

# I Thread POSIX (2/2)

- Terminazione di un thread
- Inizializzazione dinamica
- Creazione degli attributi
- Gestione degli errori
- Altri esempi

# Terminazione

- Un processo termina chiamando **exit**, oppure ritornando dal **main**; un thread termina chiamando **pthread\_exit** o ritornando dalla funzione.
- Per terminare un thread si usa:

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

```
thread_a()  
-----;  
-----;  
-----;  
-----;  
pthread_exit()    ;
```



**Note:** Termination does not free the resources owned by the thread. To release resources, a thread must be detached or joined.

# Terminazione

- Un processo termina chiamando **exit**, oppure ritornando dal **main**; un thread termina chiamando **pthread\_exit** o ritornando dalla funzione.
- Per terminare un thread si usa:

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

- *value\_ptr* è lo stato di uscita. Questo valore è ritornato al processo che chiama **pthread\_join**; se il thread è distaccato, il valore si perde.
- Un thread può uscire con un *value\_ptr* che punta ad una struttura dati contenente informazioni dettagliate.
- D'altra parte un thread può terminare con un valore intero che indica il successo o il fallimento (0 e -1), effettuando un cast da *void \** a *int*.

# Terminazione

- Un thread distaccato deve uscire con un stato *NULL*. Uno stato differente potrebbe portare a comportamenti non prevedibili.
- Dopo la terminazione del thread, il suo stack sarà reclamato dal sistema: tentare di accedere alle variabili locali di un thread terminato porta ad un comportamento indefinito.
- Lo stato di uscita di un thread non deve mai puntare a variabili locali del thread.
- Quando un thread termina le risorse non vengono rilasciate; ad esempio descrittori dei file e memoria condivisa non vengono chiusi o distaccati, anche se è un solo thread ad utilizzarle.
- Nell'esempio seguente vengono creati 3 thread e ciascuno termina in maniera differente: il **primo** esce con un **valore intero**, il **secondo** con un **puntatore**, il **terzo** ritorna con un **puntatore**.

# Esempio 5 (1/3)

```
#include <pthread.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>

void          thread1_func(), thread2_func(), fatal_error();
int          *thread3_func();
```

```
#define check_error(return_val, msg) {          \
    if (return_val != 0)                        \
        fatal_error(return_val, msg); }
```

```
int          second_thread = 2;
int          third_thread = 3;
```

```
main()
{
    pthread_t  tid[3];
    int       i, return_val;
    long      val;
```

```
/* Create thread #1 */
return_val = pthread_create(&tid[0], (pthread_attr_t *)NULL,
                          (void *(*)(void*))thread1_func,
                          (void *)NULL);
check_error(return_val, "pthread_create() - 1");
```

```
/* Create thread #2 */
return_val = pthread_create(&tid[1], (pthread_attr_t *)NULL,
                          (void *(*)(void*))thread2_func,
                          (void *)NULL);
check_error(return_val, "pthread_create() - 2");
```

## Esempio 5 (2/3)

```
/* Create thread #3 */
return_val = pthread_create(&tid[2], (pthread_attr_t *)NULL,
                           (void *(*)(void*))thread3_func,
                           (void *)NULL);
check_error(return_val, "pthread_create() - 3");
```

```
/* Wait for the threads to finish executing.
 * Print out each thread's return status.*/
for (i = 0; i < 3; i++) {
    return_val = pthread_join(tid[i], (void *)&val);
    check_error(return_val, "pthread_join()");
    printf("Thread %d returned 0x%x\n", tid[i], val);
}
```

```
/* Say Good-bye */
printf("Good-bye World!\n");
exit(0);
}
```

```
/* Print error information, exit with -1 status. */
void fatal_error(int err_num, char *function)
{
    char *err_string;

    err_string = strerror(err_num);
    fprintf(stderr, "%s error: %s\n", function, err_string);
    exit(-1);
}
```

## Esempio 5 (3/3)

```
/* Thread functions */

void thread1_func()
{
    printf("Hello World! I'm the first thread\n");
    pthread_exit((void *) 1);
}

void thread2_func()
{
    printf("Hello World! I'm the second thread\n");
    pthread_exit((void *)&second_thread);
}

int *thread3_func()
{
    printf("Hello World! I'm the third thread\n");
    return((void *)&third_thread);
}
```

# Inizializzazione dinamica

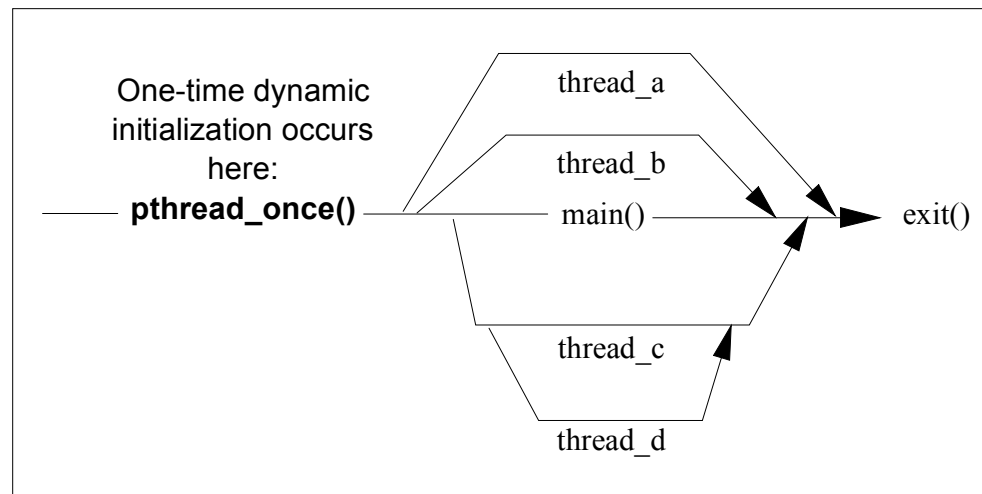
- Molte routine di libreria e applicazioni sono **inizializzate dinamicamente**, cioè l'inizializzazione di un modulo avviene automaticamente quando esso viene chiamato la prima volta.
- L'inizializzazione dinamica viene utilizzata per moduli che possono non andare in esecuzione. Ad esempio si può usare un codice del tipo:

```
static int func_x_inialized = 0;
void func_x_init( )
{
    /* Qui va il codice di inizializzazione */
    ...
}
void func_x( )
{
    if (func_x_inialized == 0){
        func_x_init ( )
        func_x_inialized = 1;
    }
    /* Qui va il codice di func_x( ) */
    ...
}
```

# Inizializzazione dinamica

- Questa strategia funziona bene nel modello a processi. Nel modello a thread, provoca problemi di **corsa critica** (*race condition*).
  - ✦ Se due thread chiamano **func\_x** simultaneamente e verificano **func\_x\_initialized == 0**, entrambi eseguono **func\_x\_init**.
- Per risolvere questo problema si usa:

```
pthread_once_t    once_control = PTHREAD_ONCE_INIT;  
  
int pthread_once(  
    pthread_once_t  
    void  
);  
  
*once_control,  
(*init_routine) (void)
```



# Inizializzazione dinamica

- Questa strategia funziona bene nel modello a processi. Nel modello a thread, provoca problemi di **corsa critica** (*race condition*).
  - ✦ Se due thread chiamano **func\_x** simultaneamente e verificano **func\_x\_initialized == 0**, entrambi eseguono **func\_x\_init**.
- Per risolvere questo problema si usa:

```
pthread_once_t    once_control = PTHREAD_ONCE_INIT;  
  
int  pthread_once(  
    pthread_once_t    *once_control,  
    void              (*init_routine) (void)  
);
```

- **pthread\_once** garantisce che una funzione di inizializzazione è chiamata una ed una sola volta. *once\_control* è utilizzato per determinare se la funzione è stata chiamata in precedenza.
- *once\_control* deve essere una variabile globale o statica, oppure non deve essere inizializzata con **PTHREAD\_ONCE\_INIT**.
- L'esempio seguente mostra come usare **pthread\_once**

```

#include <pthread.h>

static pthread_once_t  func_x_initialized = PTHREAD_ONCE_INIT;

extern void  fatal_error(int err_num, char *function);

void func_x_init()
{
    /* Initialization code for func_x() here */
    printf("In func_x_init() initialization\n");
}

void func_x()
{
    (void) pthread_once(&func_x_initialized, func_x_init);

    /* Code for func_x() here */
    printf("In func_x() main routine\n");
}

main()
{
    pthread_t  tid1, tid2;
    int  return_val;

    return_val = pthread_create(&tid1, (pthread_attr_t *)NULL,
                               (void *(*)(void*))func_x, (void *)NULL);
    check_error(return_val, "pthread_create() - 1");

    return_val = pthread_create(&tid2, (pthread_attr_t *)NULL,
                               (void *(*)(void*))func_x, (void *)NULL);
    check_error(return_val, "pthread_create() - 2");

    func_x();
}

```

## Esempio 6

# Attributi specializzati

- Uno dei parametri di **pthread\_create** sono gli attributi. Un nuovo thread è creato in accordo con quanto specificato nell'oggetto degli attributi.
- Se il parametro è **NULL**, vengono utilizzati gli attributi di default del sistema.
- Gli attributi permettono ad un'applicazione di creare thread specializzati. Ad esempio alcuni thread con grandi stack, altri con priorità alte.
- Per inizializzare gli attributi si utilizza:

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

`pthread_attr_init();`

### Attributes Object

stack size  
stack addr  
detach state

`pthread_attr_destroy();`

### Attributes Object

stack size  
stack addr  
detach state

# Attributi specializzati

- Uno dei parametri di `pthread_create` sono gli attributi. Un nuovo thread è creato in accordo con quanto specificato nell'oggetto degli attributi.
- Se il parametro è **NULL**, vengono utilizzati gli attributi di default del sistema.
- Gli attributi permettono ad un'applicazione di creare thread specializzati. Ad esempio alcuni thread con grandi stack, altri con priorità alte.
- Per inizializzare gli attributi si utilizza:

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

- Dopo la chiamata a questa funzione tutti gli attributi in *attr* hanno i valori di default e si possono quindi cambiare. Lo stesso oggetto *attr* può essere usato per inizializzare più thread.
- Una volta creato, il thread non sarà affetto dai cambiamenti apportati all'oggetto degli attributi.

# Attributi specializzati

- Quando tutti i thread che hanno bisogno degli attributi contenuti nell'oggetto *attr* sono stati creati, l'oggetto non è più necessario e può essere distrutto per non consumare inutilmente risorse.
- Per distruggere un oggetto degli attributi si usa **`pthread_attr_destroy`**.
- Poiché *pthread\_attr\_t* è un tipo opaco, non conviene operare direttamente su di esso, ma utilizzare le apposite funzioni.
- Sono disponibili i seguenti attributi:
  - ✦ *stacksize* permette di specificare una dimensione per lo stack. Il valore di default cambia da sistema a sistema.
  - ✦ *stackaddr* permette di allocare e gestire lo stack, specificando l'indirizzo base. Più thread usano lo stesso oggetto attributo => stesso stack!
  - ✦ *detachstate* permette di specificare se un thread è distaccabile o unibile. Se vale **`PTHREAD_CREATE_DETACHED`**, tutti i thread sono distaccati e non si possono usare in **`pthread_join`** e **`pthread_detach`**.

# Inizializzare gli attributi

- Il valore di *stacksize*, *stackaddr*, e *detachstate* può essere ottenuto o inizializzato nell'oggetto attributi con le seguenti funzioni.

```
int pthread_attr_getstacksize(  
    pthread_attr_t  *attr  
    size_t          *stacksize  
);  
  
int pthread_attr_getstackaddr(  
    pthread_attr_t  *attr  
    size_t          **stackaddr  
);  
  
int pthread_attr_getdetachstate(  
    pthread_attr_t  *attr  
    size_t          *detachstate  
);
```

```
int pthread_attr_setstacksize (  
    pthread_attr_t  *attr  
    size_t          stacksize  
);  
  
int pthread_attr_setstackaddr (  
    pthread_attr_t  *attr  
    size_t          *stackaddr  
);  
  
int pthread_attr_setdetachstate (  
    pthread_attr_t  *attr  
    size_t          detachstate  
);
```

- Nell'esempio seguente viene creato un oggetto degli attributi, chiesta la grandezza dello stack e, se inferiore a quattro pagine, incrementata. Il thread viene quindi creato e l'oggetto distrutto.

# Esempio 7 (1/2)

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

```
extern void    check_error(int err_num, char *function);
extern void    thread1_func();
```

```
main()
{
    pthread_t    tid;
    pthread_attr_t attr;
    size_t    stacksize;
    int    ret_val, pagesize;
```

```
/* Create the attr object */
ret_val = pthread_attr_init(&attr);
check_error(ret_val, "pthread_attr_init()");
```

```
/* Set the stacksize to (default stacksize * 4) */
ret_val = pthread_attr_getstacksize(&attr, &stacksize);
check_error(ret_val, "pthread_attr_getstacksize()");
```

```
pagesize = (int)sysconf(_SC_PAGESIZE);
if (stacksize < (pagesize * 4)) {
    stacksize = pagesize * 4;
    ret_val = pthread_attr_setstacksize(&attr, stacksize);
    check_error(ret_val, "pthread_attr_setstacksize()");
}
```

# Esempio 7 (2/2)

```
/* Create a new thread to execute thread1_func() */
ret_val = pthread_create(&tid, &attr,
                        (void *(*)(void*))thread1_func,
                        (void *)NULL);
check_error(ret_val, "pthread_create()");
```

```
/* Its safe to destroy the attr object now */
ret_val = pthread_attr_destroy(&attr);
check_error(ret_val, "pthread_attr_dest()");
```

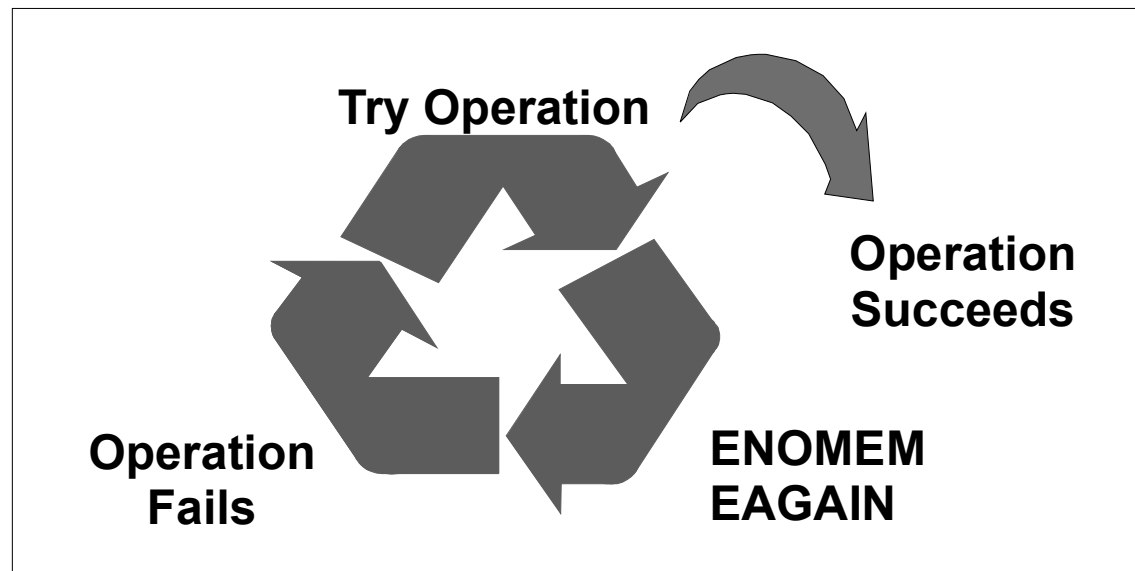
```
/* Rest of the program goes here */
```

```
/* Wait for thread to finish executing */
ret_val = pthread_join(tid, (void **)NULL);
check_error(ret_val, "pthread_join()");
}
```

```
void
thread1_func()
{
    printf("child thread executing\n");
}
```

# Gestione degli errori

- Gli esempi presentati prevedono la terminazione dell'applicazione in caso di errore. Questo comportamento potrebbe essere non sempre adeguato e per determinati errori essere più appropriato intraprendere delle azioni.
  - ◆ Ad esempio, **pthread\_create** ritorna **EAGAIN** se mancano le risorse di sistema per un nuovo thread: l'applicazione può decidere di passare il lavoro ad un thread esistente, oppure di tentare di nuovo.
  - ◆ Se **ESRCH** è ritornato da **pthread\_join** può essere che: (1) il thread si è già ricongiunto, (2) il thread è distaccato, oppure (3) il thread non esiste.
- Molti altri errori ritornati dalle funzioni di gestione dei thread possono essere trattati come fatali.



# Esempio 8

- Un processo crea due thread ed attende la loro terminazione.
  - ◆ Il primo thread stampa sullo standard output:
    - ✓ l'ID di processo,
    - ✓ il suo ID di thread.
    - ✓ Infine, si addormenta per 3 secondi, ritornando un valore NULL.
  - ◆ Il secondo thread stampa sullo standard output:
    - ✓ l'ID di processo,
    - ✓ il suo ID di thread.
    - ✓ Infine, si addormenta per 4 secondi, ritornando un valore NULL.
  
- Per la gestione degli errori, si utilizzi la funzione *check\_error*, definita nell'esempio 5.

# Esempio 8

```
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <errno.h>

void * funzione1(void *), * funzione2(void *);

int main( ) {
    pthread_t tid1,tid2;
    int retcode;

    retcode = pthread_create(&tid1,NULL,funzione1,NULL);
    check_error(retcode, "pthread_create()");

    retcode = pthread_create(&tid2,NULL,funzione2,NULL);
    check_error(retcode, "pthread_create()");

    printf("Processo padre (pid %d) tid=%ld\n", getpid(),
           pthread_self());
    retcode = pthread_join(tid1,NULL);
    check_error(retcode, "pthread_join()");

    retcode = pthread_join(tid2,NULL);
    check_error(retcode, "pthread_join()");

    printf("Fine lavori\n");
    exit(0);
}
```

# Esempio 8

```
void * funzione1(void * p)
{
    printf("Thread figlio (pid %d) tid = %ld \n",getpid(),
        pthread_self());
    sleep(3);
    return(NULL);
}

void * funzione2(void * p)
{
    printf("Thread figlio (pid %d) tid = %ld \n",getpid(),
        pthread_self());
    sleep(4);
    return(NULL);
}
```

# Esempio 9

- Un processo crea un thread per ognuno dei 10 compiti che deve eseguire. Ogni task è fatto così:

```
void * worker (void * tasks)
{
int my_task = *((int *)tasks);
int res;

sleep( mytask - 1 );
res = my_task * my_task;
printf("%d: = %d\n", pthread_self(), res);
return((void *)&res);
}
```

# Esempio 9

- Il thread iniziale ha il seguente main:

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

int tasks [] = {1,2,3,4,5,6,7,8,9,10};
#define NTASKS 10
void * worker (void *);

int main(int argc, char * argv[]) {
    int i;
    pthread_t tid[NTASKS];
    void * res;
    for(i=0;i<NTASKS;i++) {
        pthread_create(&tid[i],NULL,worker,(void *)&tasks[i]);
    }
    sleep(15);
    for(i=0;i<NTASKS;i++) {
        pthread_join(tid[i],&res);
        printf("Thread %d terminato con ris = %d\n", i, *((int *)res));
    }
    exit(0);
}
```

- Nella soluzione proposta, il risultato talvolta non e' corretto. Perché?

# Esempio 10

- Modificando la funzione **worker** del thread in questa maniera:

```
void * worker (void * tasks)
```

```
{
```

```
int my_task = *((int *)tasks);
```

```
int * res = calloc(1, sizeof(int));
```

```
sleep( mytask - 1 );
```

```
* res = my_task * my_task;
```

```
printf("%d: = %d\n", pthread_self(), res);
```

```
return((void *)&res);
```

```
}
```

- funziona. **Perché?**
- Quali altre possibilità abbiamo?

# Esercizi

- Completare gli esempi con le parti di codice mancante, compilarli e verificarne il corretto funzionamento.
- Scrivere un programma multithreaded produttore-consumatore a buffer limitato che legga righe dallo standard output e le stampi a terminale ogni N secondi.
- Riscrivere gli esempi proposti in Java.