Processi in competizione o processi cooperanti?

L'introduzione dell'astrazione "***processi***" è stata inizialmente motivata dalla necessità di utilizzare al meglio le risorse disponibili, in particolare la CPU. Da questo punto di vista ***i processi contemporaneamente attivi sono tra loro in competizione*** per l'accesso a un insieme di risorse condivise. In questo caso il Sistema Operativo deve assicurare la ***non interferenza*** tra processi.

Tuttavia un insieme di processi puo' anche ***cooperare*** per raggiungere un comune obiettivo: si possono quindi progettare applicazioni come un insieme di processi cooperanti, che possono condividere risorse: programma, dati, file aperti. Il sistema operativo dovra' fornire dei meccanismi per permettere ai processi cooperanti di ***comunicare*** e ***sincronizzarsi.***

1

Il modello a *thread* : motivazioni

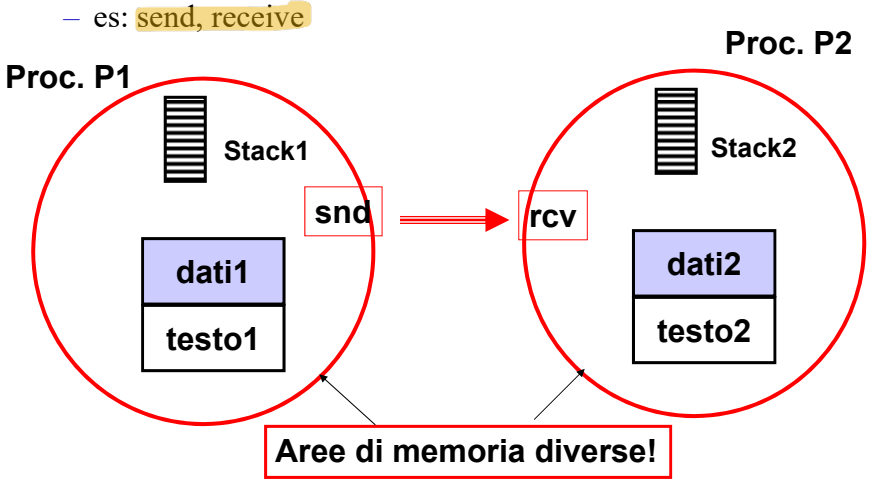
Necessità di meccanismi di comunicazione efficienti:

- Nel modello a processi, ogni processo ha il suo spazio di indirizzamento privato ed il modo per interagire è quello di utilizzare i meccanismi di IPC messi a disposizione dal sistema
- Questo implica alti costi di interazione, se i dati da scambiare sono molti!

2

Scambio dati fra due processi

- Ci sono molti meccanismi di IPC *Inter Processor Communication*)
  - es: send, receive



3

Il modello a *thread* : motivazioni (2)

Necessità di meccanismi di attivazione e cambio contesto efficienti:

- Nel modello a processi, l'attivazione di un processo, il cambio di contesto sono operazioni molto complesse che richiedono ingenti quantità di tempo per essere portate a termine
- Tuttavia a volte l'attività richiesta ha vita relativamente breve rispetto a questi tempi
  - es : invio di una pagina html da parte di un server Web è troppo 'leggera' per motivare un nuovo processo

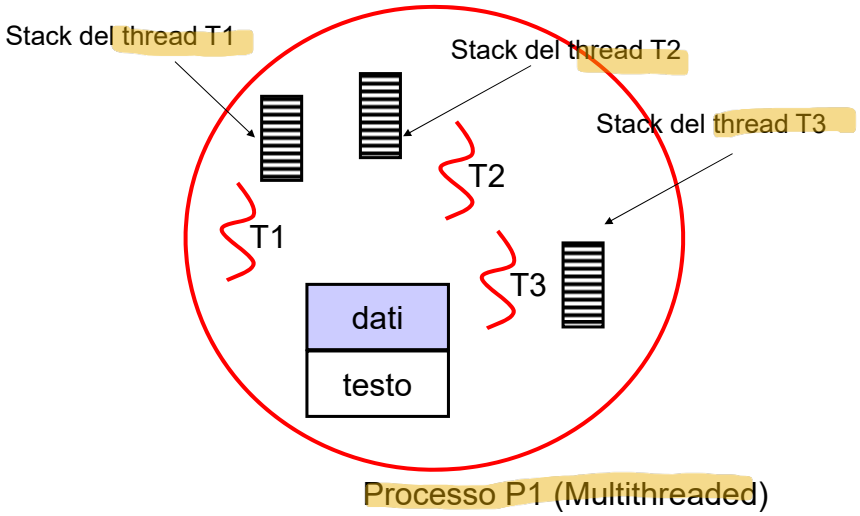
4

Il modello a *thread*

- Idee di base dietro il modello a thread :
  - permettere la definizione di attività 'leggere' (**lightweight processes**) con costo di attivazione terminazione limitato
  - possibilità di condividere lo stesso spazio di indirizzamento
- Ogni processo racchiude più flussi di controllo (thread) che condividono le aree testo e dati

5

Struttura di un processo mutithreaded



Processo P1 (Multithreaded)

6

Eseguono le stesse istruzioni ma in punti diversi.
Area dati diventa spazio di memoria condiviso, che però comporta dei rischi.

Il modello a *thread* (2)

- Se un processo P1 ammette un singolo thread di controllo

  ⇒ lo stato di avanzamento della computazione di P1 è determinato univocamente da :

  - valore del PC (prossima istruzione da eseguire)
  - valore di SP/PSW e dei registri generali
  - contenuto dello Stack (ovvero storia delle chiamate di funzione passate)
  - stato del processo : *pronto, in esecuzione, bloccato*
  - stato dell'area testo e dati
  - stato dei file aperti e delle strutture di IPC utilizzate

7

Informazioni nel process control block

Per passare da un thread all'altro ho comunque cambio contesto

Il modello a *thread* (3)

- Se un processo P1 ha più thread di controllo

  ⇒ lo stato di avanzamento della computazione di ogni thread è dato da :

  - valore del PC (prossima istruzione da eseguire)
  - valore di SP/PSW e dei registri generali
  - contenuto dello Stack privato di quel thread
  - stato del thread : *pronto, in esecuzione, bloccato*

- Sono invece comuni a tutti i thread :
  - stato dell'area testo e dati
  - stato dei file aperti e delle strutture di IPC utilizzate

} Puntatori alle aree dati e resto

8

Uso dei thread

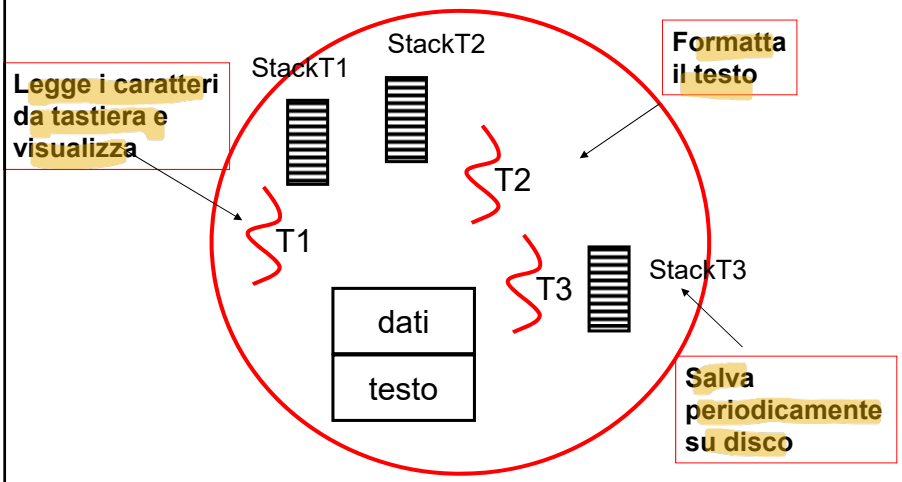
Applicazioni che :

- possono essere suddivise in più flussi di controllo
- interagiscono molto strettamente

la condivisione dello spazio di indirizzamento e delle altre risorse permette di interagire senza essere costretti a copiare dati e fare pesanti cambi di contesto

9

Struttura di un word processor multithreaded



10

Implementazione dei thread

- Ogni thread è descritto da un descrittore di thread :
  - thread identifier (`tid`)
  - PC, SP, PCW, registri generali
  - info sulla memoria occupata dallo stack privato del thread
  - stato del thread (pronto, in esecuzione, bloccato)
  - processo di appartenenza (`pid`, process identifier)
- Non vi è protezione tra i thread: un thread può modificare i dati usati da un altro thread

11

Implementazione dei thread (2)

- Thread table (TT) :
  - tabella che contiene i descrittori di thread
  - simile alla process table
  - se ne può avere una unica nel kernel o una privata di ogni processo
- Possono essere realizzati:
  - da librerie che girano interamente in stato utente (*user level thread*)
  - all'interno del kernel (*kernel level thread*)

12

User–level thread (1)

- Realizzati da una librerie di normali funzioni che girano in modo utente
  - thread_create(), thread _exit(), thread_wait()…
- Il SO e lo scheduler non conoscono l'esistenza dei thread e gestiscono solamente il processo intero
- Lo scheduling dei thread viene effettuato dal run time support della libreria

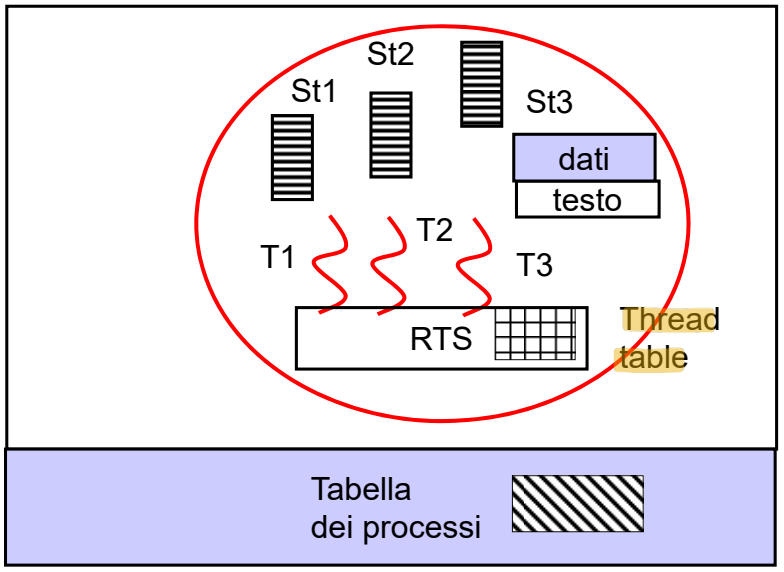
13

User–level thread (2)

- La thread table è una struttura privata del processo
- C'è una TT per ogni processo
- I thread devono rilasciare esplicitamente la CPU per permettere allo scheduler dei thread di eseguire un altro thread (ad es.thread _yield ()). All'interno di un singolo processo non ci sono interruzioni di clock

14

User–level thread (3)



15

User–level thread (problema)

- Quando un thread esegue un chiamata di sistema e si blocca in attesa di un servizio tutto il processo a cui appartiene viene bloccato
  - es. nel web server una qualsiasi lettura da disco blocca tutti i thread!

Tutto il processo è in stato waiting anche se ci sono thread ready. Poi, non avendo il supporto dell'interrupt hardware alla preparazione devo aspettare che thread lascia volontariamente la CPU.

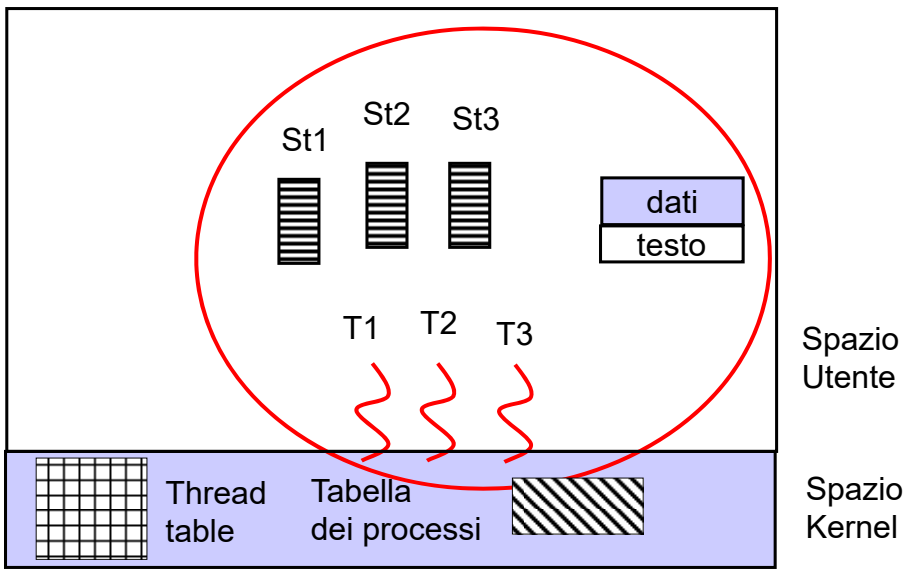
16

Kernel–*level* thread (1)

- Thread table unica (nel kernel)
- Le primitive che lavorano sui thread sono system call
  - thread_create(), thread _exit(), thread_wait()...
- Non è necessario che un thread rilasci esplicitamente la CPU
- Le system call possono bloccarsi senza bloccare tutti i thread di quel processo

17

Kernel–level thread (3)



18

User–level thread vs kernel-level thread

- Creazione di thread e thread switch molto veloce :non c'è il passaggio a kernel mode
- Si può effettuare uno scheduling "personalizzato", dipendente dall'applicazione
- Eseguibili su un SO che supporta solo i processi
- Potrebbe non essere banale gestire i thread in time sharing
- Gestione problematica delle system call bloccanti
  - librerie di SC non bloccanti

19

Modelli ibridi

- Una terza possibilità consiste nel definire separatamente il numero N di thread a livello utente associati ad un processo e il numero M (≤ N) di thread a livello kernel associati allo stesso processo.
- In questo caso occorre precisare il mapping:

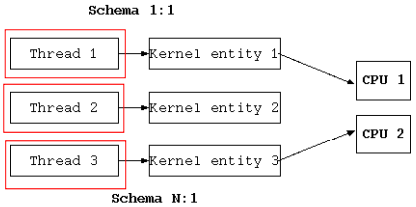
  k thread utente ---> thread del kernel

  Il sistema operativo si occupa di schedulare i thread del kernel, mentre il sistema run-time decide quale fra i k thread utente associati ad un dato thread del kernel far eseguire.

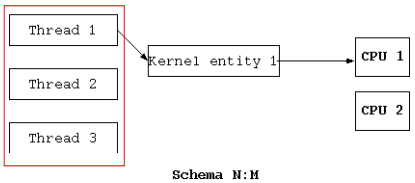
20

Thread a livello utente o a livello kernel

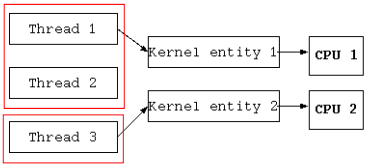
Thread a livello kernel (schema 1:1)



Thread a livello utente (schema N : 1)



Approccio misto (schema N : M)



21

Scheduling dei Thread (user level)

- Lo scheduling dei thread user level
  - il SO non conosce l'esistenza dei thread, quindi schedula i processi
  - durante l'esecuzione di un processo lo schedulatore della libreria dei thread decide quale thread mandare in esecuzione
  - le interruzioni del clock non sono visibili allo schedulatore di livello utente
  - lo schedulatore può intervenire solo se invocato esplicitamente (es. thread_yield)
  - non c'è prerilascio (all'interno di un singolo processo)

22

Scheduling dei Thread (kernel level)

- Lo scheduling dei thread kernel level
  - il SO schedula i thread (non i processi)
  - quando un thread si blocca il SO può decidere di mandare in esecuzione un altro thread di quel processo o un thread di un processo diverso
    - può scegliere se pagare il cambio di contesto o no
  - le interruzioni del clock permettono allo schedulatore di tornare in esecuzione alla fine del quanto di tempo
    - i quanti di tempo sono assegnati direttamente ai thread
    - si può effettuare prerilascio

23

Problemi di sincronizzazione fra thread dovuti alla condivisione di memoria

```
VersaSulConto(int numconto,int versamento)

{ Saldo = CC[numconto];

  Saldo = Saldo + versamento;

  CC[numconto] = Saldo; }
```

Supponiamo che due thread eseguano contemporaneamente la procedura VersaSulConto.

I thread condividono il vettore CC[] che contiene il saldo di tutti i conti correnti. La variabile Saldo è locale alla procedura, quindi diversa nei due thread (ognuno ha il suo stack dove si trovano le variabili locali). Invece CC[] è condiviso.

24

Bisogna realizzare atomicità per la regione critica in modo da non doversi preoccupare dei problemi di schedulazione

Problemi di sincronizzazione fra thread dovuti alla condivisione di memoria

CC[1200]

2.000

P1	P2
1. Saldo = CC[1200];	2. Saldo = CC[1200];
P1: Saldo = 2.000	*P2: Saldo = 2.000*
4. Saldo = Saldo + 200;	3. Saldo = Saldo + 350;
P1: Saldo = 2.200	*P2: Saldo = 2.350*
5. CC[1200] = Saldo;	
P1: CC[1200] = 2.200	

2.200

6. CC[1200] = Saldo;

P2: CC[1200] = 2.350

2.350

ERRORE !!!!!

25