Esercitazione n.1
24 Ottobre 2013

Obiettivi:

- Gestione dei thread mediante libreria **`pthreads`**:
  - creazione: pthread_create
  - terminazione: pthread_exit
  - join: pthread_join

Richiami sui thread

Processi

- **Immagine di un processo** (codice, variabili locali e globali, stack, descrittore)
- **Risorse possedute:** (file aperti, processi figli, dispositivi di I/O..),
- L' immagine di un processo e le risorse da esso possedute costituiscono il suo **spazio di indirizzamento.**
- L'allocazione dello spazio di indirizzamento dipende dalla tecnica di **gestione della memoria** adottata. Potrà essere contenuto in tutto o solo in parte nella memoria principale (registri base ed indice, impaginazione, segmentazione).

Processi (*pesanti*)

- **Proprietà dei *processi***: *spazi di indirizzamento distinti*, cioe` non condividono memoria (es.**Unix**)
- **Complessità delle operazioni di *cambio di contesto*** tra due processi: comportano il salvataggio ed il ripristino dello spazio di indirizzamento **(overhead)**. Analogamente per le operazioni di creazione e terminazione di un processo.
- L'interazione tra processi fa riferimento al modello a **scambio di messaggi** (es.Unix)

Processi & thread

Il concetto di processo e` basato su **due aspetti indipendenti:**

- **Possesso delle risorse** contenute nel suo spazio di indirizzamento.
- **Esecuzione**. Flusso di esecuzione, associato a un programma, che puo` condividere la CPU con altri flussi, possiede uno stato e viene messo in esecuzione sulla base della politica di scheduling.

- I due aspetti sono indipendenti e possono essere gestiti separatamente dal S.O.:
  - processo **leggero** (***thread***): elemento cui viene assegnata la CPU
  - processo **pesante** (***processo o task***): elemento che possiede le risorse

Un *thread* rappresenta un ***flusso di esecuzione*** all' interno di un *processo pesante*.

- **Multithreading**: molteplicità di flussi di esecuzione all' interno di un processo pesante.

- Tutti i thread definiti in un processo ***condividono*** le risorse del processo, risiedono nello **stesso spazio di indirizzamento** ed hanno **accesso a dati comuni**.

Ogni thread ha:

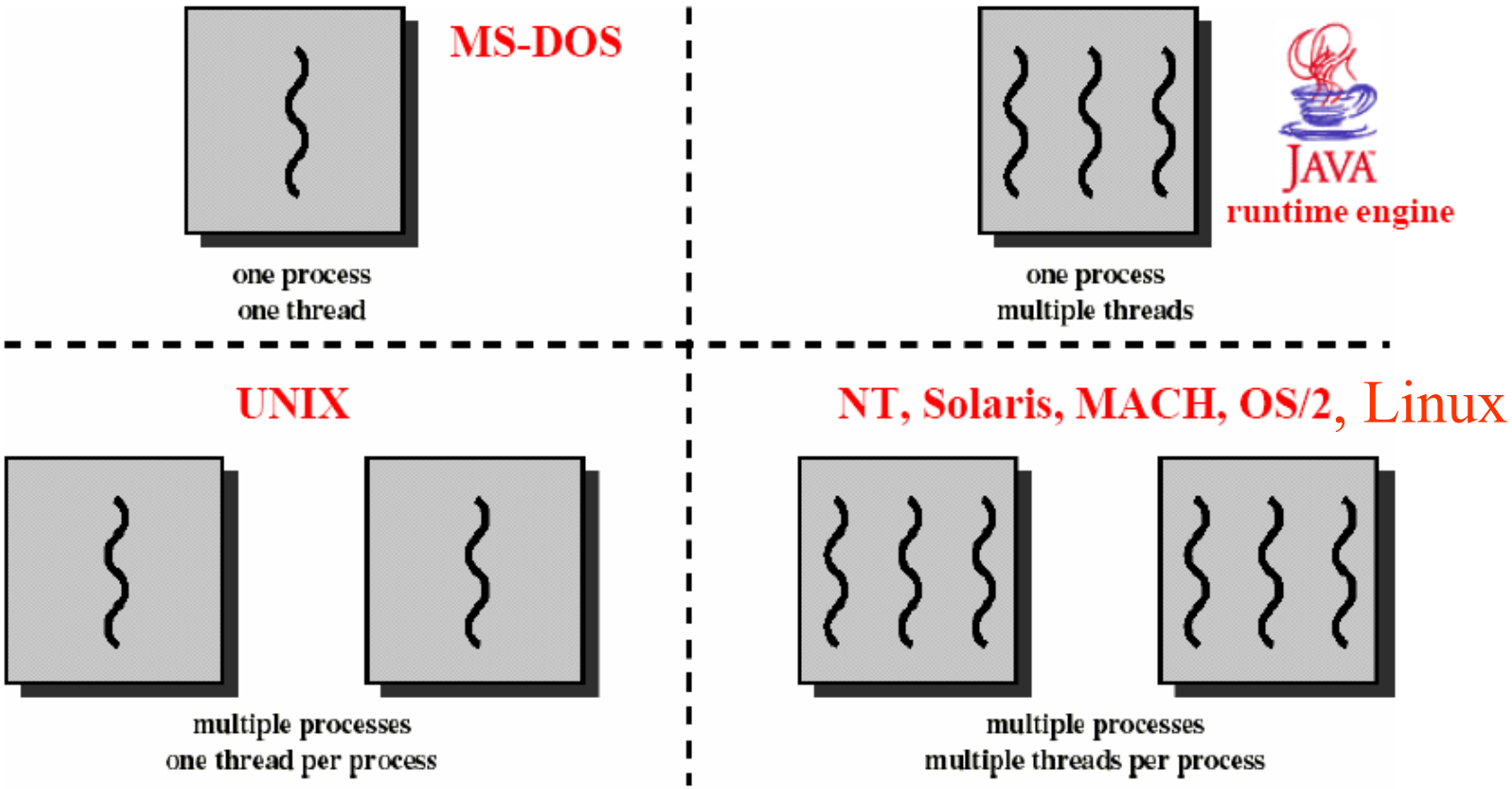
- uno **stato** di esecuzione (running, ready, blocked)
- un **contesto** che è salvato quando il thread non è in esecuzione
- uno **stack** di esecuzione
- uno spazio di memoria privato per le **variabili locali**
- accesso alla memoria e alle risorse del task **condiviso** con gli altri thread.

Vantaggi

- maggiore efficienza: le operazioni di context switch, creazione etc., sono più economiche rispetto ai processi.
- Modello adeguato per architetture multiprocessore.

Soluzioni adottate

- **MS-DOS:** un solo processo utente ed un solo thread.
- **UNIX**: più processi utente ciascuno con un solo thread.
- **Supporto run time di Java (JVM)**: un solo processo, più thread.
- **Linux, Windows NT, Solaris**: più processi utente ciascuno con più thread.


MS-DOS
one process
one thread
JAVA
runtime engine
one process
multiple threads
UNIX
multiple processes
one thread per process
NT, Solaris, MACH, OS/2, Linux
multiple processes
multiple threads per process

Realizzazione dei thread

A livello utente (es. ***Java***)

- Libreria (***thread package***) che opera a livello utente e fornisce il supporto per la creazione, terminazione, sincronizzazione dei thread e per la scelta di quale thread mettere in esecuzione (*scheduling*).
- Il sistema operativo ***ignora la presenza dei thread*** continuando a gestire solo i processi.
- Ogni processo parte con un solo thread che può creare nuovi thread chiamando una apposita funzione di libreria.

- Gerarchia di thread o thread tutti allo stesso livello.
- La soluzione è *efficiente* (tempo di switch tra thread), *flessibile* (possibilità di modificare l'algoritmo di scheduling), *scalabile* (modifica semplice del numero di thread).
- Se un thread *si blocca* in seguito ad una **chiamata ad una *funzione del package*** (es. wait), va in esecuzione un altro thread dello stesso processo.
- I thread possono **chiamare delle *system call*** (es. I/O): intervento del sistema operativo che ***blocca il processo*** e conseguentemente l'esecuzione di *tutti i suoi thread*.
- Il S.O. interviene anche nel caso allo ***scadere del quanto di tempo*** assegnato ad un processo (sistemi time sharing).
- Non è possibile sfruttare il parallelismo proprio di *architetture multiprocessore*: un processo (con tutti i suoi thread) è assegnato ad uno dei processori.

Realizzazione di thread

A livello di nucleo **(es.** *Linux***):**

- Il S.O. si fa carico di tutte le **funzioni per la gestione dei thread**. Mantiene tutti i descrittori dei thread (oltre a quelli dei processi).
- A ciascuna funzione corrisponde una ***system call***.
- Quando un thread si *blocca* il S.O. può mettere in esecuzione un altro thread dello *stesso processo*.
- Soluzione ***meno efficiente*** della precedente.
- Possibilità di eseguire thread diversi appartenenti allo stesso processo su unità di elaborazione differenti (*architettura multiprocessore).*

Realizzazione di thread

Soluzione mista (es. Solaris):

- Creazione di thread, politiche di assegnazione della CPU e sincronizzazione a *livello utente.*
- I thread a livello utente sono mappati in un numero (minore o uguale) di thread a livello nucleo.

Vantaggi:

- Thread della stessa applicazione possono essere eseguiti in parallelo su processori diversi.
- Una chiamata di sistema bloccante non blocca necessariamente lo stesso processo

L'implementazione della libreria pthread nel sistema operativo LINUX: *Linuxthreads*

Uso dei thread in Linux: system call native vs. pthread

- Processi leggeri **realizzati a livello kernel**
- System call **clone**:

```
int  clone(int (*fn) (void *arg), void *child_stack, int
      flags, void *arg)
```

➡ **E` specifica di Linux: scarsa portabilita`!**

- Libreria pthread (*LinuxThreads*): funzioni di gestione dei threads, in conformita` con lo standard POSIX 1003.1c (*pthreads*):
  - *Creazione/terminazione threads*
  - *Sincronizzazione threads:lock, [semafori], variabili condizione*
  - *Etc.*

➢ Portabilita`

LinuxThreads

Caratteristiche thread:

- Il thread e` realizzato **a livello kernel** (e` l'unita` di schedulazione)
- l' esecuzione di un programma determina la creazione di un thread iniziale che esegue il codice del main.

- Il thread iniziale può creare altri thread, ognuno dedicato all'esecuzione di una funzione data; ogni thread ne può creare altri: si crea una gerarchia di thread che condividono uno spazio di indirizzi.

- I thread vengono creati all'interno di un processo per eseguire una funzione
- ogni thread ha il suo PID (distinzione tra *task* e *threads*)

- Gestione dei segnali non conforme rispetto a POSIX (Linuxthread):
  - Non c'e` la possibilita` di inviare un segnale a un task.
  - SIGUSR1 e SIGUSR2 vengono usati per l'implementazione dei threads e quindi non sono piu` disponibili.

Gestione pthreads: sintesi

Inclusione header file: **<pthread.h>**

Rappresentazione threads: **pthread_t tid;**

Creazione di thread:

```
int pthread_create(pthread_t *thread,pthread_attr_t *attr,
                   void *(*start_routine)(void *), void * arg);
```

dove:

- **thread**: e` il puntatore alla variabile che raccogliera` il thread_ID (PID)
- **start_routine**: e` il puntatore alla funzione che contiene il codice del nuovo thread
- **arg**: e` il puntatore all'eventuale vettore contenente i parametri della funzione da eseguire
- **attr**: puo` essere usato per specificare eventuali attributi da associare al thread (di solito: NULL)

Restituisce : 0 in caso di successo, altrimenti un codice di errore (!=0)

Gestione pthreads: sintesi

Terminazione di thread:

```
void pthread_exit(void *retval);
```

Dove **retval** e` il puntatore alla variabile che contiene il valore di ritorno (puo` essere raccolto da altri threads, v. **pthread_join**).

Gestione pthreads: sintesi

Sincronizzazione con la terminazione di un altro thread con:

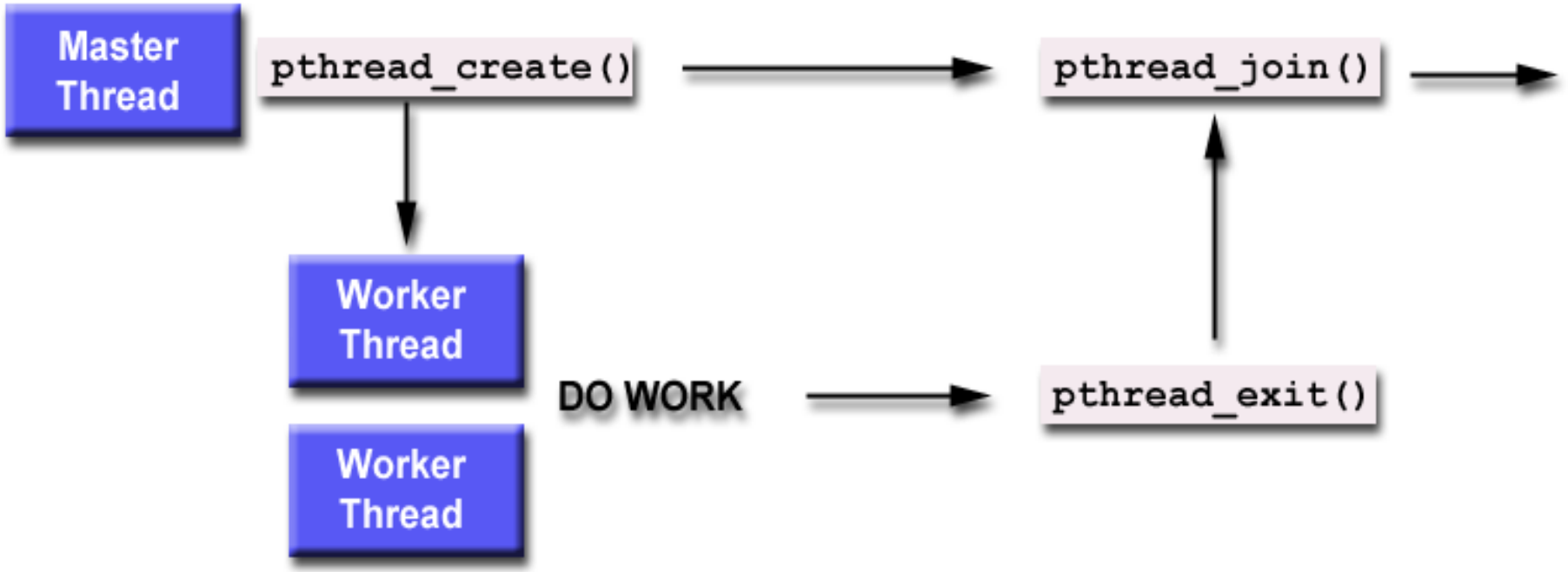
```
int pthread_join(pthread_t th, void **thread_return);
```

dove:

- **th: e` il pid del particolare thread da attendere**
- **thread_return: se thread_return non è NULL, in *thread_return viene memorizzato il valore di ritorno del thread (v. pthread_exit)**

Il valore restituto dalla pthread_join indica l'esito della chiamata: se diverso da zero significa che la pthread_join e` fallita (ad es. non vi sono thread figli).

Modello master/workers



Esempio: 4 workers

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS 4
void *Calcolo(void *t) // codice worker
{  int i;
   long tid, result=0;
   tid = (int)t;
   printf("Thread %ld è partito...\n",tid);
   for (i=0; i<100; i++)
      result = result + tid;
  printf("Thread %ld ha finito. Ris= %ld\n",tid, result);
   pthread_exit((void*) result);
}
```

```
int main (int argc, char *argv[])
{  pthread_t thread[NUM_THREADS];
   int rc;
    long t;
    void *status;

    for(t=0; t<NUM_THREADS; t++) {
       printf("Main: creazione thread %ld\n", t);
       rc = pthread_create(&thread[t], &attr, Calcolo, (void *)t);
       if (rc) {
          printf("ERRORE: %d\n", rc);
          exit(-1);   }
  }
   for(t=0; t<NUM_THREADS; t++) {
       rc = pthread_join(thread[t], &status);
       if (rc)
           printf("ERRORE join thread &d codice %d\n", t, rc);
        else
       printf("Finito thread %ld con ris. %ld\n",t,(long)status);
  }
}
```

Compilazione

Per compilare un programma che usa i linuxthreads:

```
gcc -D_REENTRANT -o prog prog.c  -lpthread
```

Esercizio 1 - calcolo "parallelo" del massimo di un insieme di interi

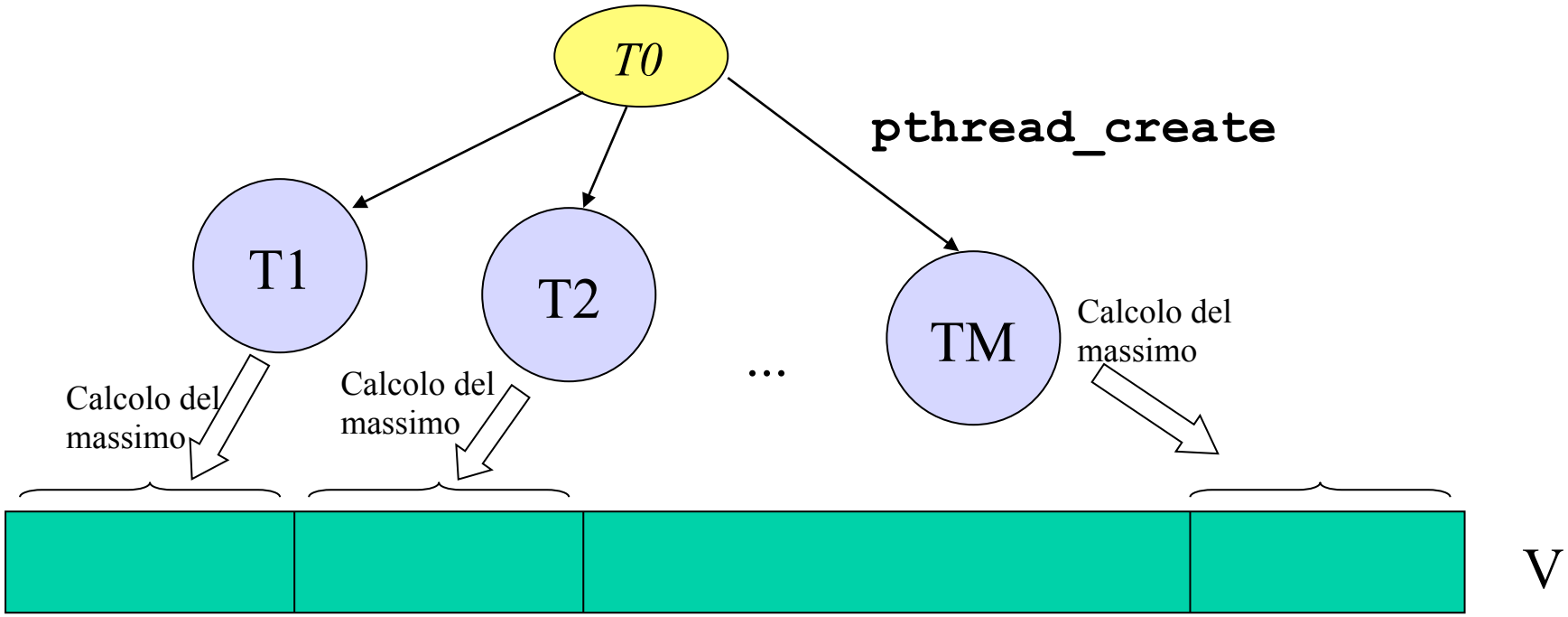
Si **calcoli il massimo in un insieme** di valori interi di N elementi, memorizzati in un vettore V.

Si vuole affidare la ricerca del massimo a un insieme di **M thread concorrenti**, ognuno dei quali si dovrà occupare della ricerca del massimo in una porzione del vettore di dimensione K data (M=N/K).

Il thread iniziale dovrà quindi:

- Inizializzare il vettore V con N valori casuali;
- Creare gli M thread concorrenti;
- Ricercare il massimo tra gli M risultati ottenuti dai thread e stamparne il valore.

Impostazione



Esercizio 2 – Prodotto scalare e Mutua esclusione

Si vuole affidare il calcolo del **prodotto scalare**(*) di due vettori A e B di dimensione **N** ad un insieme **M thread concorrenti** (si supponga che N sia multiplo intero di M, cioè N=KM).

Ogni thread T_i si dovrà occupare del calcolo del prodotto scalare tra due sottovettori di dimensione K data (M=N/K).

In altre parole, Ti dovrà calcolare il valore

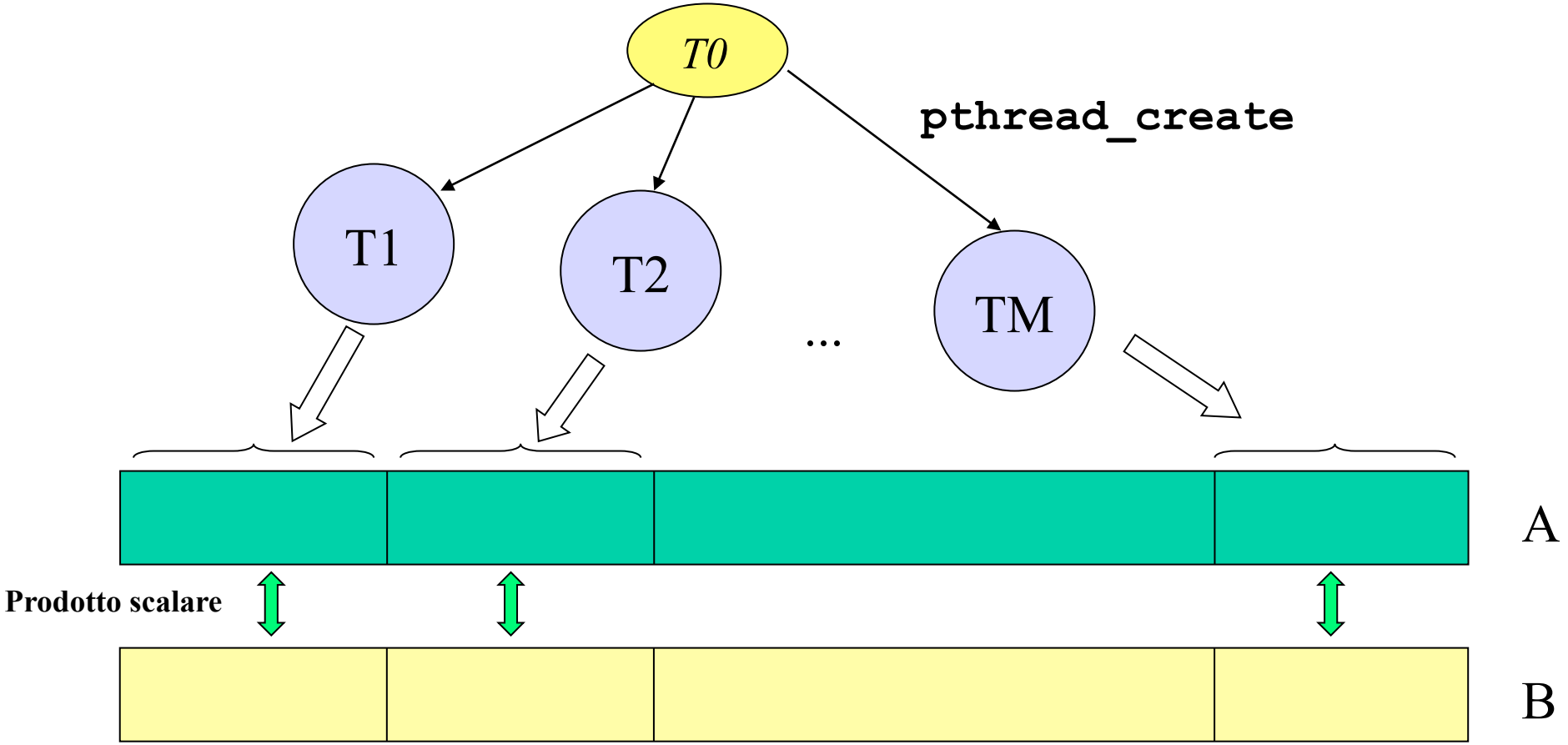
$$a_i.b_i+a_{i+1}.b_{i+1}+ \dots +a_{i+k-1}.b_{i+k-1}$$

Al termine del calcolo, ogni thread T_i dovrà aggiungere il valore calcolato alla variabile condivisa RIS[nella quale viene quindi accumulato il risultato finale A.B]; inoltre, per registrare l'ordine di aggiornamento della variabile, T_i dovrà scrivere il proprio identificatore nella prima posizione libera di un vettore LOG di M elementi.

(*) Ricordiamo che $A.B= a_1.b_1+a_2.b_2+\dots a_n.b_n$

Il thread iniziale dovrà quindi:

- Inizializzare i vettori A e B con N valori casuali;
- Creare gli M thread concorrenti;
- Attendere la terminazione dei thead e stampare il valore di RIS e il contenuto di LOG.



Problema della mutua esclusione nell'accesso a RIS e LOG

Il problema richiede che le operazioni finali di ogni thread:

<aggiornamento di RIS>
<scrittura di LOG>

vengano realizzati come **sezioni critiche**.

Gli strumenti visti finora non ci consentono di esplicitare vincoli di precedenza tra thread (sincronizzazione).

Soluzione

Applicare un algoritmo di mutua esclusione.
Per esempio: algoritmo di Peterson.

Algoritmo di Peterson (caso di 2 thread)

```
int busy1=0;
int busy2=0;
int turno=1; /*dominio {1,2}*/
```

```
/* thread 1: */
main()
{ ...
 busy1=1;
 turno=2;
 while(busy2 && turno==2);
 <sezione critica 1>;
 busy1=0;
    ...
}
```

```
/* thread 2: */
main()
{ ...
 busy2=1;
 turno=1;
 while(busy1 && turno==1);
 <sezione critica 2>;
 busy2=0;
    ...
}
```