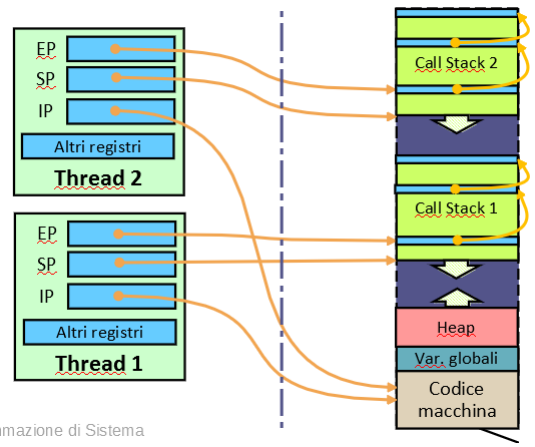
Visual Studio genera, oltre al .dll anche un file .lib (contiene uno stub delle API offerte e deve essere linkato dai progetti che intendono utilizzare in modo statico la DLL). Per usare DLL dinamicamente, il programma principale non fa riferimenti diretti ai simboli definiti dalla DLL, essa viene caricata esplicitamente tramite LoadLibrary(...) e si accede alle funzioni tramite GetProcAddress(...): si rilascia con FreeLibrary() o FreeLibraryAndExitThread()

12 - PROGRAMMAZIONE CONCORRENTE

Un programma concorrente dispone di due o più flussi di esecuzione contemporanei nello stesso spazio di indirizzamento per perseguire un obiettivo comune. All'atto della creazione, un processo dispone di un unico flusso di esecuzione, che può richiedere al SO la creazione di altri thread. Lo scheduler ripartisce, nel tempo, l'utilizzo dei core disponibili tra i diversi thread in modo non deterministico.
Vantaggi: sovrapposizione tra computazione e operazioni di I/O, riduzione del sovraccarico dovuto alla comunicazione tra processi e utilizzo delle CPU multicore. Un unico svantaggio: aumento significativo della complessità del programma (nuovi errori e non determinismo dell'esecuzione).

La memoria non può più essere pensata come un "deposito statico" in quanto può essere modificata anche dall'attività di altri thread, che devono coordinarne l'accesso. Quindi, quando un thread legge il contenuto di una locazione di memoria può trovare il valore iniziale contenuto nell'inizializzazione, quello modificato dallo stesso thread oppure un valore depositato da un altro thread: quest'ultimo caso è problematico a causa di cache hardware.
Inoltre, nascono altri problemi: atomicità (quali istruzioni devono avere effetti indivisibili?), visibilità (sotto quali condizioni un thread può vedere modifiche di altri thread?) e ordinamento (sotto quali condizioni gli effetti di più operazioni effettuate da un thread possono apparire ad altri thread in ordine differente?).
L'uso superficiale dei costrutti di sincronizzazione possono causare malfunzionamenti casuali difficili da riprodurre ed eliminare e possono addirittura essere diversi a seconda della piattaforma di esecuzione.

Ogni thread dispone di: un proprio stack delle chiamate, un proprio puntatore all'ultimo contesto per la gestione delle eccezioni e lo stato del proprio "processore virtuale". Condividono invece: le variabili globali, l'area in cui è memorizzato il codice, l'area delle costanti e lo heap.



Se due o più attività cooperano per raggiungere un obiettivo comune, occorre regolare lo svolgimento di un thread anche in base a quanto sta succedendo negli altri, tenendo conto che non è possibile fare assunzioni sulle velocità relative di avanzamento di ogni thread (se lancio 2 volte lo stesso programma, il risultato può non essere lo stesso).
Si ha interferenza quando più thread fanno accesso a un medesimo dato, modificandolo: la sua presenza da origine a malfunzionamenti casuali molto difficili da identificare. E' perciò necessario evitare che altri thread accedano a una risorsa condivisa mentre una modifica è in corso e ogni SO offre strumenti leggermente diversi, sia in termini di strutture che di API: Windows (CriticalSection, ConditionVariable, Mutex, Semaphore ...) Linux (pthread_mutex, pthread_cond ...).

Bisogna quindi controllare SE c'è una mutazione in corso e fermarsi nel caso la risposta sia positiva: è compito del programmatore riconoscere quando e dove usare la sincronizzazione (ergo maggiori errori).

In altre parole, se due thread fanno accesso alla stessa struttura dati, rispettivamente in lettura e scrittura, non c'è nessuna garanzia su quale delle due operazioni sia eseguita per prima. Se, quando osservato dall'esterno, il comportamento di un singolo thread appare indistinguibile a seguito di una modifica, sia il compilatore che la CPU possono invertire l'ordine di esecuzione delle singole istruzioni.

CreateThread è la funzione per la creazione di thread in windows che ritorna un riferimento opaco (handle): NULL se ho fallimento. *HANDLE GetCurrentThread()* restituisce una pseudo-handle del thread corrente utilizzabile solo al suo interno e per essere passata richiede una duplicazione esplicita (*DuplicateHandle*). Un thread termina quando:

- La funzione associata al thread ritorna
- Il thread invoca *ExitThread* al proprio interno (non chiamo i distruttori, occhio!)
- Un altro thread del processo invoca *TerminateThread*
- Vengono invocate le funzioni *ExitProcess(...)*(non chiamo i distruttori, occhio!) o *TerminateProcess(...)* specificando il processo a cui il thread appartiene

Per garantire la corretta distruzione di tutti gli oggetti nello stack del thread, occorre basarsi su una variabile condivisa che deve essere ispezionata periodicamente dal thread che si vuole terminare. Quando questa assume un dato valore, il thread fa in modo di ritornare dalla propria routine principale. Quando un thread termina, l'oggetto kernel corrispondente passa nello stato "segnalato".

Le handle dei thread fanno riferimento a oggetti posseduti dal SO (contengono un contatore di utilizzo e quando diventa zero, il kernel può distruggere l'oggetto) e quando un thread non ha più bisogno di usare una handle, deve segnalarlo tramite *CloseHandle()* che decrementa il contatore dell'oggetto e la rende invalida. Per i thread, il contatore parte da 2 (thread creatore e thread creato): attenzione perchè se non vengono rilasciati, gli oggetti kernel possono saturare la memoria di sistema.

13 - CONCORRENZA AD ALTO LIVELLO IN C++

Per la programmazione concorrente in C++ sono disponibili due approcci: uno di alto livello (basato su *std::async* e *std::fututre*) ed uno di basso livello (che richiede l'uso esplicito di thread e costrutti di sincronizzazione).
Spesso, un compito complesso può essere decomposto in una serie di compiti più semplici, che sono indipendenti se la computazione di uno non dipende da quella di un altro (no dati condivisi).

std::async() prende come parametro un oggetto chiamabile (puntatore a funzione o oggetto funzionale) e ritorna un oggetto di tipo *std::future*<*T*>: quando questa funzione viene eseguita, se possibile invoca in un thread separato l'oggetto chiamabile con i relativi parametri. Si accede al risultato dell'esecuzione invocando il metodo *get()* sull'oggetto future ritornato: se l'esecuzione è andata a buon fine restituisce il risultato, se si ha avuto un'eccezione la rilancia nel thread corrente, se è ancora in corso si blocca in attesa che finisca e se l'esecuzione non è ancora iniziata ne forza l'avvio nel thread corrente. L'oggetto chiamabile può essere preceduto da una politica di lancio come ASYNC (attiva un thread secondario) oppure DEFERRED (l'oggetto chiamabile sarà valutato solo se e quando qualcuno chiamerà get() o wait() sul future relativo). Attenzione! La creazione di un thread secondario ha un costo significativo, quindi se le operazioni da eseguire sono poche conviene eseguirle direttamente.

std::future<*T*> fornisce un meccanismo per accedere in modo sicuro e ordinato al risultato di un'operazione asincrona, e mantiene internamente uno stato condiviso con il blocco di codice responsabile della produzione del risultato: quando si invoca la get() lo stato condiviso viene rimosso, e si può chiamare quindi una volta sola.
Per forzare l'avvio del task e attenderne la terminazione, senza prelevare il risultato, si utilizza il metodo wait() (con le varianti wait_for e wait_until, che tuttavia non forzano l'avvio se il task è lanciato con la modalità deferred e ritornano: deferred se non è partita, ready se il risultato è pronto, timeout se il tempo è scaduto).
Quando un oggetto future viene distrutto, il distruttore ne attende la fine (possibili attese). Le attività lanciate in questo modo non sono cancellabili, se non avendo cura di condividere, con la funzione chiamata, una variabile che possa essere usata come criterio di terminazione.

std::shared_future<*T*> evita data race nell'accesso a un singolo oggetto da thread multipli. Si può muovere un future in un shared_future (non copiare) o in modo più conciso usando share(). shared_future invece, è copiabile oltre che movibile e offre gli stessi metodi di future, con la differenza che il metodo get() può essere chiamato più volte producendo sempre lo stesso risultato per realizzare catene di elaborazione asincrone in cui i risultati delle fasi iniziali sono messi a disposizione delle fasi

successive senza ridurre il grado di parallelismo.

Gli oggetti della classe *std::mutex* permettono l'accesso controllato a porzioni di codice a un solo thread alla volta: tutto il codice che fa accesso a una data informazione condivisa deve fare riferimento a uno stesso oggetto mutex e racchiudere le operazioni tra le chiamate ai metodi lock() e unlock(). Se un thread invoca il lock() di un mutex che è già posseduto da un altro thread, si blocca in attesa che quest'ultimo faccia unlock().
Non c'è corrispondenza diretta tra l'oggetto mutex e la struttura dati che esso protegge (la relazione è nella mente del programmatore) e vanno protette da mutex anche le operazioni di sola lettura. Un mutex non è ricorsivo, quindi se un thread cerca di acquisirlo due volte, senza rilasciarlo, il thread si blocca per sempre. Volendo si può usare *std::recursive_mutex.*

std::lock_guard<Lockable> semplifica il codice e garantisce che un mutex sia sempre rilasciato. Utilizza il paradigma RAII: il costruttore invoca il metodo lock() dell'oggetto passato come parametro e non offre nessun altro metodo.

In alcune situazioni, un programma non intende bloccarsi se non può acquisire un mutex, ma fare altro nell'attesa che si liberi. Si può fare ciò usando *try_lock()* che ritorna un booleano.

std::scoped_lock<Lockable> è equivalente al lock_guard ma è un template con un numero variabile di parametri, accettando più lockable su cui il lock viene acquisito contemporaneamente al momento della costruzione (e rilasciato nel momento della distruzione) e risolvendo il problema del deadlock.

std::unique_lock<Lockable> estende il comportamento di lock_guard dando la possibilità di rilasciare e riacquisire l'oggetto tramite lock() e unlock() che lanciano system_error nel caso in cui si cercasse di rilasciare qualcosa che non si possiede o viceversa.

std::shared_mutex è una primitiva di sincronizzazione che permette due forme di accesso:

- Condiviso: in cui molti thread contemporaneamente possono possedere il mutex ed entrare all'interno della regione critica (tante operazioni di lettura)
- Esclusivo: un solo thread può accedere alla sezione critica (se sono presenti delle scritture)

14 - THREAD

La classe *std::thread* modella un oggetto che rappresenta un singolo thread di esecuzione del sistema operativo. L'esecuzione inizia immediatamente dopo la costruzione dell'oggetto thread associato a partire dall'oggetto callable fornito come argomento del costruttore (che può essere un puntatore a funzione, un oggetto che implementa operator() o una funzione lambda)

```
#include <thread>
class MyThread{
    void operator()(){
        std::cout << "operator()" << std::endl;
    }
};

int main(){
    std::thread t1{ MyThread() };
    std::thread t2( [](){
        std::cout << "funzione␣lambda" << std::endl;
    });
    t1.join();
    t2.join();
}
```

Il costruttore del thread deve quindi ricevere un oggetto chiamabile e, opzionalmente, una serie di parametri, che saranno inoltrati attraverso la funzione *std::forward* che mette a disposizione del destinatario un riferimento al dato originale o un riferimento RVALUE in funzione della natura del dato passato. Per passare un dato come riferimento e non come copia, occorre incapsularlo in un oggetto di tipo *std::reference_wrapper<T>* e tali paramentri possono essere utilizzati per ospitare i valori di ritorno della funzione: in questo caso occorre garantire che il ciclo di vita di tali parametri sopravviva alla terminazione del thread e attendere che esso termini prima di esaminarne il contenuto.

Spesso è comodo utilizzare una funzione lambda come parametro ma può creare problemi di sincronizzazione e di accesso alla memoria se nel frattempo i parametri escono dal proprio scope.

Quando viene creato un oggetto *std::thread* occorre alternativamente:

- Attendere la terminazione della computazione parallela, invocando il metodo join()
- Informare l'ambiente di esecuzione che non si è interessati all'esito invocando detach()
- Trasferire le informazioni contenute nell'oggetto thread in un altro oggetto tramite movimento

Se nessuna di queste azioni avviene, e si distrugge l'oggetto thread, l'intero programma termina. Se il thread di un programma termina, tutti i thread secondari ancora esistenti terminano improvvisamente (chiama join() o detach() !).
Se un thread calcola un risultato, come fa a metterlo a disposizione degli altri thread? La soluzione richiede di appoggiarsi a una variabile condivisa, dato che la funzione callable deve avere tipo di ritorno void. Questo però introduce un secondo problema: come fanno gli altri thread a sapere che il contenuto della variabile condivisa è stato aggiornato? La proposta (banale) di introdurre un'ulteriore variabile booleana che indica la validità del dato non è una soluzione.

15 - CONDITION VARIABLE

Spesso un thread deve aspettare uno o più risultati intermedi prodotti da altri thread e, per motivi di efficienza, l'attesa non deve consumare risorse e deve terminare non appena un dato è disponibile. La coppia di classi future/promise offrono una soluzione limitata del problema, valida quando occorre notificare la disponibilità di un solo dato.
Il polling ha due limiti: consuma capacità di calcolo e batteria in cicli inutili e introduce una latenza tra il momento in cui il dato è disponibile e il momento in cui il secondo thread si sblocca.

std::condition_variable modella una primitiva di sincronizzazione che permette l'attesa condizionata di uno o più thread: fino a che non si verifica una notifica, scade un timeout o si verifica una notifica spuria. Richiede l'uso di *unique_lock<Lockable>* per garantire l'assenza di corse critiche nel momento del risveglio.
Inoltre, è offerto il metodo *wait(unique_lock)* per bloccare l'esecuzione del thread senza consumare cicli di CPU. Un altro thread può informare uno o tutti i thread attualmente in attesa che la condizione si è verificata attraverso i metodi *notify_one()* e *notify_all()*.

Si ha una lista di thread in attesa della condizione: inizialmente è vuota, e quando un thread esegue il metodo di wait, viene sospeso e aggiunto alla lista, poppando quando si riceve una notifica. La funzione di attesa è basata su un sistema a "doppia porta":

1. Aggiungo il thread corrente alla lista di quelli da risvegliare
2. Rilascio il lock
3. Sospendo il thread
4. (attesa passiva)
5. Riacquisisco il lock

La presenza di un unico lock fa si che, se più thread ricevono la notifica, il risveglio sia progressivo. La relazione tra evento e notifica è solo nella testa del programmatore. E' possibile tuttavia che il thread sia risvegliato per altri motivi: problema delle cosiddette **notifiche spurie** per cui al risveglio conviene lo stesso controllare se la condizione di attesa è verificata.
Una versione overloaded di wait accetta come parametro un oggetto chiamabile, lanciato alla notifica, che ha il compito di valutare se l'evento è proprio quello atteso, restituendo true o false.

wait_for(...) e *wait_until(...)* limitano l'attesa nel tempo e offrono l'opportunità al thread che esegue la notifica di completare la propria distruzione prima che altri abbiano l'opportunità di leggere il dato condiviso.

Le normali operazioni di accesso in lettura e scrittura non offrono nessuna garanzia sulla visibilità delle operazioni che sono eseguite in parallelo su più thread; i processori supportano alcune istruzioni specializzate per permettere l'accesso atomico ad un singolo valore.
La classe *std::atomic<T>* offre la possibilità di accedere in modo atomico a T, garantendo che gli accessi concorrenti alla variabile sono osservabili nell'ordine in cui avvengono: questo garantisce il meccanismo minimo di sincronizzazione. Le operazioni di lettura e scrittura di questi oggetti contengono al proprio interno istruzioni di *memory fence* che garantiscono che il sottosistema di memoria non mascheri il valore corrente della variabile (inoltre le operazioni non possono essere riordinate).
Le operazioni di inizializzazione non sono atomiche, mentre quelle di accesso si.

Il namespace di *std::this_thread* offre un insieme di funzioni che permettono di interagire con il thread corrente, come chiederne l'identificativo con *get_id()*. Inoltre, si hanno diverse funzioni utili alla sospensione dell'esecuzione: *sleep_for(duration)*, *sleep_until(time_point)* o *yield()* che permette di rischedulare l'esecuzione del thread.

16 - SINCRONIZZAZIONE AD ALTO LIVELLO

In alcuni casi, occorre accedere alle funzionalità di basso livello gestendo esplicitamente la creazione dei thread, la sincronizzazione, l'accesso a zone di memoria condivise o l'uso delle risorse.
async() e *future<T>* permettono di creare facilmente composizioni di attività ad alto livello a patto che i diversi compiti in cui l'algoritmo può essere suddiviso siano poco interconnessi (la computazione di uno non dipende dall'altro).

Con gli oggetti di tipo *std::thread,* non c'è scelta sulla politica di attivazione, poichè creando un oggetto di questo tipo con un parametro chiamabile, la libreria cerca di creare un thread nativo del sistema operativo: se non ci sono le risorse necessarie, la creazione dell'oggetto fallisce lanciando system_error.

std::promise<T> rappresenta l'impegno, da parte del thread, a produrre prima o poi un oggetto di tipo T da mettere a disposizione di chi lo vorrà utilizzare oppure di notificare un'eventuale eccezione che abbia impedito il calcolo dell'oggetto. Dato un oggetto promise, si può conoscere quando la promessa si avvera richiedendo l'oggetto *std::future<T>* corrispondente (devo usare get_future()).

Se si crea un oggetto thread e si invoca il metodo detach(), l'oggetto thread si "stacca" dal flusso di elaborazione corrispondente e può continuare la propria esecuzione senza, però, offrire pù nessun meccanismo specifico per sapere quando termini. Se il thread distaccato fa accesso a variabili globali o statiche, queste potrebbero essere distrutte mentre la computazione è in corso perchè sta terminando il thread principale. Se quest'ultimo termina normalmente l'intero processo viene terminato, con tutti i thread distaccati eventualmente presenti, mentre se la terminazione avviene per altre cause, questo può portare a errori sulla memoria. *std::quick_exit(...)* permette di far terminare un programma senza invocare i distruttori delle variabili globali e statiche che può essere un rimedio peggiore del male.

Se si utilizza un oggetto di tipo promise per restituire un valore, si introduce un livello sincronizzazione: il thread interessato ai risultati, invocando wait() o get() sul future corrispondente resta bloccato fino a che il thread detached non ha assegnato un valore. Questo però non significa che il thread detached sia terminato, potrebbe avere ancora del codice da eseguire (ad esempio, tutti i distruttori degli oggetti finora utilizzati. Per questo, la classe promise offre due metodi per evitare potenziali corse critiche tra la pubblicazione di un risultato nell'oggetto promise e la continuazione degli altri thread in attesa dello stesso: *set_value_at_thread_exit(T val)* e *set_exception_at_thread_exit(std::exception_ptr p).*

std::packaged_task<T(Args...)> è un'astrazione di un'attività in grado di produrre, prima o poi, un oggetto di tipo T come conseguenza di una funzione o di un oggetto funzionale che viene incapsulato nell'oggetto stesso, insieme ai suoi argomenti. Quando un packaged_task viene eseguito, la funzione incapsulata viene invocata e il risultato prodotto (o l'eventuale eccezione generata) viene utilizzato per valorizzare l'oggetto std::future associato al task.

In molte situazioni, è conveniente creare un numero limitato di thread (legato al numero di core disponibili) cui demandare l'esecuzione di compiti puntali: tail compiti non sono noti a priori e possono essere eseguiti da uno qualsiasi dei thread appositamente creati. La classe packaged_task si presta particolarmente per la realizzazione di una coda in cui ospitare le attività richieste e da cui i diversi thread del pool possono attingere le attività da svolgere.

Ci sono inoltre varie situazioni in cui è utile rimandare la creazione ed inizializzazione di strutture complesse fino a quando non c'è la certezza del loro utilizzo (lazy evaluation). In un programma sequenziale, ci si riferisce tipicamene alla struttura in questione attraverso un puntatore inizializzato a NULL: quando occorre accedere alla struttura, si controlla se il puntatore abbia già un valore lecito e, nel caso in cui valga ancora NULL, si crea la struttura e se ne memorizza l'indirizzo all'interno del puntatore. In un programma concorrente questa tecnica non può essere adottata direttamente perchè il puntatore è una risorsa condivisa e il suo accesso deve essere protetto da un mutex. Ad ogni modo, per supportare tale comportamento, abbiamo la classe *std::once_flag* e la funzione *std::call_once(...).* Registra, in modo thread safe, se è già avvenuta o sia in corso una chiamata a call_once.
std::call_once(flag, f, ...) esegue la funzione f una sola volta: se dal flag non risultano chiamate, inizia l'invocazione di f altrimenti blocca l'esecuzione in attesa del suo risultato.

17 - QT

18 - COMUNICAZIONE TRA PROCESSI

L'uso dei thread permette di sfruttare le risorse computazionali presenti in un elaboratore: la presenza di uno spazio di indirizzamento condiviso facilita la coordinazione e la comunicazione. Ci sono situazioni in cui la presenza di un singolo spazio di

indirizzamento non è possibile o desiderabile.
E' possibile decomporre un sistema complesso in un insieme di processi collegati, creandoli a partire da un processo genitore e permettendo la cooperazione indipendentemente dalla loro genesi. Ad ogni processo è associato almeno un thread (primary thread) e un sistema multiprocesso è intrinsecamente concorrente.

I processi in Windows costituiscono entità separate, senza relazioni di dipendenza esplicita tra loro, e si crea con la funzione CreateProcess(...). In Linux invece si crea un processo figlio con l'operazione fork(), che crea un nuovo spazio di indirizzamento "identico" a quello del processo genitore; dopo la for, tutte le pagine sono marcate con il flag "CopyOnWrite".
La funzione *exec*()* sostituisce l'attuale immagine di memoria dello spazio di indirizzamento, ri-inizializzandola a quella descritta dall'eseguibile indicato come parametro.

Nel caso di programmi concorrenti, l'esecuzione di fork() crea un problema, in quanto il processo figlio conterrà un solo thread e gli oggetti di sincronizzazione presenti nel padre possono trovarsi in stati incongruenti.

InterProcess Communication: il sistema operativo impedisce il trasferimento diretto di dati tra processi: per cui ogni processo dispone di uno spazio di indirizzamento separato e non è possibile sapere cosa sta capitando in un altro. Ogni SO offre alcuni meccanismi per superare tale barriera in modo controllato, permettendo lo scambio di dati e la sincronizzazione delle attività.
Indipendentemente dal tipo di meccanismo adottto, occorre adattare le informazioni scambiate così da renderle comprensibili al destinatario.

Internamente, un processo può usare una varietà di rappresentazioni, ma non è adatta ad essere esportata (i puntatori non hanno senso e alcuni informazioni non sono proprio esportabili, come gli handle).
La rappresentazione esterna è un formato intermedio che permette la rappresentazione di struttre dati arbitrarie sostituendo i puntatori con riferimenti indipendenti dalla memoria.

Le rappresentazioni esterne possono essere trattate come blocchi compatti di byte e possono essere duplicati e trasferiti senza comprometterne il significato. I dati vengono scambiati nel formato esterno: la sorgente esporta le proprie informazioni (msrshalling) e il destinatario ricostruisce una rappresentazione su cui può operare direttamente (unmarshalling).
Le operazioni di marshalling e unmarshalling possono essere codificate esplicitamente o essere eseguite da codice generato automaticamente dall'ambiente di sviluppo.
Ci sono diversi tipi di IPC:

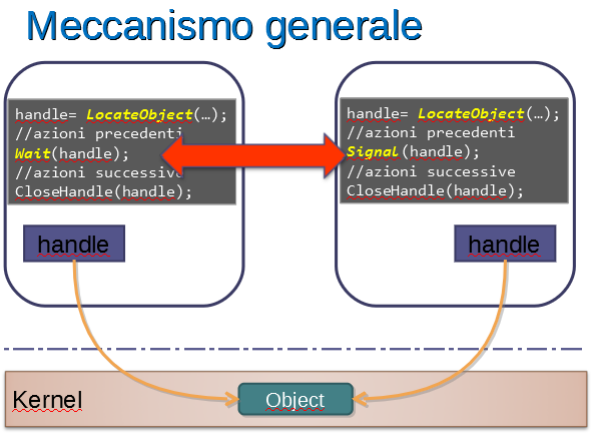
- Code di messaggi: permettono il trasferimento atomico di blocchi di byte. Comportamento FIFO, sono possibili più mittenti e l'inserimento di un blocco sincronizza mittente e destinatario
- Pipe: "tubi" che permettono il trasferimento di sequenze di byte di dimensioni arbitrarie. Occorre inserire marcatori che consentano di delimitare i singoli messaggi e la comunicazione è sincrona 1-1
- Memoria condivisa: insieme di pagine fisiche mappate in due o più spazi di indirizzamento. Richiedono ulteriori meccanismi di sincronizzazione per evitare interferenze tra i thread dei processi coinvolti
- Semafori: costrutti di sincronizzazione di baso livello basati sulla manipolazione atomica di un valore intero, gestito dal SO e MAi negativo (up incrementa il valore e down lo decrementa se non è 0, altrimenti si blocca in attesa di un incremento). Un semaforo inizializzato a 1 può essere visto come un mutex

I canali di comunicazione sono creati dal SO su richiesta dei singoli processi: questi ultimi devono accordarsi sul tipo e sull'identità del canale usato.

19 - IPC IN WINDOWS

La piattaforma Win32 offre una ricca serie di meccanismi di sincronizzazione (permettono di bloccare un thread fino a quando non si è verificato qualcosa in un altro thread), che si basano sull'uso di oggetti kernel condivisi. Essi sono più generali ma meno efficienti rispetto ai costrutti di sincronizzazione usabili in un singolo processo.
Tutte le tecniche si basano sull'uso delle funzioni di attesa *WaitForSingleObject(...)* e *WaitForMultipleObjects(...)*.
La maggior parte degli oggetti kernel può trovarsi in due stati differenti: Segnalato o Non Segnalato. Nel caso di processi e thread, il secondo indica che l'elaborazione è ancora in corso, mentre se raggiunge il primo, non è possibile tornare indietro. Gli oggetti di sincronizzazione (eventi, semafori, mutex) possono alternare i due stati di segnalazione in funzione delle proprie politiche e delle richieste eseguite nel programma.

Gli eventi sono oggetti kernel che modellano il verificarsi di una condizione (segnalata dal programmatore tramite opportuni metodi): gli eventi di tipo "manual-reset" permettono ad un numero indefinito di thread in attesa di svegliarsi, quelli "auto-reset"



tornano automaticamente allo stato non-segnalato se causano lo sblocco di un thread. Ci si collega ad un evento tramite *CreateEvent(...)* o *OpenEvent(...)* e due processi possono sincronizzarsi condividendo un evento. Si modifica lo stato attraverso le primitive:

- SetEvent: con auto_reset, un solo thread tra quelli in attesa viene rilasciato (se non ce ne sono, il prossimo ad arrivare sarà subito rilasciato). Con manual_reset, tutti i thread in attesa vengono rilasciati: l'evento resta segnalato fino ad una chiamata a ResetEvent
- ResetEvent
- PulseEvent: con auto_reset, un solo thread tra quelli in attesa viene rilasciato (se non ce ne sono, non succede nulla). Con manual_reset, tutti i thread in attesa vengono rilasciati. L'evento viene posto nello stato "non segnalato"

I semafori mantengono al proprio interno un contatore: stato segnalato se contatore maggiore di 0 e non segnalato se contatore uguale a 0. Non può mai essere negativo. La funzione WaitForSingleObject decrementa il contatore se esso è >0 altrimenti blocca il thread in attesa che un altro incrementi il contatore (invocando ReleaseSemaphore(...)).
I semafori sono usati tipicamente quando si hanno più copie di una risorse disponibili; funzioni che si possono usare: *CreateSemaphore(...), OpenSemaphore(...), WaitForSingleObject/WaitForMultipleObjects, ReleaseSemaphore.*

I mutex assicurano a più thread (anche di processi differenti) l'accesso in mutua esclusione ad una data risorsa e conservano l'ID del thread che li ha acquisiti ed hanno un contatore (che mantiene il numero di volte che il thread ha acquisito la risorsa.
Ciclo di vita di un mutex: si crea (con *CreateMutex* o *OpenMutex*), si usa *ReleaseMutex* per verificare che il thread corrente sia il possessore, decrementa il contatore e se arriva a 0 segnala il mutex. Usando *WaitForSingleObject/WaitForMultipleObjects,* si acquisice un mutex se è nello stato segnalato o se è già in possesso del thread e se ne incrementa il contatore, altrimenti attende che il mutex diventi segnalato.
Due o più oggetti kernel possono essere usati in modo congiunto per implementare particolari politiche di sincronizzazione.

Una Mailslot è una coda di messaggi asincrona per la comunicazione tra processi. Il mittente può essere un processo della macchina stessa oppure di un elaboratore appartenente allo stesso dominio di rete. Un processo può essere contemporaneamente sia un mailslot client che un mailslot server, e ciò permette di realizzare canali di comunicazione bidirezionali: è possibile inviare messaggi broadcast a tutte le mailslot con lo stesso nome nello stesso dominio di rete.
Si crea una mailslot server associandole un nome univoco: i messaggi vengono letti come normali file (con le API ReadFile(...) e ReadFileEx(...)) e vengono conservati finchè non letti; GetMailslotInfo(...) restituisce il numero di messaggi accodati e la dimensione del primo da leggere.
Il Mailslot Client accoda i messaggi (WriteFile(...) , WriteFileEx(...) sono operazioni atomiche!).
CloseHandle(...) chiude la mailslot e rilascia le relative risorse nel momento in cui tutti i suoi handle siano stati chiusi.

Le Pipe possono essere di due tipi:

- Anonymous Pipe: mezzo efficiente di comunicazione tra 2 processi "parenti". Ridirezione di stdio tra processo padre e figlio: il padre, dopo aver creato una pipe, può passare uno dei due handle al figlio. Solo monodirezionali. Un handle può anche essere duplicata nel processo con cui si intende comunicare (DuplicateHandle) e la sua identità può essere passata attraverso un segmento di memoria condiviso, facendoglielo ereditare. ReadFile e WriteFile servono a leggere e scrivere dalla pipe, ma non si può avere i/o asincrono
- Named Pipe: permettono comunicazioni tra due processi qualsiasi e possono essere create bidirezionali. Formato nome: "\\ServerName\pipe\PipeName". Funzioni per la gestione: CreateNamedPipe(...), OpenFile(...), CallNamedPipe(...), ReadFile(...), WriteFile(...)

All'atto della creazione, si può specificare la modalità di lettura dalla pipe (stream di byte o a messaggio), mentre la modalità di attesa determina il comportamento delle operazioni di lettura, scrittura e di connessione: in modalità bloccante la funzione attende per un tempo indefinito che il processo con cui si vuole comunicare termini le operazioni sulla pipe; in modalità non bloccante la funzione ritorna immediatamente nelle situazioni che richiederebbero un'attesa indefinita.

Il File Mapping è un meccanismo adatto per condividere ampie zone di memoria e a realizzare aree condivise persistenti. Permette di accedere al contenuto di un file come se fosse un blocco di memoria ma occhio, perchè occorre sincronizzare accessi concorrenti allo stesso file-mapping. Rimane un meccanismo efficiente per condividere dati tra più processi sullo stesso computer. Ciclo di vita: CreateFileMapping(...), MapViewOfFile (ritorna un puntatore che può essere usato per accedere all'area di memoria condivisa), UnmapViewOfFile, CloseHandle.

Altri meccanismi usati sono i socket, che permettono la comunicazione tra macchine dotate di SO differenti, anche se supportano solo il trasferimento di array di byte. E gli RPC (Remote Procedure Call) che è comunque basato sui socket e fornisce un meccanismo di alto livello per il trasferimento di dati strutturati.

20 - IPC IN LINUX

Ciascuna struttura IPC è identificata nel SO da un intero non negativo: all'atto della creazione, si fornisce una chiave che viene convertita dal SO nell'ID associato. Tutti i processi che conoscono la chiave possono ottenere l'ID corrispondente. Ciò non può avvere se si specifica IPC_PRIVATE(0).

Con le Message Queues i processi che intendono comunicare si accordano sul pathname di un file esistente e su un project-ID. Tramite la funzione ftok(...) convertono questi valori in una chiave univoca, e l'ultimo processo che fa accesso ad una struttura dati di IPC deve occuparsi della rimozione della stessa, altrimenti continua ad essere un oggetto kernel valido!
I messaggi sono composti da un tipo e da un payload, e sono puntatori a strutture del tipo:

```
struct message {
    long type;
    char messagetext [ MESSAGESIZE ];
}

// creo una nuova message queue, ritorna id
int msgget(key_t key, int msgflg);

/* invio di un messaggio: msqid-> id della coda, msgp->puntatore al messaggio
    il mittente deve avere il permesso di scrittura */
int msgsnd(int msqid, const void* msgp, size_t msgsz, int msgflg)

/* lettura di un messaggio: msgtyp indica il messaggio di interesse (0 viene restituito il pr
ssize_t msgrcv(int msqid, void* msgp, size_t msgsz, long msgtyp, int msgflg)

/* controllo di una coda: esegue l'operazione specificata da cmd, se IPC_RMID rilascia le ris
int msgctl(int msqid, int cmd, struct msqid_ds* buf)
```

Le Pipe permettono le IPC tra processi padre-figlio , creati con la funzione pipe(). Le Named pipe sono FIFO e permettono la comunicazione tra processi generici, create con la funzione *int mkfifo(const char* path, mode_t mode)* e a cui si accede come ad un file.

Esiste anche un meccanismo chiamato Shared Memory, che viene creato e acceduto con la funzione shmget() che restituisce l'id del segmento condiviso associato alla chiave key ed eventualmente alloca il segmento in base ai parametri shmflag . La dimensione restituita da shmget è uguale alla dimensione effettiva del segmento, arrotondata ad un multiplo di PAGE_SIZE.

```
int shmget(key_t key, size_t size, int shmflg)

// ottenere informazioni sul segmento, impostare permessi, rilasciare risorse
int shmctl(int shmid, int cmd, struct shmid_ds* buf)

// connette il segmento allo spazio di indirizzamento del processo chiamante
void* shmat(int shmid, const void* shmaddr, int shmflg)

//disconnette il segmento dallo spazio di indirizzamento del chiamante
int shmdt(const void* shmaddr)
```

Altre possibili operazioni sono: IPC_STAT (copia le informazioni dalla struttura dati del kernel associata alla memoria condivisa all'interno della struttura puntata da buf), IPC_SET (scrive i valori contenuti nella struttura dati puntata da buf nell'oggetto kernel corrispondente alla memoria condivisa specificata) e IPC_RMID (marca il segmento come da rimuovere. Il segmento viene rimosso dopo che dell'ultimo processo ha effettuato il detach).

Il memory mapped file sono porzioni di file mappate in memoria per condividere il contenuto del file tra processi in lettura/scrittura:

```
// mappo N byte del file FD a partire dall'offset OFF all'indirizzo START
void* mmap(void* start, size_t n, int prot, int flags, int fd, off_t off)

// rilascio il mapping: invalido indirizzi
// se il processo termina, rilascio automaticamente (ma non se chiudo il file)
int munmap(void* start, size_t length)
```

I semafori System-V vengono utilizzati per la sincronizzazione di thread di processi differenti e sono costituiti da un array di contatori: se è necessario proteggere più risorse, si può ottenere il lock contemporaneo.
Si crea con semget che restituisce l'id dell'insieme di semafori associati alla chiave.

```
int sid = semget( mySemKey, 1, IPC_CREAT | 0700 )

// sops->operazioni da compiere, n->numero elementi del semaforo
int semop(int semid, struct sembuf* sops, unsigned n)

struct sembuf{
    unsigned short sem_num; // indice del semaforo su cui operare
    short sem_op;
    // operazione da effettuare: >0 acquisizione, <0 rilascio
    sem_flg; // solitamente 0
}

// effettuo operazione cmd sull'i-esimo semaforo identificato da semid
int semctl(int semid, int i, int cmd, [union semun arg])

// rilascio risorse relative ad un semaforo
semctl( sid, 0, IPC_RMID, 0);
```

Le possibili operazioni sono: IPC_STAT (copia le informazioni dalla struttura dati del kernel associata al semaforo all'interno della struttura semid_ds puntata da arg.buf), IPC_SET (scrive i valori contenuti nella struttura dati semid_ds puntata da arg.buf nell'oggetto kernel corrispondente all'insieme di semafori e aggiorna sem_ctime), IPC_SETALL (imposta semval per tutti i semafori del set utilizzando arg.array e aggiorna sem_ctime) e IPC_GETALL (restituisce i semval correnti di tutti i semafori).