

POSIX Thread

- I Thread sono stati standardizzati. IEEE POSIX 1003.1c (1995) specifica l'interfaccia di programmazione (Application Program Interface - API) dei thread. I thread POSIX sono noti come **Pthread**.
- Le API per Pthread distinguono le funzioni in 3 gruppi:
 - **Thread management:** funzioni per creare, eliminare, attendere la fine dei pthread
 - **Mutexes:** funzioni per supportare un tipo di sincronizzazione semplice chiamata "mutex" (abbreviazione di **mutua esclusione**). Comprende funzioni per creare e eliminare la struttura per mutua esclusione di una risorsa, **acquisire** e **rilasciare** tale risorsa.
 - **Condition variables:** funzioni a supporto di una **sincronizzazione** più complessa, **dipendente** dal **valore** di **variabili**, secondo i modi definite dal programmatore. Comprende funzioni per creare e eliminare la struttura per la sincronizzazione, per **attendere** e **segnalare** le modifiche delle variabili.

POSIX Thread API

Convenzione sui nomi delle funzioni:

Gli identificatori della libreria dei Pthread iniziano con **pthread_**

pthread_	indica gestione dei pthread in generale
pthread_attr_	funzioni per gestione proprietà dei thread
pthread_mutex_	gestione mutua esclusione
pthread_mutexattr_	proprietà delle strutture per mutua esclusione
pthread_cond_	gestione delle variabili di condizione
pthread_condattr_	proprietà delle variabili di condizione
pthread_key_	dati speciali dei thread

il file **pthread.h** contiene le definizioni dei pthread

API per creazione ed esecuzione di Pthread

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

- ❑ crea una thread e lo rende eseguibile, cioè lo mette a disposizione dello scheduler che prima o poi lo farà partire.
- ❑ Il primo parametro **thread** è un puntatore ad un identificatore di thread in cui verrà scritto l'identificatore del thread creato.
- ❑ Il terzo parametro **start_routine** è il nome (indirizzo) della procedura da fare eseguire dal thread. Deve avere come unico argomento un puntatore.
- ❑ Il secondo parametro seleziona caratteristiche particolari: può essere posto a NULL per ottenere il comportamento di default.
- ❑ Il quarto parametro è un puntatore che viene passato come argomento a start_routine. Tale puntatore solitamente punta ad una area di memoria allocata dinamicamente.

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

- ❑ termina l'esecuzione del thread da cui viene chiamata, immagazzina l'indirizzo retval, restituendolo ad un altro thread che attende la sua fine.
- ❑ Il sistema libera le risorse allocate al thread.
- ❑ Il caso del programma principale (main) è particolare.
- ❑ Se il main termina prima che i thread da lui creati siano terminati e non chiama la funzione pthread_exit, allora tutti i thread sono terminati. se invece il main chiama pthread_exit allora i thread possono continuare a vivere fino alla loro teminazione.

Identificatori dei Pthread

Il tipo di dato **pthread_t** è il tipo di dato che contiene l'identificatore univoco di un pthread.

Due funzioni utili a maneggiare gli identificatori di pthread sono :

pthread_t **pthread_self** (void);

restituisce l'identificatore del thread che la chiama.

int **pthread_equal** (pthread_t pthread1, pthread_t pthread2);

restituisce 1 se i due identificatori di pthread sono uguali

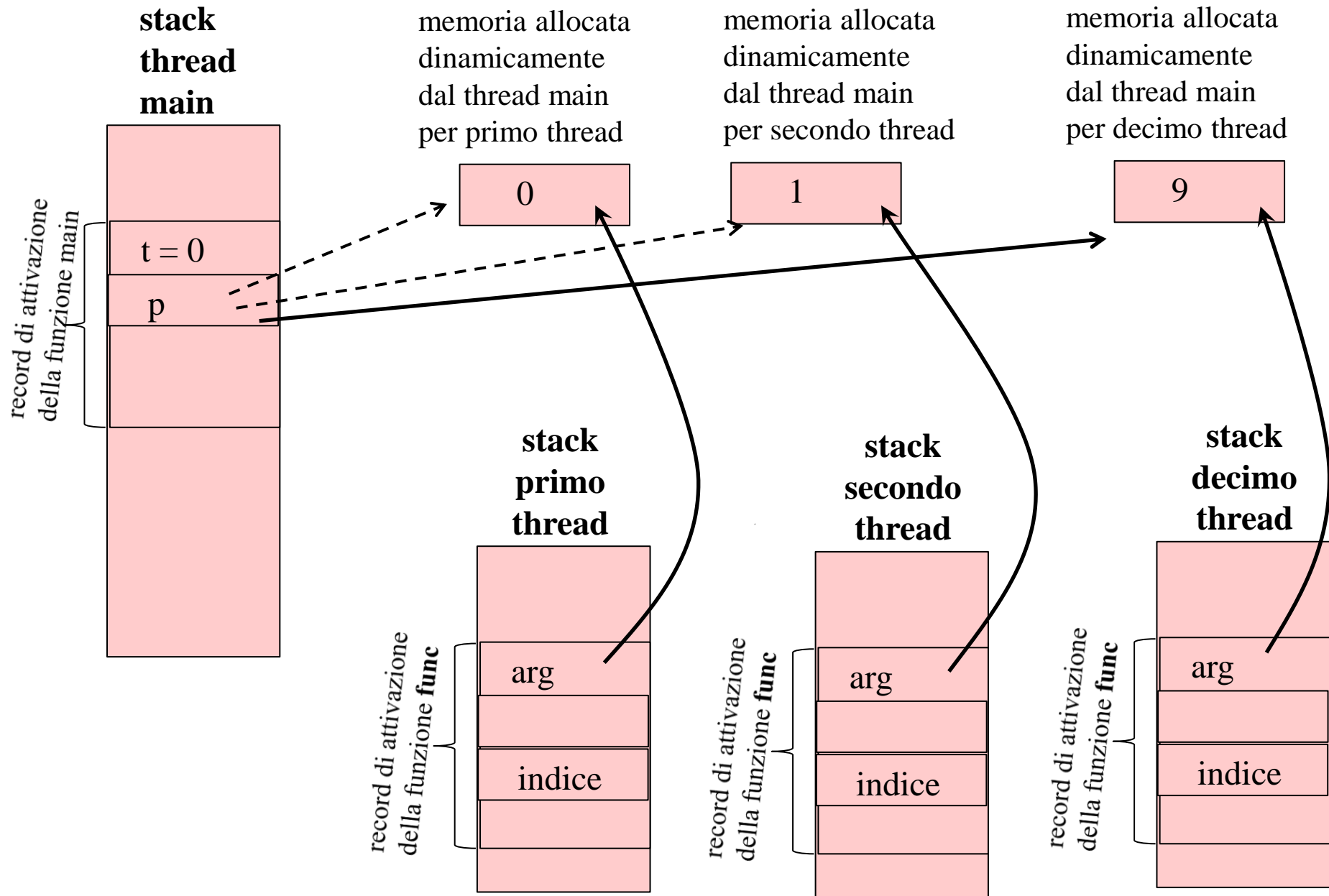
Creazione di un Pthread

La creazione di un Pthread viene effettuata chiamando la funzione:

```
int pthread_create ( pthread_t *threadId ,  const pthread_attr_t *attr,  
                    void *(*start_routine) (void *),  void *arg );
```

- threadId è l'indirizzo di una variabile di tipo *identificatore di Pthread* in cui verrà messo l'identificatore del Pthread creato.
- attr è l'indirizzo di una variabile di tipo pthread_attr_t in cui posso specificare alcune caratteristiche del Pthread da creare. Il parametro attr può essere NULL per usare le impostazioni di default.
- start_routine è il nome della funzione che implementa il Pthread da creare. Tale funzione formalmente deve prendere come argomento un puntatore generico void* e deve restituire un puntatore generico void*.
- arg è un puntatore generico void* che punta ad un'area di memoria in cui sono stati messi gli argomenti da passare alla funzione start_routine da eseguire. Il formato di tale area di memoria è user defined, cioè è concordato tra chi implementa il Pthread e chi implementa la chiamata al Pthread. Il parametro arg può essere NULL se voglio creare un Pthread senza passargli alcun argomento.
- Il risultato restituito dalla funzione pthread_create è un intero. Vale 0 se la funzione è riuscita a creare il Pthread. Vale !=0 se non è stato possibile creare il Pthread.
- Se diverso da 0, il valore restituito identifica il tipo di errore avvenuto.

Schema di Passaggio di argomenti a Pthread



Passaggio degli argomenti ad un Pthread

La funzione **pthread_create** richiede un puntatore per il passaggio dei parametri al Pthread che sta creando. Nel momento in cui il Pthread comincerà l'esecuzione avrà a disposizione questo puntatore.

Chi chiama la pthread_create deve allocare dinamicamente una struttura dati in cui collocare i parametri da passare al pthread. Il Pthread creato userà quest'area di memoria e poi la deallocherà.

Formalmente, la funzione che implementa il Pthread prende come argomento un puntatore generico void* e restituisce un puntatore generico.

Nell'esempio che segue, il main crea 10 Pthread passando a ciascuno un valore intero, ciascun pthread legge il valore intero ricevuto ricevuto e lo mette in una propria variabile indice.

codice del main

```
#define NUM_THREADS 10
pthread_t tid;
int t, rc;
int *p;

CORRETTO
for(t=0;t < NUM_THREADS;t++) {
    p = (int *) malloc( sizeof(int) );
    *p = t;
    rc = pthread_create( & tid , NULL, func, (void *) p );
}
```

SBAGLIATO

```
for(t=0;t < NUM_THREADS;t++) {
    rc = pthread_create( & tid , NULL, func, (void *) & t );
}
```

codice dei Pthread

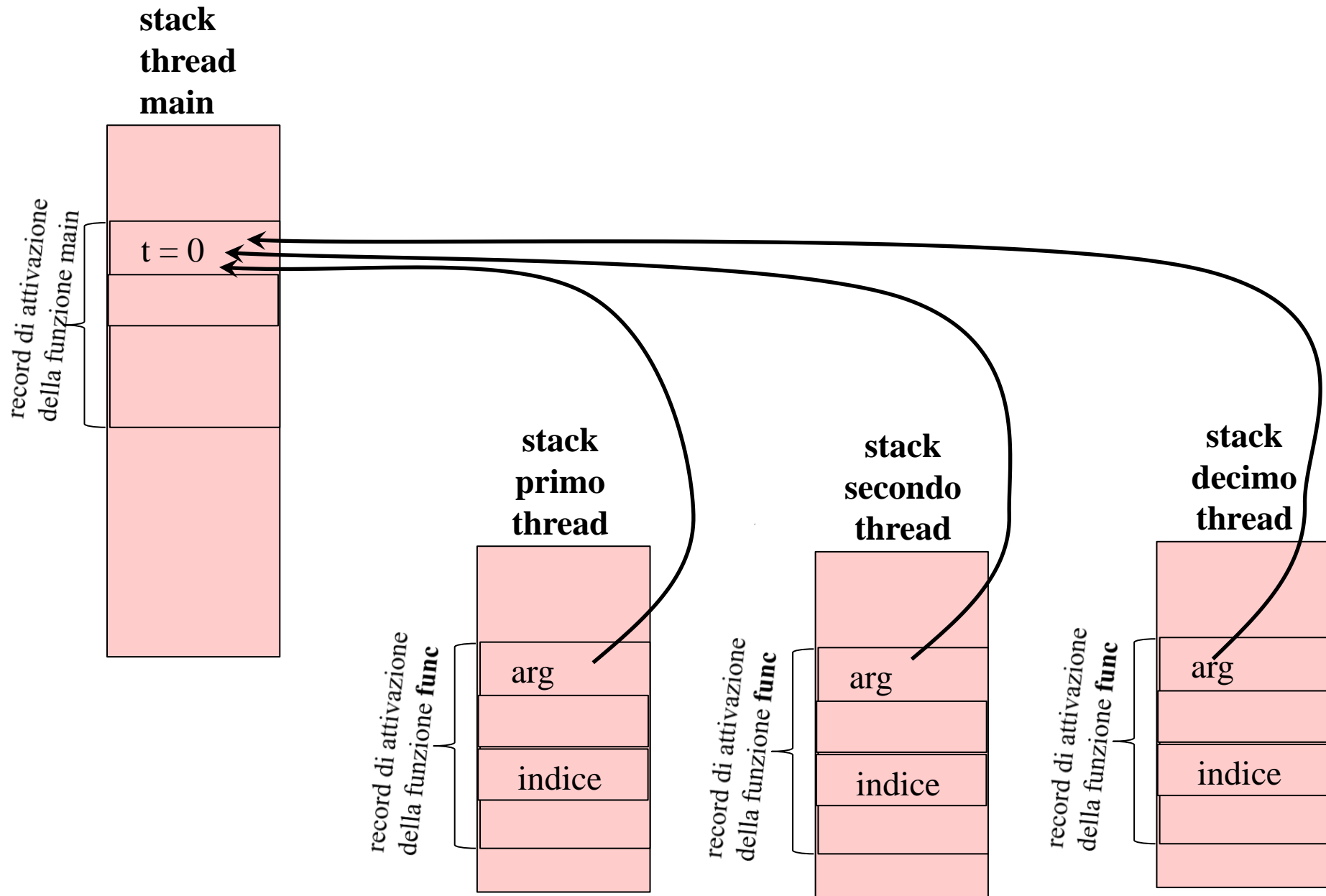
```
void * func( void *arg ) {
    int indice;
    /* uso arg come puntatore a int */
    indice = * ((int*)arg);

    /* rilascio memoria */
    free( arg ); /* non fare in caso sbagliato */

    printf(" ho ricevuto %i \n", indice );

    pthread_exit( NULL );
}
```

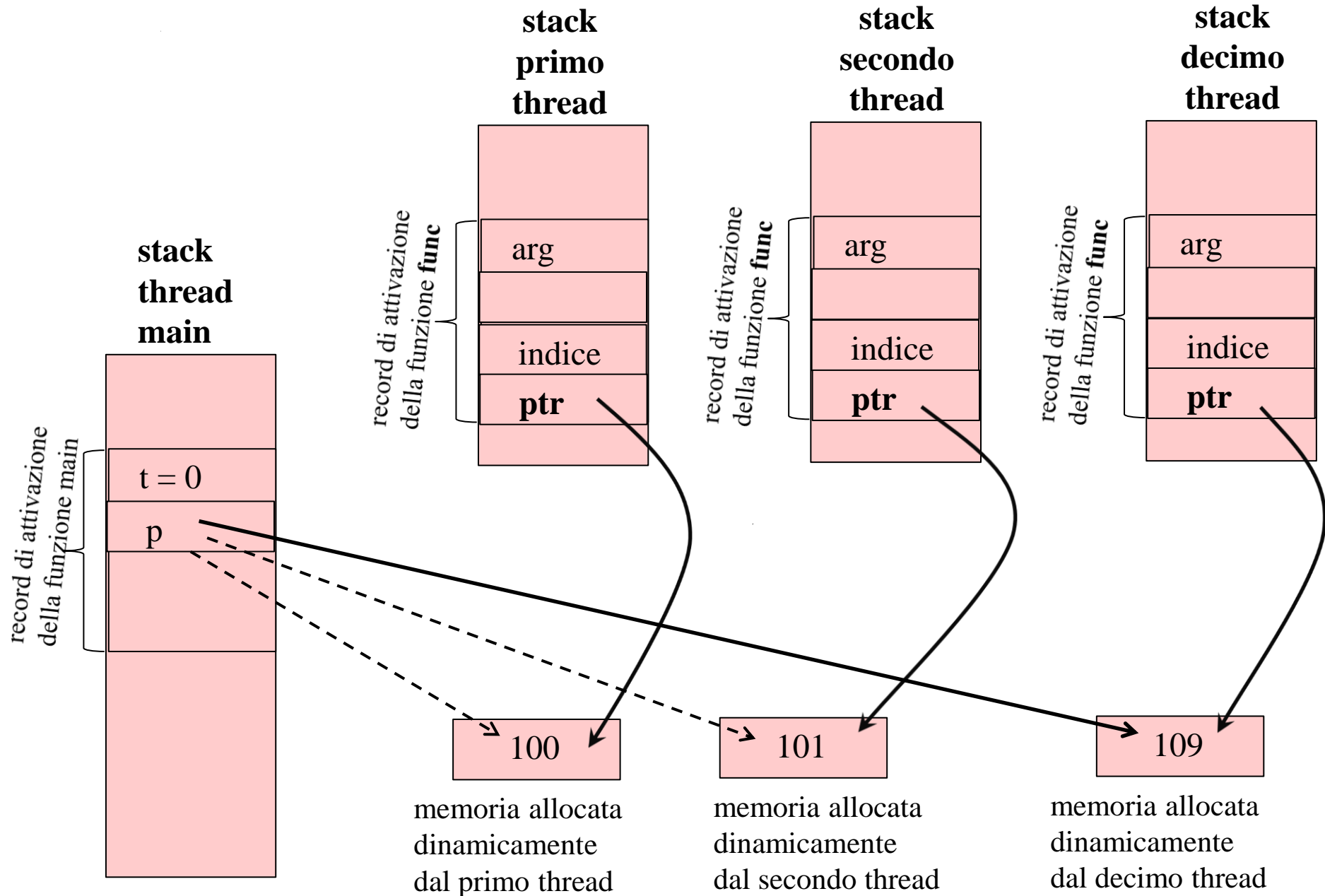
Errore nel Passaggio di argomenti a Pthread



Terminazione di Pthread e processo

- Il main di un processo è un Pthread come gli altri, tranne per quanto riguarda la terminazione.
- Quando un Pthread qualunque chiama l'istruzione `exit()` provoca la terminazione di tutto il processo con tutti i suoi Pthread.
- Quando il main (solo il main, non gli altri Pthread) chiama l'istruzione `return`, provoca la terminazione di tutto il processo con tutti i suoi Pthread.
- La funzione `pthread_exit()` viene chiamata da un Pthread per terminare se stesso (inteso come Pthread) e restituire un risultato.
 - Se ci sono altri Pthread in esecuzione, il processo continua l'esecuzione degli altri Pthread.
 - Se invece il Pthread che ha chiamato la `pthread_exit` era l'ultimo Pthread ancora in esecuzione in quel processo, allora il processo viene terminato.

Schema di Passaggio di risultato di un Pthread



Terminazione e Risultato di un Pthread

Quando un Pthread termina, può voler far conoscere a qualche altro Pthread il risultato del suo lavoro. Occorre un meccanismo con cui **un Pthread può restituire** un risultato (o, più in generale, **un'area di memoria** in cui potrebbe stare anche una struttura dati) a qualche altro Pthread.

La funzione **pthread_exit** permette ad un Pthread di stabilire quale area di memoria deve essere restituita come risultato al termine del Pthread.

Formalmente, ogni Pthread deve restituire come risultato un puntatore generico **void***. Da questo deriva che la funzione `pthread_exit` prevede come argomento un puntatore generico `void*` che sarà consegnato a chi vuole sapere il risultato del pthread.

```
void pthread_exit ( void* ptr );
```

Il puntatore `ptr` può essere `NULL` se il Pthread non vuole restituire un qualche risultato. Il Pthread che invece intende restituire un risultato, deve allocare dinamicamente una struttura dati in cui collocare i dati che intende restituire come risultato.

Nota Bene:

Spiegare **dove** viene inserito il puntatore da restituire, quando si parlerà dell'esempio relativo alla funzione `pthread_join`.

Dove viene mantenuto il Puntatore che funge da risultato di un Pthread

- Il risultato restituito da un Pthread mediante la chiamata a `pthread_exit()` è un puntatore ad un'area di memoria allocata dal Pthread stesso.
- **Il valore di questo puntatore viene mantenuto nello stack del Pthread anche dopo la terminazione del Pthread stesso, e fino a che un altro Pthread non richiede quel risultato mediante la chiamata alla funzione `pthread_join`.**
- **La memoria occupata dallo stack del Pthread già terminato viene rilasciata**
 - solo dopo che un altro Pthread ha chiamato la `pthread_join` per conoscere il risultato del Pthread terminato.
 - oppure ovviamente quando termina il processo.
- .Se nessun Pthread chiede il risultato di un Pthread già terminato, la parte di memoria occupata dallo stack di quel Pthread terminato rimane occupata fino alla terminazione del processo.
 - Però, se un processo gira a lungo e continua a creare nuovi Pthread senza far rilasciare la memoria dello stack dei suoi Pthread che terminano, prima o poi la memoria disponibile si riempie e il processo smette di funzionare.
- E' però possibile configurare i Pthread, al momento della loro creazione, per consentire che **il loro stack possa essere liberato immediatamente dopo la loro terminazione, senza aspettare la `pthread_join`.** Questi Pthread vengono detti **detached**.
 - C'è uno svantaggio, però: i Pthread detached non possono far sapere a nessun altro Pthread il loro risultato.

Attesa di terminazione di un Pthread

E' possibile per un Pthread attendere la terminazione di un altro Pthread dello stesso processo.

Occorre conoscere l'identificatore del Pthread di cui vogliamo attendere la terminazione.

La funzione **pthread_join** vuole come argomento l'indirizzo di un puntatore generico, in cui verrà collocato l'indirizzo restituito dal Pthread il cui identificatore è passato come primo argomento.

Nello scheletro di esempio che segue, il main crea 10 Pthread, poi aspetta la terminazione dei 10 Pthread, da ciascuno riceve un puntatore ad intero e lo usa per stampare il valore intero lì contenuto. Poi rilascia la memoria allocata dal Pthread terminato

codice del main

```
#define NUM_THREADS 10
pthread_t  tid [NUM_THREADS];
int t, rc; int *p;
/* creo i pthread */
for(t=0; t < NUM_THREADS; t++) {
    p = (int *) malloc( sizeof(int) );          *p = t;
    rc = pthread_create( & tid[t] , NULL, func, (void*) p );
}
/* attendo terminazione dei pthread */
for(t=0; t < NUM_THREADS; t++) {
    int ris;

    rc = pthread_join( tid[t] , (void**) & p );
    ris = *p ;
    printf( " pthread restituisce %i \n", ris );
    free (p);
}
```

codice del pthread

```
void * func( void *arg ) {
    int *ptr;
    int indice = *((int*) arg); free(arg);
    /* qui il pthread fa qualcosa ... */

    ptr = (int *) malloc( sizeof(int));

    /* pthread vuole passare un valore
       intero, ad es 100+indice */

    *ptr = 100+indice;
    pthread_exit( (void*) ptr );
}
```

Mutua Esclusione e Sincronizzazione

Perché Garantire la Mutua Esclusione - race condition

Esempio: Due thread devono decrementare il valore di una variabile globale data, ma solo se questa è maggiore di zero. Inizialmente data vale 1.

THREAD1

if(data>0)

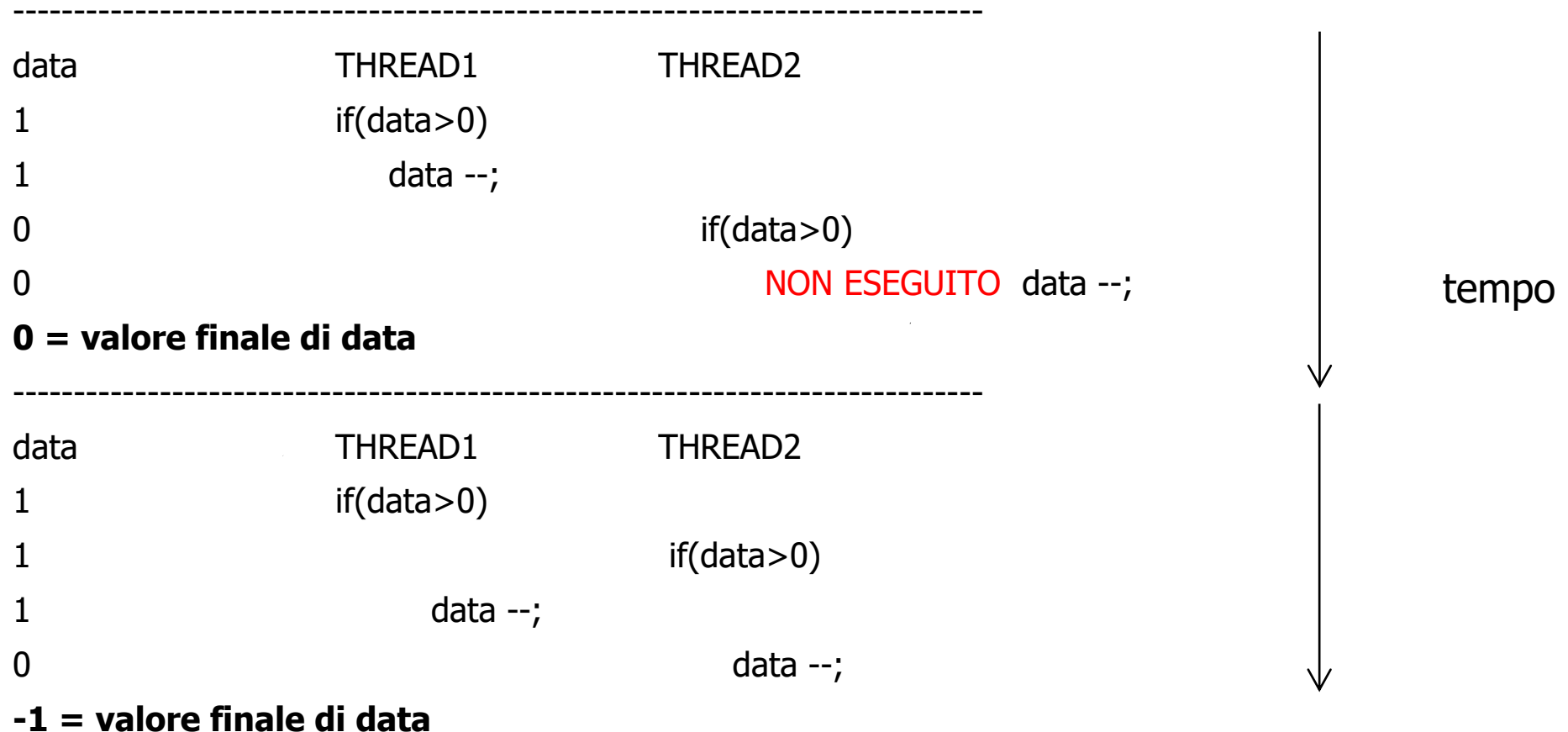
data --;

THREAD2

if(data>0)

data --;

A seconda del tempo di esecuzione dei due thread, la variabile data assume valori diversi (race condition).



Mutex Variables

- ❑ **Mutex** è l'abbreviazione di "**mutua esclusione**".
- ❑ Una variabile mutex (formalmente di tipo **pthread_mutex_t**) serve per regolare l'accesso a dei dati che debbono essere protetti dall'accesso contemporaneo da parte di più thread.
- ❑ Noi usiamo le cosiddette FAST mutex. Ne esistono altri tipi più evoluti che vengono determinati in fase di inizializzazione, **ma noi NON le usiamo e non le potrete usare all'esame.**
- ❑ Ogni thread, prima di accedere a tali dati, deve effettuare una operazione di **pthread_mutex_lock** su una stessa variabile mutex. L'operazione detta "lock" di una variabile mutex blocca l'accesso da parte di altri thread.
- ❑ Infatti, se più thread eseguono l'operazione di lock **su una stessa variabile mutex**, solo uno dei thread termina la lock() e prosegue l'esecuzione, gli altri rimangono bloccati nella lock. In tal modo, il processo che continua l'esecuzione può accedere ai dati (protetti mediante la mutex).
- ❑ Finito l'accesso, il thread effettua un'operazione detta **pthread_mutex_unlock** che libera la variabile mutex. Un'altro thread che ha precedentemente eseguito la lock della mutex potrà allora terminare la lock ed accedere a sua volta ai dati.
- ❑ La tipica sequenza d'uso di una mutex è quindi:
 - creare ed inizializzare la mutex
 - più thread cercano di accedere alla mutex chiamando ciascuno la **pthread_mutex_lock**
 - un solo thread termina la lock e diviene proprietario della mutex, gli altri sono bloccati.
 - il thread proprietario accede ai dati, o esegue funzioni.
 - il thread proprietario libera la mutex eseguendo la **pthread_mutex_unlock**
 - un'altro thread termina la lock e diventa proprietario di mutex
 - e così via
 - al termine del programma, o quando non serve più, la mutex viene distrutta.

Operazioni con Mutex (1)

Creare e Distruggere variabili Mutex (per FAST Mutex)

- Le variabili Mutex devono essere create usando il tipo di dato `pthread_mutex_t` e devono essere inizializzate prima di poter essere usate.

- **Inizializzazione statica**, all'atto della dichiarazione;

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

- **Inizializzazione dinamica**, mediante invocazione di funzione, in un momento successivo alla dichiarazione, nel caso si voglia specificare proprietà aggiuntive; Se il puntatore agli attributi è NULL si utilizzano gli attributi di default.

```
pthread_mutex_t mymutex;
```

```
pthread_mutex_init( & mymutex , &attribute);
```

 attribute può essere NULL

- Finito l'uso delle variabili mutex, bisognerebbe rilasciarle.

```
pthread_mutex_destroy (&mymutex);
```

Operazioni con Mutex (2)

Attesa per Mutua Esclusione (per Fast Mutex)

- ❑ **int pthread_mutex_lock (pthread_mutex_t *mutex);** blocca la variabile mutex passata come argomento.
 - ❑ Se la variabile mutex è attualmente sbloccata, diventa bloccata e di proprietà del thread chiamante e immediatamente la funzione pthread_mutex_lock termina e restituisce il controllo al chiamante, con risultato 0. In caso di problemi viene restituito un risultato diverso da zero che identifica il tipo di errore.
 - ❑ Se invece la variabile mutex è già bloccata da un altro thread, allora la funzione pthread_mutex_lock sospende il thread chiamante finché la mutex non viene sbloccata.
 - ❑ **Attenzione**, se la mutex è già bloccata dal thread chiamante (perché lo stesso pthread ha già fatto una lock senza fare una successiva unlock) allora, chiamando ancora la pthread_mutex_lock il thread va in stallo e si blocca per sempre. In altri tipi di mutex NON FAST, invece, il comportamento è differente.
- ❑ **int pthread_mutex_unlock (pthread_mutex_t *mutex);** sblocca la variabile mutex.
 - ❑ La funzione DEVE ESSERE CHIAMATA su una variabile mutex che sia già bloccata e di proprietà del thread chiamante. In tal caso la funzione pthread_mutex_unlock sblocca la variabile mutex e termina restituendo 0. In caso di problemi viene restituito un risultato diverso da zero che identifica il tipo di errore.
 - ❑ **Attenzione:** Se la funzione pthread_mutex_unlock viene chiamata da un thread che NON E' PROPRIETARIO della variabile mutex (cioè non ha fatto una precedente lock su quella variabile) allora si genera un comportamento imprevedibile, poiché la mutex viene sbloccata anche se è di proprietà di un thread diverso. Questo è un comportamento su cui non si deve fare affidamento.
 - ❑ Nelle mutex di tipo NON FAST, invece, vengono effettuati controlli aggiuntivi ed il comportamento è diverso.

Operazioni con Mutex (3)

Attesa per Mutua Esclusione (per Fast Mutex)

Evitare il blocco

```
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

- questa funzione è come la lock(), ma **se** si accorge che la mutex è già in possesso di altro thread (e quindi si rimarrebbe bloccati) restituisce immediatamente il controllo al chiamante con risultato EBUSY
- Se, invece, si ottiene la proprietà della mutex, restituisce 0.
- **Attenzione**, questa funzione porta a creare codice che causa busy waiting.

Esempio di protezione con Mutex Variable

```
#define NUM_THREADS 10
pthread_t  tid [NUM_THREADS ];
pthread_mutex_t mutexdata;
int data=4;

int main (void) {
    int rc , t;
    int *p;

    pthread_mutex_init (&mutexdata, NULL);

    /* creo i pthread */
    for(t=0;t < NUM_THREADS;t++) {
        rc = pthread_create( & tid[t] , NULL, func, NULL ) ;
    }

    /* attendo terminazione dei pthread */
    for(t=0;t < NUM_THREADS;t++) {

        rc = pthread_join( tid[t] , (void**) & p );
    }
    printf ("data = %d \n", data);

    pthread_mutex_destroy (&mutexdata);

    pthread_exit(NULL);
}
```

codice del pthread

```
void * func( void *arg ) {

    pthread_mutex_lock (&mutexdata);

    if(data>0)
        data--;

    pthread_mutex_unlock (&mutexdata);

    pthread_exit( NULL );
}
```

Condition Variables

- ❑ Le Mutex consentono di effettuare sincronizzazioni tra pthreads nel momento in cui ciascun pthread tenta l'accesso in mutua esclusione. L'accesso viene consentito senza poter valutare a priori il valore delle variabili condivise.
- ❑ Al contrario, le condition variables sono delle variabili che consentono di effettuare delle sincronizzazioni tra pthreads mentre questi già detengono la mutua esclusione, bloccandoli o facendoli continuare in base a condizioni definite dal programmatore e dipendenti dal valore attuale delle variabili condivise.
- ❑ Senza le condition variables, il programmatore dovrebbe scrivere dei pthread che effettuano un loop continuo in cui prima viene presa la mutua esclusione, poi viene controllato il valore delle variabili condivise. Se il valore corrisponde alle condizioni richieste allora il pthread esegue le operazioni richieste e poi rilascia la mutua esclusione. Se invece il valore NON corrisponde alle condizioni richieste allora il pthread rilascia immediatamente la mutua esclusione e poi la riprende e ricontrolla i valori delle variabili condivise e così via all'infinito, **effettuando busy waiting e spreco di tempo di CPU**.
- ❑ Le condition variables devono sempre essere utilizzate assieme ad una mutex per consentire di accedere in mutua esclusione alle variabili condivise.
- ❑ Ci sono tre principali funzioni per sincronizzare con le condition variables:
- ❑ **pthread_cond_wait**: blocca un thread fino a che un altro thread lo risveglia e può prendere la mutua esclusione.
- ❑ **pthread_cond_signal**: sveglia un altro thread bloccato sulla pthread_cond_wait, ma l'altro thread prima di ripartire deve ottenere la mutua esclusione.
- ❑ **pthread_cond_broadcast**: sveglia tutti i threads bloccati sulla pthread_cond_wait, ma ciascuno di questi, prima di ripartire uno alla volta, deve ottenere la mutua esclusione.

ESEMPIO di SINCRONIZZAZIONE

Volpe mangia uova deposte da gallina. La volpe aspetta che la gallina abbia deposto un uovo per poi mangiarselo, poi aspetta di nuovo e così via all'infinito.

- Una prima implementazione (Orrenda) senza Condition variables, provoca busy waiting.
- Una seconda implementazione corretta con Condition variables, efficiente.
 - Stessa soluzione può essere usata avendo contemporaneamente più volpi e più galline, se si considera che solo una gallina alla volta può deporre l'uovo e che solo una volpe per volta può mangiare un uovo.

Volpe mangia uova

SENZA CONDITION VARIABLES - BUSY WAITING ☹ ☹ ☹

```
int uova = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *gallina_produce_uova(void *)
{
    while(1) {
        sleep(3); /* gallina produce uova impiegando tempo ....*/
        pthread_mutex_lock(&mutex);
        uova++; /* gallina depone uovo */
        pthread_mutex_unlock(&mutex);
    }
}

void *volpe_mangia_uova(void *)
{
    while(1) {
        pthread_mutex_lock(&mutex);
        if( uova > 0 ) {
            mangia(); /* volpe mangia uovo impiegando poco tempo ....*/
            uova--;
            pthread_mutex_unlock(&mutex);
            burp(); /* volpe digerisce impiegando tempo */
        } else {
            ; /* do nothing, ma la valutazione di condizione if l'hai fatta
               e la fai in continuazione usando CPU ..... */
            pthread_mutex_unlock(&mutex);
        }
    }
}

int main(void)
{
    pthread_t tid;
    pthread_create( &tid, NULL, gallina_produce_uova, NULL )
    pthread_create( &tid, NULL, volpe_mangia_uova, NULL )
    pthread_exit(NULL);
}
```

CON CONDITION VARIABLES - NO BUSY WAITING ☺

```
int uova = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void *gallina_produce_uova(void *arg)
{
    while(1) { /* gallina produce uova impiegando tempo ....*/
        /* poi gallina depone uovo e fa coccodrillo */
        pthread_mutex_lock(&mutex);
        uova++;
        pthread_cond_signal(&cond); /* coccodrillo' avvisa volpe che c'è uovo */
        pthread_mutex_unlock(&mutex);
    }
}

void *volpe_mangia_uova(void *arg)
{
    while(1) {
        pthread_mutex_lock(&mutex);
        while( uova==0 ) { /* attendi che venga prodotto uovo, attendi signal */
            pthread_cond_wait(&cond, &mutex);
        }
        /* quando arrivo qui c'è almeno un uovo */
        mangia(); /* volpe mangia uovo impiegando tempo ....*/
        uova--;
        pthread_mutex_unlock(&mutex);

        burp(); /* volpe digerisce impiegando tempo */
    }
}

int main(void)
{
    pthread_t tid;
    pthread_create( &tid, NULL, gallina_produce_uova, NULL )
    pthread_create( &tid, NULL, volpe_mangia_uova, NULL )
    pthread_exit(NULL);
}
```


Più attori, stesso codice, NO BUSY WAITING ☺

```
int uova = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void *gallina_produce_uova(void *arg)
{
    while(1) { /* gallina produce uova impiegando tempo ....*/
        /* poi gallina depone uovo e fa coccodrillo */
        pthread_mutex_lock(&mutex);
        uova++;
        pthread_cond_signal(&cond); /* coccodrillo' avvisa volpe che c'è uovo */
        pthread_mutex_unlock(&mutex);
    }
}

void *volpe_mangia_uova(void *arg)
{
    while(1) {
        pthread_mutex_lock(&mutex);
        while( uova==0 ) { /* attendi che venga prodotto uovo, attendi signal */
            pthread_cond_wait(&cond, &mutex);
        }
        /* quando arrivo qui c'è almeno un uovo */
        mangia(); /* volpe mangia uovo impiegando tempo ....*/
        uova--;
        pthread_mutex_unlock(&mutex);

        burp(); /* volpe digerisce impiegando tempo */
    }
}

int main(void)
{
    pthread_t tid; int i;
    for (i=0,i<10;i++) pthread_create( &tid, NULL, gallina_produce_uova, NULL )
    for (i=0,i<30;i++) pthread_create( &tid, NULL, volpe_mangia_uova, NULL )
    pthread_exit(NULL);
}
```

Operazioni con Condition Variables (1)

Creare e distruggere Condition Variables

- Le Condition variables devono essere create usando il tipo di dato `pthread_cond_t` e devono essere inizializzate prima di poter essere usate.
 - **Inizializzazione statica**, all'atto della dichiarazione;
`pthread_cond_t mycond = PTHREAD_COND_INITIALIZER;`
 - **Inizializzazione dinamica**, mediante invocazione di funzione, quando si vuole specificare proprietà aggiuntive;
`pthread_cond_t mycond;`
`pthread_cond_init(& mycond , &attribute);`
- Finito l'uso delle condition variable, bisognerebbe rilasciarle.
`pthread_cond_destroy(&mycond);`

Operazioni con Condition Variables (2)

Attesa

- ❑ **pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex)** è eseguita da un pthread quando questo vuole bloccarsi ed aspettare che una condizione sia vera.
- ❑ Prima di chiamare la wait il pthread deve prendere la mutua esclusione sulla mutex specificata nel secondo parametro della wait.
- ❑ Quando la wait blocca il pthread, automaticamente la wait rilascia la mutua esclusione e poi si mette in attesa di essere abilitata al risveglio da una pthread_cond_signal() chiamata da qualche altro pthread.
- ❑ La wait continua e termina solo dopo che si verificano due condizioni
 - 1) un altro pthread l'ha risvegliata con una chiamata a pthread_cond_signal() e che
 - 2) la wait ha potuto riprendere la mutua esclusione.
- ❑ Quando la wait termina, il pthread si trova a detenere la mutua esclusione sulla variabile mutex specificata nel secondo parametro della wait.

Operazioni con Condition Variables (3)

Abilitazione al risveglio

- **pthread_cond_signal(pthread_cond_t *cond)** controlla se c'è qualche thread in attesa con una wait sulla condition variable specificata nel parametro passato alla signal. Se c'è lo avvisa e lo abilita a risvegliarsi, ma per potersi risvegliare quel thread deve attendere di poter prendere la mutua esclusione.
- Dopo aver abilitato il thread a proseguire, la funzione signal termina.
- Se non c'è un thread in attesa da abilitare, la funzione semplicemente termina.
- Non è specificato qual' è l'ordine con cui le signal abilitano i thread in attesa su una wait. Non è affatto detto che l'ordine di risveglio sia lo stesso ordine in cui i threads sono entrati in wait.
- **IMPORTANTE: Le chiamate alle pthread_cond_signal() e pthread_cond_broadcast() devono essere effettuate da un thread che detiene la mutua esclusione sulla mutex specificata nelle wait per quella condition variable. In tal modo si protegge l'accesso alla condition variable che è essa stessa una variabile condivisa.**
- In pratica, prima di chiamare la pthread_cond_signal() o la pthread_cond_broadcast() occorre prima avere chiamato la pthread_mutex_lock().
- Per abilitare tutti i thread in attesa su una wait per una data condition variable, posso fare una iterazione di chiamate alla signal, oppure posso chiamare la funzione **pthread_cond_broadcast(pthread_cond_t *cond)**.

Riassumendo: Operazioni con Condition Variables

□ Waiting and signaling on condition variables

□ Routines

□ `pthread_cond_wait(condition, mutex)`

- ▶ Blocks the thread until the specific condition is signalled.
- ▶ Should be called with mutex locked
- ▶ Automatically release the mutex lock while it waits
- ▶ When return (condition is signaled), mutex is locked again

□ `pthread_cond_signal(condition)`

- ▶ Wake up a thread waiting on the condition variable.
- ▶ Called after mutex is locked, and must unlock mutex after

□ `pthread_cond_broadcast(condition)`

- ▶ Used when multiple threads blocked in the condition