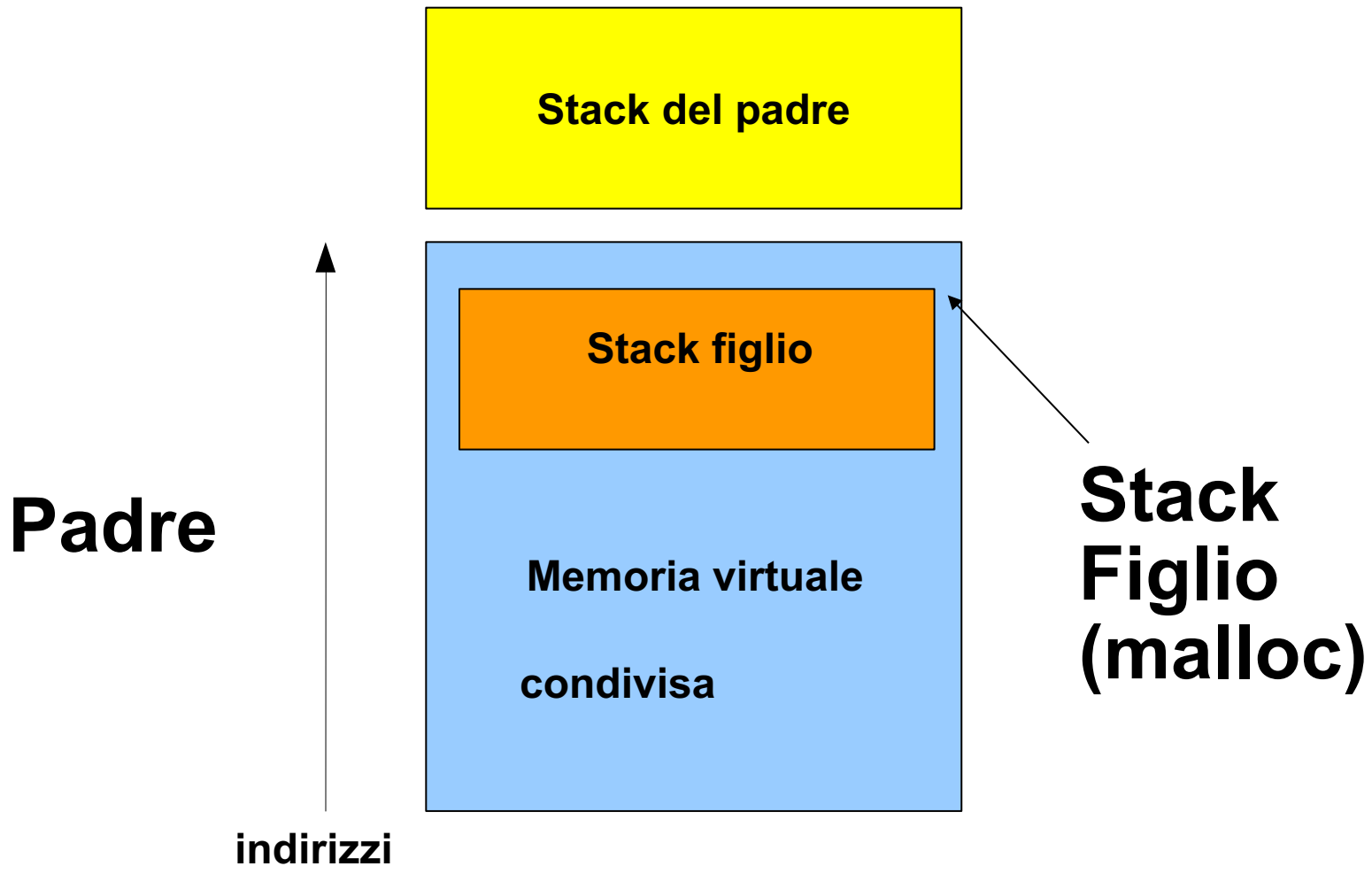
Thread in C

Clone

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

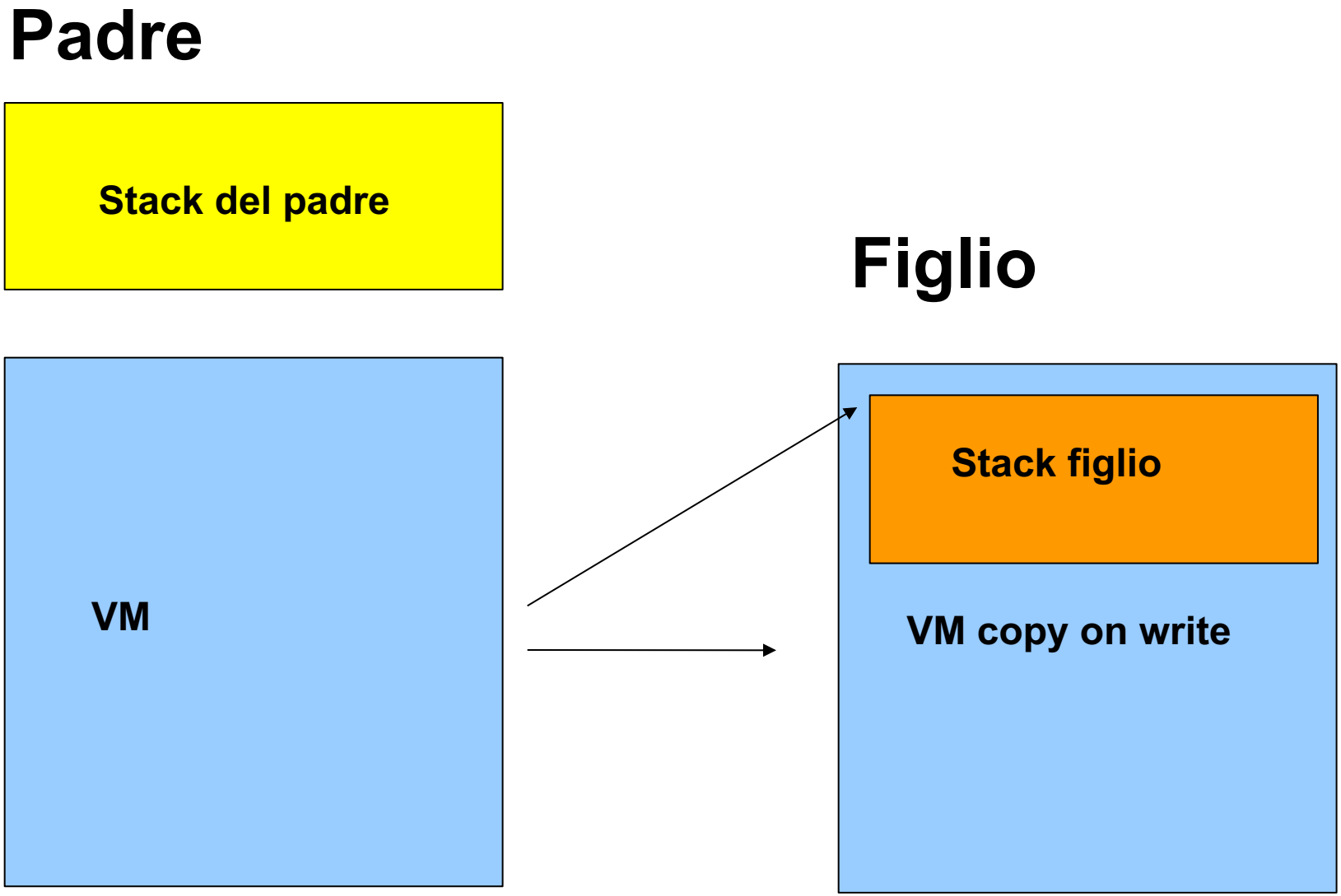
Clone


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

```
….
#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!

Fork via clone



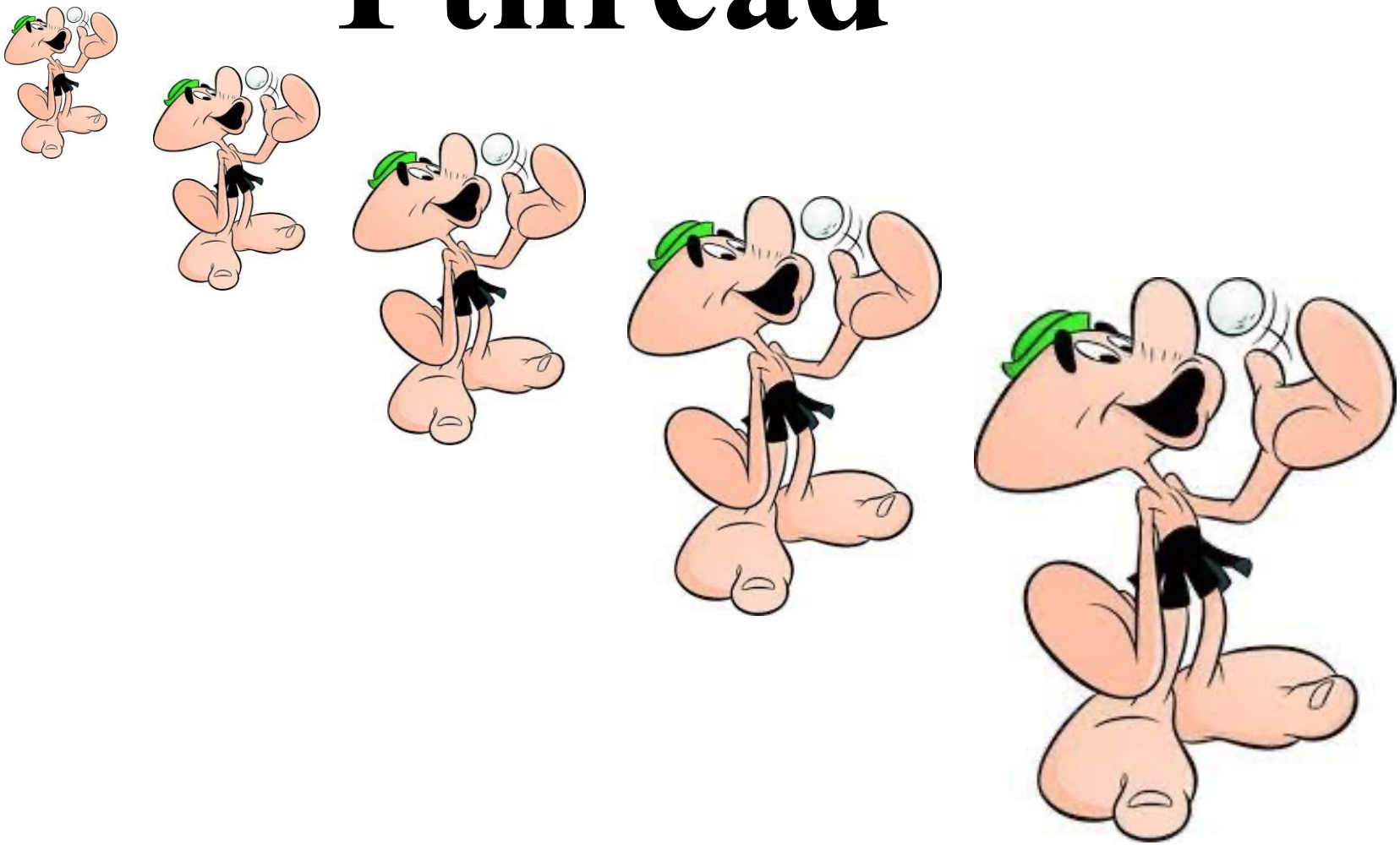
```
....
#define STACK_SIZE 65536

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

Pthread



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 (Linux Thread, NPTL)

  I thread hanno tipo opaco pthread_t

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

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):

Attesa della terminazione

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.

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).

Sincronizzazione

Libreria pthread:

- mutex (semaforo binario)
- Condition variable (monitor)

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

Mutex e Monitor

La libreria pthread fornisce le seguenti strutture dati concorrenti:

- mutex (semaforo binario)
- condition variable (monitor)

Inoltre in Linux si possono usare 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:

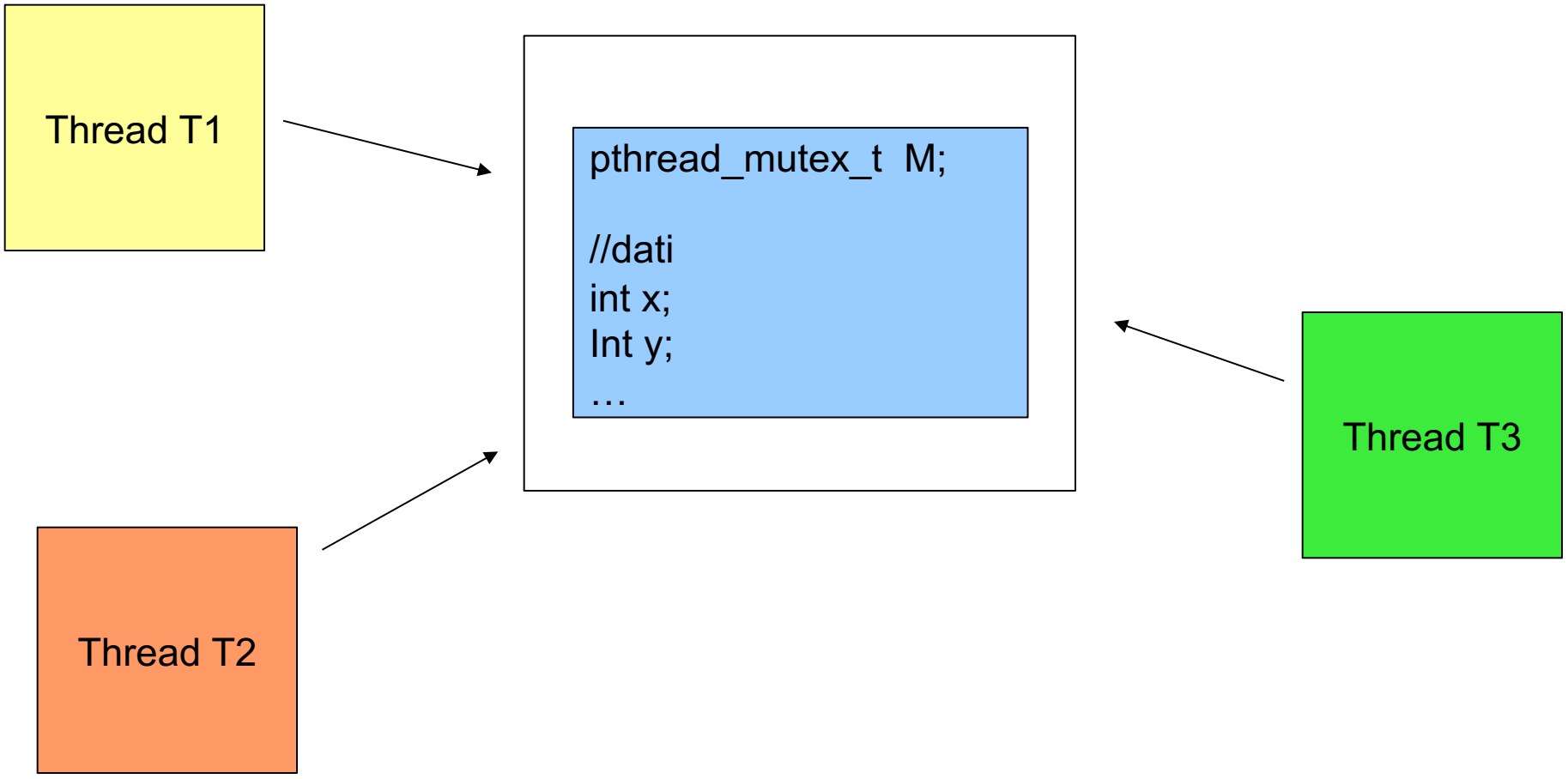
pthread_mutex_lock (pthread_mutex_t *M)

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

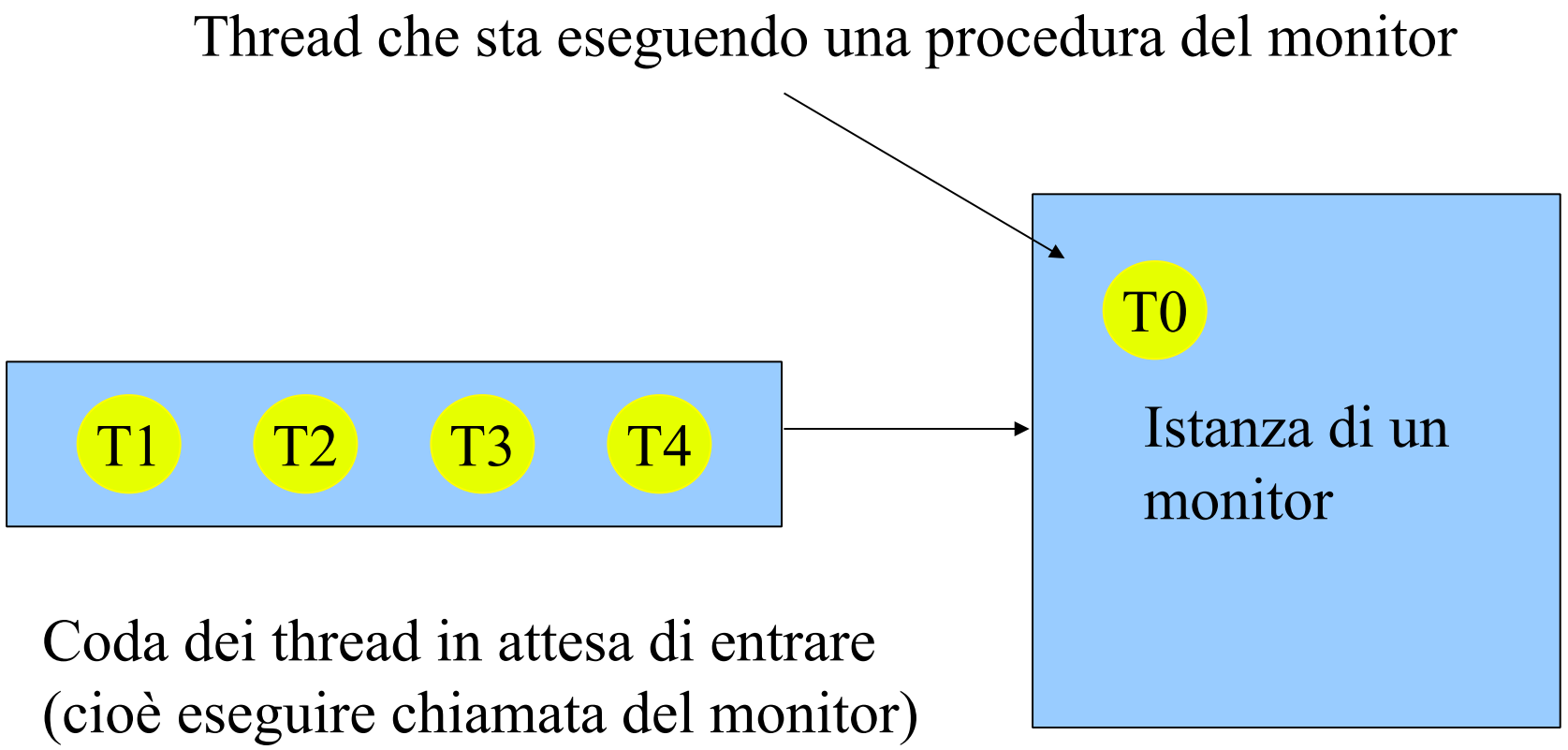
pthread_mutex_unlock (pthread_mutex_t *M)

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

Verso i Monitor: Proteggere dati con i mutex



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

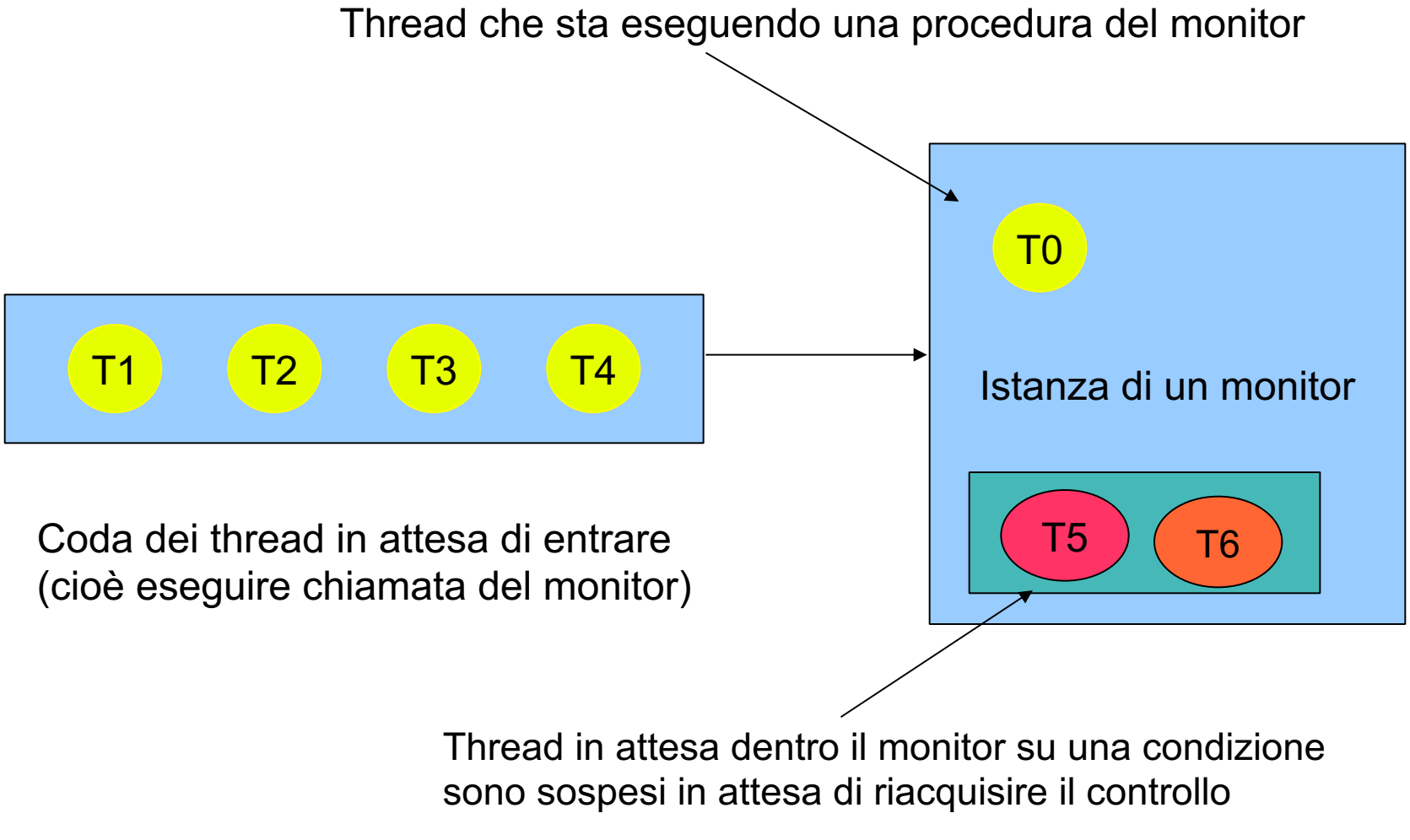
Condition Variable

Strumento di sincronizzazione associato ai lock

Consente la sospensione dei thread in attesa che sia soddisfatta

Ad ogni condition variable viene associata una coda di attesa per i thread in attesa

Condition Variable



Condition Variable nei Pthread

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

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 condition alla entry_queue

T2 potrà rientrare nel monitor solo dopo l'uscita di T1

Attenzione!

Altri thread potrebbero entrare nel monitor prima di T2 e modificare le variabili condivise

La condizione E utilizzata per entrare in attesa andrebbe ricontrollata dopo l'ingresso di T2 nel monitor

Per evitare questo tipo di situazioni l'operazione wait viene quindi solitamente usata con questo pattern:

```
while (E)  pthread_cond_wait(&C,&m);
```

In questo modo si continua l'esecuzione nel monitor solo quando E diventa falsa

Esempio: Produttore/consumatore

Produttore

```
while(true){
  //produci x
  lock(mtx)
  //sc:
  // inserisci x nel buffer
  signal(C)
  unlock(mtx)
}
```

Consumatore

```
while(true){
  lock(mtx)
  while (empty) wait(C,mtx);
  //sc:
  //  estrai y dal buffer
  unlock(mtx);
  //elabora 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;
```

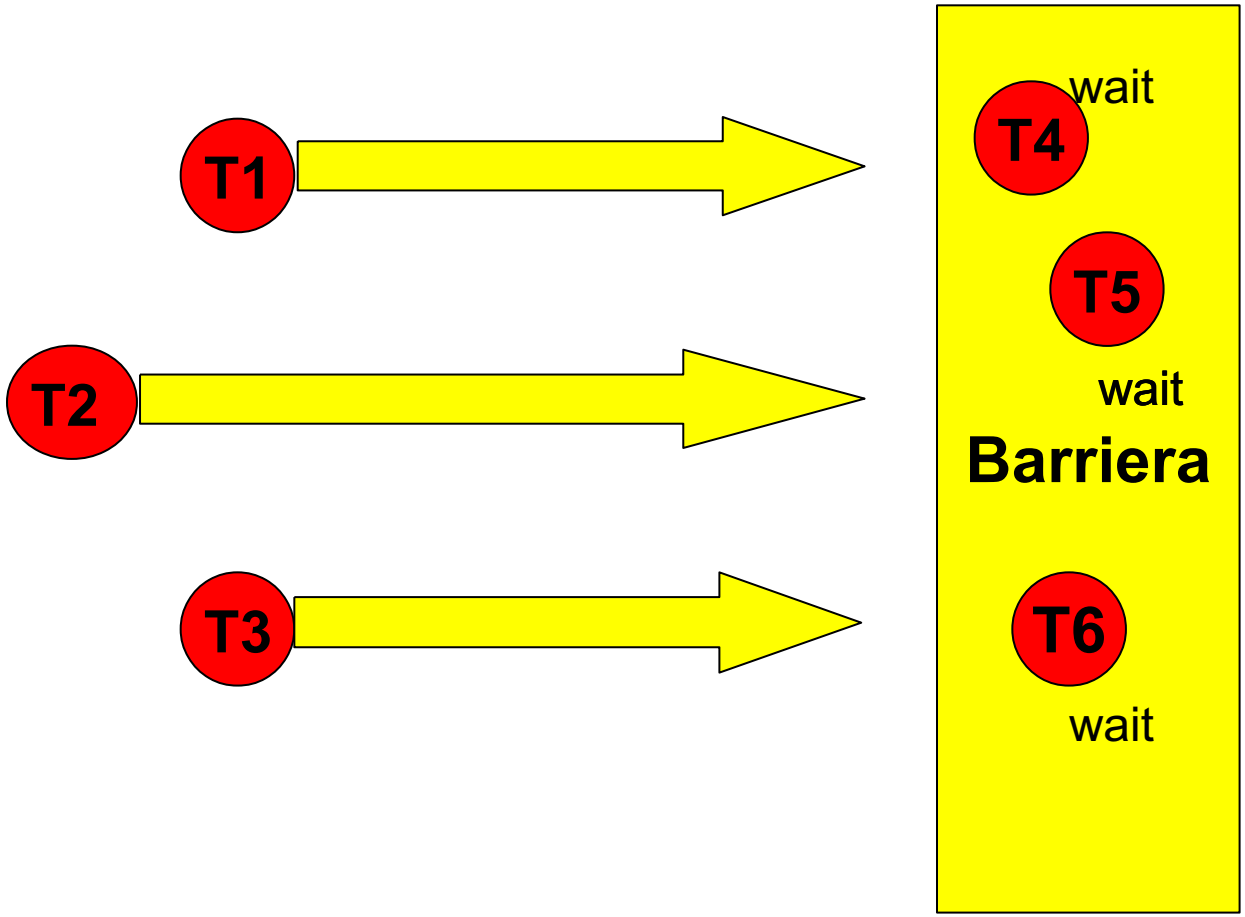
```
static void* consumer (void*arg) {
struct node * p;
 while(true) {
      pthread_mutex_lock(&mtx);
      while (head == NULL){
      pthread_cond_wait(&cond, &mtx);
      printf(”Waken up!\n”); fflush(stdout);
 }
 p=estrai();
 pthread_mutex_unlock(&mtx);
 /* elaborazione p … … */
  }
}
```

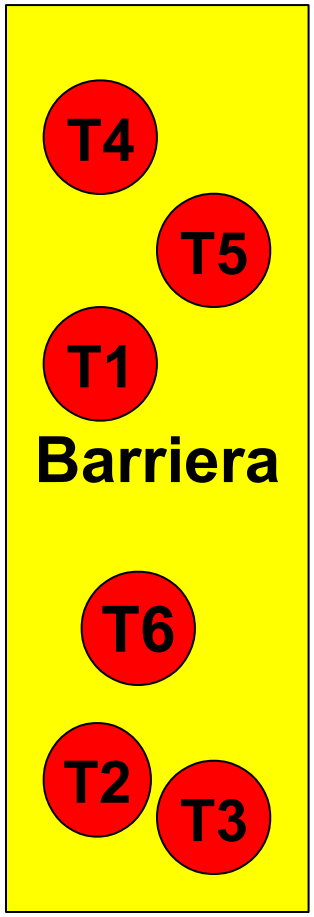
```
static void* producer (void*arg) {
struct node * p;

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

}
```

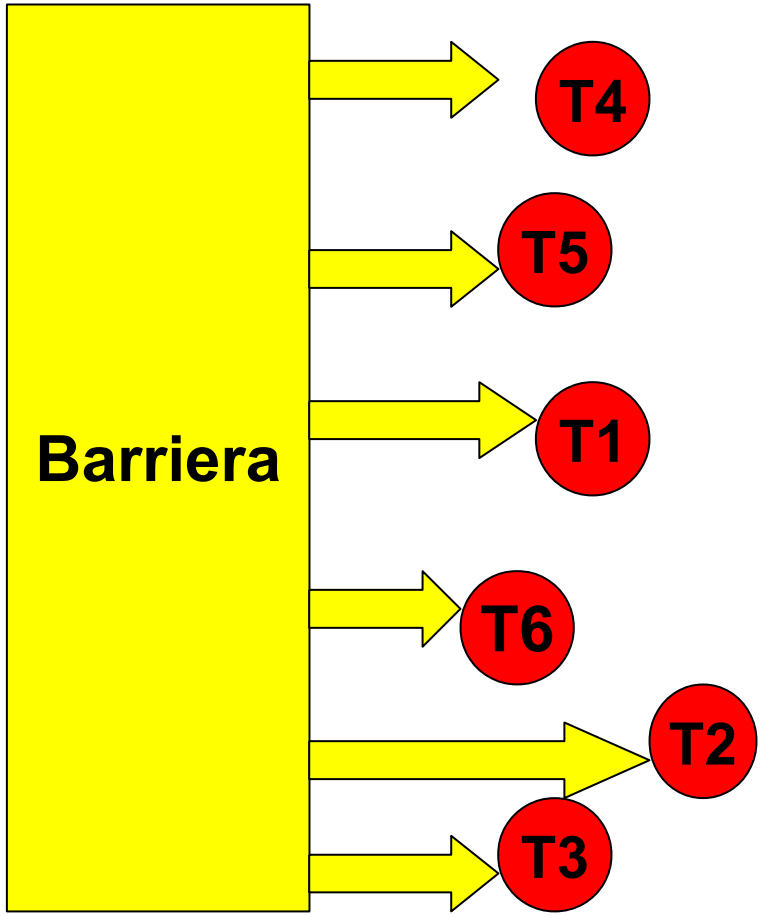
Barriere di Sincronizzazione

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
T4
T5
T6
wait
wait
wait
Barriera


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

Barriere di Memoria (memory barrier)

Memory Barrier

Una memory barrier è una classe di istruzioni forniti dale architetture per imporre un certo ordine nelle operazioni di lettura e scrittura in memoria.

In particolare le operazioni di scrittura prima di una certa barriera vengono consolidate prima di eseguire quelle successive alla barriera.

```
X=1;   // write(X,1) potrebbe essere nel buffer di scrittura
memory_fence();
if (Y==0) …          // X ha adesso valore 1 in memoria
```

Dove le trovo?

Extensions to the C Language Family

https://gcc.gnu.org/onlinedocs/gcc-4.6.2/gcc/
sez. 6.5 Atomic builtins:

__sync_synchronize (...)

This built-in issues a full memory barrier: