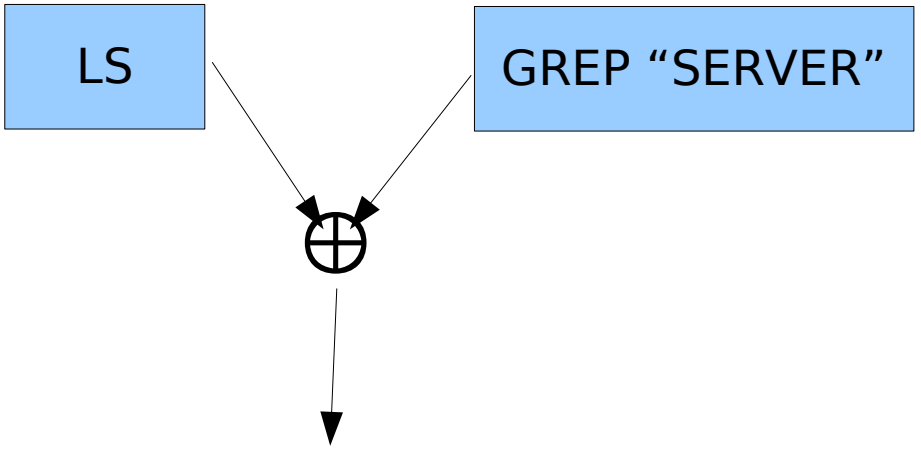
Esempio

ls: comando Unix che lista il contenuto di una directory

grep: comando Unix che identifica linee di un file che fanno match con un pattern dato (es. che contengono una stringa)



Se potessi far collaborare i due processi potrei fornire all'utente l'elenco dei contenuti di una directory i cui dati rispettano un certo schema!

Memoria condivisa

- tramite system call un processo richiede l'allocazione di una porzione di memoria accessibile anche ad altri processi, che dovranno successivamente agganciarla al proprio spazio degli indirizzi
- l'area di memoria può essere "plasmata" secondo qualsiasi tipo di dati utile ai programmi comunicanti, per esempio un buffer (di dimensione limitata)

```
#define D 10

typedef ... elemento;

typedef struct _b {
   elemento dato[D];
   int inserisci, preleva;
} buffer_cond;
```

Memoria condivisa

(1)

Area di memoria:
sequenza di byte

(2)

I processi hanno bisogno di dati in un formato specifico, devono sapere come interpretare I byte condivisi

```
typedef struct _b {
   elemento dato[D];
   int inserisci, preleva;
} buffer_cond;
```

(3)

I processi condividono un tipo di dato: il tipo di dato secondo il quale vanno interpretati I byte della memoria condivisa

Memoria condivisa

PRODUTTORE

```
...
alloca b di tipo buffer_cond come memoria condivisa

while (1) {
    if (! pieno(b) ) {
       nuovo = ... produci ...;
       b.dato[b.inserisci] = nuovo;
       b.inserisci = (b.inserisci+1) % D;
    }
}
```

CONSUMATORE

```
...
aggancia b al proprio spazio indirizzi

while (1) {
    if (! vuoto(b) ) {
       nuovo = b.dato[b.preleva];
       b.preleva = (b.preleva+1) % D;
    }
}
```

Memoria condivisa

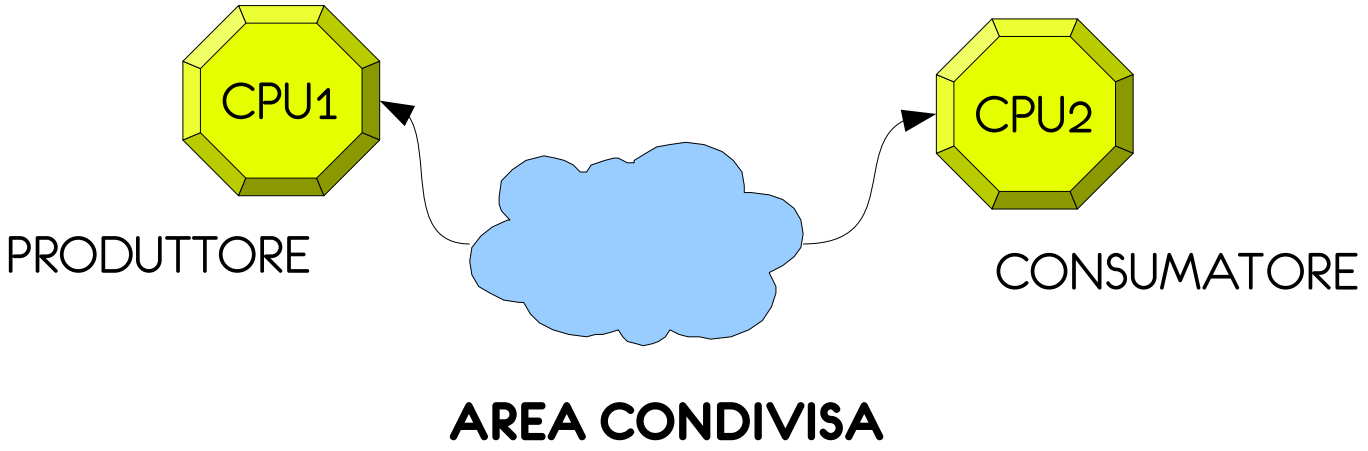
- Gli accessi vanno controllati per evitare inconsistenze!!!

```
pieno(b): (b.inserisci+1)%D == b.preleva
vuoto(b): (b.inserisci == b.preleva)
```

- Se produttore ha eseguito `b.dato[b.inserisci] = nuovo` ma non ancora `b.inserisci = (b.inserisci+1) % D` consumatore può ritenere il buffer vuoto, erroneamente
- Peggio ancora: e se ci fossero tanti consumatori? Se il buffer contiene un solo elemento e un consumatore ha già eseguito `nuovo = b.dato[b.preleva]` ma non ancora `b.preleva = (b.preleva+1) % D` un altro consumatore potrebbe ritenere il buffer erroneamente pieno!!!

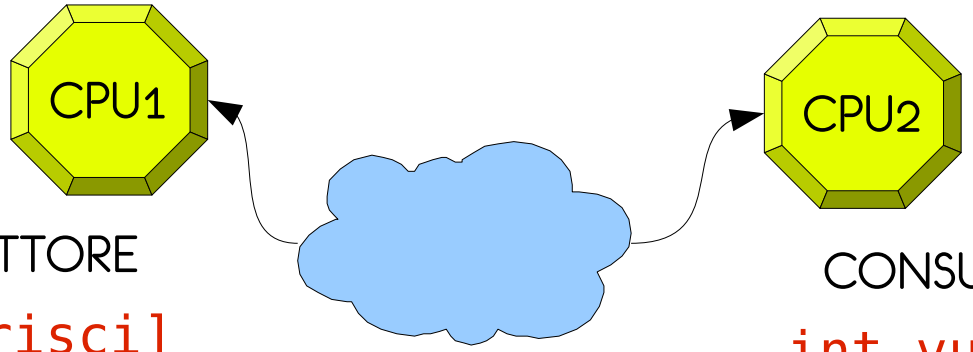
Inconsistenza dei dati

- Abbiamo visto che le inconsistenze dei dati del tipo descritto possono essere causate dallo scheduling della CPU
- Se avessimo due processori, uno per il produttore e uno per il consumatore, il problema potrebbe presentarsi comunque?



Inconsistenza dei dati

- Sì
- Supponiamo che all'inizio
  b.inserisci == b.preleva == 0
- è un problema intrinseco all'interleaving delle istruzioni del produttore e del consumatore



PRODUTTORE

```
b.dato[b.inserisci]
        = nuovo
```

ma non ha ancora aggiornato b.inserisci

AREA CONDIVISA

CONSUMATORE

```
int vuoto() {
return
b.inserisci==b.preleva;
}
```

Scambio di messaggi

- Consente a due processi di comunicare senza condividere una stessa area di memoria. Questo meccanismo può essere caratterizzato in modi diversi, a livello logico:
  - diretto o indiretto: è diretto se un processo deve fornire il PID del processo con cui desidera comunicare
  - sincrono (bloccante) o asincrono (non bloccante):
    - **invio sincrono**: il mittente si blocca in attesa che il ricevente riceva il messaggio
    - **recezione sincrona**: il ricevente rimane in attesa di un messaggio fintantoché non ne viene effettivamente ricevuto uno
    - **rendez vous**: invio sincrono + recezione sincrona
  - a gestione automatica o esplicita del buffer

Send e receive dirette

- A livello logico due processi che intendono comunicare devono essere connessi da un canale. Per scambiarsi messaggi usano send e receive

- **comunicazione diretta:**
  - **send(P, msg)**: invia msg al processo P
  - **receive(P, msg) / receive(id, msg)**: attendi un messaggio dal processo P, tale messaggio verrà memorizzato in msg oppure ricevi un messaggio e salva in id il PID del mittente e in msg il messaggio
  - la reciproca conoscenza del PID definisce un canale logico

Send e receive indirette

- in questo caso l'invio/recezione sono effettuati non a processi ma a porte o mailbox, distinte dall'identità del ricevente

- **comunicazione indiretta:**
  - **send(M, msg)**: invia msg alla mailbox M
  - **receive(M, msg)**: attendi un messaggio alla mailbox M
  - il canale logico è definito dalla mailbox
  - NB: più mittenti/riceventi possono usare la stessa mailbox, una stessa coppia di processi può esare diverse mailbox per comunicare

Buffering dei messaggi

- la mailbox una una capacità

- **tipi di buffering:**
  - **capacità 0**: il canale non ha memoria (meccanismo no buffering); il mittente rimane sospeso se il ricevente non ha ancora consumato il messaggio ricevente (gestione esplicita del buffer)
  - **capacità N>0**: il mittente rimane in attesa solo se il buffer è pieno (meccanismo automatic buffering)
  - **capacità illimitata**: il mittente non attende mai (meccanismo automatic buffering)

Un paio di esempi

- socket: definizione di un canale di comunicazione fra processi in esecuzione su macchine diverse
- remote procedure call: invocazione di una procedura definita ed eseguita da un altro sistema

Socket

- usato in sistemi client-server
- socket: è il nome dato a un estremo (endpoint) di un canale di comunicazione fra due processi
- due processi interagenti in rete usano una coppia di socket (uno per ciascuno), ciascuno dei quali ha per identificatore l'IP della macchina concatenato a un identificatore di porta:

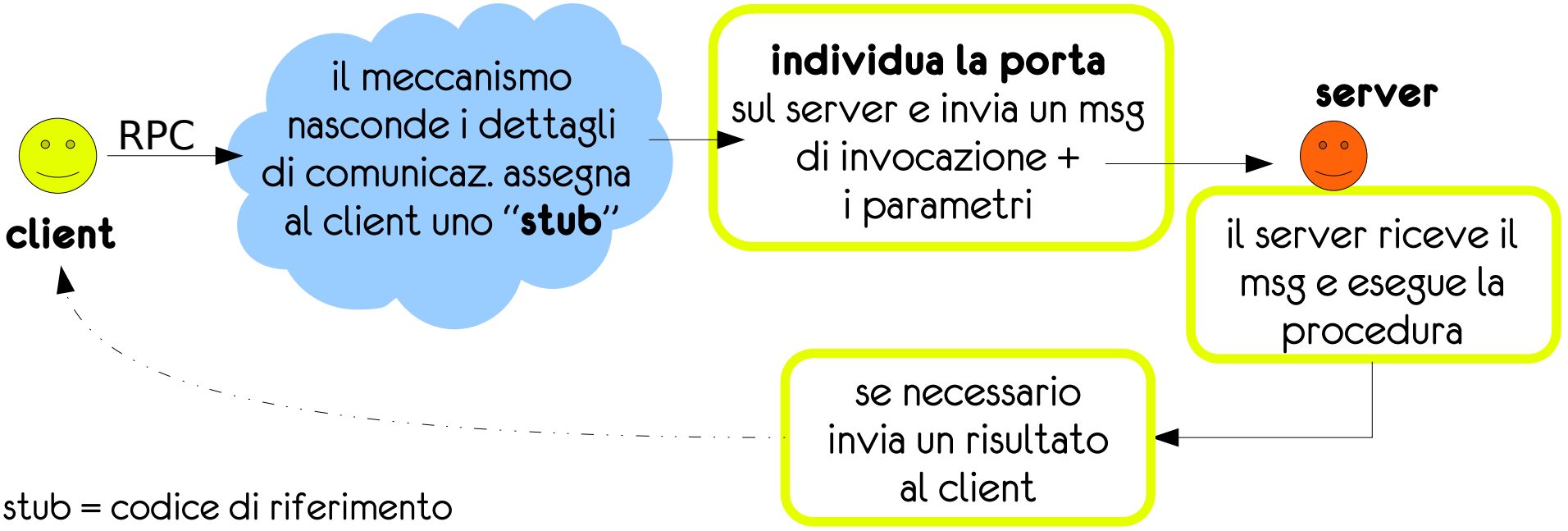
- Esempio di id: **140.228.112**:**80** la parte in blu (140.228.112) è l'IP di una macchina, quella in rosso (80) un numero di porta, sono concatenati da un due punti.
- I messaggi sono semplici pacchetti dati non strutturati

Socket

- tutte le porte < 1024 sono riservate
- un processo utente che richieda una porta riceverà un numero maggiore di 1024 e lo stesso numero non potrà essere assegnato a due processi diversi. La coppia **IP:PORTA** è un identificatore univoco.
- esempi di porte predefinite:
  - telnet 23
  - ftp 21
  - web 80
- Caso particolare: tramite l'indirizzo di loopback **127.0.0.1** un computer può fare riferimento a se stesso, quindi il meccanismo visto è usabile anche per far comunicare processi diversi su di uno stesso computer.

Remote Procedure Call

- meccanismo usato in sistemi client-server
- in certi casi è utile consentire a un processo di invocare l'esecuzione di una procedura che risiede su di un'altra macchina connessa in rete: meccanismo noto come remote procedure call (RPC)
- i messaggi sono ben strutturati, sono costituiti dall'identificatore della procedura da eseguire e dai parametri su cui viene invocata

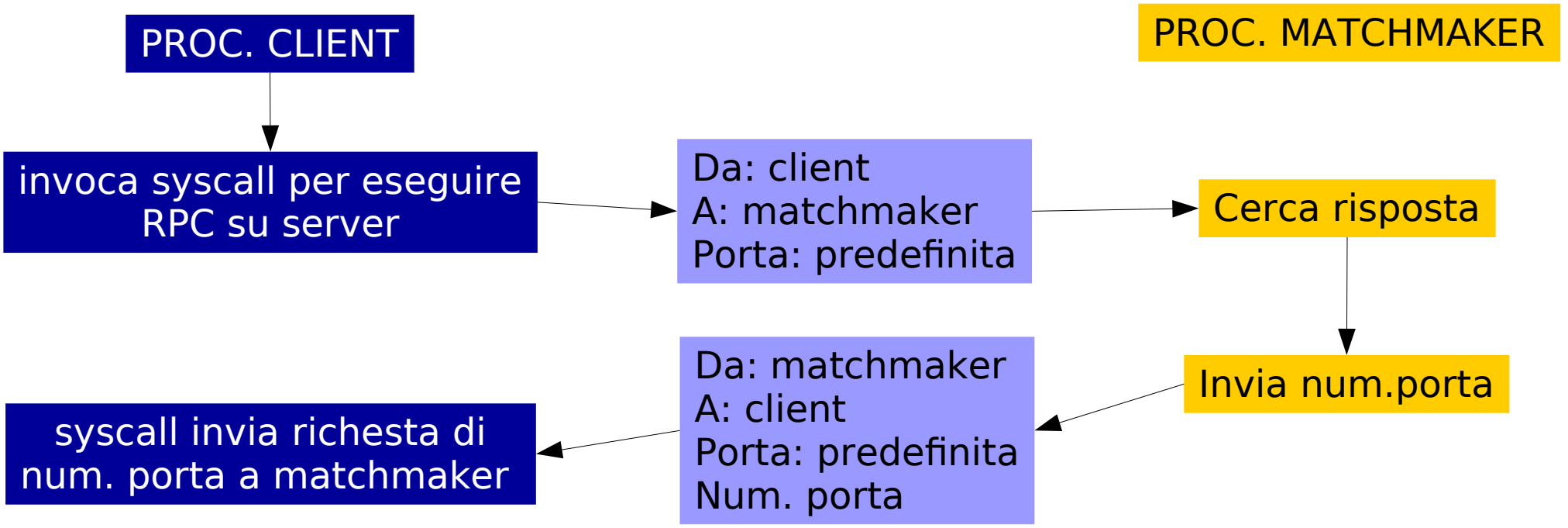


stub = codice di riferimento

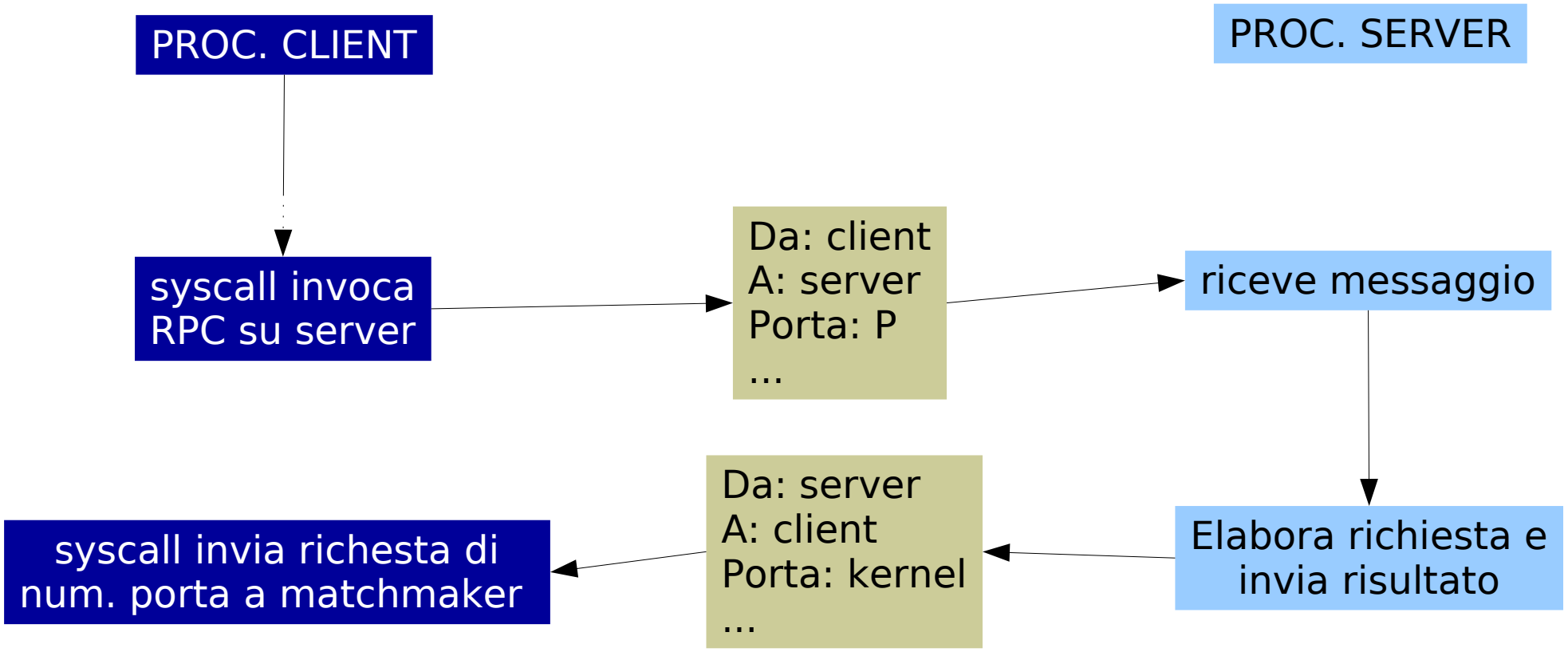
Associazione porte a RPC

- Il client deve conoscere l'associazione fra le RPC di un server e le relative porte. Come/quando avviene tale associazione?
- Client e server non condividono memoria!
- Soluzioni
  - Associazione Predefinita: fissata in fase di compilazione delle RPC
  - Associazione Dinamica:
    - si introduce un servizio intermedio di rendez-vous, detto "**matchmaker**" invocabile su di una porta fissa
    - **Interazione con due demoni**: il matchmaker e il demone in ascolto sulla porta identificata

Interazione col matchmaker



Interazione col server



Remote Procedure Call

- **Problemi**

  1) macchine diverse possono adottare rappresentazioni diverse dei dati!! **Soluzione**: **usare un formato intermedio**, es. **XDR** (external data representation). Il client converte i dati in XDR, il server converte da XDR al proprio formato

  2) una RPC va eseguita una e una sola volta però come essere sicuri che l'esecuzione è avvenuta? E se problemi di connessione facessero perdere la richiesta? **Soluzione**:

     a. il server archivia tutte le richieste pervenute associando loro un timestamp (sicurezza che una richiesta venga eseguita al + 1 volta)

     b. meccanismo di riveuta: il client continua a reinviare la richiesta fino a quando non riceve una ricevuta dal server

Pipe

- Canale di comunicazione fra processi
- Pipe anonima:
  - Canale simplex, FIFO
  - interazione basata sul meccanismo produttore consumatore
  - Estremità di scrittura, estremità di lettura (unidirezionalità)
  - Consentono la comunicazione fra una singola coppia di processi, tipicamente un padre crea una pipe anonima e la usa per interagire con un figlio
  - Non sopravvive al processo creatore

PROCESSO PRODUTTORE PROCESSO CONSUMATORE

Named pipe

- Named Pipe (o FIFO):
  - interazione basata sul meccanismo produttore consumatore
  - FIFO
  - In Unix è unidirezionale, in Windows è bidirezionale
  - Consente la comunicazione di più di due processi
  - Sopravvive alla terminazione del processo creatore
  - Va disallocata esplicitamente
  - Spesso realizzata come file
  - VMware virtualizza le porte tramite named pipe

Process tree

- Per visualizzare l'albero dei processi si possono utilizzare, in alternativa, i comandi:
  - **ps axjf**
  - **ps -ejH**

ESEMPIO DI OUTPUT

```
 1913  tty2      Sl+  /usr/lib/gnome-terminal/gnome-terminal-server
 1945  pts/0     Ss    \_ bash
 2142  pts/0     S+    |  \_ alpine
20606 pts/1     Ss    \_ bash
20798 pts/1     R+    |  \_ ps f
20742 pts/2     Ss    \_ bash
20778 pts/2     S+       \_ man vmstat
20790 pts/2     S+          \_ pager
 1882  tty2      Sl+   /usr/lib/tracker/trac
```

“vedere” code e mem. Cond.

baroglio@rhialto:~$ **ipcs**

------ Message Queues --------
key msqid owner perms used-bytes messages

------ **Shared Memory Segments** --------

key	**shmid**	**owner**	perms	**bytes**	**nattch**	status
0x00000000	131072	baroglio	600	524288	2	dest
0x00000000	229377	baroglio	600	4194304	2	dest
0x00000000	393218	baroglio	600	524288	2	dest
0x00000000	294915	baroglio	600	67108864	2	dest
0x00000000	1605636	baroglio	600	524288	2	dest
0x00000000	13172741	baroglio	600	2304	2	dest
0x00000000	6717446	baroglio	600	36864	2	dest
0x00000000	2392071	baroglio	600	4915200	2	dest

...

------ Semaphore Arrays --------
key semid owner perms nsems

“vedere” code e mem. Cond.

baroglio@rhialto:~$ **ipcs -cu**

------ Messages Status --------
allocated queues = 0
used headers = 0
used space = 0 bytes

------ Shared Memory Status --------
segments allocated 30
pages allocated 30971
pages resident 6906
pages swapped   0
Swap performance: 0 attempts   0 successes

------ Semaphore Status --------
used arrays = 0
allocated semaphores = 0

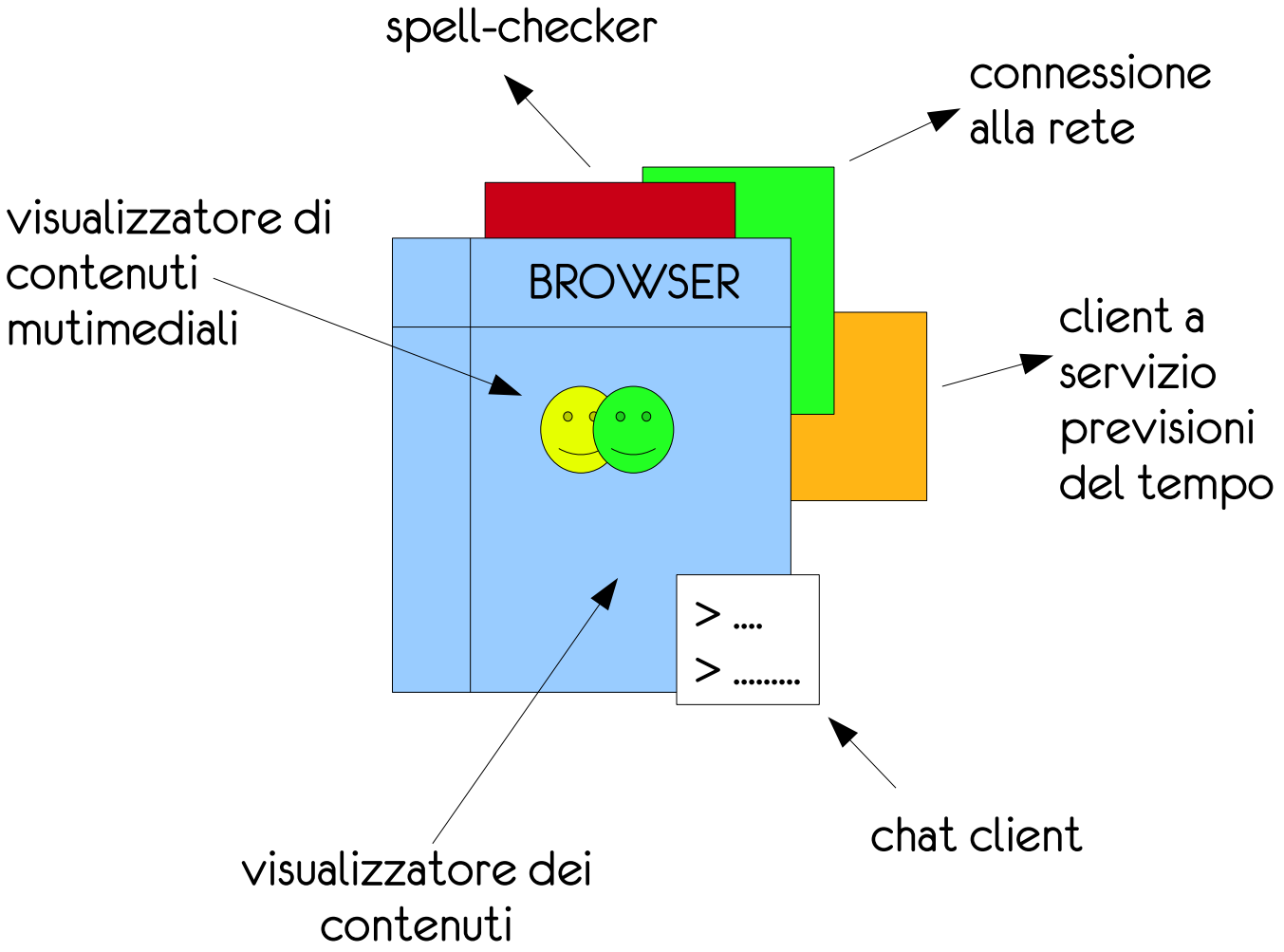
thread

capitoli 4 e 5.5 del libro (VII ed.)
+
cap. 4 di Sistemi Operativi, di H. Deitel,
P. Deitel e D. Choffnes

Introduzione

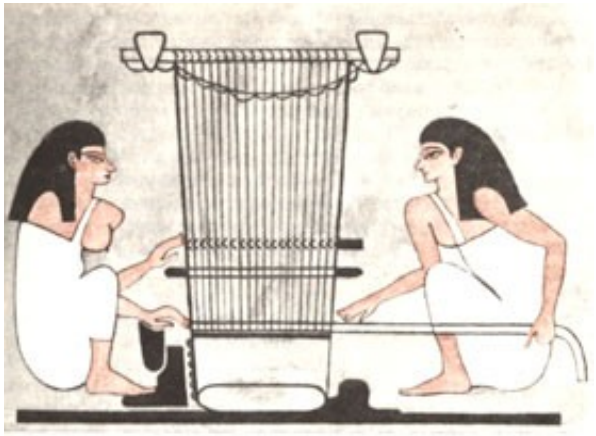
io avvio il programma con un click oppure digitando il nome del browser in una shell ...

ma tutto questo è un processo solo?



Thread

Thread: esecuzione sequenziale di codice



Ткачество на фресках Бени-Хасана

Thread

```
Main() {
    int ris = 0;
    ...
    ris = f1(x1, x2, ...) + f2(y1, y2, ...);
}
```

Se f1 e f2 non hanno dipendenze, eseguire l'una prima dell'altra non fa differenza. Allora è anche possibile pensare a una loro esecuzione parallela ...

f1 e f2 condividono il loro contesto di esecuzione, ha senso pensare a combinare due processi per consentirne l'esecuzione parallela?

Introduzione

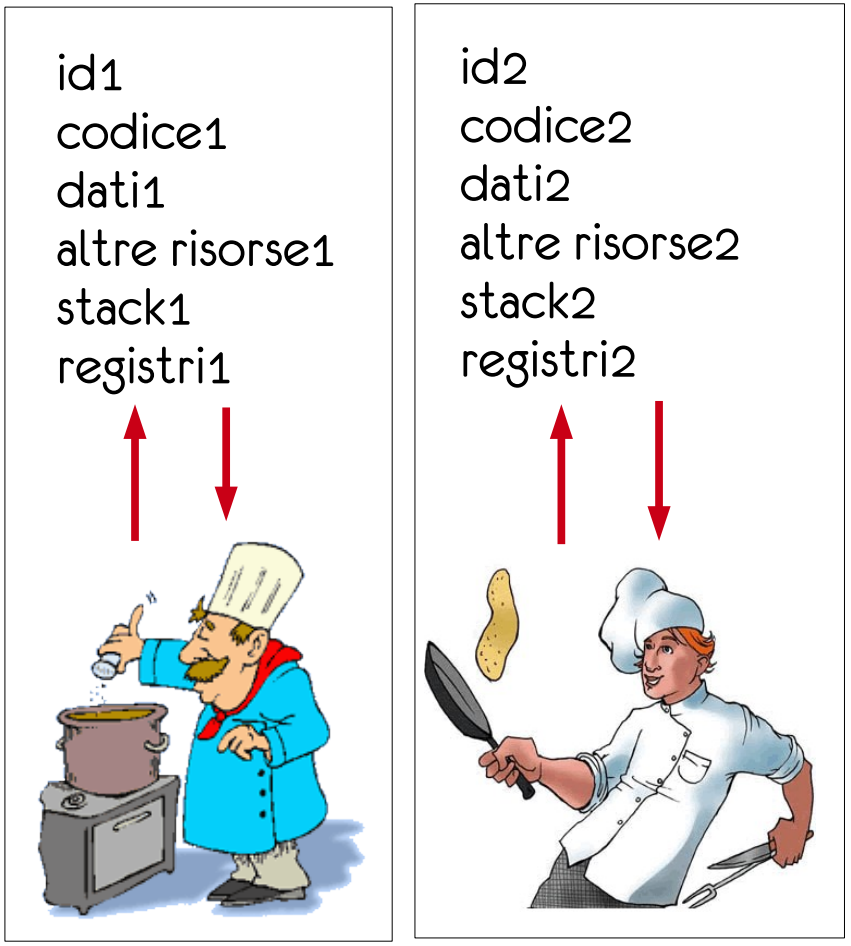
- molti SO moderni considerano come unità di base d'uso della CPU non il processo ma il THREAD. Un processo può essere organizzato in un insieme di thread cooperanti

Thread:

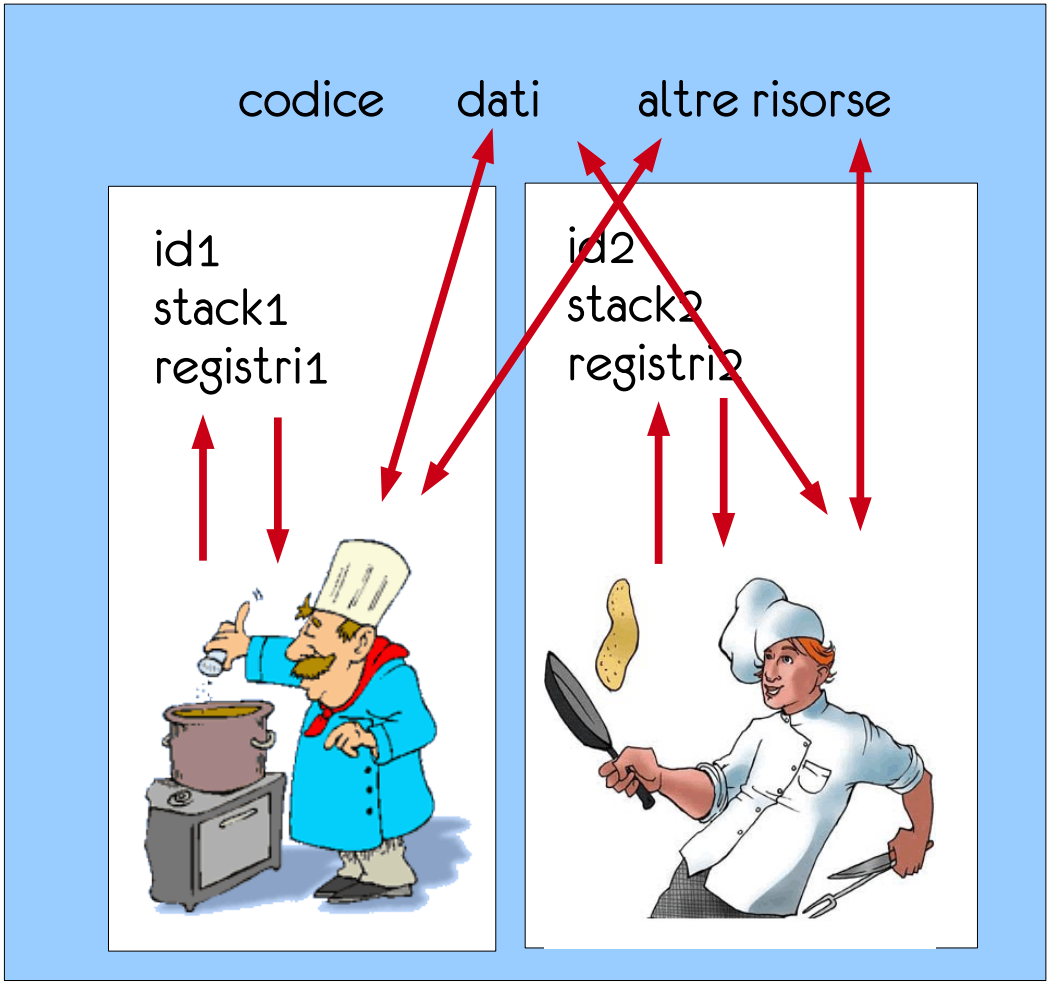
- è costituito da: un identificatore, un program counter, un insieme di valori di registri, uno stack
- condivide con gli altri thread dello stesso processo: il codice, la sezione dati, file aperti, segnali e altre risorse di sistema

- Un processo costituito da un solo thread è detto **heavyweight process**, processo pesante

Cose proprie e comuni



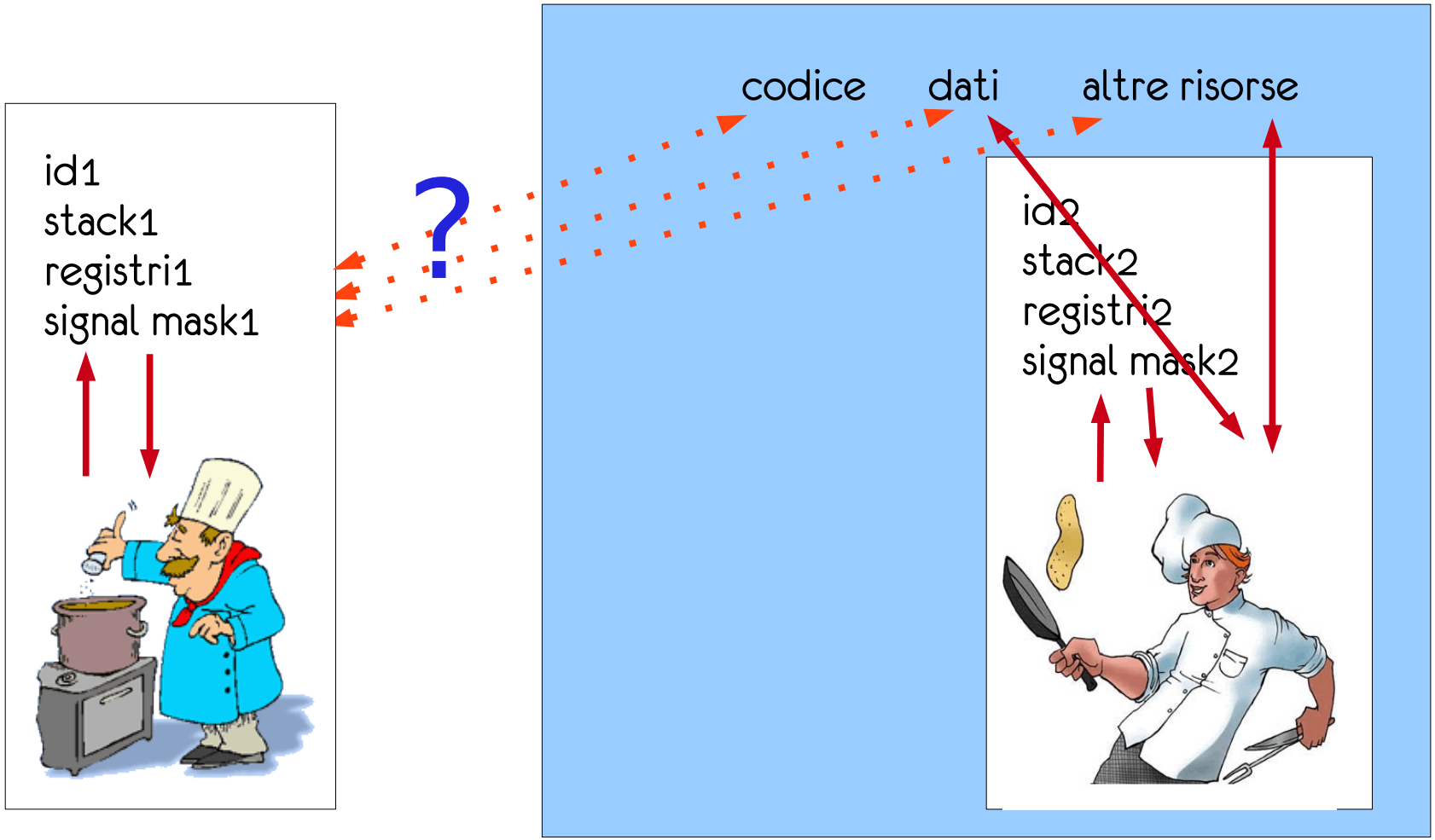
processi



thread di uno stesso processo

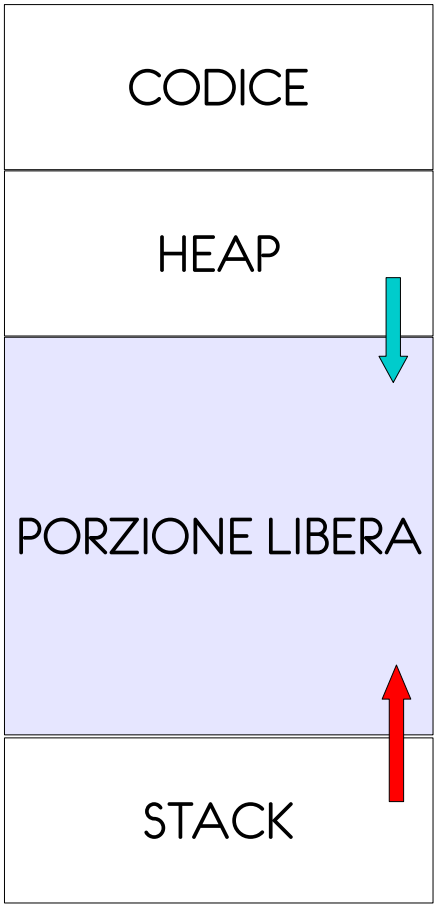
comunicano tramite un meccanismo a memoria condivisa!!

Thread senza processi?

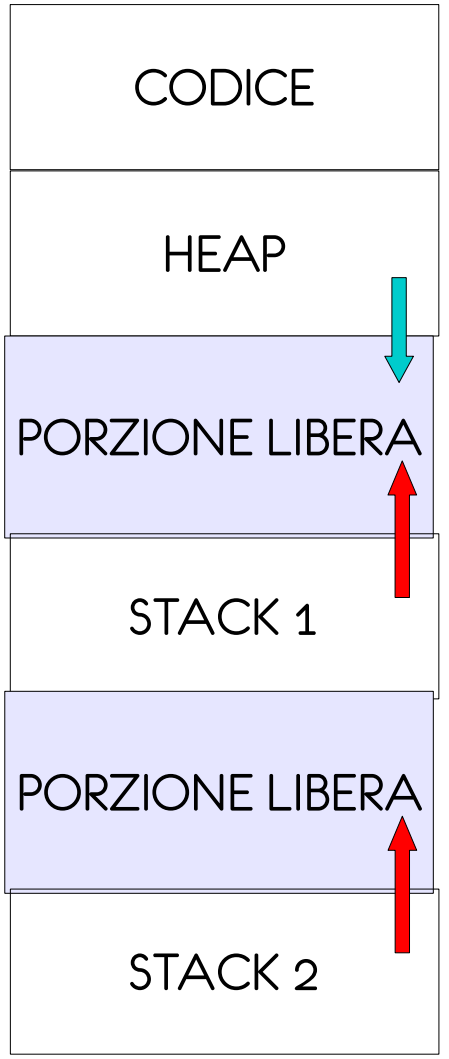


un thread non può esistere al di fuori di un processo perché non contiene tutte le informazioni necessarie per poter effettuare l'esecuzione: codice, dati, risorse fanno parte del processo

Organizzazione della memoria



Processo monolitico



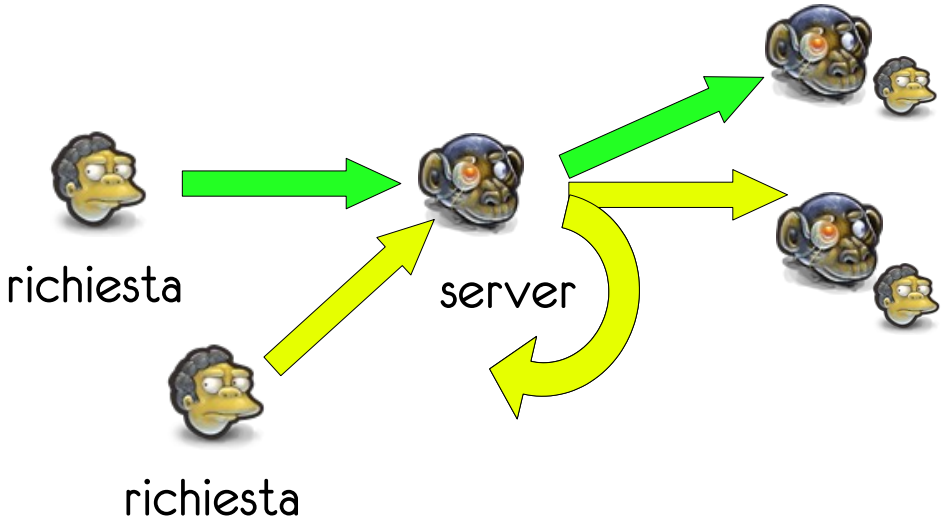
Processo con due thread

Vantaggi

- Una soluzione a thread di solito è più efficiente di una soluzione a processi cooperanti
- Molti programmi contengono sezioni di codice eseguibili indipendentemente dal resto del programma.
- Incremento delle prestazioni:
  - Il **context switch** è più rapido,
  - è richiesta l'allocazione di una **quantità inferiore di risorse** (le condividono),
  - la **comunicazione è più veloce** perché avviene tramite variabili condivise (spesso non occorre appoggiarsi a strutture di IPC)
- in architetture multicore diventa possibile spezzare l'esecuzione di un processo su più processori

Esempio

- molti server (RPC, web, ...) sono implementati a thread: l'arrivo di una nuova richiesta comporta la **creazione** di un thread servitore, dedicato a soddisfare quella richiesta mentre il thread principale si rimette in attesa di nuove richieste



- Server per cui il tempo di esecuzione è critico, sono talvolta organizzati in un pool di thread creati all'avvio.
- Tali server possono gestire in parallelo al più un numero di richieste pari alla cardinalità del pool di thread.

Thread e linguaggi

- I thread sono definiti dal programmatore, creati e gestiti da programma
- Molti linguaggi di programmazione offrono specifiche istruzioni per la creazione e il controllo dei thread e per controllare l'accesso alle variabili condivise
- **Esempio**:
  Java, C#, Python

Programmazione III

- Al contrario i processi sono per lo più generati in modo invisibile all'utente,i linguaggi di programmazione non forniscono costrutti sintattici ad hoc e occorre l'esplicita invocazione di system call
- Linguaggi come C e C++ sono detti a singolo flusso di controllo
- **NB**: anche questi linguaggi possono essere usati per scrivere programmi a multithead ma richiedono l'inclusione di apposite librerie

Esempio: Android

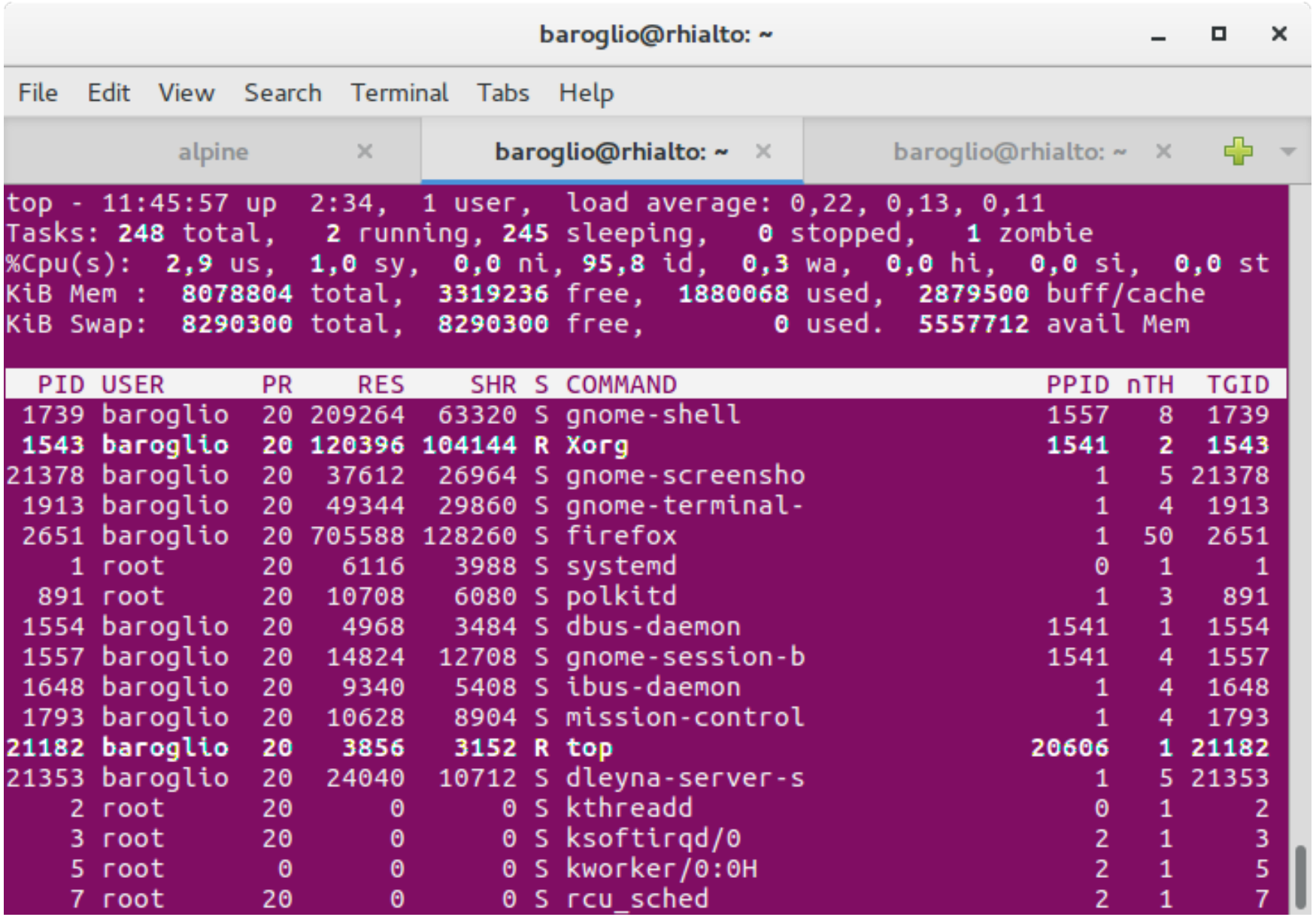
- Il SO Android fa dei thread un elemento centrale della programmazione
- Tutte le componenti di ogni applicativo sono realizzate come diversi thread di uno stesso processo
- All'avvio di un applicativo, viene generato il thread main che ha l'importante funzione di effettuare il dispatch degli eventi alle diverse componenti dell'interfaccia grafica

“vedere” I thread

baroglio@rhialto:~$ **ps m -L | more**

```
PID   LWP TTY      STAT   TIME COMMAND
...
 1541     - tty2     -      0:00 /usr/lib/gdm3/gdm-x-session ...
    -  1541 -      Ssl+   0:00 -
    -  1542 -      Ssl+   0:00 -
    -  1555 -      Ssl+   0:00 -
 1543     - tty2     -      2:01 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth...
    -  1543 -      Sl+    2:00 -
    -  1544 -      Sl+    0:00 -
 1554     - tty2     -      0:00 dbus-daemon --print-address 4 --session
    -  1554 -      S+     0:00 -
 1557     - tty2     -      0:00 /usr/lib/gnome-session/gnome-session-binary...
    -  1557 -      Sl+    0:00 -
    -  1694 -      Sl+    0:00 -
    -  1695 -      Sl+    0:00 -
    -  1697 -      Sl+    0:00 -
 1654     - tty2     -      0:00 /usr/lib/gvfs/gvfsd
    -  1654 -      Sl+    0:00 -
    -  1655 -      Sl+    0:00 -
...
```

“vedere” I thread: top



fork exec e thread

- abbiamo visto le system call per i processi: fork ed exec
- un processo multithread può usare queste system call? Che effetto hanno?
- **fork**:
  - ??
- **exec**:
  - ??

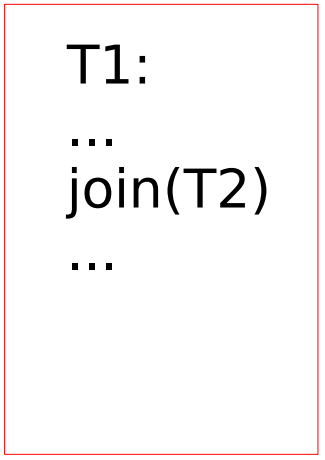
fork exec e thread

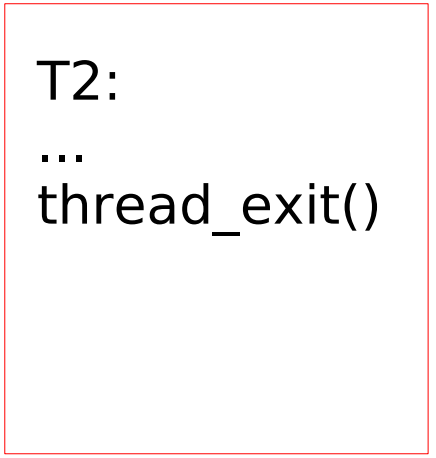
- abbiamo visto le system call per i processi: fork ed exec
- un processo multithread può usare queste system call? Che effetto hanno?
- **fork**:
  - in certi SO causa la creazione di un nuovo processo con la duplicazione di tutti i thread
  - in altri SO causa la sola duplicazione del thread chiamante
- **exec**:
  - causa la sovrascrittura del codice dell'intero processo con un nuovo programma: NB, tutti i thread vengono sostituiti!

Operazioni sui thread

- **creazione**: implica la creazione di una struttura dati specifica che mantiene le informazioni relative al nuovo thread;
- **terminazione**: è più rapida di quella dei processi perché, per es., non richiede la gestione delle risorse
- sospensione/blocco
- recupero/risveglio
- **join**: specifica per I thread, comporta l'attesa da parte di un thread della terminazione di un altro

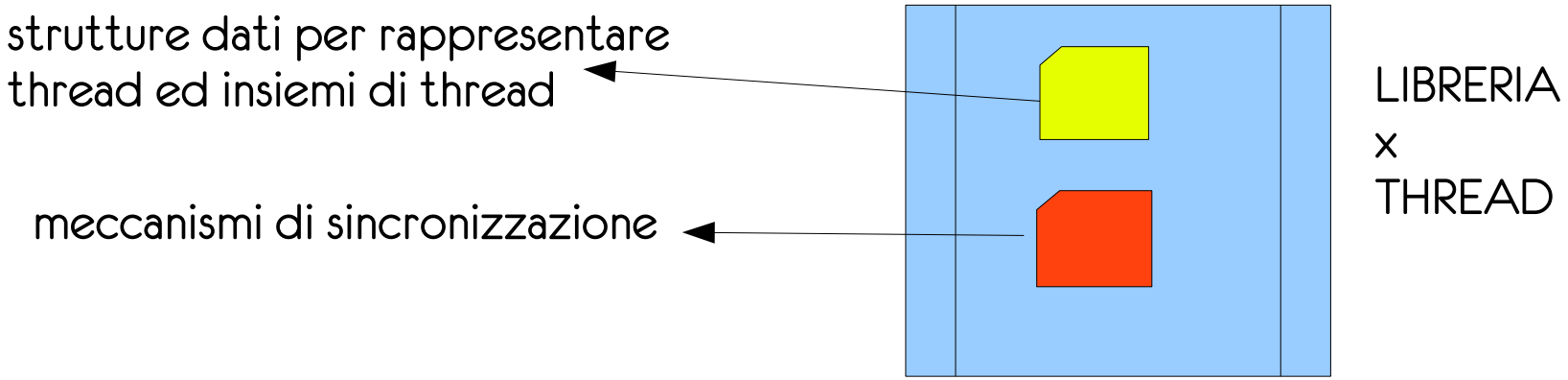
T1 si blocca sulla join fino a quando T2 non termina





Modelli di thread (a)

- Primi sistemi operativi: consentivano un solo contesto di esecuzione per processo -> ogni processo multithread deve gestire le info dei suoi thread, il loro scheduling, la comunicazione fra I suoi thread.
- Si parla di thread a livello utente
- Sono creati da funzioni di libreria che non possono eseguire istruzioni privilegiate
- Sono trasparenti al SO, che vede il processo come una sola entità indistinta



Modelli di thread (a)

- Vantaggi:
  - I thread a livello utente sono portabili anche su SO che non prevedono il multi-threading perché sono gestiti internamente al processo, ad un livello di astrazione più alto;
  - I criteri per effettuare lo scheduling possono essere facilmente adattati alle esigenze dello specifico programma;
  - esecuzione più rapida in quanto non richiede né l'uso di interruzioni (e dei context switch che conseguono alla loro gestione) né l'invocazione di system call
- Svantaggi:
  - Non sono adattabili a sistemi multiprocessore
  - Se un thread richiede di eseguire un'operazione di I/O tutto il processo rimane bloccato fino al suo termine

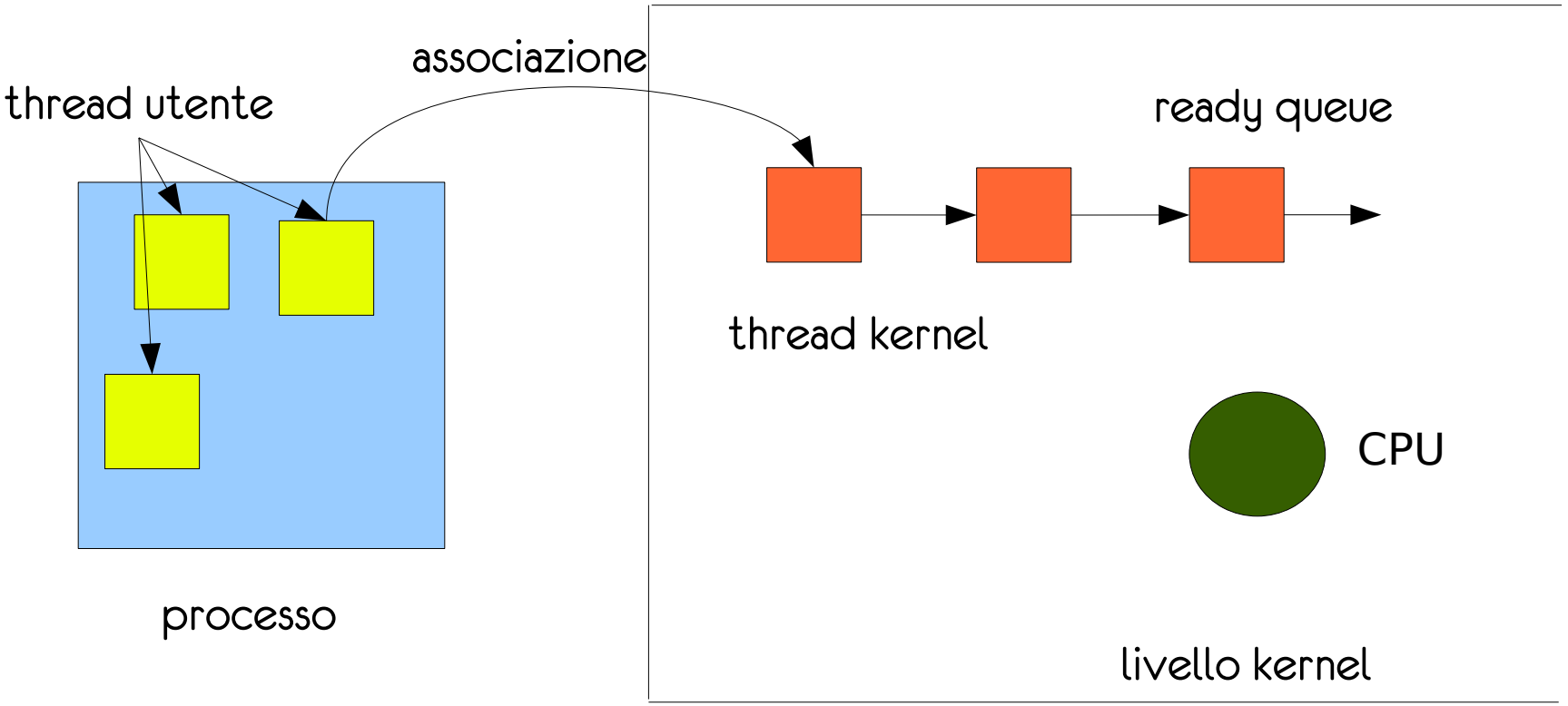
Modelli di thread (b)

- Un approccio diverso consiste nell'associare a un processo diversi contesti di esecuzione (corrispondenti ai vari thread), il cui scheduling sulla CPU è gestito esplicitamente dal SO
- thread a livello kernel: sono creati e gestiti dal SO, se presenti lo scheduling della CPU è fatto a livello di thread kernel.
- Sono più costosi per il SO perché richiedono che esso mantenga appositi descrittori nonché l'associazione fra tali descrittori e i processi che ne definiscono il contesto di esecuzione
- L'utente genera dei thread a livello utente tramite le apposite librerie e il SO associa a tali thread proprio strutture, che implementano thread a livello kernel

modelli di thread (b)

- Vantaggi:
  - è possibile distribuire i thread di uno stesso processo su più processori (se disponibili)
  - singole operazioni di I/O non bloccano processi multithread
  - maggiore interattività con l'utente
  - migliori prestazioni dei singoli processi
- Svantaggi:
  - minore portabilità: non portabili su SO non multithread, SO multithread diversi implementano i thread in modo diverso
  - in presenza di applicazioni fortemente multithread il SO potrebbe dover gestire migliaia di thread contemporaneamente, ciò potrebbe causare un sovraccarico e un calo di prestazioni

Livello utente ⇨ livello kernel

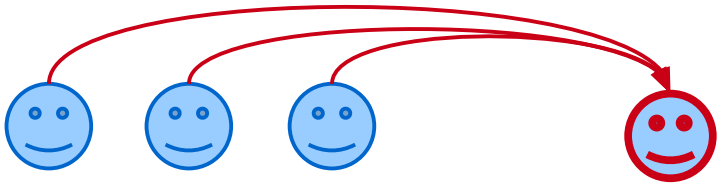


- Per far effettuare lo scheduling della CPU a livello di thread occorre associare thread utente a thread kernel.
- Vi sono tre modelli secondo i quali questa associazione può essere fatta: uno a uno (thread kernel), molti a uno (thread utente), molti a molti (soluzione ibrida)

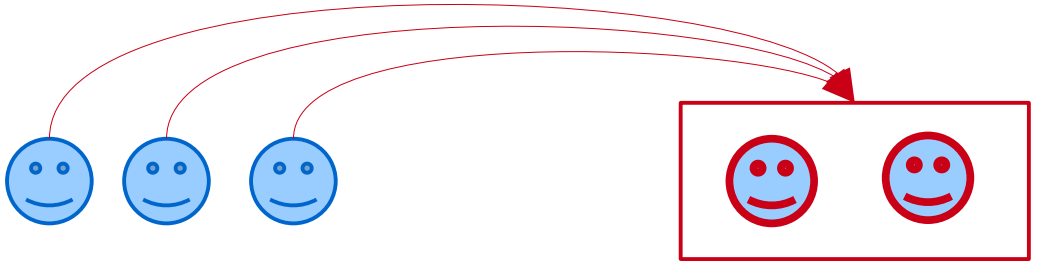
Livello utente e livello kernel

- **uno** (utente) **a uno** (kernel): soluzione adottata da Linux e Windows

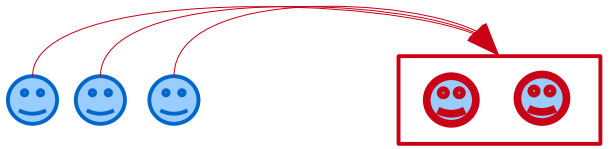
- **molti a uno**: a un pool di thread utente è assegnato un thread kernel. Inefficiente, un solo thread utente per volta può accedere al kernel



- **molti a molti**: a un insieme di thread utente è associato un insieme (di solito + piccolo) di thread kernel. Quando consente anche di vincolare un thread utente a un thread kernel, si parla di modello a due livelli

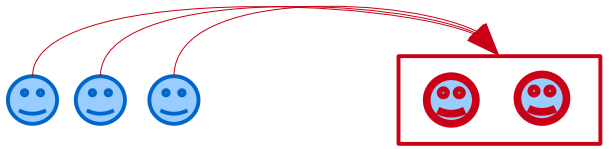


Lightweight process



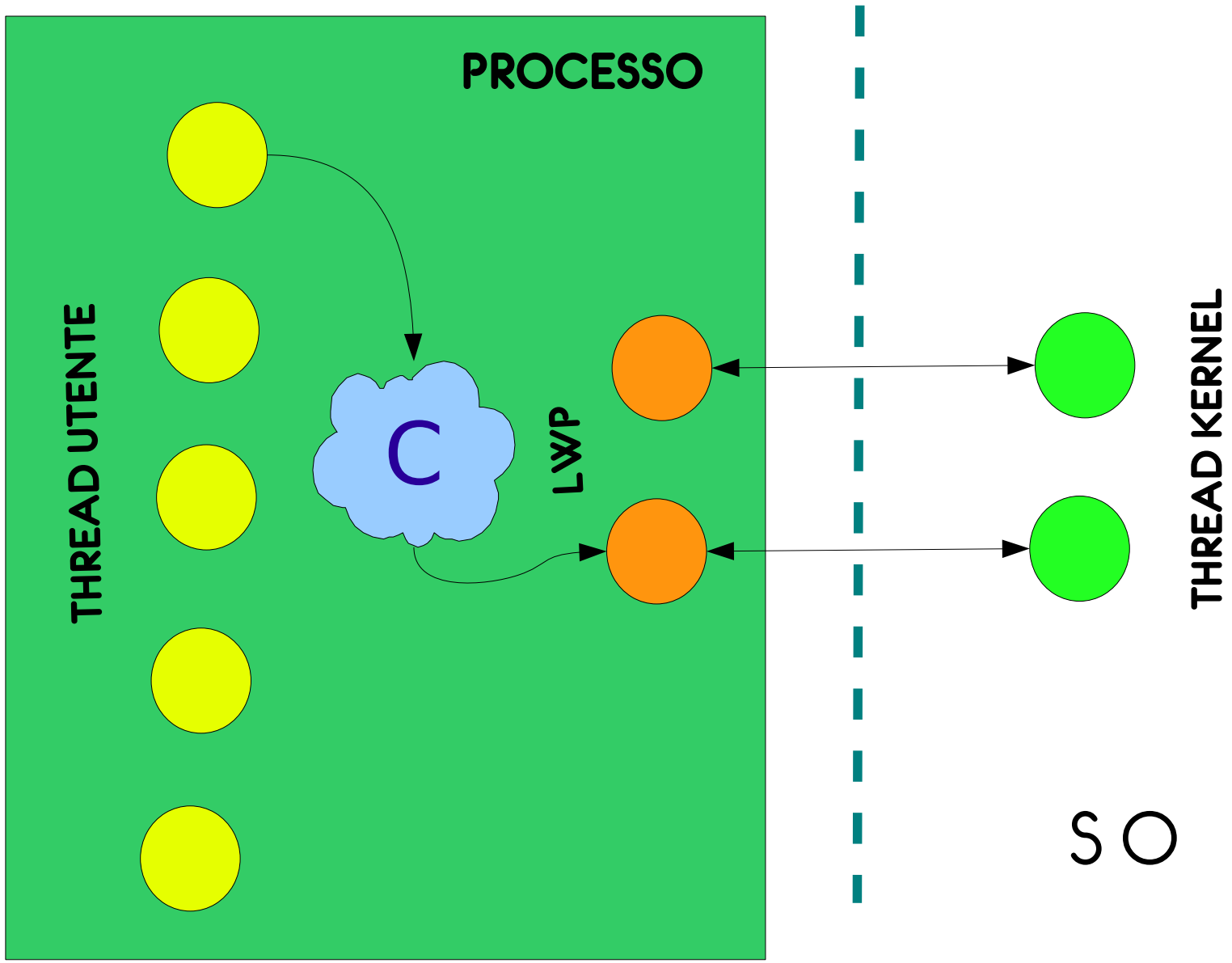
- nei SO che usano il modello molti a molti (o quello a due livelli) si ha la necessità di effettuare uno scheduling dei thread utente per l'accesso ai thread kernel
- questo spesso viene fatto introducendo un nuovo oggetto fra thread utente e thread kernel, detto lightweight process (**LWP**, processo leggero)

Lightweight process



- nei SO che usano il modello molti a molti (o quello a due livelli) si ha la necessità di effettuare uno scheduling dei thread utente per l'accesso ai thread kernel
- questo spesso viene fatto introducendo un nuovo oggetto fra thread utente e thread kernel, detto lightweight process (**LWP**, processo leggero)
- Gli LWP sono visti dagli applicativi utente come **processori virtuali** (un vero e proprio pool di risorse ad essi assegnate) e sono usati per effettuare lo scheduling dei thread utente al kernel. Si parla di associazione indiretta.
- **Ogni LWP corrisponde a un thread kernel**
- L'assegnamento di un LWP a un thread utente è gestito in modo esplicito, da programma, dall'applicativo stesso, che deve includere procedure speciali per la gestione di "**upcall**" ...

Processori virtuali assegnati a un processo



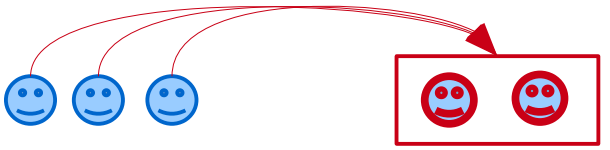
Scheduling

- L'**applicativo utente** esegue il proprio scheduling dei thread su un insieme di LWP messi a disposizione dal kernel. Ogni thread dell'applicativo può essere pronto, in esecuzione o in attesa:
  - **un thread utente è in esecuzione se ha assegnato un LWP**

Scheduling

- L'**applicativo utente** esegue il proprio scheduling dei thread su un insieme di LWP messi a disposizione dal kernel. Ogni thread dell'applicativo può essere pronto, in esecuzione o in attesa:
  - **un thread utente è in esecuzione se ha assegnato un LWP**
- Quando un thread esegue una system call bloccante, il SO informa l'applicativo (**upcall**).

Scheduling

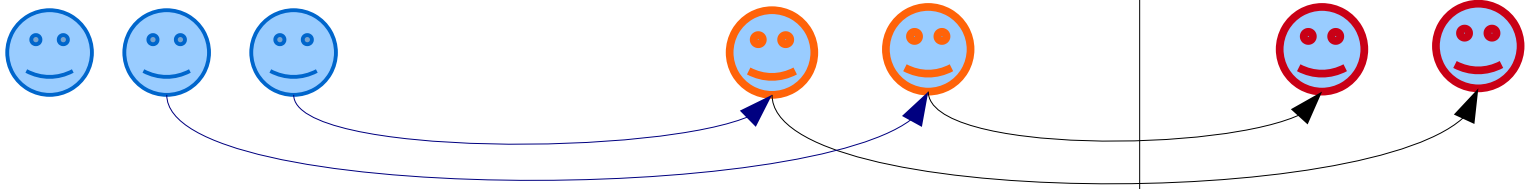


- L'**applicativo utente** esegue il proprio scheduling dei thread su un insieme di LWP messi a disposizione dal kernel. Ogni thread dell'applicativo può essere pronto, in esecuzione o in attesa:
  - **un thread utente è in esecuzione se ha assegnato un LWP**
- Quando un thread esegue una system call bloccante, il SO informa l'applicativo (**upcall**).
- L'applicativo esegue un **gestore della upcall** che salva lo stato del thread bloccante e rilascia l'LWP su cui era eseguito, che viene riassegnato dall'applicativo stesso a un suo altro thread pronto per l'esecuzione.
- Quando si verificherà l'evento che sveglia il thread sospeso, il SO farà un'altra upcall. L'applicazione segnerà come pronto il thread e lo inserirà nel pool dei thread pronti da assegnare a un LWP.

Lightweight process

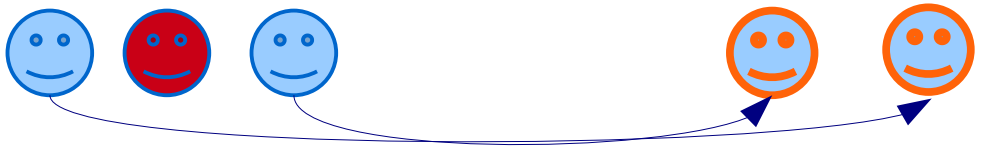
lo scheduling della CPU viene effettuato fra i thread kernel che vanno in ready queue

NB: non è uno scheduling della CPU !!!



un applicativo ha pronti più thread di LWP a disposizione, deve decidere a chi assegnarli: due saranno in esecuz. e uno no

ogni LWP ha associato un thread kernel



upcall

quando un thread sta per bloccarsi il SO fa una upcall e l'applicativo riassegna l'LWP

quando può proseguire il SO fa un'altra upcall e il thread torna fra quelli pronti

upcall

Scheduling della CPU

- in presenza di thread lo scheduling della CPU è quindi a due livelli:
  - **process-contention scope** (PCS): è lo scheduling effettuato all'interno di un processo per decidere a quali thread utente vanno assegnati gli LWP a disposizione del processo. I thread kernel associati agli LWP in uso vengono gestiti come i PCB già visti
  - **system-contention scope** (SCS): lo scheduling della CPU viene fatto fra tutti i thread kernel in ready queue (a una granularità più fine rispetto a quanto visto per i PCB), indipendentemente dal processo di appartenenza
- se l'associazione fra thread utente e thread kernel è 1-a-1 allora si ha solo SCS