Programmazione concorrente

Programmazione di Sistema

A.A. 2017-18

Argomenti

- Programmazione concorrente
- Modello di esecuzione dei thread
- Sincronizzazione
- Thread nativi in Windows
- Thread nativi in Linux

Programmazione concorrente

- Un programma concorrente dispone di due o più flussi di esecuzione
  - Nello stesso spazio di indirizzamento
  - Per perseguire un obiettivo comune
- All'atto della creazione, un processo dispone di un unico flusso di esecuzione
  - Thread principale
  - Esso può richiedere al S.O. la creazione di altri thread
- Il S.O. e/o le librerie di supporto allocano le risorse fisiche necessarie
  - 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
  - È possibile sfruttare i tempi di attesa delle operazioni di I/O per eseguire altre parti dell'algoritmo
- Riduzione del sovraccarico dovuto alla comunicazione tra processi
  - Lo spazio di memoria è condiviso, quindi non occorre copiare i risultati parziali né serializzarne i contenuti
- Utilizzo delle CPU multicore
  - Vero parallelismo
  - Più flussi di esecuzione possono svolgersi contemporaneamente, riducendo il tempo totale di elaborazione

Svantaggi

- Aumento significativo della complessità del programma
  - Nuove fonti e tipologie di errore
  - Non determinismo dell'esecuzione

Complessità

- La memoria non può più essere pensata come un “deposito statico”
  - I dati scritti possono cambiare in conseguenza dell’attività di altri thread
- I thread devono coordinare l’accesso alla memoria
  - Tramite opportuni costrutti di sincronizzazione

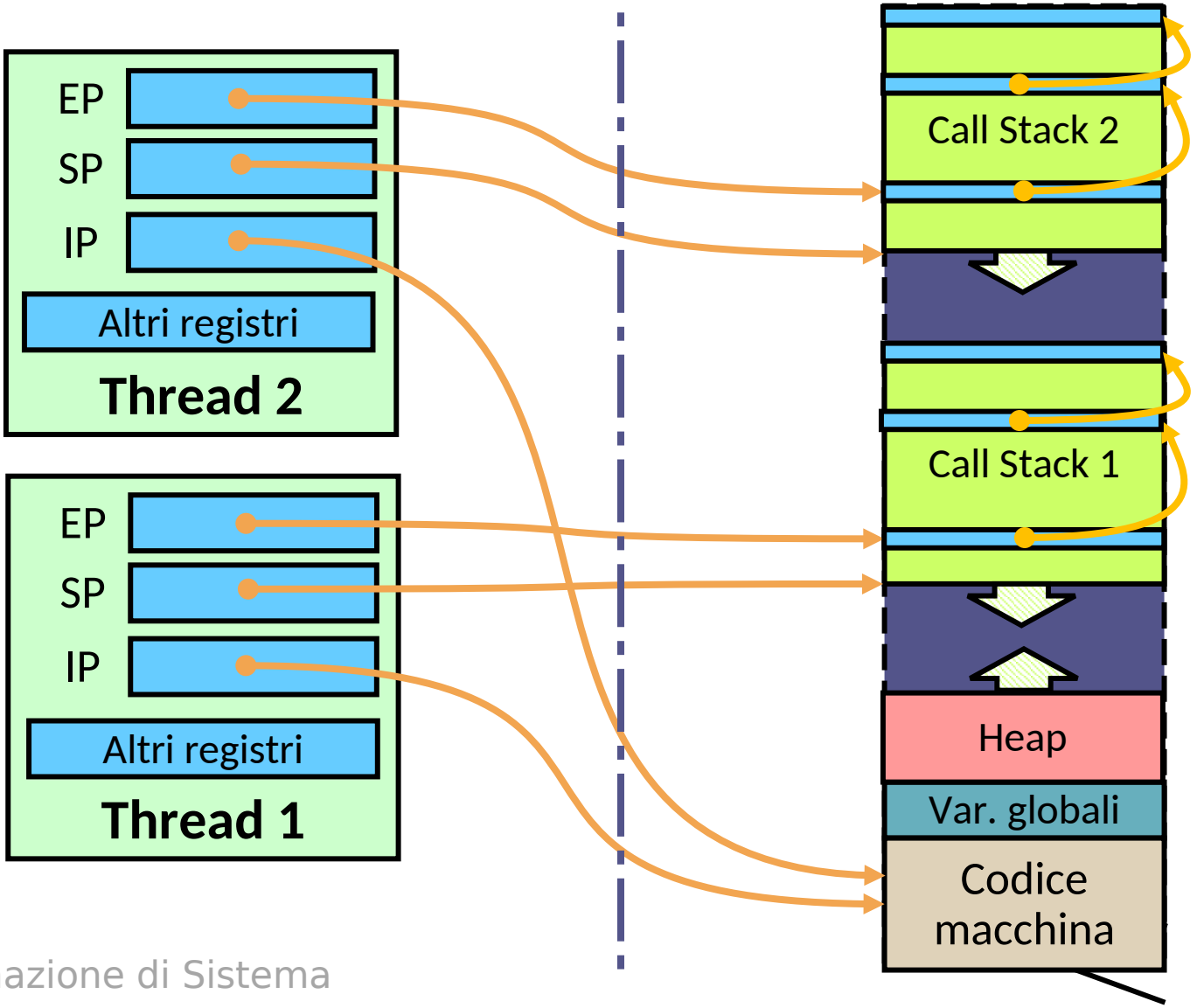
Errori

- L'uso superficiale dei costrutti di sincronizzazione porta a blocchi passivi o attivi del programma ...
- Si possono causare malfunzionamenti casuali
  - Dovuti al comportamento non deterministico e asincrono dell'esecuzione concorrente
  - Estremamente difficili da riprodurre e da eliminare
- Gli errori possono manifestarsi cambiando la piattaforma di esecuzione
  - Oppure soltanto dopo numerose esecuzioni

Thread e memoria

- Ogni thread dispone di
  - Un proprio stack delle chiamate
  - Un proprio puntatore all'ultimo contesto per la gestione delle eccezioni
  - Lo stato del proprio "processore virtuale"
- I thread dello stesso processo condividono
  - Le variabili globali
  - L'area in cui è memorizzato il codice
  - L'area delle costanti
  - Lo heap

Thread e memoria



Esecuzione concorrente

- L'esecuzione si alterna
  - Ogni thread accede a variabili locali elementari, memorizzate nello stack (il cui puntatore può essere dischiuso ad altri thread), a dati condivisi sullo heap ed alle variabili globali
- 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
  - Occorre essere in grado di comunicare delle informazioni e sapere quando esse sono valide/disponibili

Esempio

```
int data[10];

void thread1() {
  for (int i=0; i<10; i++)
    data[i]= calcola_nuovo_valore();
}

void thread2() {
  for (int i=0; i<10; i++)
    usa_valore(data[i]);
    //Quando è pronto questo dato?
}
```


Esecuzione e non determinismo

- L'esecuzione di un singolo thread procede secondo le normali regole sequenziali
  - Per cui è possibile dire cosa avvenga «prima» e cosa «dopo»
- Se più thread sono in esecuzione, non è possibile fare assunzioni sulle velocità relative di avanzamento
- L'esecuzione ripetuta dello stesso programma (con gli stessi ingressi) può portare a risultati differenti
  - A seguito delle interazioni (complesse) tra il S.O. e i dispositivi HW

Esempio

```
#include <thread>
#include <iostream>
#include <string>

void run(std::string msg) {
  for (int j=0; j<10; j++) {
    std::string s= msg + std::to_string(j) + "\n";
    std::cout << s;
  }
}
int main() {
  std::thread t1(run, "aaaa");
  std::thread t2(run," bbbb");
  t1.join();
  t2.join();
}
```

```
aaaa0
 bbbb0
aaaa1
 bbbb1
aaaa2
 bbbb2
aaaa3
 bbbb3
aaaa4
 bbbb4
aaaa5
aaaa6
aaaa7
 bbbb5
aaaa8
aaaa9
 bbbb6
 bbbb7
 bbbb8
 bbbb9
```


Osservazioni

- L’output del programma precedente è solo uno dei possibili risultati
  - Se lo stesso programma viene eseguito più volte, si ottengono risultati differenti
- È assolutamente possibile (e a volte succede) che tutte le righe di uno dei thread precedano quelle dell'altro
- L’unica certezza è che le righe che cominciano con “aaaa” sono tra loro ordinate in modo crescente
  - Così come le righe che cominciano con “bbbb”
- Il non determinismo dà origine a comportamenti del tutto inattesi in un contesto di elaborazione sequenziale

Esempio

```
#include <iostream>
#include <thread>

int a=0;

void run() {
  while (a>=0) {
    int before=a;
    a++;
    int after=a;
    if (after-before!=1)
      std::cout<< before<<" -> "<<
after<<"("
                << after-before<<")\n";
  }
}
```

Esempio

```
//creo due thread e ne attendo la
terminazione

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

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


Esecuzione

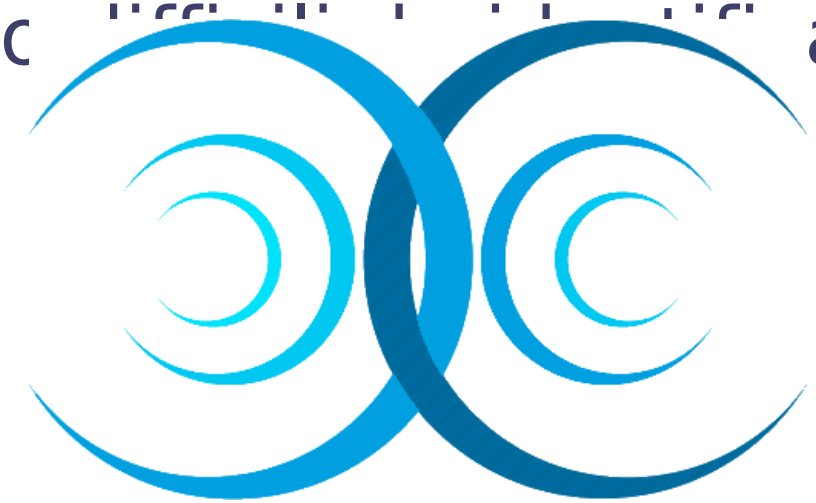
```
119944578 -> 119944552 (-26)
123397102 -> 123397584 (482)
128314912 -> 128314956 (44)
395835151 -> 395835236 (85)
396049424 -> 396098482 (49058)
412859791 -> 412859826 (35)
419214490 -> 419214537 (47)
419406880 -> 419406877 (-3)
433982464 -> 433982472 (8)
436005364 -> 436215900 (210536)
441453011 -> 441454010 (999)
446802106 -> 446802106 (0)
```


Domande

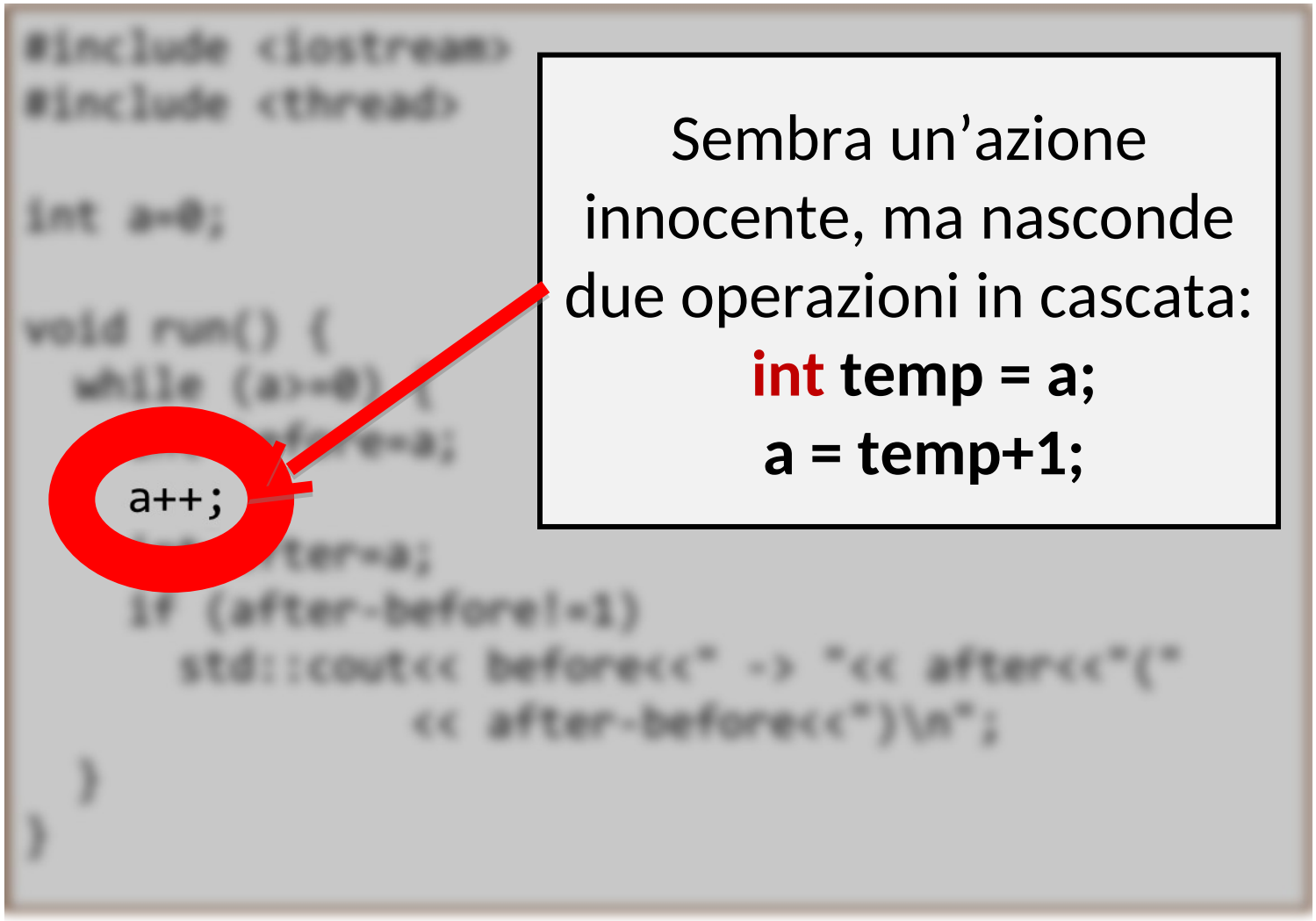
- Esaminando l'uscita del programma precedente si vedono molti casi in cui la differenza tra after e before è superiore a 1
  - In alcuni casi tale valore è anche molto grande: perché?
- Talora capita che la differenza sia nulla o negativa
  - Come è possibile, se entrambi i flussi incrementano sempre la variabile a?

Interferenza

- Si verifica quando più thread fanno accesso ad un medesimo dato, modificandolo
  - La sua presenza dà origine a **malfunzionamenti casuali**, molto difficili da identificare



Interferenza



Sembra un'azione innocente, ma nasconde due operazioni in cascata:

int temp = a;

a = temp+1;

Sincronizzazione

- È necessario evitare che altri thread accedano ad una risorsa condivisa mentre una modifica è in corso
  - Meccanismo che regoli l'accesso alle zone pericolose
- Ogni S.O. offre strumenti leggermente diversi, sia in termini di strutture dati che di API
  - La libreria C++ 2011 introduce un meccanismo standard per la sincronizzazione, indipendente dalla piattaforma soggiacente

Sincronizzazione

- Windows
  - CriticalSection
  - ConditionVariable
  - Oggetti kernel
    - Mutex, Event, Semaphore, Pipe, Mailslot, …
- Linux
  - Oggetti della libreria PThreads
    - pthread_mutex,pthread_cond
  - Oggetti kernel
    - Semafori, pipe, segnali, futex

Correttezza

- Occorre fare in modo che non capiti mai che un thread “operi” su un dato, alterandone il contenuto
  - Mentre un altro sta già operando sullo stesso oggetto
- In particolare, non devono essere visibili stati “transitori” dell’oggetto
  - Dovuti al meccanismo di aggiornamento
- Gli oggetti condivisi mutabili devono godere di questa proprietà
  - Questi mantengono al proprio interno degli “invarianti”
  - Perché gli oggetti immutabili non sono soggetti ad interferenza?

Correttezza

- Bisogna impedire che gli invarianti siano violati
  - Si effettuano le mutazioni (cambi di stato) con metodi che garantiscono la validità degli invarianti prima e dopo l'esecuzione e che bloccano l'accesso concorrente
- Si accede allo stato attraverso altri metodi
  - Che controllano che non ci sia una mutazione in corso
  - E che impediscono che essa inizi mentre si sta facendo accesso allo stato condiviso
- È compito del programmatore riconoscere quando e dove utilizzare la sincronizzazione
  - Un uso sbagliato porta a risultati disastrosi

Accesso condiviso: le cause del problema

- Atomicità
  - Quali operazioni di memoria hanno effetti indivisibili?
- Visibilità
  - La scrittura di una variabile può essere osservata da una lettura eseguita da un altro thread?
- Ordinamento
  - Sotto quali condizioni, sequenze di operazioni effettuate da un thread sono visibili nello stesso ordine da parte di altri thread?

Accesso non sincronizzato ad un dato

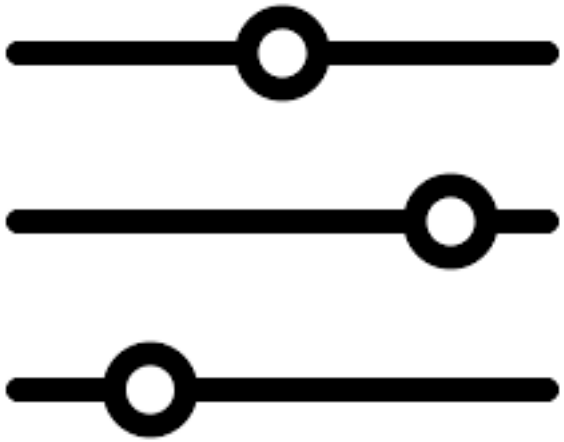
- Se due thread fanno accesso allo **stessa struttura dati**, rispettivamente in lettura e scrittura
  - Non c'è nessuna garanzia su quale delle due operazioni sia eseguita per prima

Accesso ad un dato mentre un modifica è in corso

- Se un thread legge un dato che un altro thread sta modificando
  - Il valore letto può essere **diverso** sia dal valore iniziale che da quello finale

Ri-ordinamento

- 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'e delle singole ist



Accesso non sincronizzato

```
int val; //variabili globale
void f(int v) { ... }

void thread1() {
// invoco f con |val|
  if (val >=0)
    f(val) // non è detto che a
questo
           // punto val sia ancora
>=0
} else {
    f(-val);  //idem
  }
}
```

Accesso non sincronizzato

```
std::vector<int> v; //globale


void threadFunc() {
  //...
  if (!v.empty()) {
    std::cout<<v.front()<<std:endl;
    // anche qui, v.front()
    // potrebbe non esistere più
  }
}
```


Accesso non sincronizzato

- In generale, non è lecito operare né il lettura né in scrittura su una qualsiasi struttura dati mentre è in corso un'altra scrittura
- Fanno eccezione
  - L'accesso a elementi distinti di uno stesso contenitore
  - L'uso dei flussi di caratteri std::cin, std::cout, std::cerr

Lettura durante una modifica

- Se, nello stesso istante, due thread differenti leggono e scrivono una stessa cella di memoria il risultato è impredicibile
  - Potrebbe essere restituito il valore precedente alla scrittura
  - Potrebbe essere restituito il valore scritto
  - Potrebbe essere restituito un qualsiasi altro valore

I problemi introdotti

- Mancata sincronizzazione
  - Malfunzionamenti casuali anche gravi
- Errata sincronizzazione
  - Blocco
- Eccessiva sincronizzazione
  - Scarse prestazioni

Meccanismi di sincronizzazione

- La complessità di gestione della sincronizzazione spinge all'utilizzo di tecniche note ed affidabili (pattern)
  - È sempre necessaria molta cautela nella realizzazione di programmi multi-thread

Uso dei thread

- Le API dei S.O. permettono la gestione del ciclo di vita dei thread
  - Creazione e terminazione di thread
  - Meccanismi di sincronizzazione
  - Aree private di memoria
- I dettagli relativi a ciascuna piattaforma differiscono alquanto
  - Rendendo complessa la portabilità delle applicazioni
- La versione 2011 del linguaggio C++ ha introdotto una standardizzazione nella creazione e gestione dei thread

Thread in windows

- Creazione di un thread

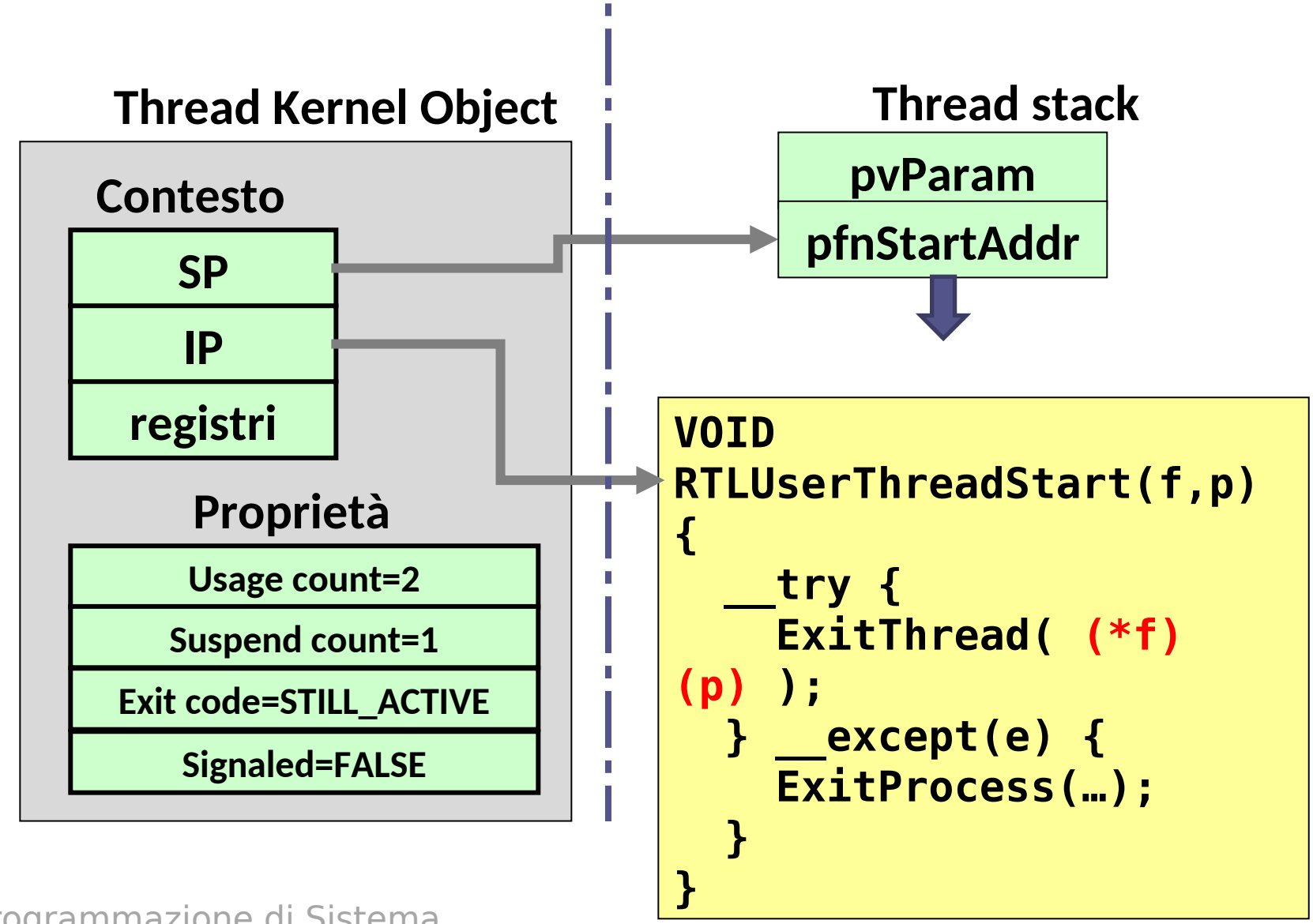
```
HANDLE WINAPI CreateThread(
  LPSECURITY_ATTRIBUTES
lpThreadAttributes,
  SIZE_T           dwStackSize,
  LPTHREAD_START_ROUTINE
lpStartAddress,
  LPVOID           lpParameter,
  DWORD            dwCreationFlags,
  LPDWORD          lpThreadId
);
```


Parametri

- lpThreadAttributes
  - Puntatore al contesto di sicurezza in cui il thread verrà eseguito
- stackSize
  - Dimensione dello stack
- lpStartAddress
  - Indirizzo della funzione principale del thread
- lpParameter
  - Puntatore ad un eventuale parametro passato alla funzione principale
- dwCreationFlag
  - Indica se il thread dovrà essere attivato subito (0) o sospeso (CREATE_SUSPENDED)
- lpThreadId
  - Indirizzo di una variabile in cui sarà memorizzato l'id del thread

CreateThread

- La funzione ritorna un riferimento opaco (handle)
  - Utilizzato per fare riferimento al thread nelle altre funzioni del S.O.
  - NULL indica un fallimento



Thread e librerie C/C++

- La libreria standard del C utilizza variabili globali il cui accesso non è sincronizzato
  - Se usate (direttamente o indirettamente) da un thread secondario, il risultato non è deterministico
- Con i compilatori Microsoft, sono disponibili le funzioni wrapper
  - _beginthreadex(...)
  - _exitthreadex(...)
  - Definite in <process.h>
- Garantiscono che vengano create e rilasciate versioni locali al thread delle variabili globali della libreria standard

Identificare un thread

- HANDLE GetCurrentThread()
  - Restituisce una pseudo-handle del thread corrente utilizzabile solo al suo interno
  - Per essere passata ad altri richiede una duplicazione esplicita (DuplicateHandle)

Terminare un thread

- Un thread termina quando
  - La funzione associata al thread ritorna
  - Il thread invoca ExitThread al proprio interno
  - Un altro thread del processo invoca TerminateThread
- Un thread termina quando
  - Vengono invocate le funzioni ExitProcess(...) o TerminateProcess(...) specificando il processo cui il thread appartiene
  - Attenzione al codice di startup!

Terminare un thread

- In C++, è opportuno evitare di chiamare ExitThread e ExitProcess
  - I distruttori degli oggetti allocati nello stack del thread e quelli delle variabili globali non verrebbero eseguiti
- Fare in modo che un thread causi la terminazione ordinata di un altro thread può essere complesso

Terminare un thread

- 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
  - Occorre che sia dichiarata volatile e che l'accesso avvenga in modo sincronizzato

Attendere la terminazione di un thread

- Quando un thread termina, l'oggetto kernel corrispondente passa nello stato "segnalato"
  - Si può attendere tale condizione senza consumare CPU attraverso le funzioni di attesa

Funzioni di attesa

```
DWORD WINAPI WaitForSingleObject(
  HANDLE       hHandle,
  DWORD        dwMilliseconds
);
```

```
DWORD WINAPI WaitForMultipleObjects(
  DWORD            nCount,
  const HANDLE *lpHandles,
  BOOL             bWaitAll,
  DWORD            dwMilliseconds
);
```


Handle e oggetti kernel

- Le handle dei thread fanno riferimento ad oggetti posseduti dal S.O.
  - Contengono un contatore di utilizzo
  - Quando diventa zero, il kernel può distruggere l'oggetto
- Quando un thread non ha più bisogno di usare una handle, deve segnalarlo tramite CloseHandle()
  - Decrementa il contatore dell'oggetto e rende invalida la handle

Handle e oggetti kernel

- Nel caso dei thread, il contatore inizialmente vale 2
  - L'oggetto corrispondente è accessibile al thread creatore ed al thread creato
- Se non vengono rilasciati, gli oggetti kernel possono saturare la memoria di sistema
  - Problematico nel caso di processi che durano a lungo

Thread in Linux

- Linux offre un insieme di chiamate a sistema per la gestione dei thread
  - Interfaccia a basso livello, complessa da utilizzare
  - Usate dalla libreria PThread per offrire un accesso più semplice

La libreria PThread

- Libreria per la gestione di thread per i sistemi UNIX
  - Specificata dallo standard IEEE POSIX 1OO3.1c
  - Definita nel file pthread.h
- Offre un'interfaccia basata sul linguaggio C
  - Strutture dati e funzioni per la gestione del ciclo di vita di un thread
  - Per la compilazione ed il collegamento, usare il flag "–pthread"

Tipi e funzioni

- pthread_t
  - Identifica in modo univoco un thread in un processo
- pthread_t pthread_self()
  - Identifica il thread corrente
- int pthread_equal(t1,t2)
  - Confronta due identificatori

Creazione di thread

```
int pthread_create(
  pthread_t*      tidp,
  pthread_attr_t*  attr,
  void*           (* function)(void*),
  void*           arg
);

// restituisce 0 in caso di successo
```


Attendere la terminazione

```
int pthread_join(
  pthread_t       tid,
  void**          retvalp
);

// restituisce 0 in caso di successo
// retvalp punta ad un blocco in cui
// viene salvato il valore ritornato
```


Spunti di riflessione

- Si scriva un programma Linux che crei un thread secondario e che ne attenda la terminazione
- Si riimplementi il programma in Windows