

Thread POSIX

- **Introduzione ai thread POSIX**
 - operazioni elementari sui thread
- **Sincronizzazione**
 - **Semafori**
 - ⇒ semafori di mutua esclusione
 - ⇒ semafori generali
 - ⇒ utilizzo ed esempi
 - **Variabili condition**
 - ⇒ generalità
 - ⇒ utilizzo ed esempi

Thread POSIX: aspetti preliminari

Programmazione concorrente



- **La programmazione concorrente è l'insieme di tecniche e di strumenti necessari a poter supportare più attività simultanee in una applicazione software.**
- **Caratteristica dei sistemi multiprogrammati.**
- **Consente a più utenti di accedere contemporaneamente ad un sistema informatico**
- **Consente ad un solo utente l'esecuzione di più programmi simultaneamente**
- **Consente ad un singolo programma di scomporre la propria attività in più attività concorrenti**

Programma vs Processo



- **Programma** : è un'entità statica che rimane immutata durante l'esecuzione ed è costituita dal codice oggetto generato dalla compilazione del codice sorgente.
- **Processo**: è l'entità utilizzata dal sistema operativo per rappresentare una specifica esecuzione di un programma. Esso è quindi un'entità dinamica, che dipende dai dati che vengono elaborati e dalle operazioni eseguite su di essi.

Il processo è caratterizzato:

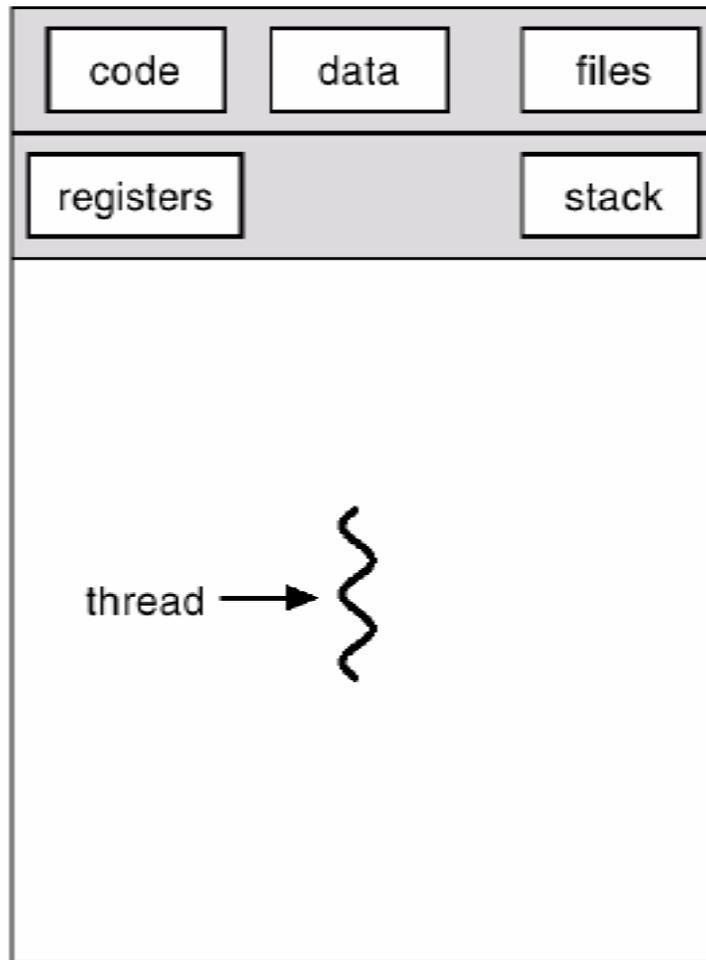
- dal codice eseguibile
- dall'insieme di tutte le informazioni che ne definiscono lo stato
 - ⇒ contenuto della memoria indirizzata
 - ⇒ i thread
 - ⇒ i descrittori dei file e delle periferiche in uso.



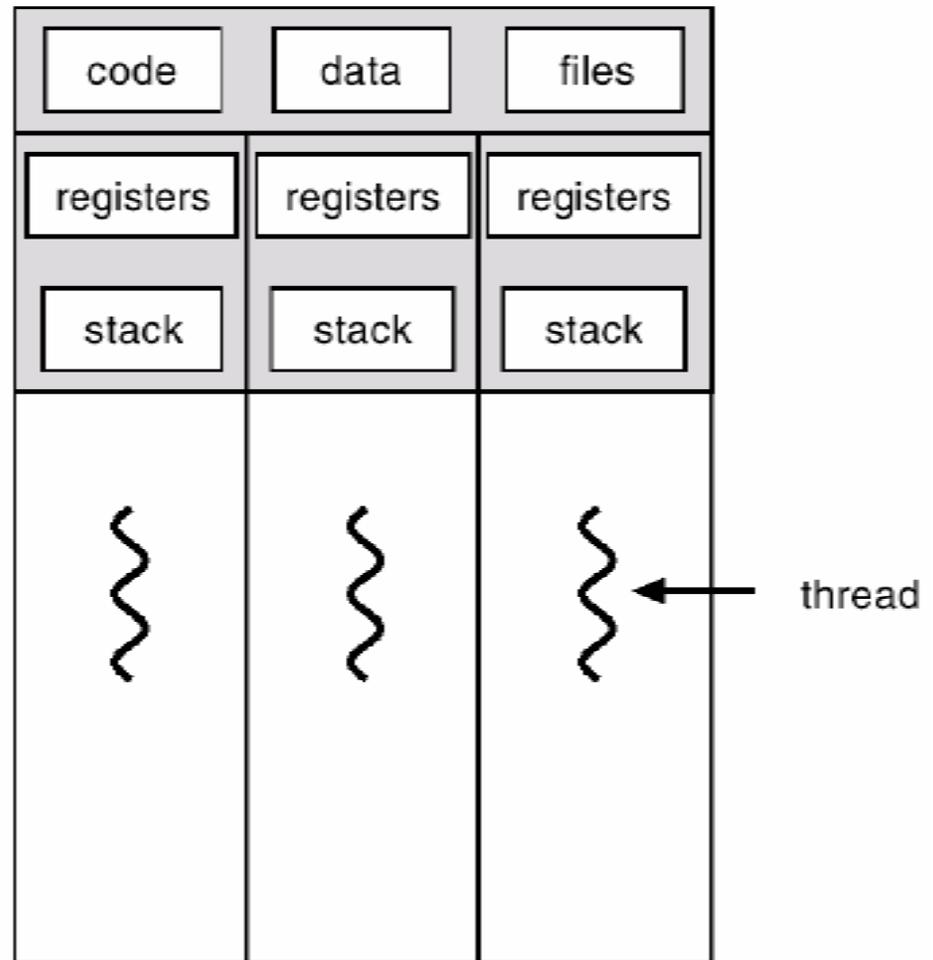
- **Processo:** è l'unità di condivisione delle risorse alcune delle quali possono essere inizialmente ereditate dal padre.
 - Ogni processo ha:
 - ⇒ spazio di indirizzamento privato
 - ⇒ stack
 - ⇒ heap.
 - Due processi non dividono mai lo spazio di indirizzamento
- **Thread :** è un flusso di esecuzione indipendente che condivide tutte le sue risorse, incluso lo spazio di indirizzamento, con altri thread
 - I thread girano su memoria condivisa nell'ambito dello stesso programma, quando un thread viene creato condivide il suo spazio di memoria con gli altri threads che fanno parte del processo
 - Sono anche chiamati *lightweight process* o *processi leggeri* perchè possiedono un contesto più snello rispetto ai processi

Processo vs Thread

(2 di 2)



single-threaded



multithreaded

- **Thread**
 - è l'unità granulare in cui un processo può essere suddiviso e che può essere eseguito in parallelo ad altri thread
 - è parte del processo che viene eseguita in maniera concorrente ed indipendente internamente al processo stesso
 - insieme di istruzioni che vengono eseguite in modo indipendente rispetto al main
- **Stato di un thread**
 - stack
 - registri
 - proprietà di scheduling
 - stato dei segnali
 - dati privati

Un esempio



- Un esempio di applicazione che può far uso di più thread è un browser Web, che usa un thread distinto per scaricare ogni immagine in una pagina Web contenente più immagini.
- Un altro esempio è costituito dai processi server che possono rispondere contemporaneamente alle richieste provenienti da più utenti.

- **Visibilità dei dati globali**
 - **condivisione di oggetti semplificata.**
- **Più flussi di esecuzione.**
- **Gestione semplice di eventi asincroni (I/O)**
- **Comunicazioni veloci**
 - **Tutti i thread di un processo condividono lo stesso spazio di indirizzamento, quindi le comunicazioni tra thread sono più semplici delle comunicazioni tra processi.**
- **Context switch veloce**
 - **Nel passaggio da un thread ad un altro di uno stesso processo viene mantenuta buona parte dell'ambiente.**

- **Concorrenza invece di parallelismo**
 - gestire la mutua esclusione
- **I thread di un programma usano il sistema operativo mediante system call che usano dati e tabelle di sistema dedicate al processo.**
 - **Le syscall devono essere costruite in modo da poter essere utilizzate da più thread contemporaneamente. Ad esempio la funzione `char *inet_ntoa()` scrive il proprio risultato in una variabile di sistema (del processo) e restituisce al chiamante un puntatore a tale variabile. Se due thread di uno stesso processo eseguono “nello stesso istante” la chiamata a due `inet_ntoa()` ognuno setta la variabile con un valore.**

- **Standard ANSI/IEEE POSIX 1003.1 (1990)**
 - Lo standard specifica l'interfaccia di programmazione (Application Program Interface - API) dei thread.
 - I thread POSIX sono noti come Pthread.

Funzioni delle API per 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” (mutua esclusione).
 - ⇒ funzioni per creare ed eliminare la struttura per la 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.
 - ⇒ funzioni per creare ed eliminare la struttura per la sincronizzazione, per attendere e segnalare le modifiche delle variabili.

■ Utilizzo

- includere l'header della libreria che contiene le definizioni dei pthread

```
#include <pthread.h>
```

⇒ Per interpretare correttamente i messaggi di errore è necessario anche includere l'header `<errno.h>`

- compilare specificando la libreria

```
gcc <opzioni> -lpthread
```

⇒ Libreria pthread (libpthread) → `lpthread`

⇒ Per ulteriori informazioni sulla compilazione fare riferimento alla documentazione della piattaforma utilizzata `man pthread` o `man pthreads`

Convenzione sui nomi delle funzioni



- **Gli identificatori della libreria dei Pthread iniziano con pthread_**
 - pthread_
 - ⇒ **indica la gestione dei thread in generale**
 - pthread_attr_
 - ⇒ **funzioni per gestire proprietà dei thread**
 - pthread_mutex_
 - ⇒ **gestione della mutua esclusione**
 - pthread_mutexattr_
 - ⇒ **proprietà delle strutture per la mutua esclusione**
 - pthread_cond_
 - ⇒ **gestione delle variabili di condizione**
 - pthread_condattr_
 - ⇒ **proprietà delle variabili di condizione**
 - pthread_key_
 - ⇒ **dati speciali dei thread**



- **POSIX Threads Programming Tutorial**
 - <http://www.llnl.gov/computing/tutorials/pthreads/>
- **Libri (consultazione)**
 - B. Lewis, D. Berg, “*Threads Primer*”, Prentice Hall
 - D. Butenhof, “*Programming With POSIX Threads*”, Addison Wesley
 - B. Nichols et al, “*Pthreads Programming*”, O’Reilly



■ Manpages

- **pacchetto** `manpages-posix-dev` (**Debian**)
- `man pthread.h`
- `man <nomefunzione>`

■ Manuale GNU libc

- http://www.gnu.org/software/libc/manual/html_node/POSIX-Threads.html

Gestione dei thread

Tipi definiti nella libreria pthread



- **All'interno di un programma un thread è rappresentato da un identificatore**
 - tipo opaco `pthread_t`
- **Attributi di un thread**
 - tipo opaco `pthread_attr_t`

tipo opaco: si definiscono così strutture ed altri oggetti usati da una libreria, la cui struttura interna non deve essere vista dal programma chiamante (da cui il nome) che li deve utilizzare solo attraverso dalle opportune funzioni di gestione.

Identificatore del thread



- **Processo:** process id (pid) pid_t
- **Thread:** thread id (tid) pthread_t

```
pthread_t pthread_self( void )
```

- restituisce il tid del thread chiamante

Confronto tra thread



```
int pthread_equal( pthread_t t1, pthread_t t2 )
```

- **confronta i due identificatori di thread.**
 - 1 se i due identificatori sono uguali



```
int pthread_create( pthread_t *thread,  
                  const 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.**



- `pthread_t *thread`
 - **puntatore ad un identificatore di thread in cui verrà scritto l'identificatore univoco del thread creato (se creato con successo)**
- `const pthread_attr_t *attr`
 - **attributi del processo da creare: può indicare le caratteristiche del thread riguardo alle operazioni di join o allo scheduling**
 - **se NULL usa valori di default**
- `void *(*start_routine)(void *)`
 - **è il nome (indirizzo) della funzione da eseguire alla creazione del thread**
- `void *arg`
 - **puntatore che viene passato come argomento a `start_routine`.**
- **Valore di ritorno**
 - **0 in assenza di errore**
 - **diverso da zero altrimenti**
 - ⇒ attributi errati
 - ⇒ mancanza di risorse

Terminazione di un thread



```
void pthread_exit( void *value_ptr )
```

- Termina l'esecuzione del thread da cui viene chiamata
- Il sistema libera le risorse allocate al thread.
- **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 terminazione.**
- `void *value_ptr`
 - valore di ritorno del thread consultabile da altri thread attraverso la funzione `pthread_join`

Esempio 1: creazione e terminazione(1 di 3)



```
/* Include */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS      5

/* Corpo del thread */
void *PrintHello(void *num) {
    printf("\n%d: Hello World!\n", num);
    pthread_exit(NULL);
}
```

Continua ⇒

Esempio 1: creazione e terminazione(2 di 3)



```
/* Programma */
int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0; t<NUM_THREADS; t++){
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```



- **Eseguire più volte il programma**
 - Cosa accade?
- **Provare a commentare le funzioni**
`pthread_exit(NULL)`
 - Cosa accade?



- La `pthread_create` prevede un puntatore per il passaggio dei parametri al thread nel momento in cui comincia l'esecuzione.
- Si ponga attenzione nel caso il thread debba modificare i parametri, oppure il chiamante debba modificare i parametri, potrebbero insorgere problemi, meglio dedicare una struttura dati ad ok, per il passaggio.



- Per riferimento con un cast a `void*`
- Esempio (errato)
 - il ciclo modifica il contenuto dell'indirizzo passato come parametro

```
int rc, t;

for(t=0; t<NUM_THREADS; t++) {
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
    ...
}
```



- **Esempio (corretto)**
 - **struttura dati univoca per ogni thread**

```
int *taskids[NUM_THREADS];
for(t=0; t<NUM_THREADS; t++){
    taskids[t] = (int *) malloc(sizeof(int));
    *taskids[t] = t;
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *) taskids[t]);
    ...
}
```



```
/* Include */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS      5

/* Corpo del thread */

void *PrintHello(void *num) {
    printf("\n%d: Hello World!\n", *(int *) num);
    pthread_exit(NULL);
}
```



```
/* Programma */
int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;

    for(t=0; t<NUM_THREADS; t++) {
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```



```
/* Include */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS      5

/* Corpo del thread */

void *PrintHello(void *num) {
    printf("\n%d: Hello World!\n", *(int *) num);
    pthread_exit(NULL);
}
```



```
/* Programma */
int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    int *taskids[NUM_THREADS];
    for(t=0; t<NUM_THREADS; t++){
        taskids[t] = (int *) malloc(sizeof(int));
        *taskids[t] = t;
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)
taskids[t] );
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Esempio 2: passaggio parametri



- Qual è la differenza?

Sincronizzazione

- **Forma elementare di sincronizzazione**
 - il thread che effettua il join si blocca finché uno specifico thread non termina
 - il thread che effettua il join può ottenere lo stato del thread che termina
- **Attributo `detachstate` di un thread**
 - specifica se si può invocare o no la funzione join su un certo thread
 - un thread è joinable per default

Operazione di join



```
int pthread_join( pthread_t *thread, void **value )
```

- `pthread_t *thread`
 - **identificatore del thread di cui attendere la terminazione**
- `void **value`
 - **valore restituito dal thread che termina**
- **Valore di ritorno**
 - **0 in caso di successo**
 - **EINVAL se il thread da attendere non è joinable**
 - **ERSCH se non è stato trovato nessun thread corrispondente all'identificatore specificato**



```
int pthread_attr_init( pthread_attr_t *attr )
```

- **Inizializza gli attributi del pthread**

```
int pthread_attr_destroy ( pthread_attr_t *attr)
```

- **Dealloca il pthread**



- Un thread può essere:
 - **Joinable:** i thread non sono rilasciati automaticamente ma rimangono come zombie finché altri thread non effettuano delle join
 - **Detached:** i thread detached sono rilasciati automaticamente e non possono essere oggetto di join da parte di altri thread.

```
int pthread_attr_setdetachstate( pthread_attr_t *attr,  
                                int detachstate )
```

- Detach può essere:
 - PTHREAD_CREATE_DETACHED
 - PTHREAD_CREATE_JOINABLE.



```
/* Attributo */
pthread_attr_t attr;

/* Inizializzazione esplicita dello stato joinable */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
...
pthread_attr_destroy(&attr);
```



```
int main (int argc, char *argv[]) {
    pthread_t thread[NUM_THREADS];
    ...
    pthread_attr_destroy(&attr);
    for(t=0; t<NUM_THREADS; t++)
    {
        rc = pthread_join(thread[t], (void **)&status);
        if (rc) {
            printf("ERROR: return code from pthread_join() is %d\n", rc);
            exit(-1);
        }
        printf("Completed join with thread %d status= %d\n",t, status);
    }
    pthread_exit(NULL);
}
```



```
/* Include */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS      5

void *PrintHello(void *num) {
    printf("\n%d: Hello World!\n", num);
    pthread_exit(NULL);
}
```

Continua ⇒

Esempio 3: thread join

(2 di 3)



```
int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    void *status;
    int rc, t;
    pthread_attr_t attr;

    /* Inizializzazione esplicita dello stato joinable */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for(t=0; t<NUM_THREADS; t++){
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
}
```

Continua ⇨



```
for(t=0; t<NUM_THREADS; t++){
    rc = pthread_join(threads[t], (void **)&status);
    if (rc) {
        printf("ERROR: return code from pthread_join() is %d\n", rc);
        exit(-1);
    }
    printf("Completed join with thread %d status= %d\n",t, status);
}

printf ("Main(): Atteso su %d threads. Fatto \n", NUM_THREADS);

/*Rimuovi oggetti ed esci*/
pthread_attr_destroy(&attr);

pthread_exit(NULL);
}
```

Esempio 3 : thread join



- **Cosa succede se eliminiamo il join?**

Sincronizzazione tra thread



- **Attraverso variabili globali**
 - condivise tra thread
 - meccanismi di protezione
- **Compito del programmatore**
 - corretto utilizzo delle funzioni di sincronizzazione
- **Meccanismi forniti dalla libreria**
 - semafori di mutua esclusione
 - semafori generali
 - variabili condition

Esercizio 1



- `pthread-1a-simple.c`
 - **analizzare l'output**
 - **cambiare `pthread_exit(NULL)` in `return(0)`**
 - **cosa succede?**
 - **aggiungere il passaggio di un parametro ai thread passando a tutti lo stesso valore**
- `pthread-1b-simple.c`
 - **cosa cambia rispetto al precedente?**
- `pthread-1c-simple.c`
 - **soluzione dell'esercizio precedente**

pthread-1a-simple.c



```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_THREADS 3

void *thread_function(void* arg){
    printf("[Thread] Waiting for termination...\n");
    sleep(5);
    printf("[Thread] ...thread finished!\n");
    pthread_exit(NULL);
}

int main(void){
    pthread_t tids[NUM_THREADS];
    int i, rc;
    printf("[Main] Starting...\n");
    for (i=0; i<NUM_THREADS; i++) {
        printf("[Main] Creating thread %d..\n", i);
        rc = pthread_create(&tids[i], NULL, thread_function, NULL);
        if (rc) {
            perror("[Main] ERROR from pthread_create()\n");
            exit(-1);
        }
    }
    printf("[Main] ...done!\n");
    pthread_exit(NULL);
}
```

pthread-1b-simple.c (1 di 2)



```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_THREADS 3

void *thread_function(void* arg){
    printf("[Thread] Waiting for termination...\n");
    sleep(5);
    printf("[Thread] ...thread finished!\n");
    pthread_exit(NULL);
}
```

Continua ⇒

pthread-1b-simple.c (2 di 2)



```
int main(void){
    pthread_t tids[NUM_THREADS];
    int i, rc;
    printf("[Main] Starting...\n");
    for (i=0; i<NUM_THREADS; i++) {
        printf("[Main] Creating thread %d..\n", i);
        rc = pthread_create(&tids[i], NULL, thread_function, (void*)NULL);
        if (rc) {
            perror("[Main] ERROR from pthread_create()\n");
            exit(-1);
        }
    }
    printf("[Main] Waiting for threads termination...\n");
    for(i=0; i<NUM_THREADS; i++) {
        rc = pthread_join(tids[i], (void *)NULL);
        if (rc) {
            perror("[Main] ERROR from pthread_join()\n");
            exit(-1);
        }
        printf("[Main] Completed join with thread %d\n",i);
    }
    printf("[Main] ...done!\n");
    return(0);
}
```

Esercizio 2



- `pthread-2a-args.c`
 - **analizzare l'output**
 - **modificare in modo da ottenere un funzionamento corretto**
- `pthread-2b-args.c`
 - **soluzione dell'esercizio precedente**

pthread-2a-args.c



```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_THREADS 3

void *thread_function(void* arg)
{
    int i = *(int*)arg;
    printf("[Thread %d] Waiting for termination...\n",i);
    sleep(5);
    printf("[Thread %d] ...thread finished!\n",i);
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t tids[NUM_THREADS];
    int i, rc;
    printf("[Main] Starting...\n");
    for (i=0; i<NUM_THREADS; i++) {
        printf("[Main] Creating thread %d..\n", i);
        rc = pthread_create(&tids[i], NULL, thread_function, (void*)&i);
        if (rc){
            perror("[Main] ERROR from pthread_create()\n");
            exit(-1);
        }
    }
    printf("[Main] ...done!\n");
    pthread_exit(NULL);
}
```