Programmazione concorrente

(introduzione)

Processi e thread

Processi e thread

Un **processo** è generato da un programma in esecuzione.
Un programma eseguito su un processore può dare origine a più processi.

Ogni **processo** ha il proprio contesto, ovvero il proprio:

process ID,
Program Counter,
Stato dei Registri,
Stack,
Codice eseguibile,
Dati
.....

Il Codice eseguibile tradizionale è pensato per eseguire procedure **sequenzialmente**.
Il processo (o **processo pesante**) equivale a un task con un solo thread.

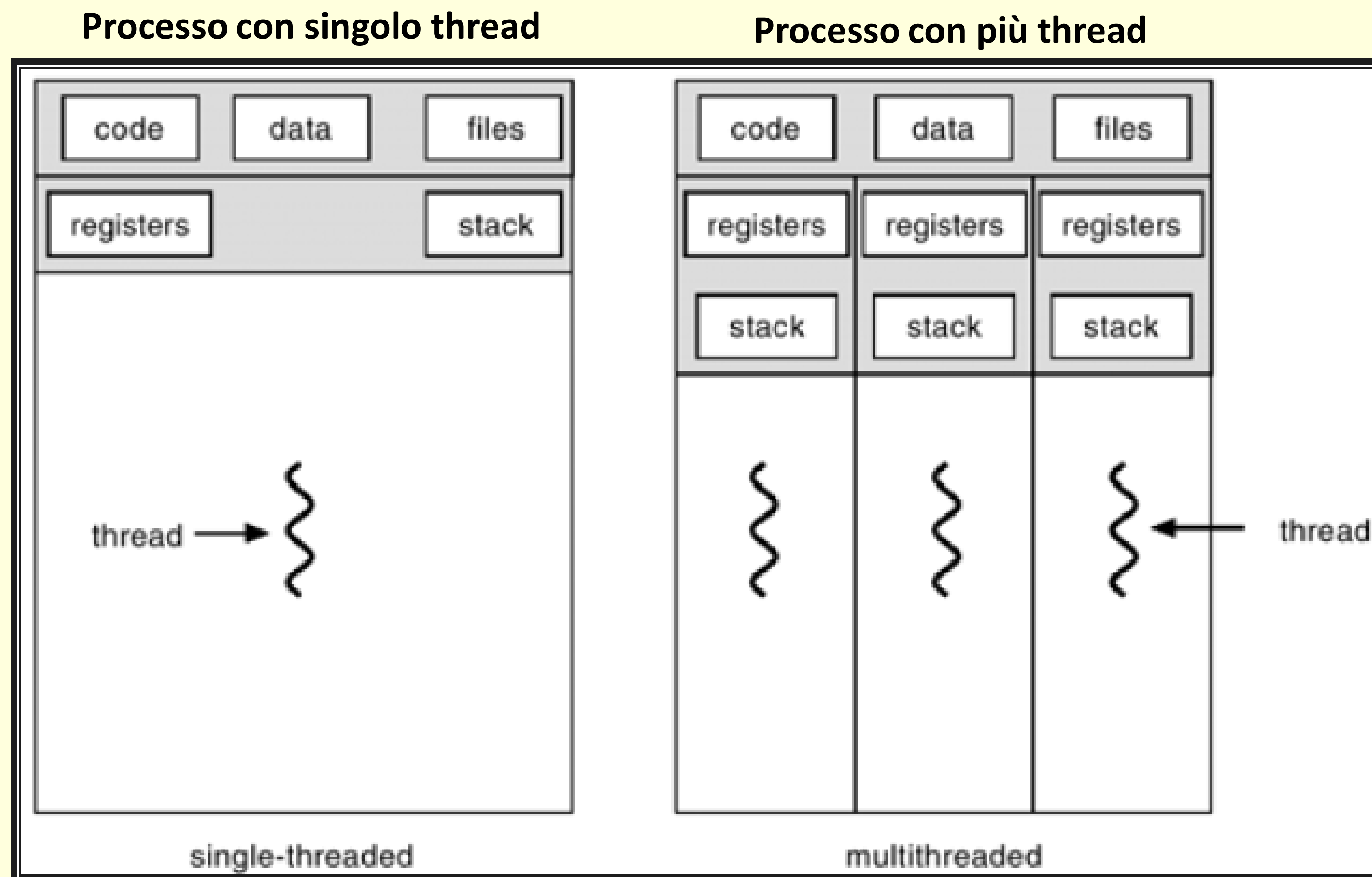
Cos'è un thread?

Un **thread** (o **processo leggero**) è una sequenza di istruzioni di un programma in esecuzione.

Praticamente

il **flusso esecutivo di un processo**
viene scomposto in
più flussi concorrenti.

Processi e thread



L'astrazione dei thread vuole consentire di **eseguire procedure concorrentemente** (in parallelo), ovviamente scrivendo tali procedure in modo opportuno.

Processi e thread

Thread significa "**trama**" /"**fibra**" e designa un percorso di **esecuzione parallela** (**execution path**).

> Un **thread** è la **minima unità di esecuzione** che il sistema operativo può gestire in modo indipendente.

Ogni thread **condivide** con altri thread:

- il codice eseguibile (stesso spazio degli indirizzi)
- area dati globali
- *heap*
- “ambiente“
- La tabella di descrittori dei file
- ……

ma ha **elementi propri**:

- dati propri
- uno stack (variabili locali)
- un Program Counter
- un set di registri

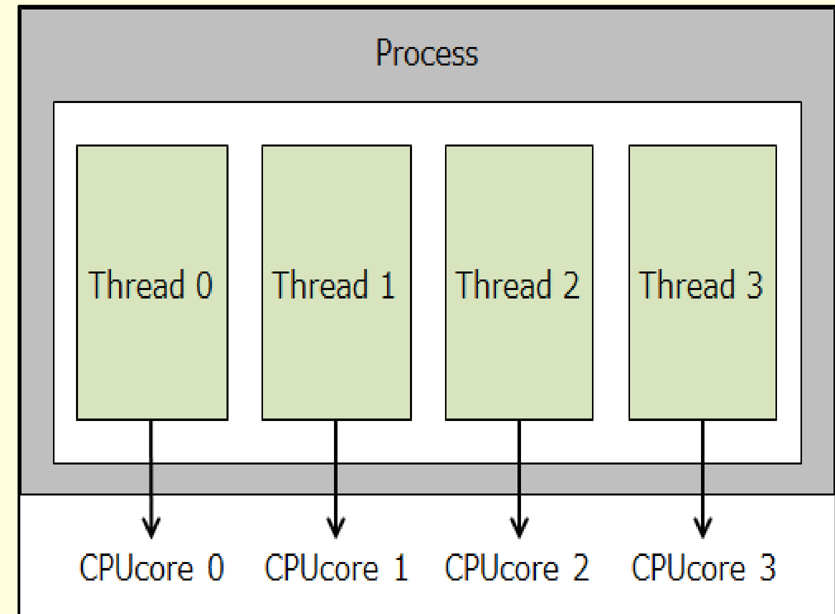
questi ultimi indispensabili per permettere un **execution path indipendente** per ogni thread.

Thread

In generale con i thread si ottiene un **miglioramento delle prestazioni** del sistema dato che alcuni algoritmi vengono eseguiti in parallelo da più thread.

Attenzione: si ottiene un **aumento della velocità** complessiva di esecuzione solo se **si dispone di un adeguato numero di CPU** assegnate ai vari thread o processi.

Nei computer moderni dotati di processori con CPU multicore, l'aumento delle prestazioni di esecuzione dei programmi è spesso basato sulla suddivisione dei dati da elaborare in sezioni su cui è possibile agire indipendentemente.



Parallelizzando l'esecuzione degli algoritmi, **se la mole dei dati da elaborare non è sufficientemente elevata**, si rischia che **il tempo richiesto dal sistema operativo per creare e sincronizzare processi o thread sia superiore a quello necessario per l'elaborazione sequenziale dei dati**!

Thread

Vantaggi: THREAD

- **Visibilità dei dati globali**: condivisione di oggetti semplificata.
- **Più flussi di esecuzione** (minor numero di core inutilizzati).
- **gestione semplice di eventi asincroni** (I/O per esempio).
- **Comunicazioni veloci**. Tutti i thread di un processo condividono lo stesso spazio di indirizzamento, quindi le comunicazioni tra thread sono più semplici delle comunicazioni tra processi.
- **Context switch veloce**. Nel passaggio da un thread ad un altro di uno stesso processo viene mantenuto buona parte dell'ambiente.

Svantaggi:

- **Concorrenza**: gestire la mutua esclusione.
- **Routine di libreria devono essere rientranti** (thread safe call): le routine devono essere costruite in modo da poter essere utilizzate da più thread contemporaneamente (non accedere in lettura/scrittura a variabili globali) Es: se una funzione scrive il proprio risultato in una variabile di sistema (del processo) e restituisce al chiamante un puntatore a tale variabile e due thread di uno stesso processo eseguono "nello stesso istante" la chiamata a quella funzione ognuno setta la variabile con un valore che potrebbe non essere corretto.

Garantire la Mutua Esclusione

due thread devono decrementare il valore di una **variabile globale M**, se questa è maggiore di zero. Inizialmente M vale 1.

THREAD1	THREAD2
if(M > 0)	if(M > 0)
M--;	M--;

A seconda del tempo di esecuzione dei due thread,la variabile **M** assume valori diversi.

M	THREAD1	THREAD2
1	if(M > 0)	
1	M--;	
0		if(M > 0)
0		M--;

0 = valore finale di M

M	THREAD1	THREAD2
1	if(M > 0)	
1		if(M > 0)
1		M--;
0	M--;	

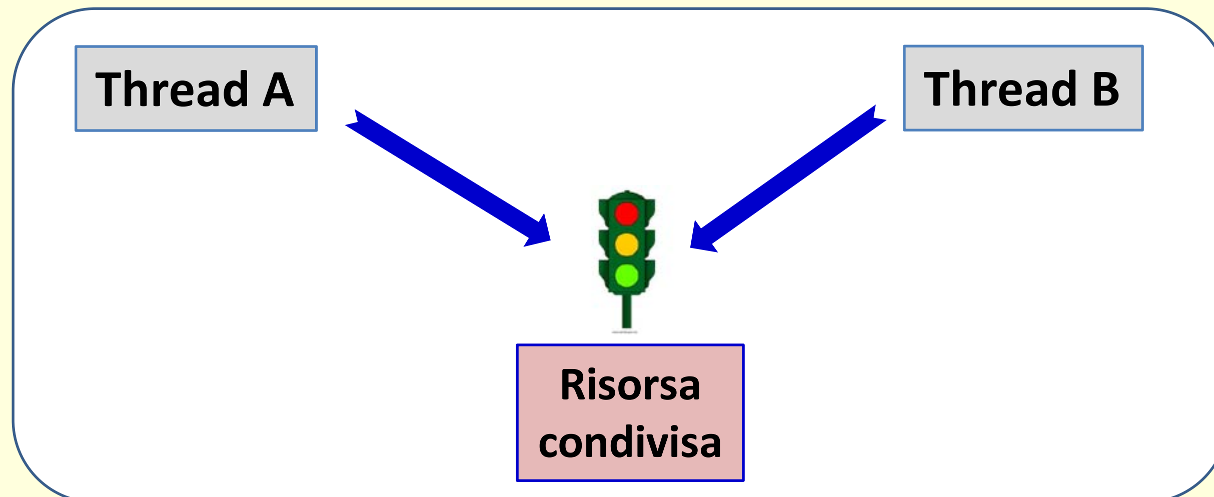
-1 = valore finale di M

Garantire la Mutua Esclusione

Quando due thread hanno necessità di **interagire** tra loro devono "**sincronizzarsi**" in modo da evitare situazioni non deterministiche.
La **sincronizzazione** è uno degli aspetti più complessi della **programmazione concorrente**.
Si può attuare con la **mutua esclusione**.

La ***mutua esclusione*** è utilizzata per:

> **consentire l'accesso concorrente** a **strutture dati condivise** impedendo che le elaborazioni portino a situazioni non corrette dal punto di vista dei valori contenuti nelle aree dati condivise.

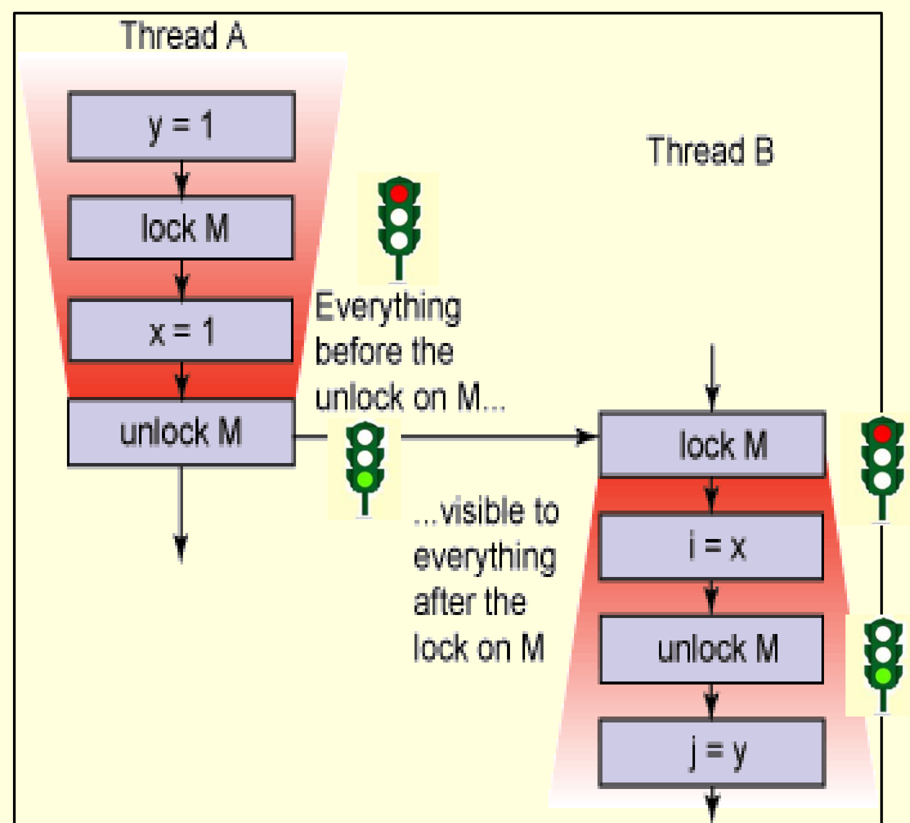


Garantire la Mutua Esclusione

Ogni thread, prima di accedere a tali dati, deve effettuare una **operazione di lock** su una stessa variabile.

L'operazione detta "**lock**" di una variabile **blocca l'accesso da parte di altri thread**. Infatti, se più thread eseguono l'operazione di lock su una stessa variabile, solo uno dei thread termina la lock() e prosegue l'esecuzione, gli altri rimangono bloccati nella lock. In tal modo, il processo che continua l'esecuzione può accedere ai dati (protetti).

Finito l'accesso, il thread effettua un'operazione detta "**unlock**" che libera la variabile.
Un altro thread che ha precedentemente eseguito la lock sulla variabile potrà allora terminare la lock ed accedere a sua volta ai dati.



Processi e thread in Windows

Processi e thread
in Windows

Nella letteratura Microsoft **ogni eseguibile**, anche se singolo, **è costituito da almeno un thread**.

Il thread che nasce per primo è detto **main thread**, ed è esso che eventualmente **crea thread figli**.

I **thread figli** condividono → le **risorse** del **main thread**,
e in particolare la **memoria** e quindi le **variabili**

Le **funzioni API** (Application Programming Interface) dell'ambiente Windows consentono di **creare** con facilità **thread** che il sistema operativo esegue *contemporaneamente* nel contesto del processo che li ha creati.

Il **codice di un thread** (eseguibile) coincide normalmente con una funzione la cui esecuzione avviene in parallelo con l'esecuzione del codice del processo che crea il thread e con quella degli altri thread creati dallo stesso processo.

Processi e thread in Windows

Funzione CreateThread

Sintassi

```
HANDLE WINAPI CreateThread(   _In_opt_    LPSECURITY_ATTRIBUTES  lpThreadAttributes,
                              _In_        SIZE_T   dwStackSize,
                              _In_        LPTHREAD_START_ROUTINE  lpStartAddress,
                              _In_opt_    LPVOID   lpParameter,
                              _In_        DWORD   dwCreationFlags,
                              _Out_opt_   LPDWORD   lpThreadId );
```

Parameters

***lpThreadAttributes* [in, optional]**

A pointer to a **SECURITY_ATTRIBUTES** structure that determines whether the returned handle can be inherited by child processes. The lpSecurityDescriptor member of the structure specifies a security descriptor for the new thread. If lpThreadAttributes is **NULL**, the thread gets a default security descriptor.

***dwStackSize* [in]**

The initial size of the stack, in bytes. The system rounds this value to the nearest page. If this parameter is zero, the new thread uses the default size for the executable.

***lpStartAddress* [in]**

A pointer to the application-defined function to be executed by the thread. This pointer represents the starting address of the thread.

***lpParameter* [in, optional]**

A pointer to a variable to be passed to the thread.

***dwCreationFlags* [in]**

The flags that control the creation of the thread. 0= The thread runs immediately after creation.

***lpThreadId* [out, optional]**

A pointer to a variable that receives the thread identifier. If this parameter is **NULL**, the thread identifier is not returned.

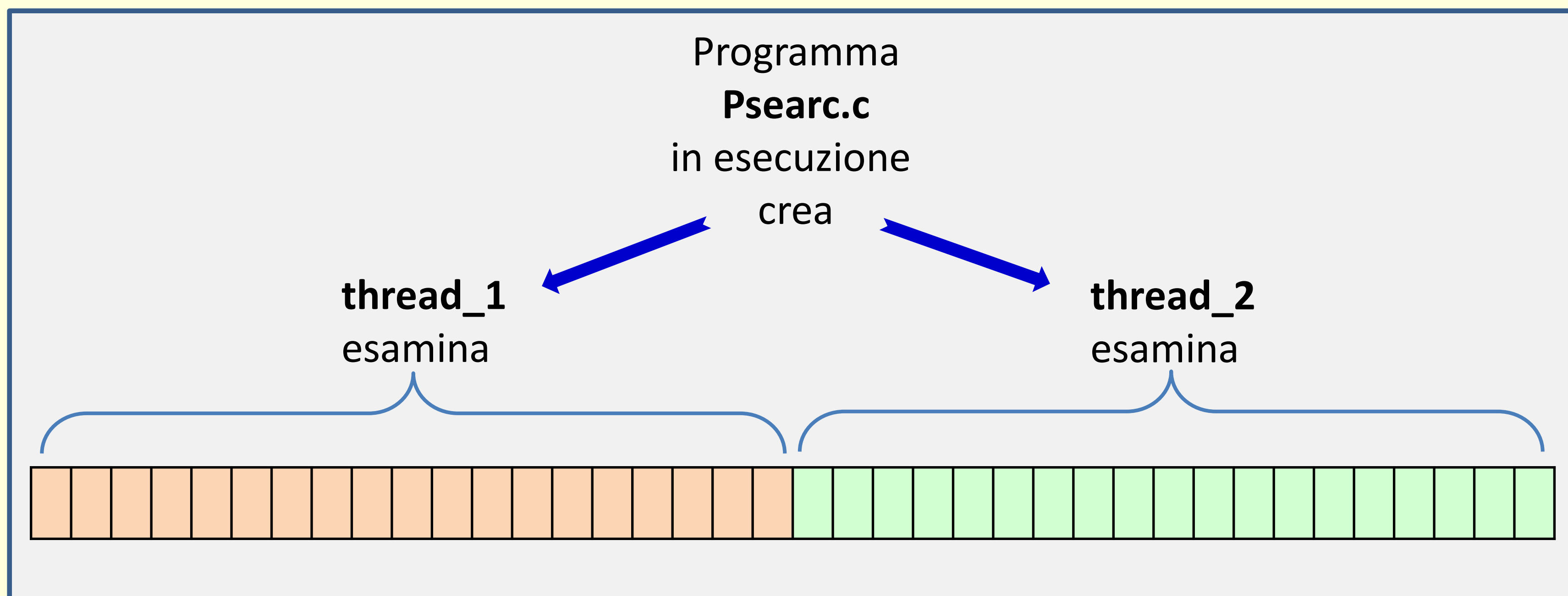
Return value

If the function succeeds, the **return value** is a **handle** to the new thread. If the function fails, the return value is **NULL**.

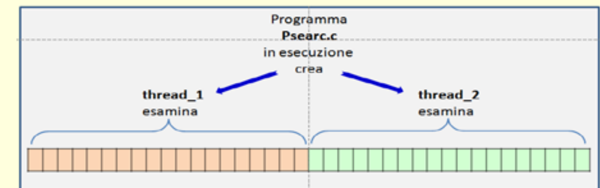
Processi e thread in Windows

Esercizio

Vogliamo realizzare un programma in linguaggio C per ambiente Windows che visualizza tutti gli indici degli elementi di un vettore di valori, generati casualmente, corrispondenti a un valore fornito dall'utente, creando due thread che analizzano due distinte sezioni del vettore.



Processi e thread in Windows



Soluzione

- Deve essere inclusa la libreria **windows.h**
  Il file di intestazione **windows.h** *comprende i prototipi delle funzioni API* del sistema operativo Windows.
- Dichiarazione di una **struttura** che contiene i parametri da passare alla **funzione search** descritta al punto seguente.

```
struct PAR
{
   int first_index;      //  1° indice del vettore
   int last_index;       //  2° indice del vettore
   int search_value;  //  valore da ricercare
};
```

- Implementazione della **funzione search** che effettua la ricerca di un valore in una sezione del vettore compresa tra due indici e visualizza gli indici degli elementi corrispondenti al valore ricercato.
- **Programma main**
  Dopo avere **inizializzato il vettore** con numeri casuali, **crea due thread** basati sulla funzione search, **che ricercano in parallelo il valore fornito dall'utente** rispettivamente nella prima e nella seconda metà del vettore e ne attende la conclusione.

Implementazione in C: ***Psearch.c***

Processi e thread
in Windows

- Il file di intestazione *windows.h* comprende i prototipi delle funzioni API del sistema operativo Windows. In questo file è definita la costante di compilazione ***WINAPI***, che nei prototipi precede il nome della funzione stessa[2].

- La funzione che implementa il *thread* deve obbligatoriamente avere il seguente **prototipo**:

```
unsigned long WINAPI fun(void* par);
```

  L'unico parametro che la funzione può ricevere come argomento è un puntatore generico: la tecnica utilizzata dal programma per aggirare questa limitazione consiste nel dichiarare una struttura i cui elementi sono i parametri della funzione e fornire come argomento un puntatore a questa struttura. La funzione che implementa il *thread* si conclude con l'invocazione della funzione **API *ExitThread***, che sostituisce la classica `return`.

2. Nel compilatore Microsoft C/C++ *WINAPI* definisce la speciale *keyword* `_stdcall`, che stabilisce una particolare convenzione per il passaggio dei parametri di una funzione.

OSSERVAZIONE Il codice della funzione *search* viene eseguito in modo indipendente e contemporaneo dai due *thread* creati dal programma. Il parametro ricevuto come argomento e le variabili locali sono duplicate al momento della creazione del *thread*.

Processi e thread
in Windows

- La funzione API ***CreateThread*** istanzia il *thread* a partire dalla funzione fornita come argomento. I parametri sono i seguenti:

  – un puntatore a una struttura definita nel file di intestazione *windows.h* che consente di gestire in modo avanzato i criteri di sicurezza relativi all'esecuzione del *thread*; l'uso del valore `NULL` consente di non specificare tali criteri;
  – la dimensione in byte dell'area di memoria necessaria per la duplicazione dei parametri e delle variabili locali della funzione che implementa il codice del *thread*;
  – l'indirizzo della funzione che implementa il codice del *thread*[3];
  – l'argomento da fornire alla funzione che implementa il codice del *thread* (un puntatore che può eventualmente essere `NULL` se la funzione non utilizza parametri);
  – un puntatore a un valore intero che consente di gestire in modo avanzato i criteri di creazione del *thread* da parte del sistema operativo; l'uso del valore «0» consente di specificare criteri standard;
  – un puntatore a una variabile intera dove la funzione API restituisce l'identificativo numerico del *thread* creato: se `NULL`, il valore non viene restituito.

  La funzione API *CreateThread* restituisce un riferimento (***handle*** nella terminologia Windows) al *thread* che crea; questo riferimento deve essere fornito come argomento alle funzioni API che interagiscono con il *thread*. Nel caso che il sistema operativo non riesca a creare il *thread*, la funzione *CreateThread* restituisce un *handle* non valido (il valore `NULL`).

3. Nel linguaggio C è possibile fornire una funzione come argomento a un'altra funzione, passando come parametro un puntatore alla funzione stessa.

Processi e thread in Windows

- La funzione API *GetCurrentThreadId* restituisce l'identificativo numerico del *thread* all'interno del quale viene invocata.

- La funzione API *WaitForSingleObject* consente di attendere la terminazione di un *thread* specificandone l'*handle*. L'attesa può avere un limite temporale espresso in millisecondi fornito alla funzione come secondo argomento; la costante di compilazione `INFINITE` definita nel file di intestazione *windows.h* implica l'attesa senza limiti.

Le funzioni di libreria _beginthreadex e _endthreadex

Le funzioni API *CreateThread* e *ExitThread* per la creazione e la terminazione di un *thread* in ambiente Windows possono essere invocate indirettamente utilizzando le funzioni di libreria

_beginthreadex e
_endthreadex

che richiedono l'inclusione del file di intestazione *process.h*. Quando i *thread* invocano le funzioni della libreria standard del linguaggio C, è consigliato utilizzare queste funzioni anziché invocare direttamente le API del sistema operativo.

OSSERVAZIONE Nel caso di attesa di più *thread* è possibile utilizzare la funzione API *WaitForMultipleObjects*, i cui parametri sono i seguenti:

- il numero di *thread*;
- un vettore degli *handle* dei *thread*;
- una variabile intera che specifica se attendere un solo *thread* (valore `FALSE` = 0) o tutti i *thread* (valore `TRUE` = 1);
- il limite temporale in millisecondi (eventualmente la costante di compilazione `INFINITE`).

Volendo utilizzare questa funzione API il codice del programma di esempio deve essere così modificato:

```
void main(void)
{
  int i, n;
  struct PAR par_1, par_2;
```

Processi e thread
in Windows

```
  unsigned long thread[2];
  …
  …
  …
  thread[0] = CreateThread(NULL,4096,&search,&par1,0,NULL);
  thread[1] = CreateThread(NULL,4096,&search,&par2,0,NULL);

  WaitForMultipleObject(2,thread,1,INFINITE);
}
```

OSSERVAZIONE Il vettore su cui viene effettuata la ricerca degli elementi nel programma di esempio precedente è effettivamente condiviso tra i due *thread*, contrariamente a quanto avviene in ambiente Linux, dove l'invocazione della funzione API *fork* causa la duplicazione del contenuto della memoria del processo clonato.

Processi e thread
in Windows

Esercizi

1. Realizzare un programma che, anziché effettuare la ricerca completa di tutte le occorrenze di un valore tra gli elementi di un vettore, si limita a ricercare un elemento corrispondente al valore stesso, è semplice fare in modo che l'esecuzione di tutti i thread si arresti al momento in cui si verifica la prima corrispondenza. A tale scopo si impieghi una variabile globale di segnalazione (flag) che tutti i thread condividano:

   ```
   bool  flag =true      //0=non trovato, 1=trovato
   ```

2. Scrivere un programma che accetti un intero N da riga di comando, con $1<N\leq10$, crei N thread e poi aspetti la loro terminazione. (Utilizzare nel programma principale la funzione WaitForMultipleObject(…)).
   Ciascun thread aspetta un numero di secondi casuale tra 1 e 10, poi incrementa una variabile globale intera ed infine ne stampa il valore.

3. Scrivere un programma che calcoli la somma degli elementi di un vettore dichiarato nell'area globale, utilizzando quattro thread che operano in parallelo.