Dai Cloni ai Thread in Linux

Clone in Linux

Clone è una libreria di sistema che viene usata per creare dei processi lightweight cioè che possono condividere parte del contesto di esecuzione: **spazio memoria**, tabella dei segnali, tabella dei descrittori

Viene usata quindi per creare **thread**

Al contrario della fork il thread creato esegue una funzione passata come parametro su un argomento anch'esso passato come parametro

Clone in Linux

```
#include <sched.h>

int clone(int (*fn)(void *),
              void *child_stack,
              int flags,
              void *arg, ...
     /*       pid_t *ptid, struct user_desc *tls, pid_t *ctid
     */
);
```

Clone in Linux

fn: puntatore ad una funzione chiamata dal processo creato all'inizio dell'esecuzione

arg: argomento passato alla funzione fn

Quando fn(arg) termina, il processo figlio termina (può terminare anche con exit o segnali)

child_stack: specifica la locazione di memoria dello stack usato dal processo figlio

Lowbyte di **flags**: contiene il numero del segnale di terminazione da inviare al padre

Clone in Linux

flag :
si può mettere in bitwise end con maschere per decidere cosa condividere con il padre

CLONE_VM: condivide spazio di indirizzi

CLONE_SIGHAND: condivide la tabella dei segnali

CLONE_FILES: condivide i file descriptor (file aperti)

CLONE_FS; condivide info su file system (root, workingdir,...)

Poi si possono condivere PID, messaggi di sistema, creare nuovi namespace, ecc

child_stack

Padre e figlio condividono lo spazio di memoria

ma **non possono condividere lo stack**.

Serve quindi allocare spazio per lo stack del figlio

In Linux lo stack cresce verso la parte bassa della memoria virtuale

Quindi bisogna passare alla clone un puntatore all'indirizzo

più in alto nella memoria virtuale del figlio

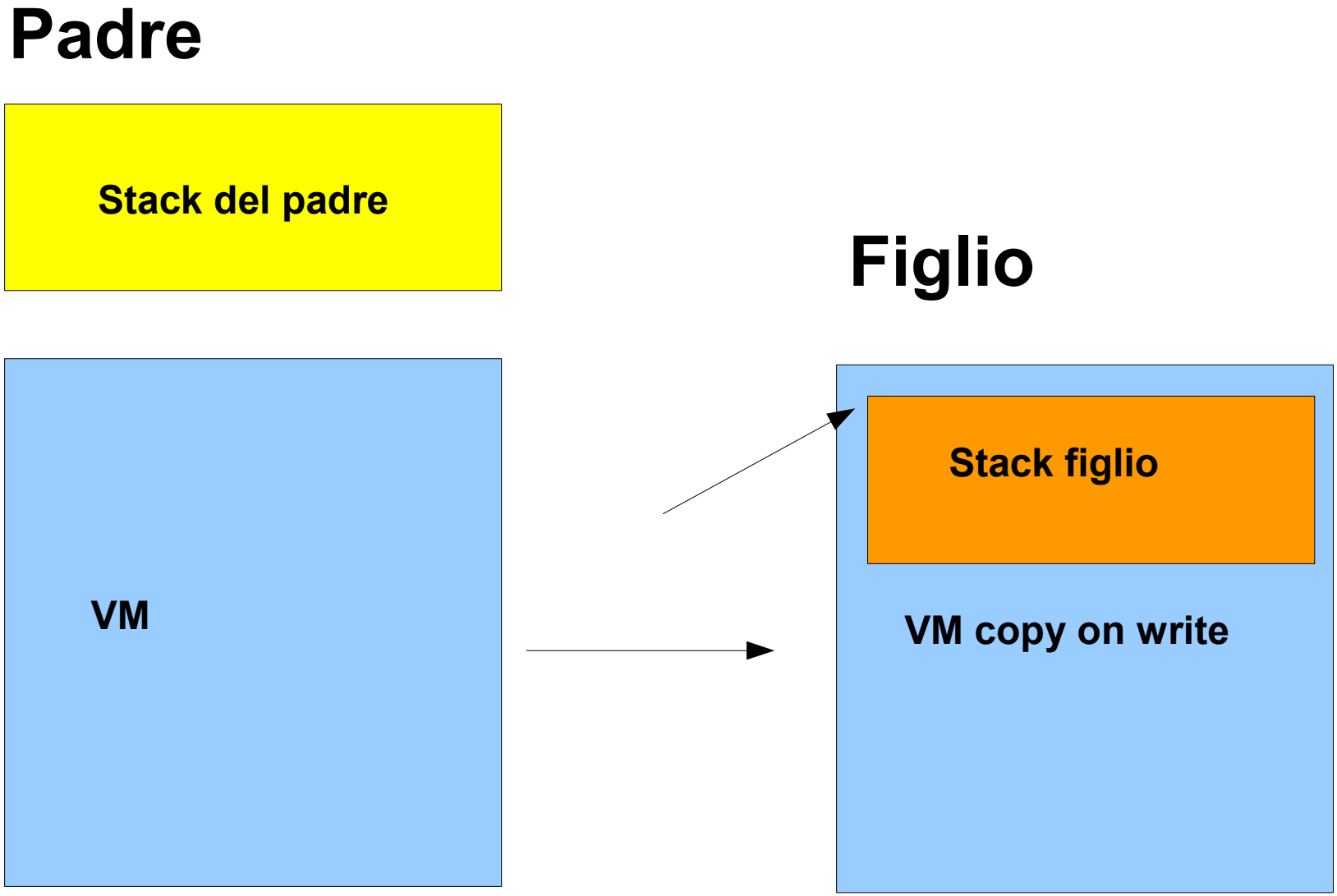
```
....
#define STACK_SIZE 65536

int n = 0;
int Child(void *);

int main() {
 pid_t pid1;
 pid_t pid2;
 char *stackf;
 char *stacks;
 stackf = malloc(STACK_SIZE);
 stacks = malloc(STACK_SIZE);
 pid1 = clone(Child,stackf + STACK_SIZE, NULL);
 pid2 = clone(Child,stacks + STACK_SIZE, NULL);
 waitpid(-1, NULL, __WALL);
 ....
}
```

Non condividono VM con padre

Pointers to raw memory

Indirizzo pù alto nell'area allocata

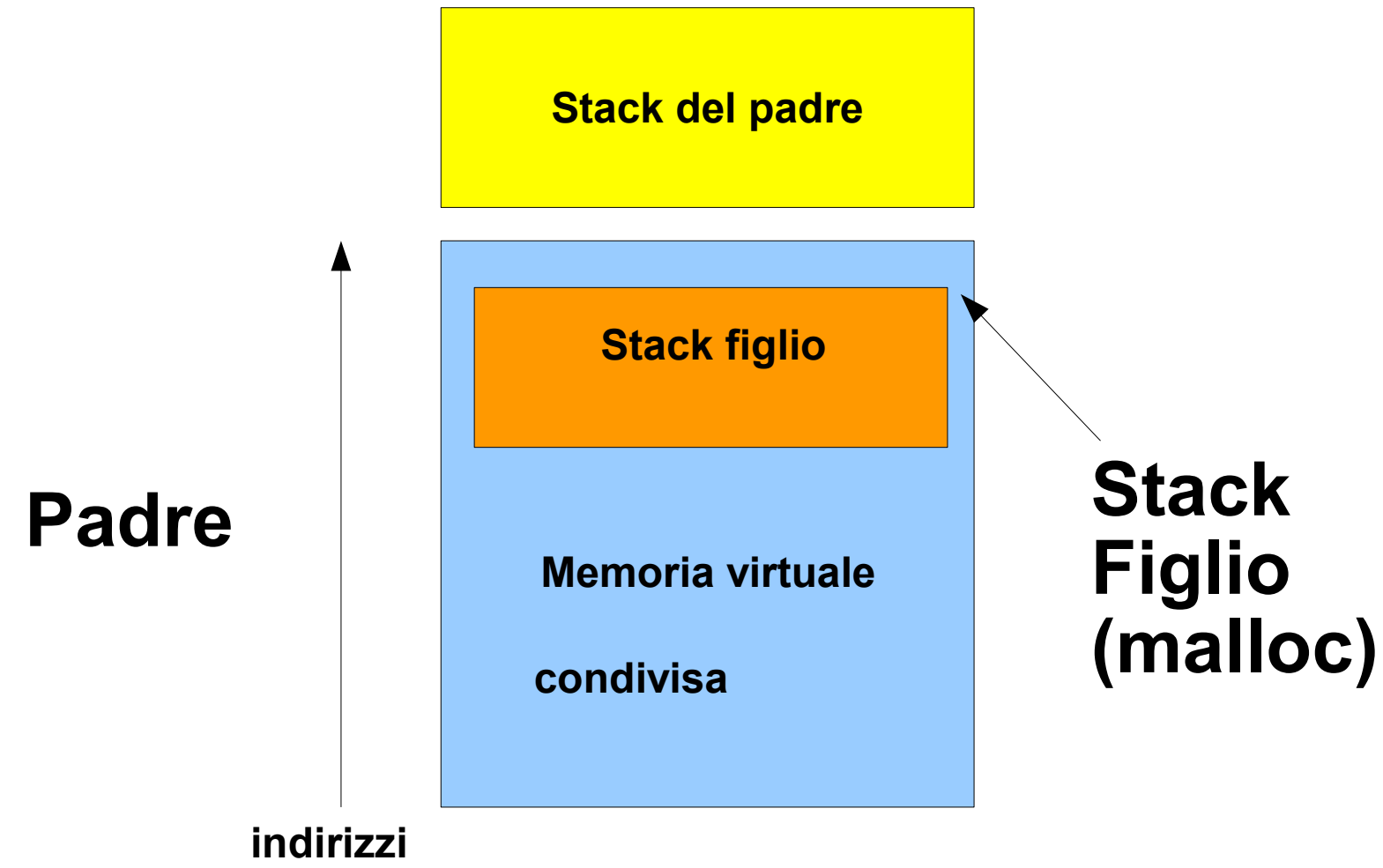
Fork


Padre
Stack del padre
Figlio
VM
Stack figlio
VM copy on write

```
….
#define STACK_SIZE 65536

int n = 0;
int Child(void *);

int main() {
 pid_t pid1;
 pid_t pid2;
 char *stackf;
 char *stacks;
 stackf = malloc(STACK_SIZE);
 stacks = malloc(STACK_SIZE);
 pid1 = clone(Child,stackf + STACK_SIZE, CLONE_VM);
 pid2 = clone(Child,stacks + STACK_SIZE,CLONE_VM);
 waitpid(-1, NULL, __WALL);
 ….
}
```

Condividono VM con padre
Sono thread con memoria condivisa!

Clone


Stack del padre
Stack figlio
Memoria virtuale
condivisa
Padre
indirizzi
Stack
Figlio
(malloc)

Pthread in Linux

Libreria Pthread

- **Definita in ambito POSIX definisce un insieme di primitive per la programmazione di applicazioni multithreaded realizzate in C**

- **Esistono diverse versioni della libreria per diversi sistemi operativi es**
  Linux Thread (basata su clone)
  NPTL (reimplementazione kernel level)

Rappresentazione dei thread

- **Il thread è l'unità di scheduling, ed è univocamente individuato da un indentificatore (intero):**

  pthread_t tid;

- **Il tipo pthread_t è dichiarato nell'header file**

  <pthread.h>

Creazione

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

dove:

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

Terminazione

Un thread può terminare chiamando:

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

dove:

retval:è il puntatore alla variabile che contiene il valore di ritorno (può essere raccolto da altri threads, vedi pthread_join).

Attesa

Un thread può sospendersi in attesa della terminazione di un altro thread con:

int pthread_join(pthread_t th, void *thread_return);

dove:

th: è il pid del particolare thread da attendere
thread_return: è il puntatore alla variabile dove verrà memorizzato il valore di ritorno del thread (vedi pthread_exit)

Normalmente è necessario eseguire la pthread_join per ogni thread che termina la sua esecuzione, altrimenti rimangono allocate le aree di memoria ad esso assegnate.

- **In alternativa si può "staccare" il thread dagli altri con:**

  int pthread_detach(pthread_t th);

  il thread rilascia automaticamente le risorse assegnategli quando termina.

Sincronizzazione

Libreria pthread:

- mutex (semaforo binario)
- variabili condizione

Semafori (*POSIX 1003.1b*, *libreria <semaphore.h>)*.

Mutex

Astrazione simile al concetto di semaforo binario.

Il valore può essere 0 oppure 1 (*occupato* o *libero*)

Sono utilizzati per risolvere problemi di *mutua esclusione.*

Un mutex è definito dal tipo pthread_mutex_t che rappresenta:

- **lo *stato* di mutex**
- **la *coda* dei processi *sospesi* in attesa che mutex sia libero.**

Lock/Unlock

Sui mutex sono possibili solo due operazioni:
locking e unlocking (equivalenti a *wait* e *signal* sui semafori):

pthread_mutex_lock (pthread_mutex_t *M)

se M è *occupato* (stato 0), il thread chiamante si *sospende* nella coda associata a M; altrimenti *occupa M*.

pthread_mutex_unlock (pthread_mutex_t *M)

se vi sono processi *in attesa* di M, ne *risveglia* uno, altrimenti *libera* M.

Come funzionano i semafori

In generale i semafori sono un tipo di dato astratto con due operazioni up e down

Un semaforo S po' essere visto come un contatore condiviso

- **up(S):**
  - **incrementa il contatore di 1 (invio di un segnale)**
- **down(S):**
  - **bloccante se S=0 (cioè il thread si mette in attesa che S>0)**
  - **se S>0 decrementa S**

Esempi semafori: Mutua Esclusione

Semaphor S=1;

Thread T1:

```
while(true){
    down(S);
    critical section
    up(S);
}
```

Thread T2:

```
while(true){
    down(S);
    critical section
    up(S);
}
```

Esempi semafori: Sincronizzazione Ordinare thread

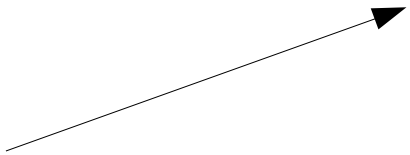
Semaphor S2=0;

Thread T1:

```
while(true){
    ...
    up(S2);
}
```

Thread T2:

```
while(true){
    down(S2);
    ...
}
```



Come sono implementati i semafori

I semafori sono costituiti da

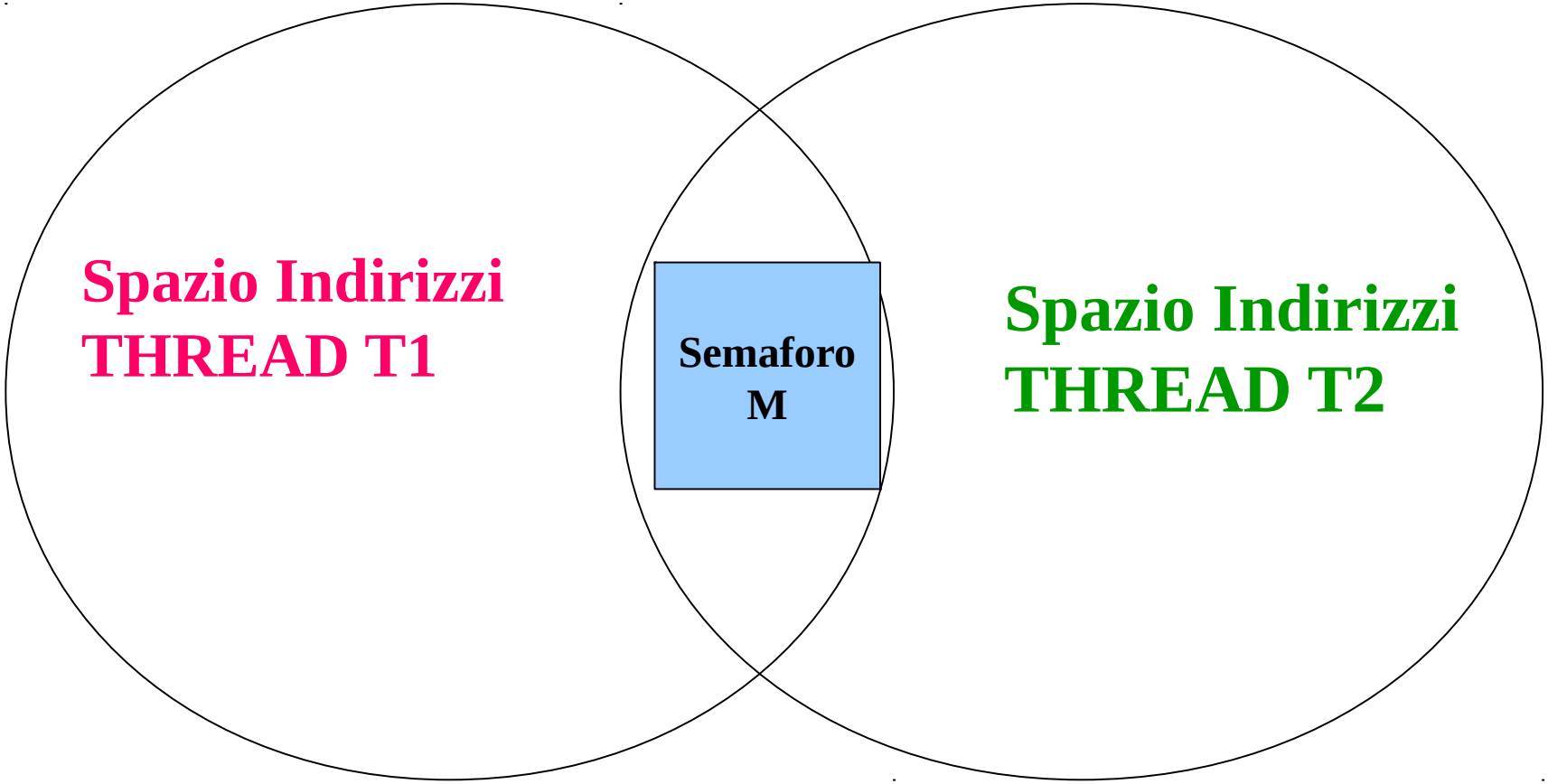
- **Un contatore che tiene traccia dei segnali inviati**
- **Una coda FIFO di (puntatori a) thread che rappresenta l'insieme (ordinato) dei thread in attesa**

down():
se il contatore è a zero sospende il thread e lo mette in coda

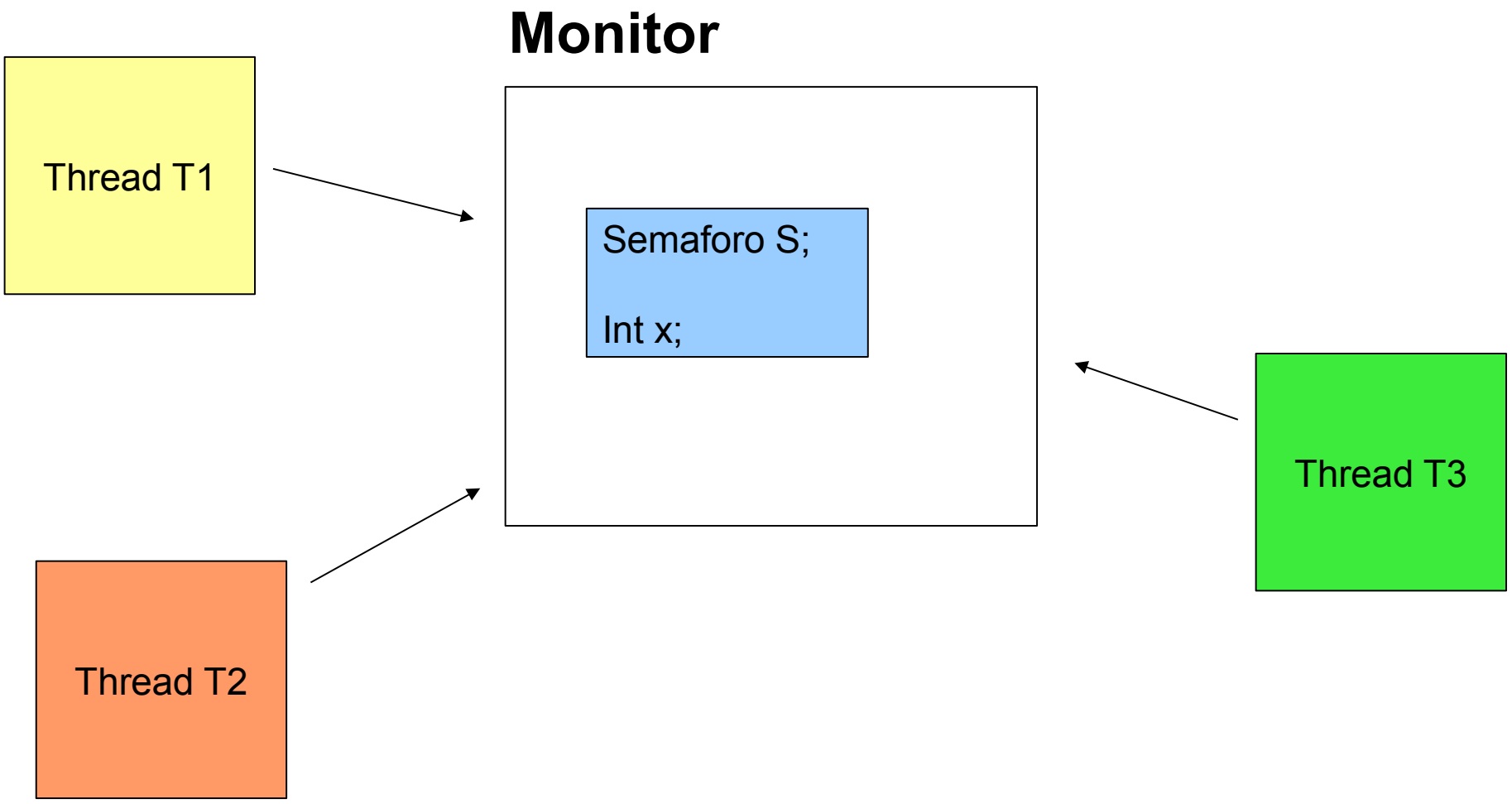
up():
incrementa il contatore e, se la coda non è vuota, riattiva il primo thread in coda

Le operazioni vanno fatte controllando race condition...

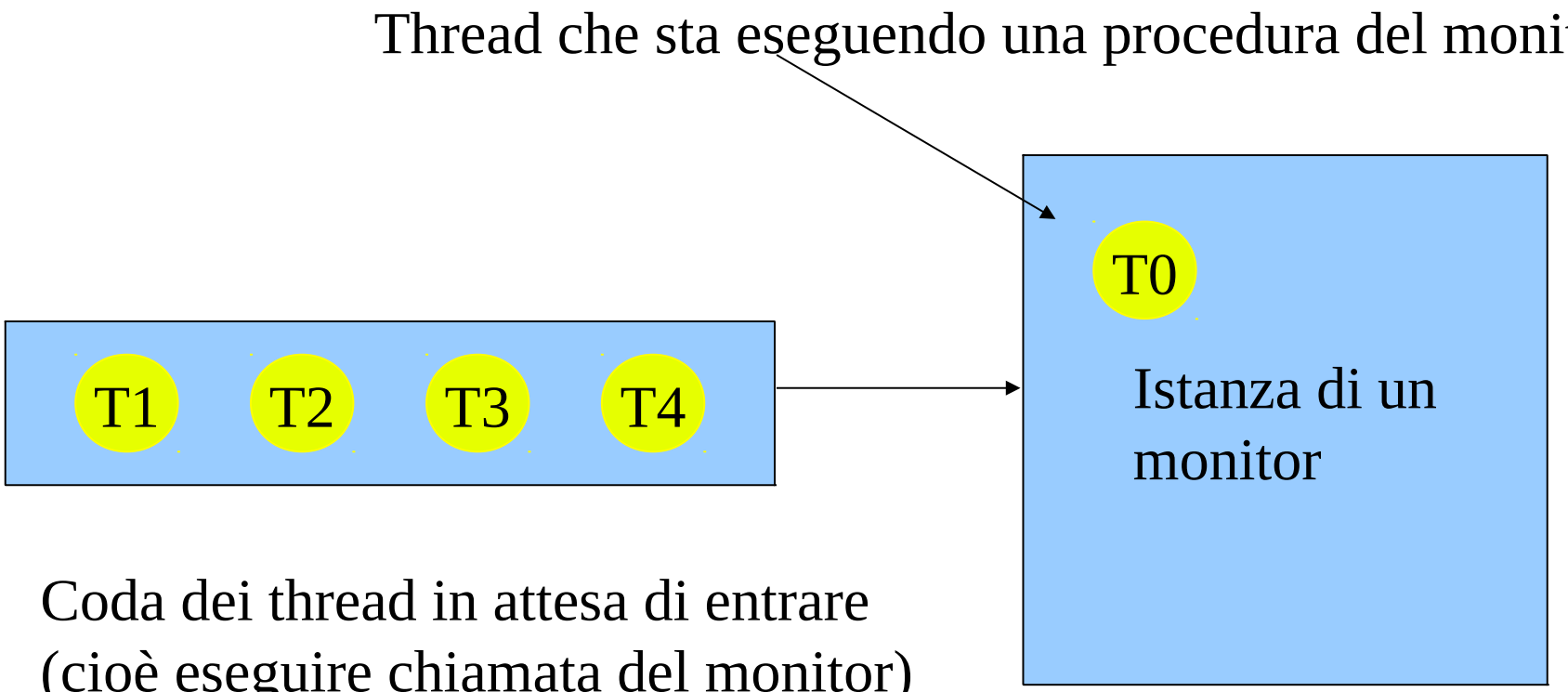
Thread e Semaforo: Spazio di Indirizzamento
Un'istanza di un semaforo è un dato condiviso



Semafori e Monitor



Mutua Esclusione



Basta la coda di ingresso?

La coda di ingresso garantisce la serializzazione

Tuttavia se un thread è entrato nel monitor ma non riesce ad effettuare un'operazione perchè ad esempio una qualche condizione non è verificata (es. la condizione x==0 su una variabile privata del monitor) cosa deve fare?

- **uscire e rientrare?**
- **sospendersi in attesa della condizione con priorità maggiore rispetto a quelli che sono in attesa fuori?**

La seconda strategia si implementa con le variabili condizione

Variabili condizione (condition)

Strumento di sincronizzazione associato ai lock:
consente la sospensione dei thread in attesa che sia soddisfatta una condizione logica.
Ad ogni condition viene associata una coda per la sospensione dei thread.

La variabile condizione non ha uno stato: rappresenta solo una coda di thread.

Operazioni fondamentali:

wait ***(sospensione)***
signal ***(risveglio)***

Una variabile condizione C viene creata e inizializzata con attributi nel modo seguente:

p_thread_cond_t C;
p_thread_cond_ init (&C,attr);

attr è l'indirizzo della struttura che contiene eventuali attributi

Le operazioni fanno sempre riferimento ad un mutex

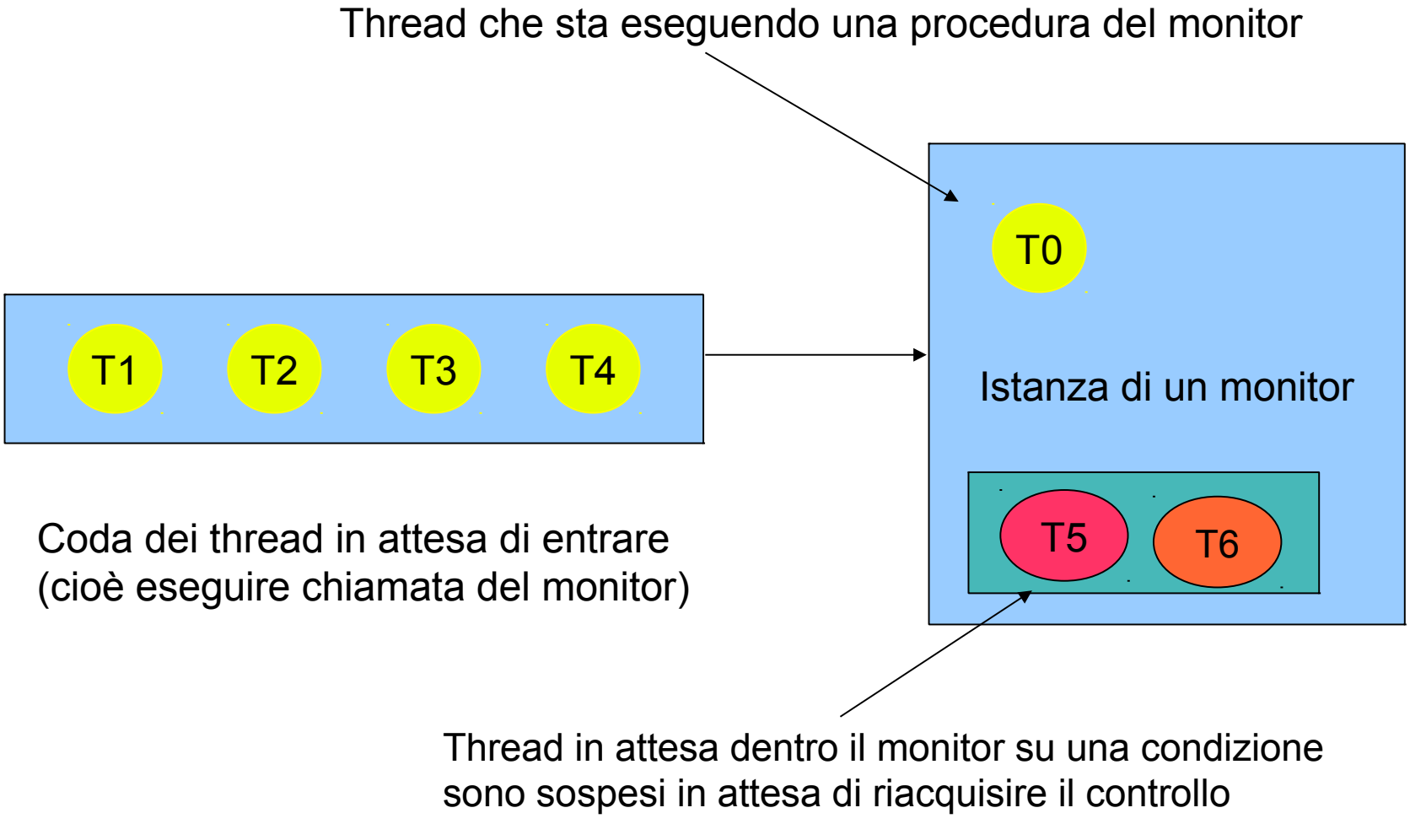
pthread_cond_wait(&C,&m):
sospensione ed ingresso nella coda C nel monitor associato al mutex m

pthread_cond_signal(&C,&m):
segnalazione ad un thread sospeso nella coda C del mutex m

pthread_cond_broadcast(&C,&m):
notifica a tutti i thread sospesi nella coda C del mutex m

Le operazioni vanno eseguite dopo aver acquisto un lock

Variabili Condizione



Semantica di signal

Il thread T1 esegue signal e il thread T2 è in attesa:
non vogliamo due thread in esecuzione (deve valere mutua esclusione)

- **Il thread segnalato T2 viene trasferito dalla coda associata alla variabile condizione alla entry_queue e potrà rientrare nel monitor una volta che T1 l'abbia rilasciato.**

- **Poiché altri processi possono entrare nel monitor prima di T2 è necessario che quando T2 rientra nel monitor controlli nuovamente la condizione**

- **Si usa il pattern:**

  while(!B) wait (cond);

Esempio: Produttore/consumatore con buffer unbounded

Produttore

```
while(true){
  produce x
  lock(mtx)
  inserisci x
  signal(C)
  unlock(mtx)
}
```

Consumatore

```
while(true){
  produce x
  lock(mtx)
  while (empty)
     wait(C,mtx);
  estrai y
  unlock(mtx);
  elab y
}
```

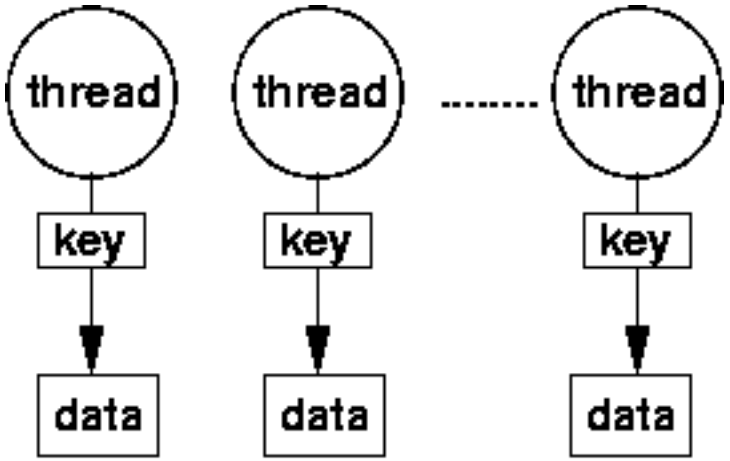
Strutture dati

```
struct node {
int info;
struct node * next;
};

/* testa lista inizialmente vuota*/
struct node * head=NULL;

/* mutex e cond var*/
pthread_mutex_t mtx=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
```

```
static void* consumer (void*arg) {
struct node * p;
 while(true) {
      pthread_mutex_lock(&mtx);
      while (head == NULL){
      pthread_cond_wait(&cond, &mtx);
      printf(”Waken up!\n”); fflush(stdout);
 }
 p=estrai();
 pthread_mutex_unlock(&mtx);
 /* elaborazione p … … */
 }
}
```

```
static void* producer (void*arg) {
struct node * p;

for (i=0; i<N; i++) {
      p=produci();
      pthread_mutex_lock(&mtx);
      inserisci(p);
      pthread_cond_signal(&cond);
      pthread_mutex_unlock(&mtx);
}

}
```

TSD: Dati Privati

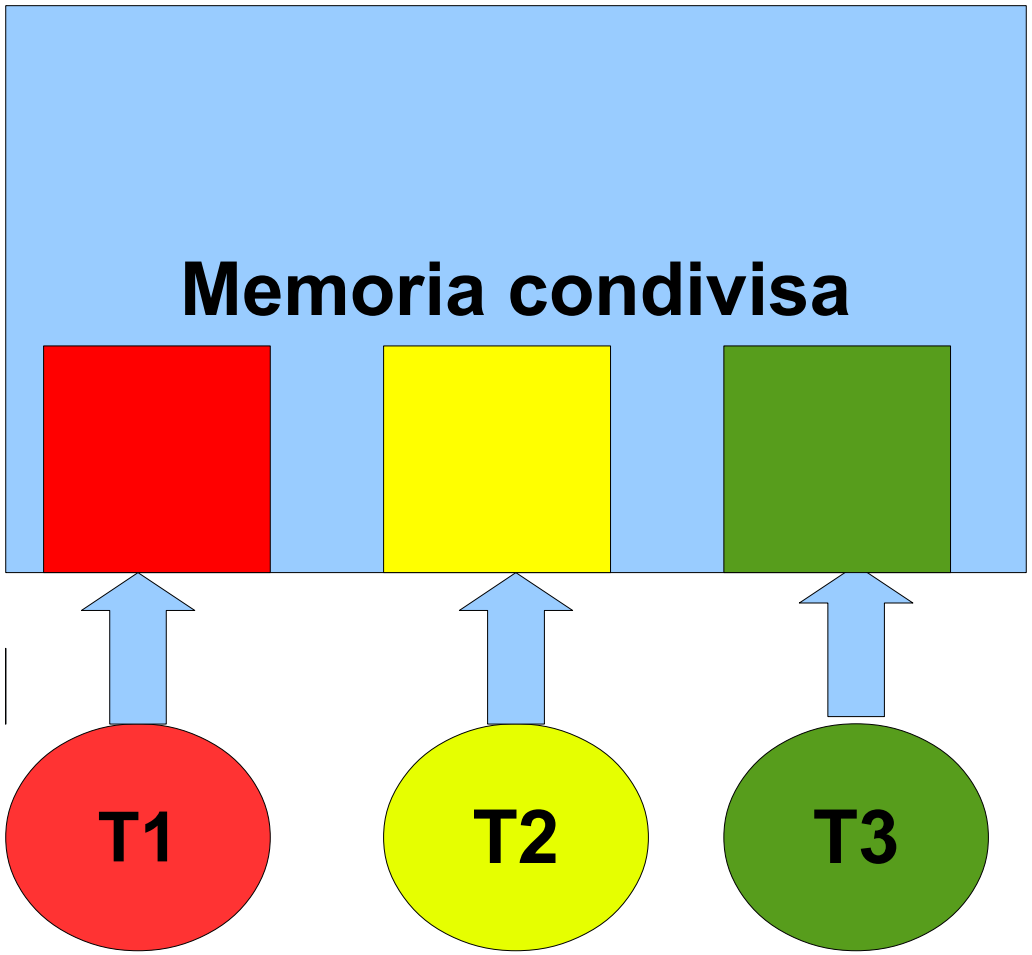


Dati Privati di un Thread

- I thread condividono il segmento dati
- Complementarietà rispetto ai processi
  - thread
    - semplice scambiare dati con altri thread
    - appositi meccanismi per disporre di dati privati
      - TSD
  - processi
    - semplice disporre di dati privati del processo
    - appositi meccanismi per dialogare con altri processi
      - IPC

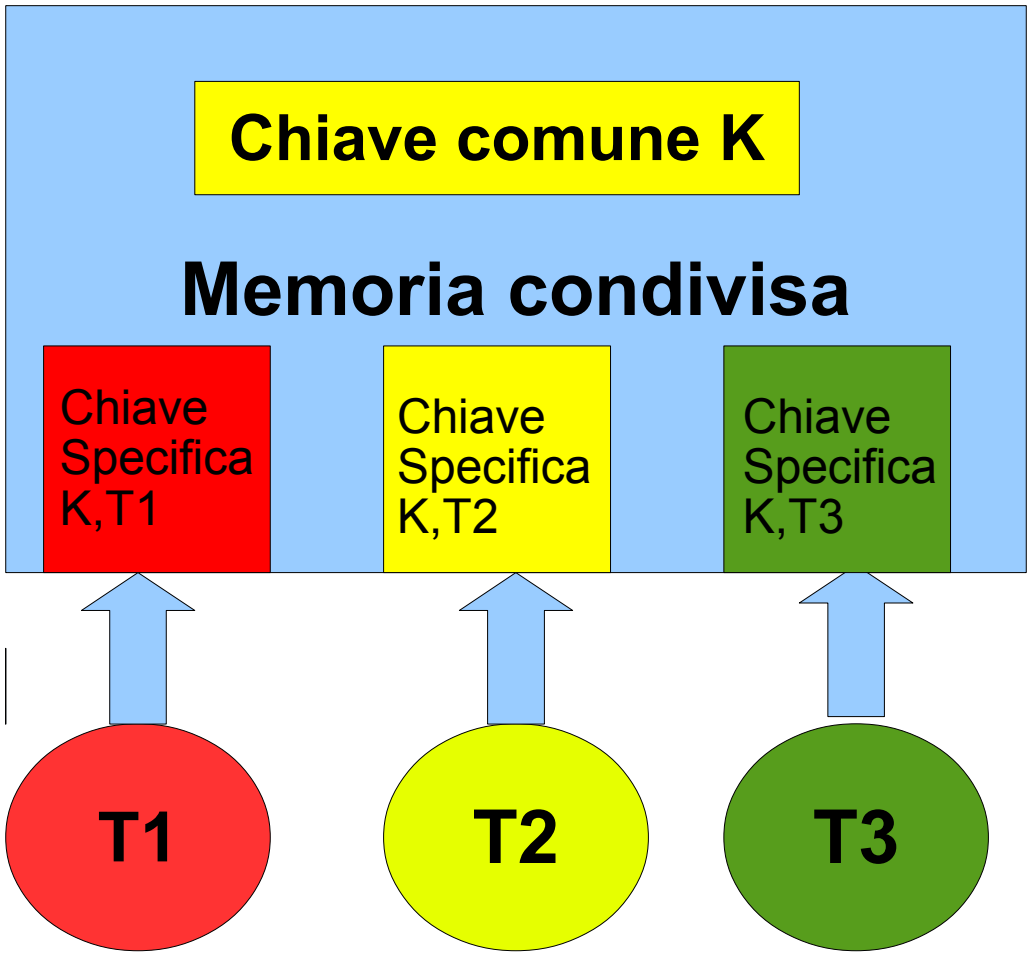
Thread Specific Data (TSD)

- Può servire disporre di dati che siano globalmente visibili all'interno di un singolo thread ma distinti da thread a thread
- Ogni thread possiede un'area di memoria privata, la TSD area, indicizzata da chiavi
- La TSD area contiene associazioni tra le chiavi ed un valore di tipo `void *`
  - diversi thread possono usare le stesse chiavi ma i valori associati variano di thread in thread
  - inizialmente tutte le chiavi sono associate a `NULL`


Memoria condivisa
T1
T2
T3

Funzioni per Thread Specific Data

- `int pthread_key_create()`
  - per creare una chiave TSD
- `int pthread_key_delete()`
  - per deallocare una chiave TSD
- `int pthread_setspecific()`
  - per associare un certo valore ad una chiave TSD
- `void *pthread_getspecific()`
  - per ottenere il valore associato ad una chiave TSD


Chiave comune K
Memoria condivisa
Chiave Specifica K,T1
Chiave Specifica K,T2
Chiave Specifica K,T3
T1
T2
T3

TSD: `pthread_key_create()`

```
#include <pthread.h>
int pthread_key_create(          pthread_key_t *key,
                                 void (*destr_function) (void *));
```

- Alloca una nuova chiave TSD e la restituisce tramite key
- destr_function, se ≠ NULL, é il puntatore ad una funzione "distruttore" da chiamare quando il thread esegue pthread_exit()
- restituisce 0 in caso di successo oppure un codice d'errore ❹0

pthread_setspecific() & pthread_getspecific()

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key,
                        const void *pointer);
```

- Cambia il puntatore TSD associato ad una data key per il thread chiamante
- restituisce 0 in caso di successo oppure un codice d'errore $\neq 0$

```
void * pthread_getspecific(pthread_key_t key);
```

- Restituisce il puntatore TSD associato ad una data key per il thread chiamante oppure `NULL` in caso di errore

Main:

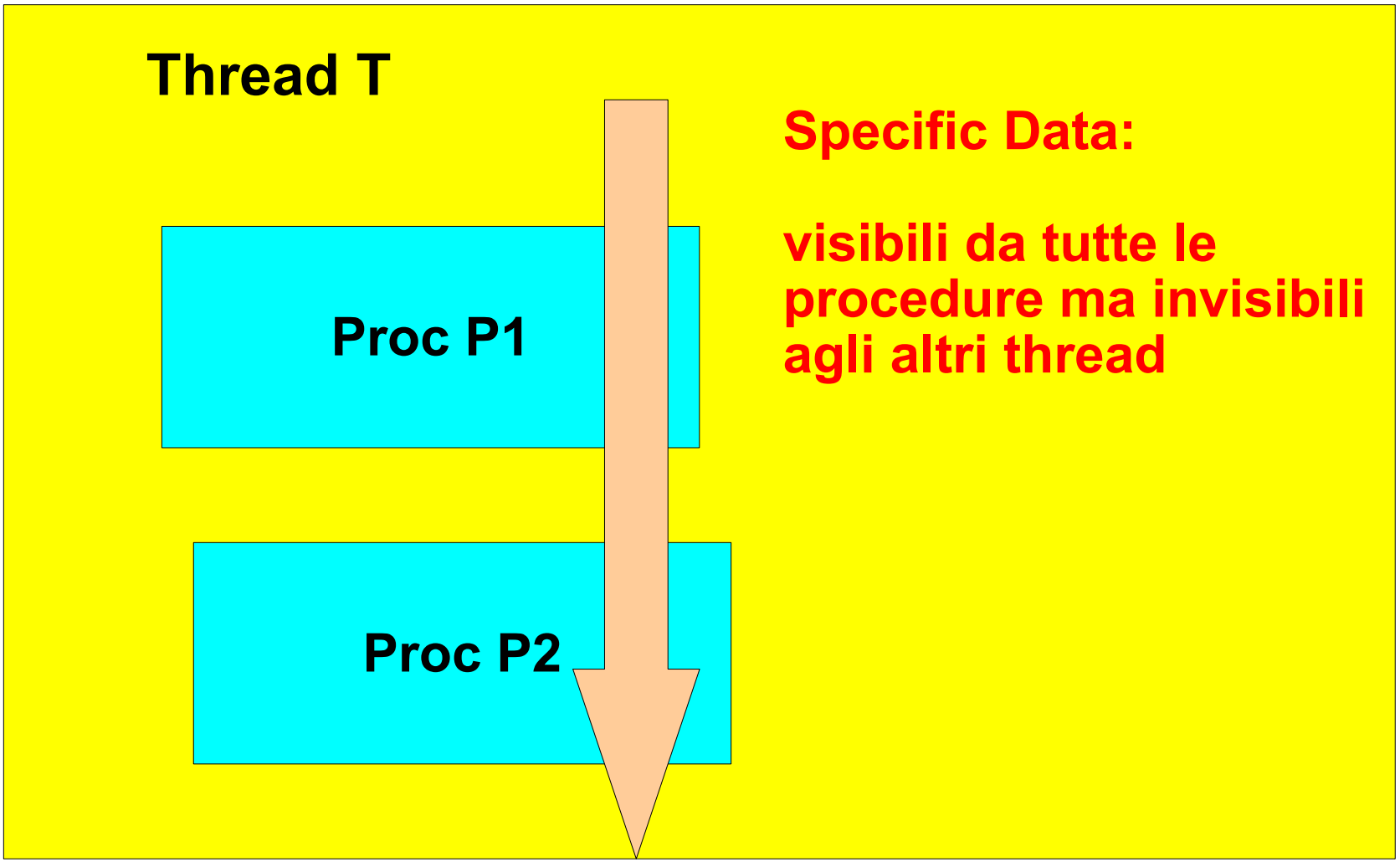
Creo chiave comune K

Thread:

Vede la chiave comune K

Genera chiave specifica associata a K e ad un puntatore a dati privati

Usa la chiave specifica come globale


Thread T
Proc P1
Proc P2
Specific Data:
visibili da tutte le procedure ma invisibili agli altri thread

tsd.c(1)

```
#include …
static pthread_key_t thread_log_key; /* tsd key per thread */

void write_to_thread_log (const char* message); //Scrive log
void close_thread_log (void* thread_log); //Chiude file log
void* thread_function (void* args);       //Eseguita dai thread
int main() {
  // Crea una chiave da associare al log Thread-Specific
  // Crea 5 worker thread per il lavoro
  // Aspetta che tutti finiscano
  return 0;
}
…
```

tsd.c (2)

```
…
int main() {
  int i;
  pthread_t threads[5];
  // Crea una chiave da associare al punt. TSD al log file
  pthread_key_create(&thread_log_key, close_thread_log);

  for (i = 0; i < 5; ++i) // wroker thread
    pthread_create(&(threads[i]), NULL, thread_function,
   NULL);

  for (i = 0; i < 5; ++i) // Aspetta che tutti finiscano
    pthread_join (threads[i], NULL);
  return 0;
}
…
```

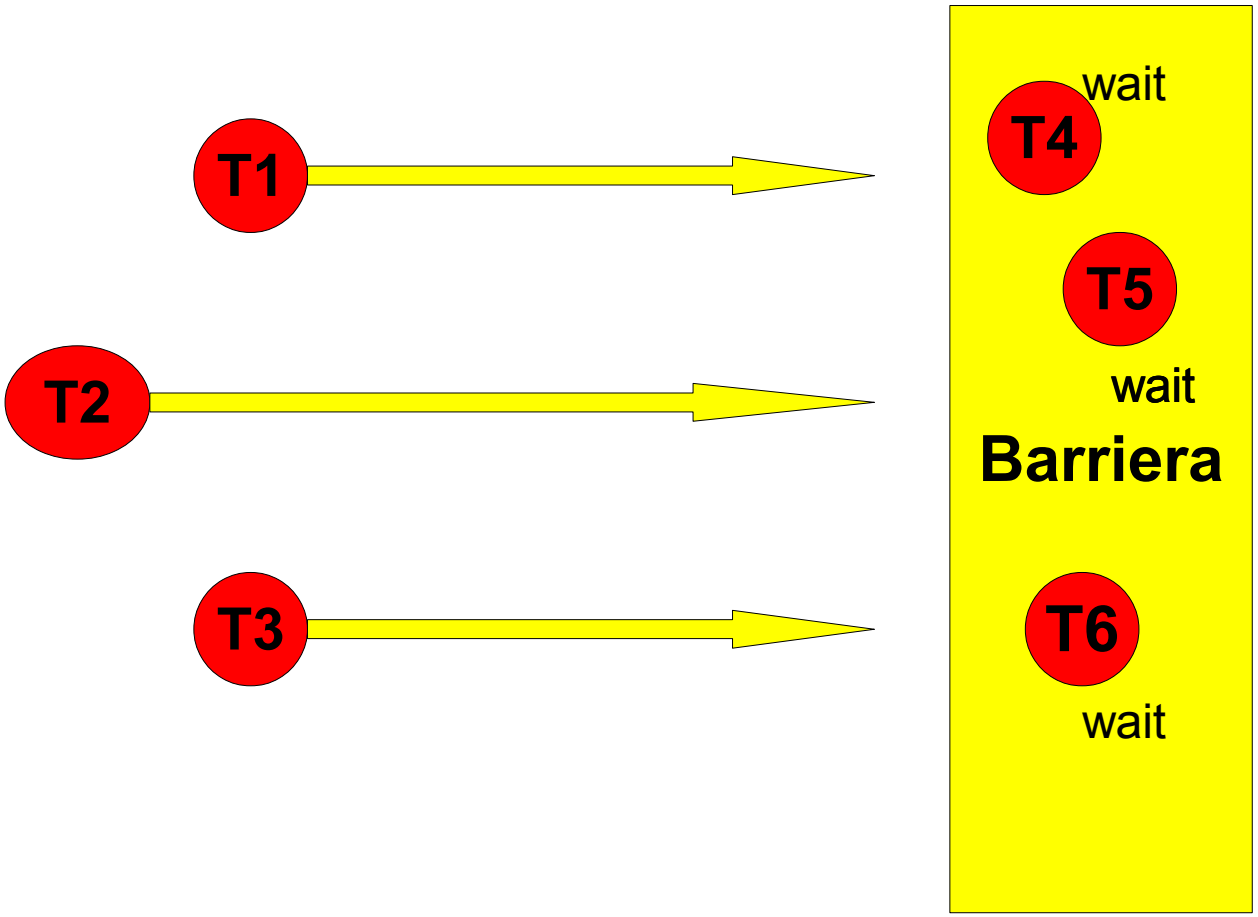
tsd.c (3)

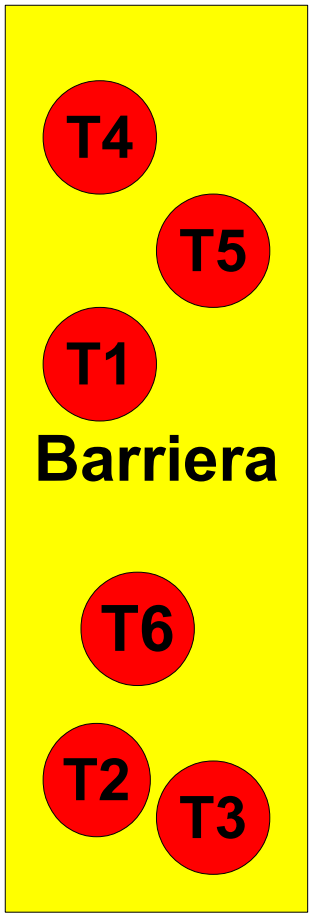
```
…
void write_to_thread_log (const char* message) {
  FILE* thread_log = (FILE*)pthread_getspecific(thread_log_key);
  fprintf (thread_log, "%s\n", message);
}
void close_thread_log (void* thread_log) {
  fclose ((FILE*) thread_log);
}
void* thread_function (void* args) {
  char thread_log_filename[20];
  FILE* thread_log;
  sprintf(thread_log_filename,"thread%d.log",(int)pthread_self ());

  thread_log = fopen (thread_log_filename, "w");
  /* Associa la struttura FILE TSD a thread_log_key.  */
  pthread_setspecific (thread_log_key, thread_log);

  write_to_thread_log ("Thread starting.");
  /* Fai altro lavoro qui... */  return NULL;
}
```

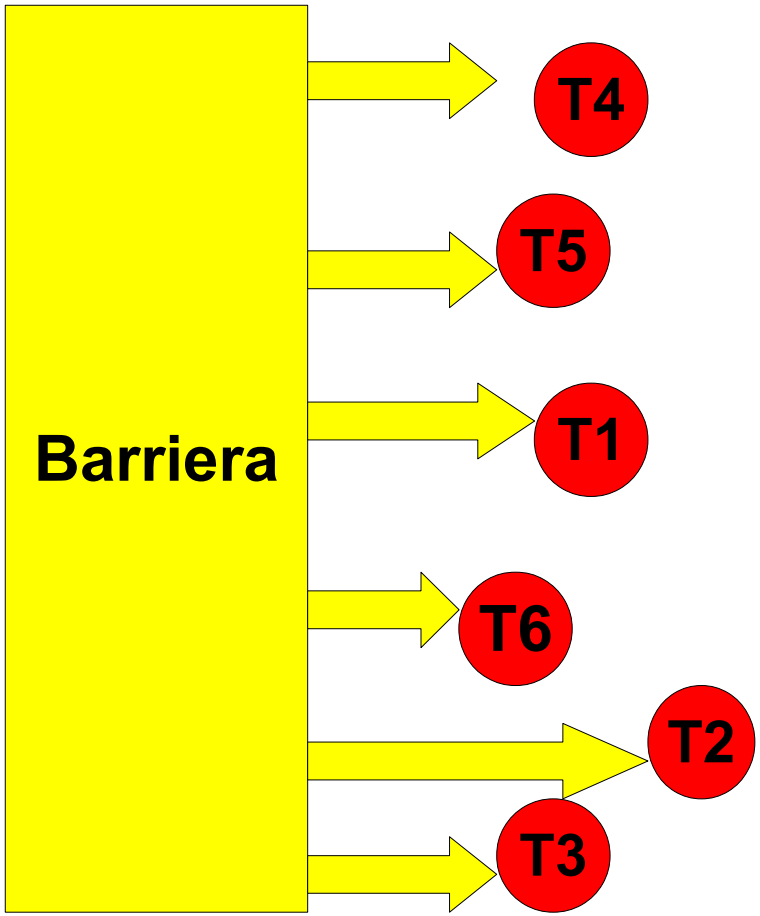
Barriere

Barriere nei Pthread

- Le barriere sono un meccanismo di sincronizzazione utilizzato ad esempio nel calcolo parallelo
- Idea: una barriera rappresenta un punto di sincronizzazione per N thread
  - I thread che arrivano alla barriera aspettano gli altri
  - Solo quando tutti gli N thread arrivano alla barriera allora possono proseguire


T1
T2
T3
wait
T4
T5
wait
Barriera
T6
wait


T4
T5
T1
Barriera
T6
T2
T3


Barriera
T4
T5
T1
T6
T2
T3

Pthread_barrier

- Creazione di una barriera

  int pthread_barrier_init(pthread_barrier_t *barrier, pthread_barrierattr_t *barrier_attr, unsigned int count);

  oppure

  pthread_barrier_t barrier = PTHREAD_BARRIER_INITIALIZER(count);

- count=numero di thread da sincronizzare

Pthread_barrier

- Attesa su una barriera all'interno del codice di un thread:
- int pthread_barrier_wait(pthread_barrier_t *barrier);
- Quando N thread hanno eseguito wait sono tutti sbloccati e possono proseguire l'esecuzione!

Main:
creo barriera per N thread

Thread:

Chiamo wait appena raggiungo il punto di sincronizzazione

Schema di uso:
Fork e Join x N thread

- Main: creo barriera per N thread
- Spezzo il task da eseguire in N sottotask paralleli
- Creo i thread che eseguono i task
- Ogni thread esegue un task e poi aspetta gli altri
- Quando tutti hanno terminato si prosegue con il calcolo

Esercizio

Come si implementa una
BARRIERA
usando mutex e condition?

Esercizio

Come si implementa un SEMAFORO GENERALE usando mutex e condition?

Esercizio

Come si implementa un
MONITOR
usando mutex e condition?