

I thread

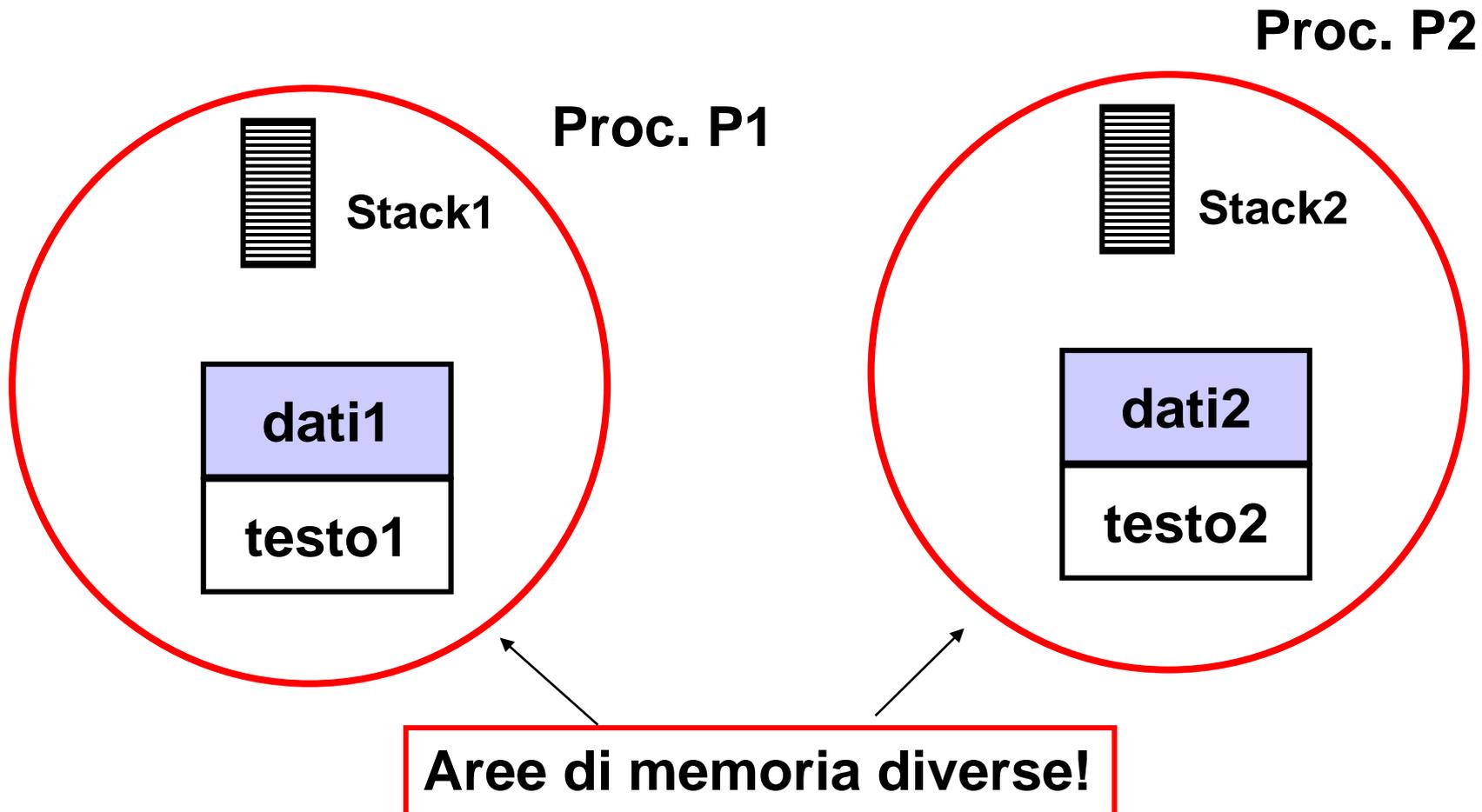
Non bastavano i processi?

Il modello a *thread* : motivazioni

- Nel modello a processi,
 - ogni processo ha il suo spazio di indirizzamento,
 - vediamo cosa significa ...

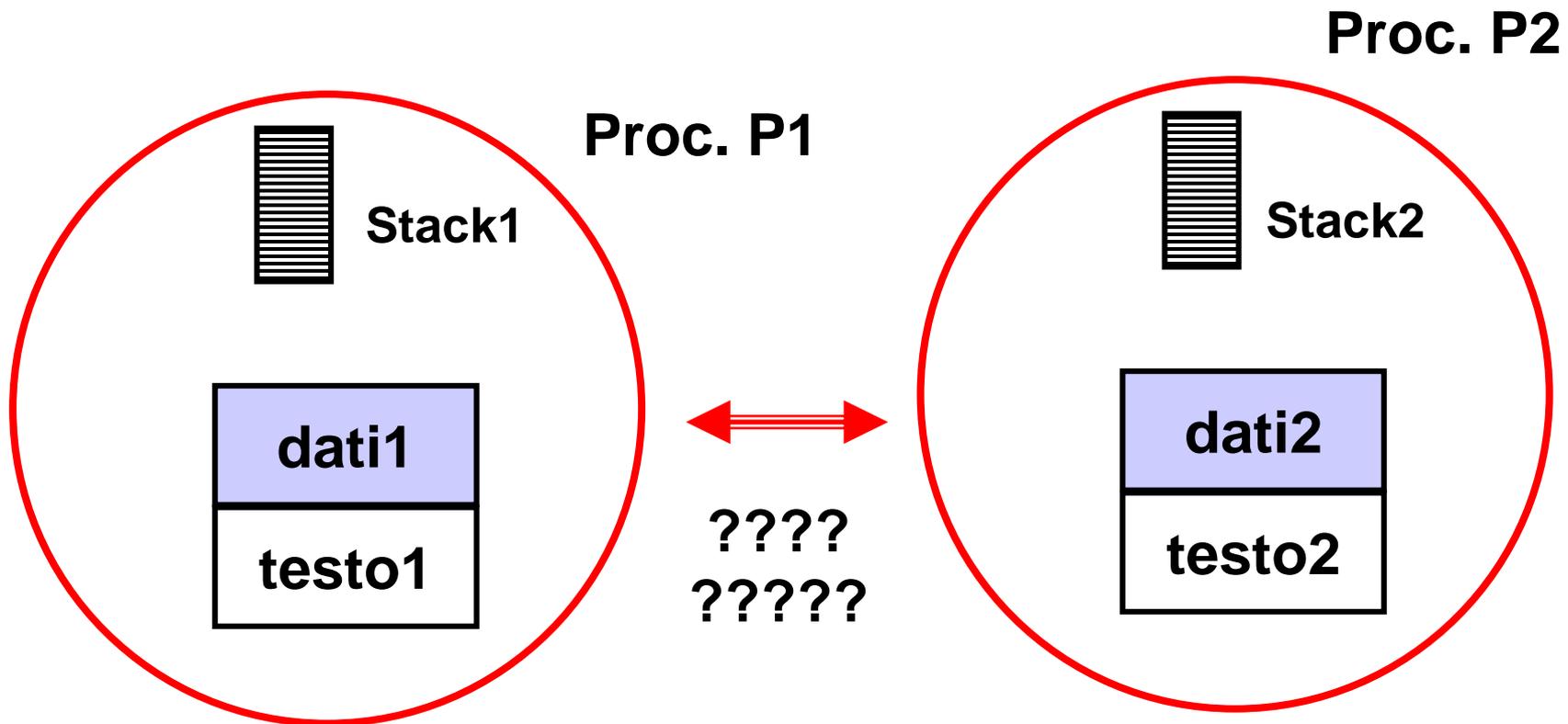
Spazi di indirizzamento distinti

- Due processi diversi non hanno spazio di indirizzamento comune!!!



Condivisione dati fra due processi

- Come posso condividere dati fra P1 e P2 ?



Condivisione dati fra due processi

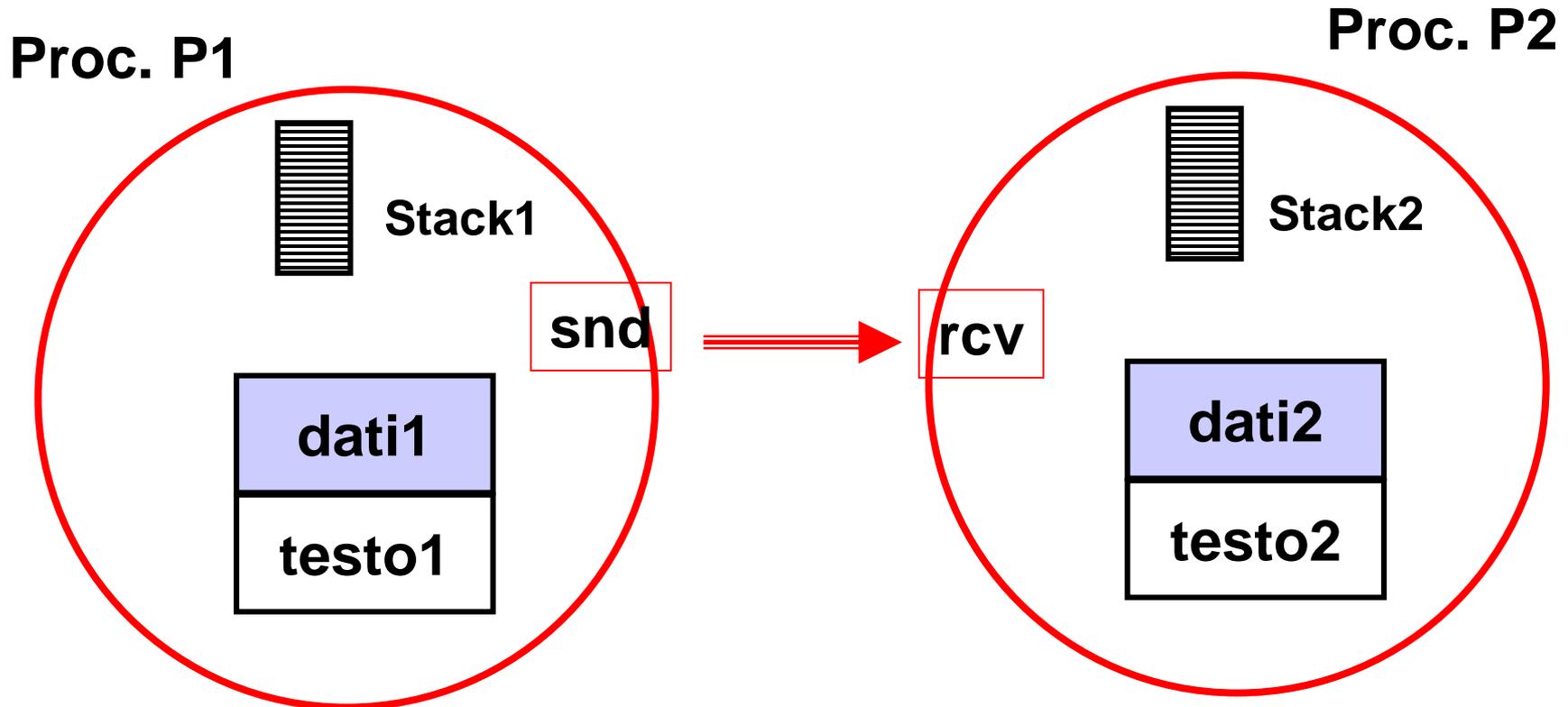
- Possiamo passare una copia dei dati:
 - pipe, socket (si macchine diverse)
 - costoso
 - molte system call, overhead passaggi utente-kernel
 - copia dati (possibilmente di grosse dimensioni)
- Possiamo condividere esplicitamente parti dello spazio di indirizzamento
 - mmap, sulla stessa macchina
 - complicato da programmare
 - richiede diverse system call concertare sui vari processi

Condivisione dati fra due processi (2)

- In entrambi i casi:
 - poco efficace se i dati sono molti e le interazioni frequenti
 - per ogni flusso di controllo abbiamo un processo diverso che occupa un insieme di risorse del sistema
 - memoria, tabella dei processi, descrittori dei file etc.
 - la creazione di nuovi processi è costosa

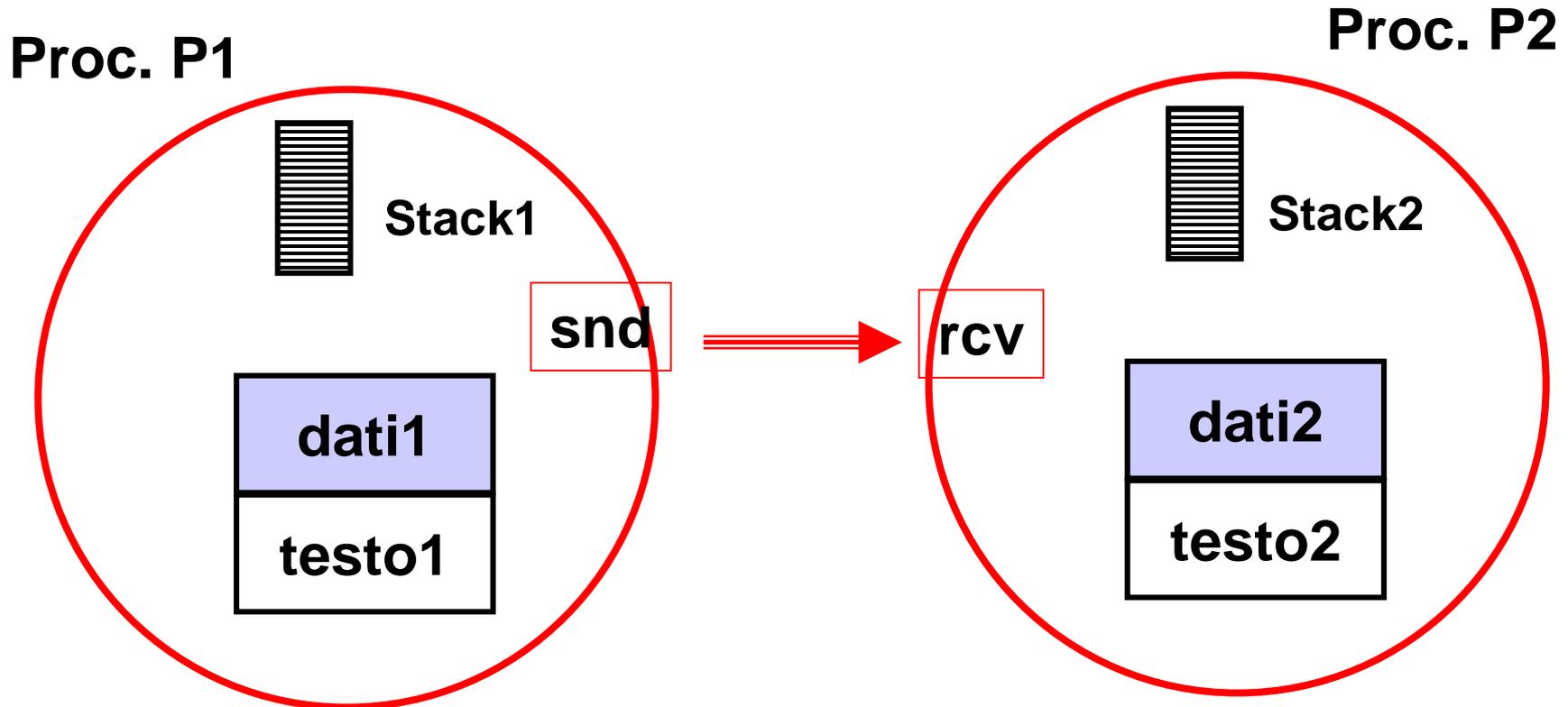
Esempio: send-receive

- Le interazioni avvengono con alcune primitive
 - es: send, receive



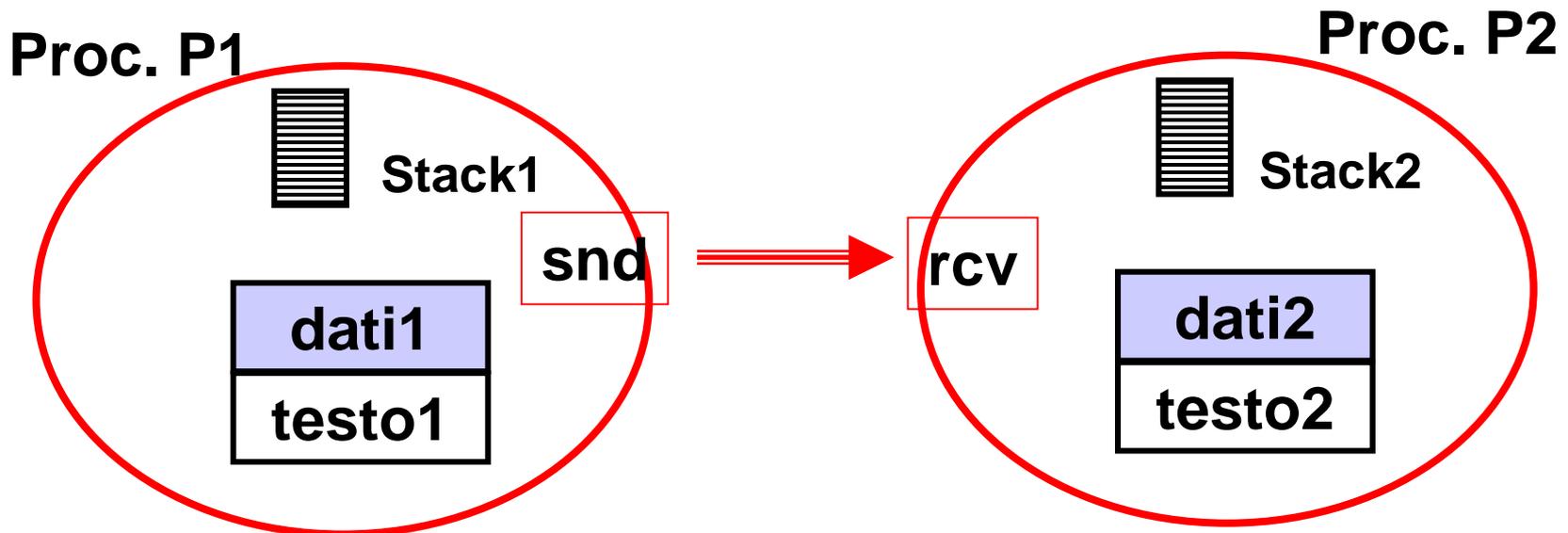
Esempio: send-receive (2)

- Una possibile implementazione :
 - Il canale è una struttura dati del *kernel*
 - *snd* e *rcv* sono due chiamate di sistema



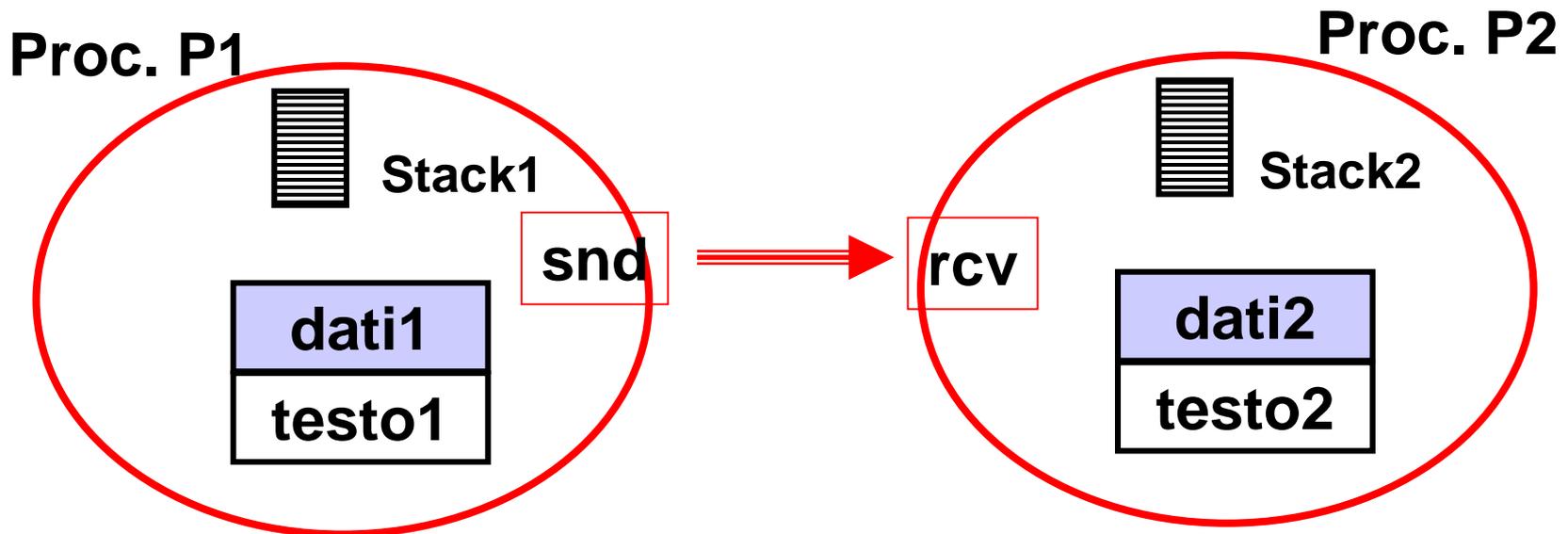
Esempio: send-receive (3)

- Costi :
 - Dati sono copiati almeno una volta dal processo mittente al processo destinatario
 - Si paga : il cambio di contesto, l'overhead delle *chiamate di sistema* per `snd` e `rcv`



Esempio: send-receive (4)

- Altra possibile implementazione :
 - Il canale è una pipe o un socket
 - *snd* e *rcv* corrispondono a chiamate di sistema che effettuano letture/scritture su un file
 - **il costo non migliora!!!!!!!!!!!!!!!**



Quindi : se i dati da scambiare sono molti e le interazioni frequenti avere spazi di indirizzamento separati è estremamente costoso !

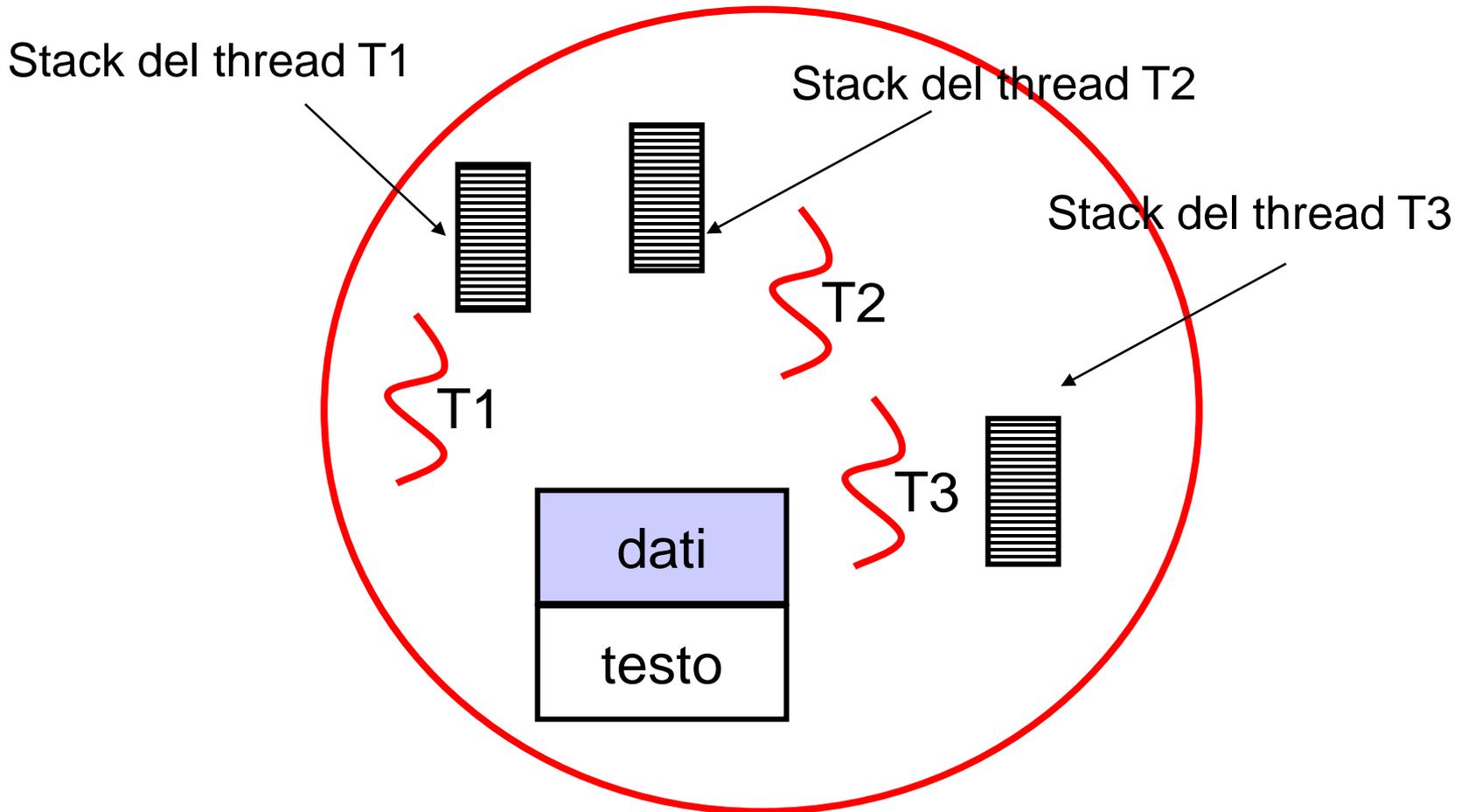
Il modello a *thread* : motivazioni (2)

- Inoltre:
 - ci sono tipologie di applicazioni che si prestano male al modello a processi!
- Creare e schedulare un processo costa!
 - il cambio di contesto ordine dei millisecondi
- 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
 - troppo 'leggera' per motivare un nuovo processo

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

Struttura di un processo multithreaded



Processo P1 (Multithreaded)

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

Il modello a *thread* (3)

- Se un processo P1 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

Il modello a *thread* (4)

- Quindi, lo stato di avanzamento di un processo multithreaded è dato da
 - lo stato di avanzamento di tutti i suoi thread
 - stato dell'area testo e dati
 - stato dei file aperti e delle strutture di IPC utilizzate

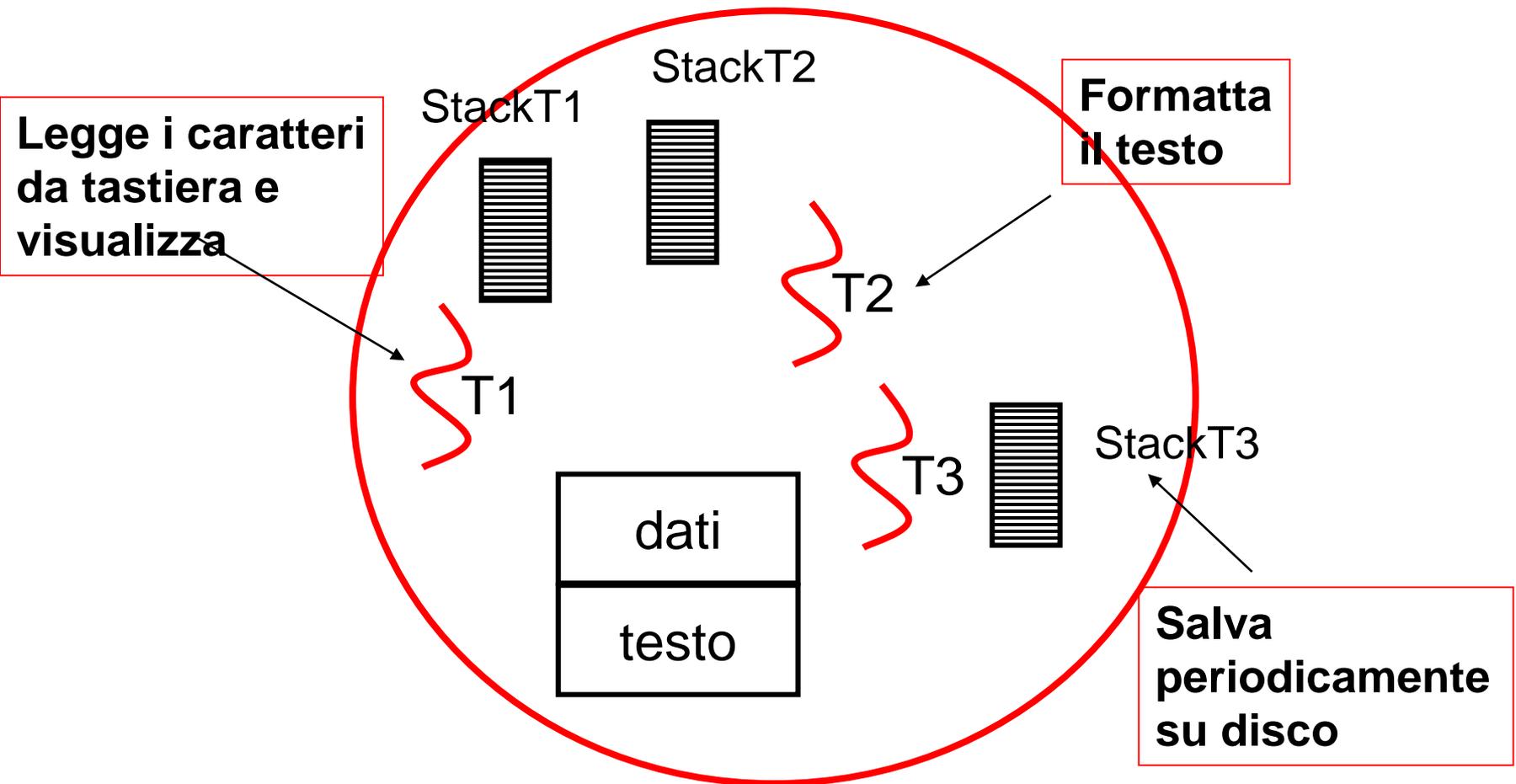
Uso dei thread (1)

Applicazioni che :

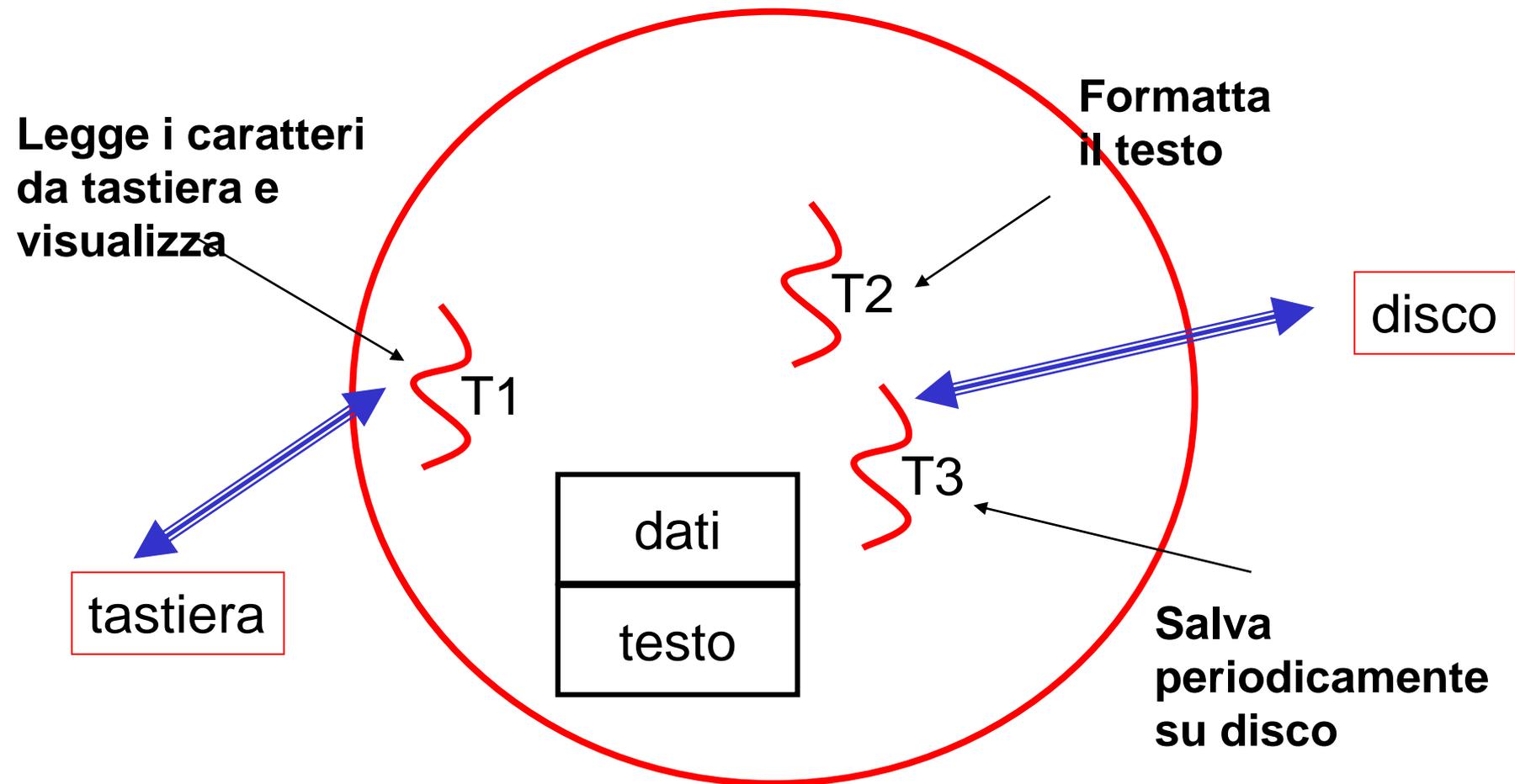
- possono essere suddivise in più flussi di controllo
- interagiscono molto strettamente

la condivisione dello spazio di indirizzamento e delle altre risorse permette di interagire senza pagare copie e cambi di contesto

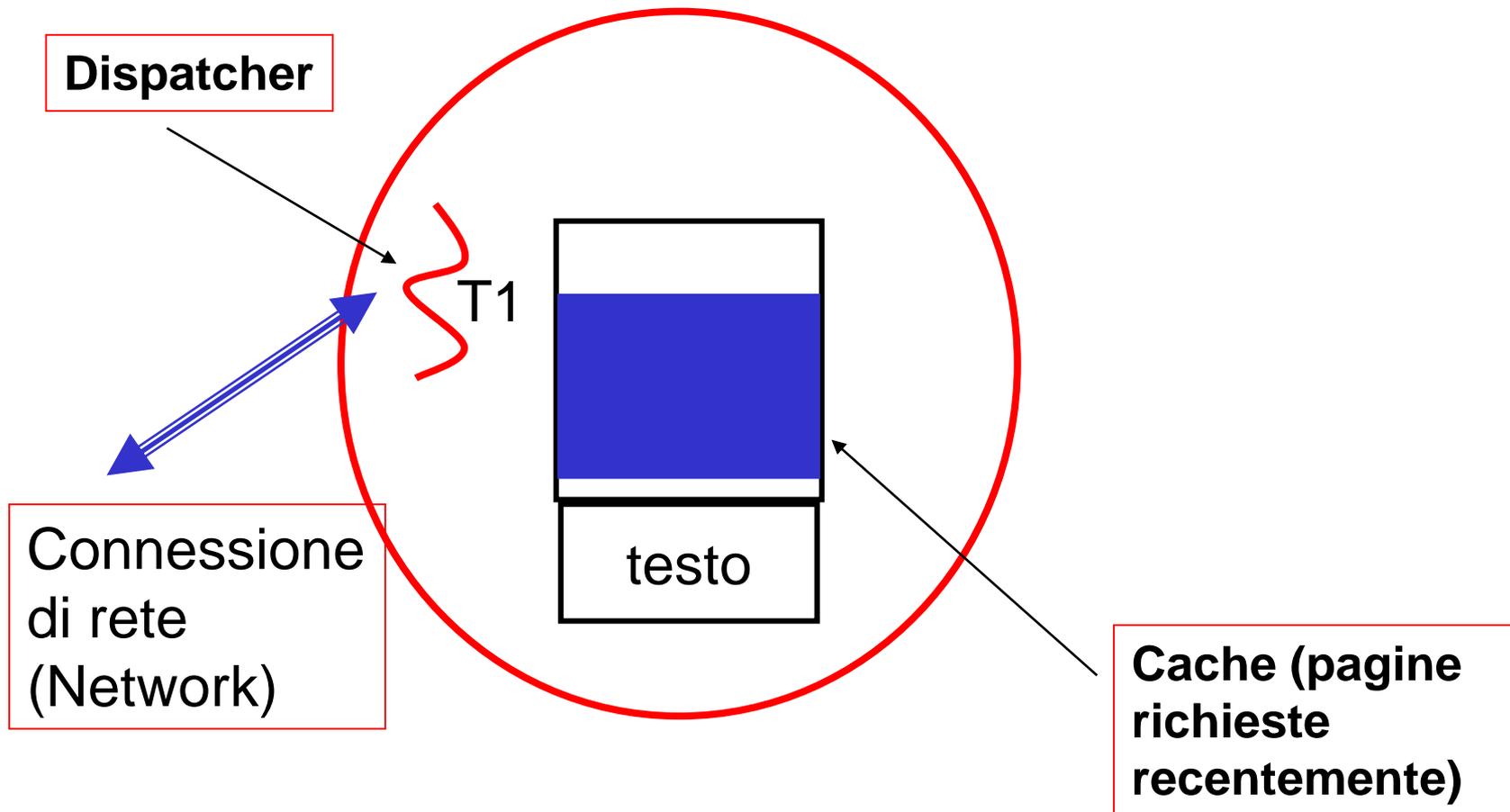
Struttura di un word processor multithreaded



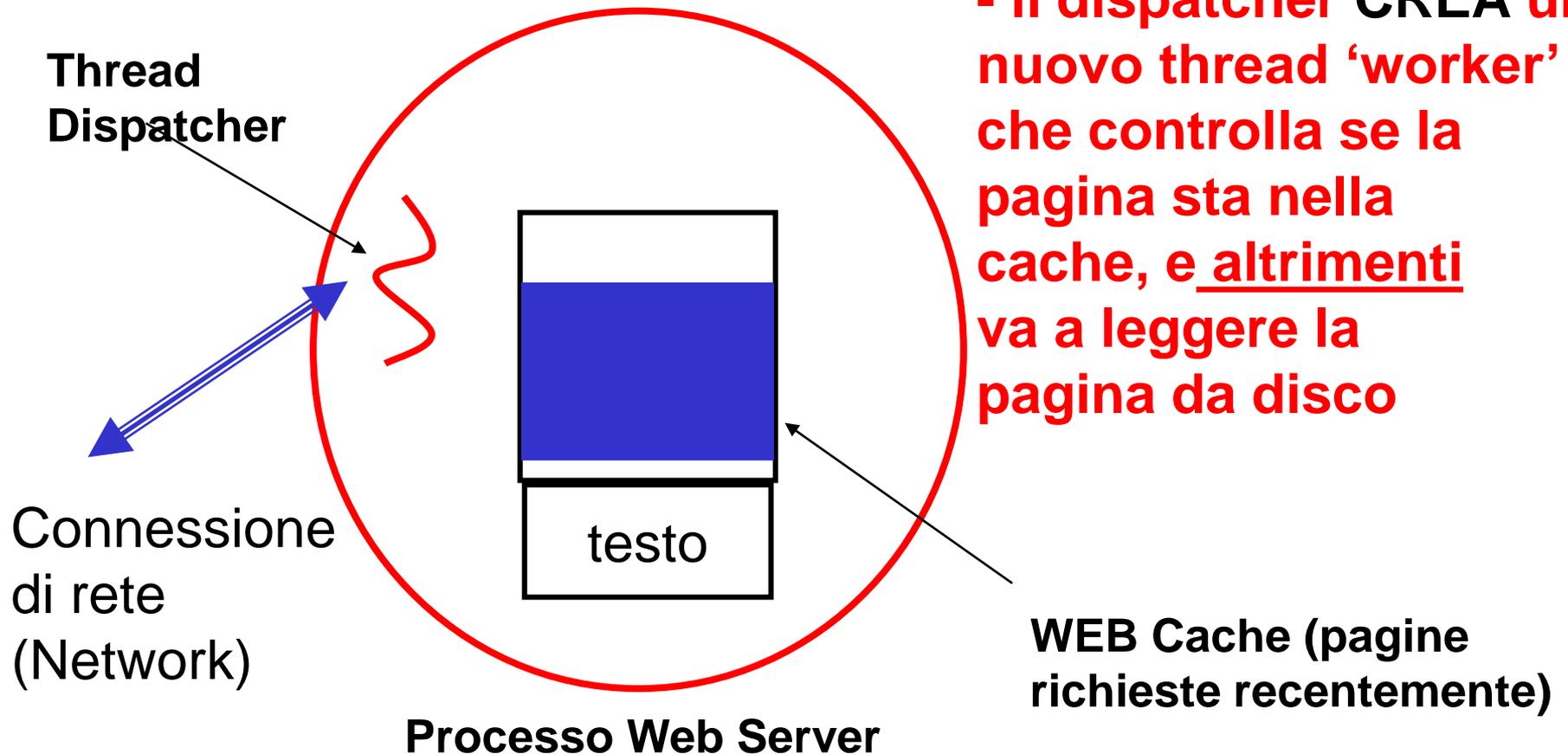
Struttura di un word processor multithreaded (2)



Un web server multithreaded

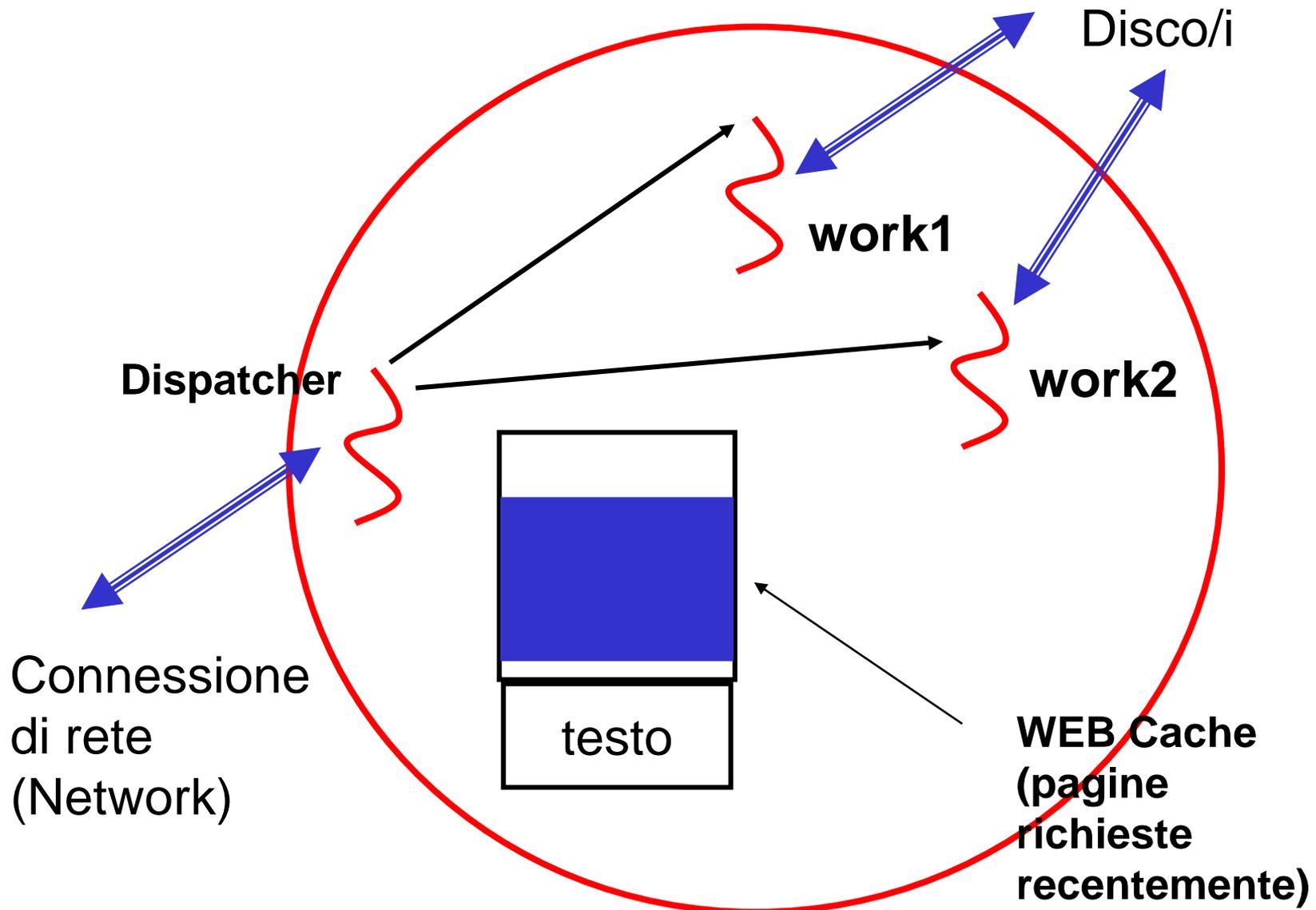


Un web server multithreaded (2)

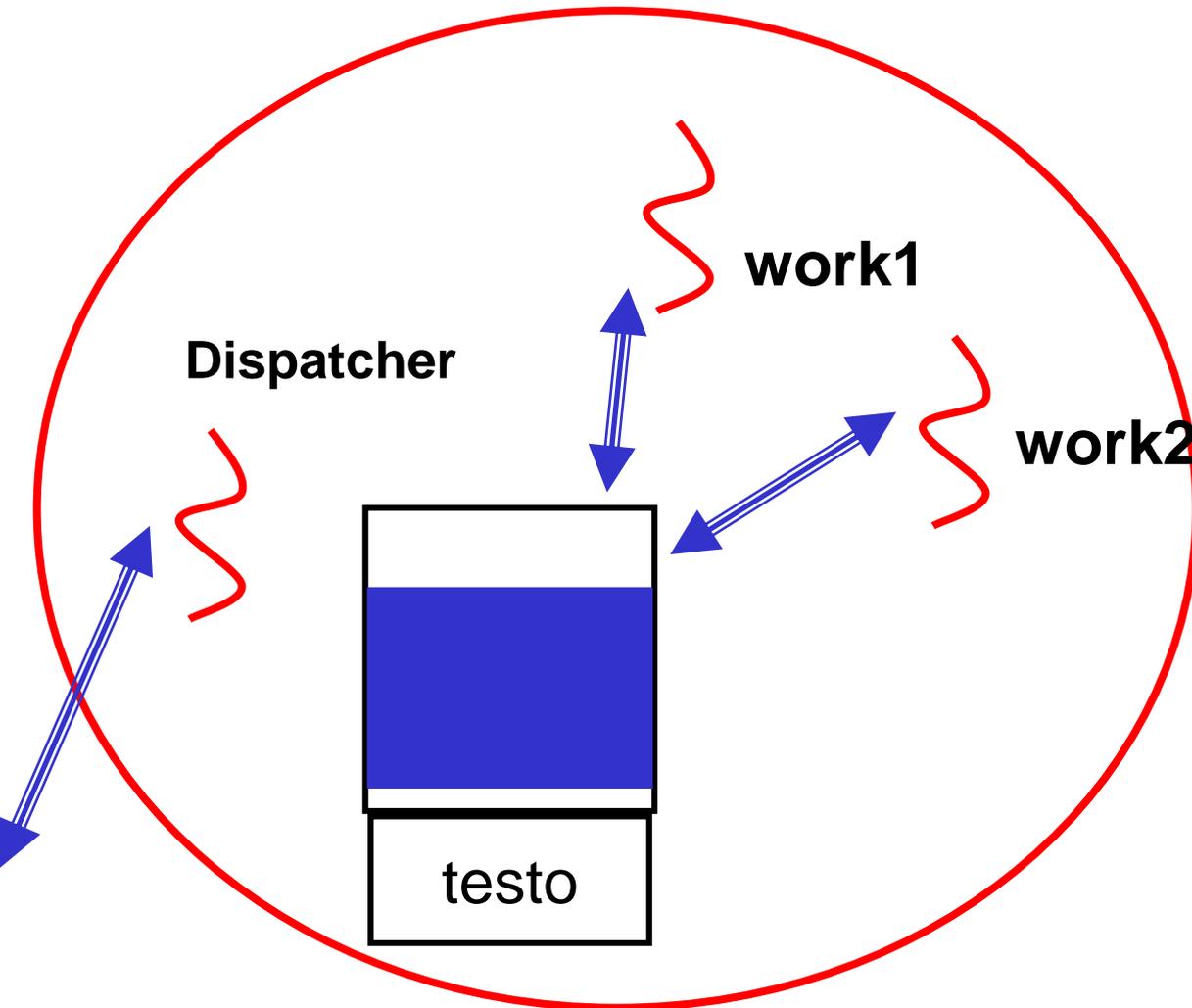


All'arrivo di una nuova richiesta dalla rete :
- il dispatcher **CREA** un nuovo thread 'worker' che controlla se la pagina sta nella cache, e altrimenti va a leggere la pagina da disco

Un web server multithreaded (3)



Un web server multithreaded (4)



- Ogni worker
- 1) legge le informazioni su cache o disco
 - 2) aggiorna la cache
 - 3) risponde alla richiesta,
 - 4) TERMINA

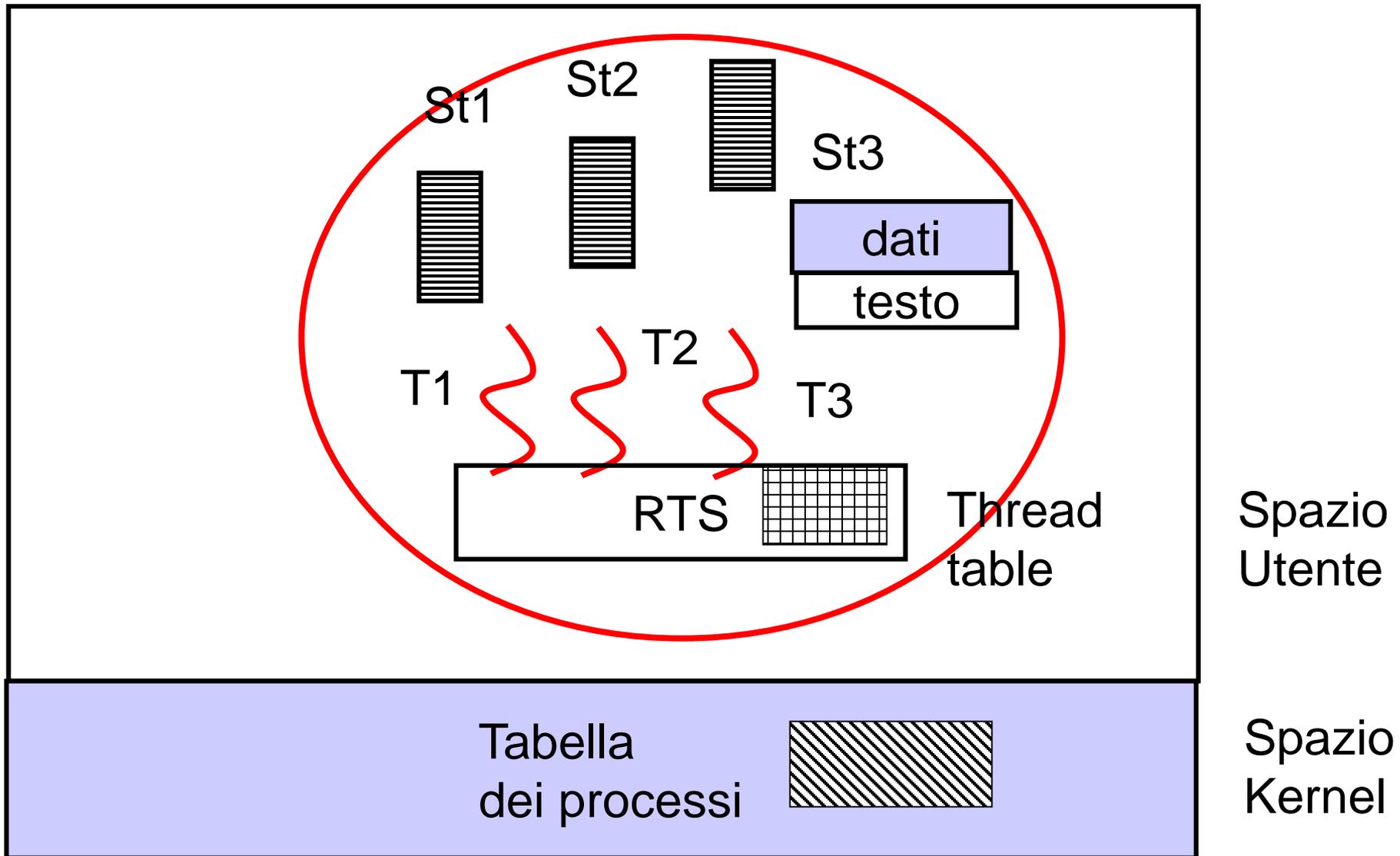
User-level thread (1)

- 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

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
 - `thread_yield ()`

User-level thread (3)



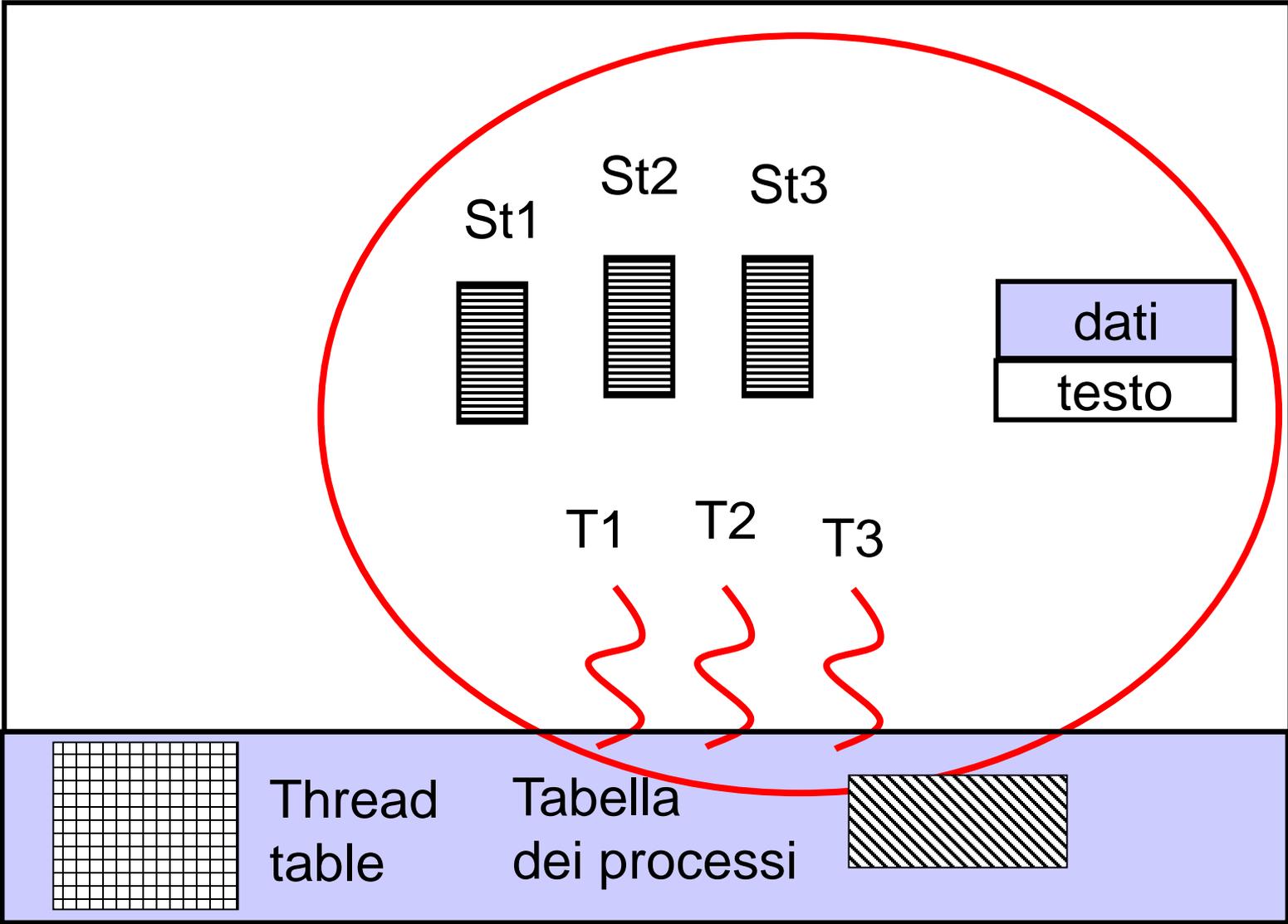
User-level thread (4)

- 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!
 - Addio parallelismo ...

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

Kernel-level thread (3)



Spazio Utente

Spazio Kernel

User–level thread vs kernel-level thread

- Creazione di thread e thread switch molto veloce
- Si può effettuare uno scheduling “personalizzato”, dipendente dall’applicazione
- Eseguibili su un SO che supporta solo i processi
- Gestione problematica delle system call bloccanti
 - librerie di SC non bloccanti

Modelli ibridi

