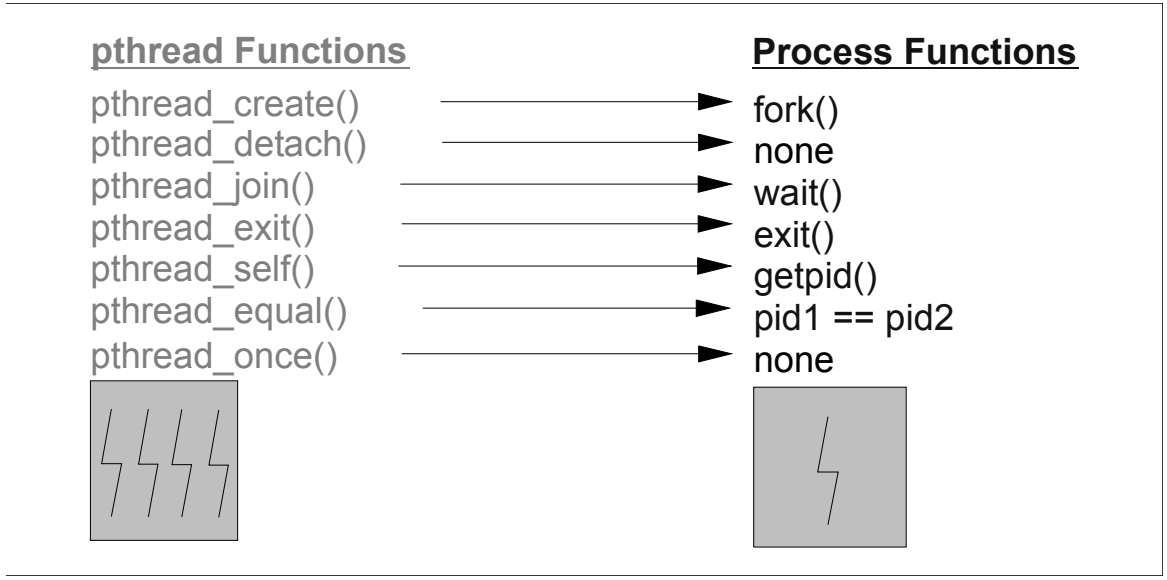
I Thread POSIX

- Relazione tra modello a processi e a thread
- Creazione di un thread
- Attesa
- Distacco
- Terminazione
- Ricerca e confronto
- Inizializzazione dinamica

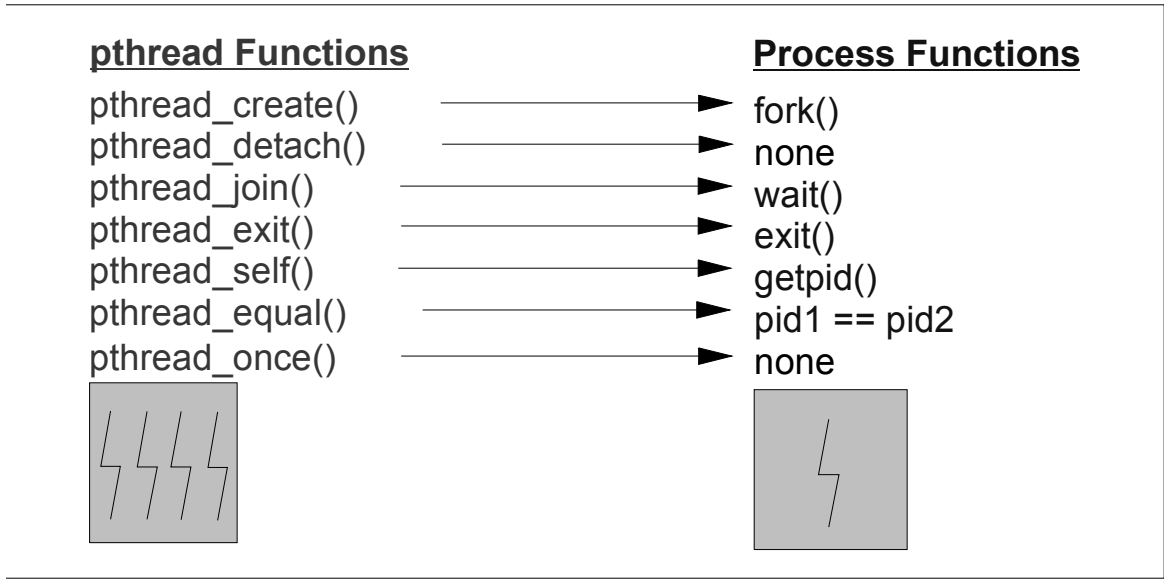
Relazione tra thread e processi

- Nel modello a processi, un processo è creato, ha una vita ed è terminato. All'atto della creazione è associato con un segmento di codice (un programma) da eseguire.
- Durante la sua vita, eseguirà dormirà, e si bloccherà molte volte. Alla sua terminazione le sue risorse saranno rilasciate in due tempi:
  - Quanto termina, la memoria assegnata e molte delle sue risorse verranno ritornate al sistema, eccetto lo stack del kernel e il PCB (stato di uscita).
  - Quando il suo genitore esegue **wait**, le statistiche di esecuzione verranno salvate e lo stato di terminazione del figlio ritornate al genitore.



Relazione tra thread e processi

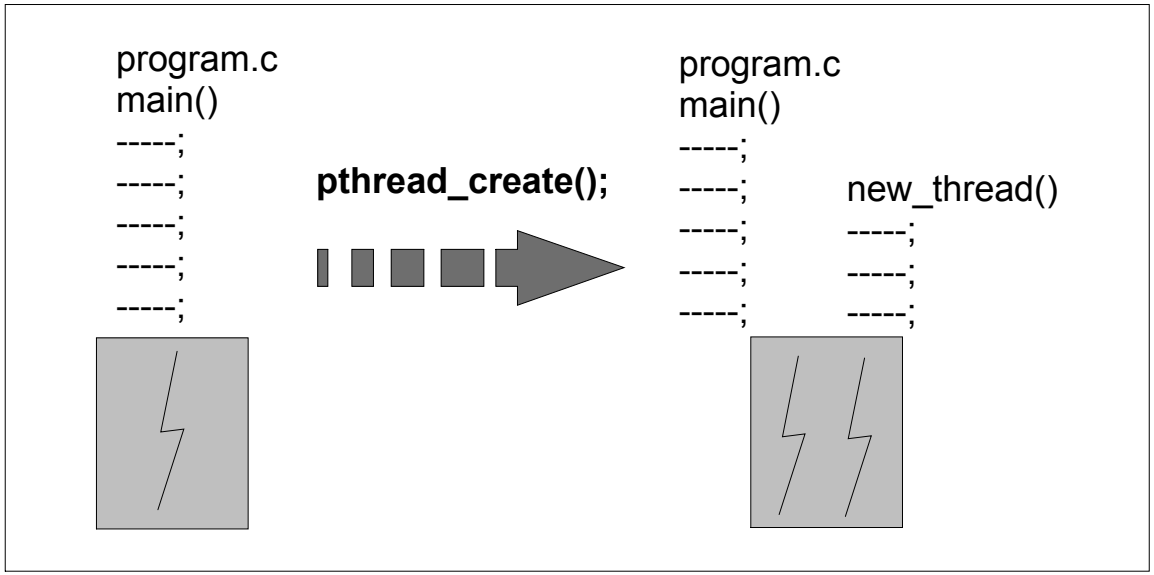
- Nel modello a thread, un thread è creato, ha una vita ed è terminato. All'atto della creazione è associato con un segmento di codice:
  - un programma da eseguire, se il processo ha un singolo thread,
  - Una funzione, se il processo è multithreaded.
- Durante la sua vita, eseguirà dormirà, e si bloccherà molte volte. Alla sua terminazione *il programmatore deve fare in modo che le sue risorse siano ritornate al sistema*, con un'operazione di **wait** a levello del thread.



Creare un thread

- Tutti i processi contengono almeno un thread, detto thread **principale** o **iniziale**, creato automaticamente all'atto della creazione del processo.
- Tutti i thread aggiuntivi vengono creati con

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



Creare un thread

- Tutti i processi contengono almeno un thread, detto thread **principale** o **iniziale**, creato automaticamente all'atto della creazione del processo.
- Tutti i thread aggiuntivi vengono creati con

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

- **pthread_create** è simile alla funzione **fork** usata per creare un nuovo processo, ma con delle differenze; ad esempio, con **fork** l'intero codice è schedulato per l'esecuzione.
- **pthread_create** ritorna in *thread_id* l'identificativo del nuovo thread.
- Il parametro *attr* è l'oggetto degli attributi, quali la grandezza dello stack, l'indirizzo dello stack, le politiche di priorità e scheduling e lo stato di detach.
- Un thread eredita la maschera dei segnali, la politica di scheduling, e la priorità, ma non eredita i segnali pendenti.

Esempio: Hello World!

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

void            fatal_error(int err_num, char *function);
void            thread1_func();

/* Print "Hello World!", then "Good-bye World!" */
main()
{
   pthread_t    tid;
   int          return_val;

    /* Create a new thread to execute thread1_func(). */
   return_val = pthread_create(&tid, (pthread_attr_t *)NULL,
                              (void *(*)())thread1_func,
                              (void *)NULL);
   if (return_val != 0)
        fatal_error(return_val, "pthread_create()");

/* Wait for the thread to finish executing, then... */
   return_val = pthread_join(tid, (void **)NULL);
   if (return_val != 0)
        fatal_error(return_val, "pthread_join()");

   /* 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);
}


/* Function to print "Hello World!". */
void
thread1_func()
{
   printf("Hello World!\n");
   pthread_exit((void *)NULL);
}
```

- Le funzioni Pthread non inizializzano **errno**; ritornano **0** in caso di successo
- Non è detto che un thread è in esecuzione dopo **pthread_create**; è possibile che esso abbia già finito la sua esecuzione quando la funzione ritorna.
- Se *arg* è un puntatore, **non deve** puntare ad una variabile locale del thread

Aspettare un thread

- Nel modello a processi, un processo può sospendere la sua esecuzione e attendere la terminazione di un figlio chiamando **wait** o **waitpid**.
- Così il genitore conosce lo stato di terminazione del figlio. Se non lo attende, le risorse di sistema non vengono rilasciate fino alla terminazione del padre.
- Nel modello a thread, le risorse di un thread possono essere reclamate con

```
int pthread_join(
        pthread_t       thread_id,
        void            **return_val
);
```

Time 0	*thread_a* calls pthread_join() on *thread_b*
Time 1	*thread_b* executes
Time 2	*thread_b* terminates
Time 3	*thread_a* resumes execution and optionally reaps the exit status from *thread_b*

Aspettare un thread

- Nel modello a processi, un processo può sospendere la sua esecuzione e attendere la terminazione di un figlio chiamando **wait** o **waitpid**.
- Così il genitore conosce lo stato di terminazione del figlio. Se non lo attende, le risorse di sistema non vengono rilasciate fino alla terminazione del padre.
- Nel modello a thread, le risorse di un thread possono essere reclamate con

```
int pthread_join(
        pthread_t       thread_id,
        void            **return_val
);
```

- **pthread_join** sospende il chiamante fino a che *thread_id* non è terminato.
- Nel modello a processi solo i genitori possono attendere i figli; un thread può invece attendere un qualsiasi altro thread *joinable* nel processo.
- Lo stato di uscita del processo è conservato in *return_val.*

Esempio: Hello World! (1/4)

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

#define MAX_THR_MSG    100
#define HELLO_MSG      "Hello World!"

struct thr_rtn {
   int  completed;
   char in_msg[MAX_THR_MSG + 1];
   char out_msg[MAX_THR_MSG + 1];
};

void   fatal_error(int err_num, char *function);
void   thread1_func(struct thr_rtn *msg);

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

Esempio: Hello World! (2/4)

```
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);
}

void thread1_func(struct thr_rtn *msg)
{
   printf("%s\n", msg->in_msg);
   strcpy(msg->out_msg, msg->in_msg);
   msg->completed = 1;
   pthread_exit((void *)msg);
}
```

Esempio: Hello World! (3/4)

```
main()
{
   pthread_t   tid;
   int         return_val;
   struct thr_rtn     *ptr;
   struct thr_rtn     msg;

   /* Initialize parameter to thread1_func() */
   strcpy(msg.in_msg, HELLO_MSG);
   msg.out_msg[0] = NULL;
   msg.completed = 0;

   /* Create a new thread to execute thread1_func() */
   return_val = pthread_create(&tid,(pthread_attr_t *)NULL,
                               (void *(*)())thread1_func,
                               (void *)&msg);
   check_error(return_val, "pthread_create()");

   /* Wait for thread to finish executing */
   return_val = pthread_join(tid, (void **)&ptr);
   check_error(return_val, "pthread_join()");
```

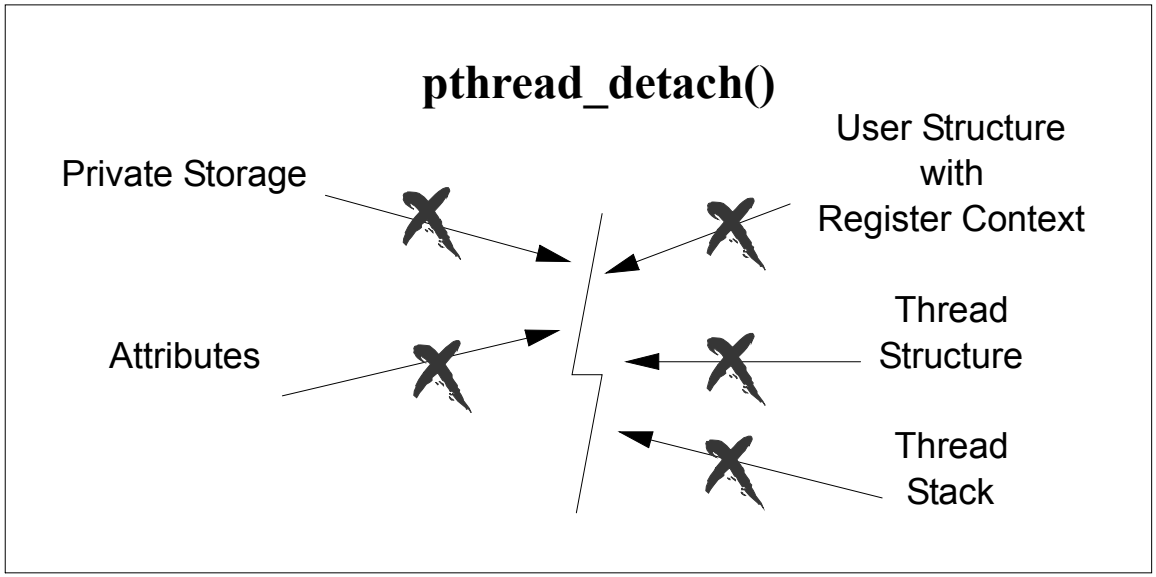
Esempio: Hello World! (4/4)

```
   /* Check return parameter for validity */
   if (ptr != &msg) {
       fprintf(stderr, "ptr != &msg\n");
       exit(-1);
   } else if (ptr->completed != 1) {
       fprintf(stderr, "ptr->completed != 1\n");
       exit(-1);
   } else if (strcmp(ptr->in_msg, HELLO_MSG) != 0) {
       fprintf(stderr, "ptr->in_msg != %s\n", HELLO_MSG);
       exit(-1);
   } else if (strcmp(ptr->out_msg, HELLO_MSG) != 0) {
       fprintf(stderr, "ptr->out_msg != %s\n", HELLO_MSG);
       exit(-1);
   }

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

Distacco

- Quando un thread si distacca, all'atto della terminazione tutte le sue risorse sono reclamate dal sistema.
- Se un thread è distaccato, il suo stato non è disponibile agli altri thread; per distaccarsi si usa:

```
int pthread_detach(
        pthread_t       pthread_id
);
```



Distacco

- Quando un thread si distacca, all'atto della terminazione tutte le sue risorse sono reclamate da sistema.
- Se un thread è distaccato, il suo stato non è disponibile agli altri thread; per distaccarsi si usa:

```
int pthread_detach(
        pthread_t       pthread_id
        );
```

- In un qualche momento, per ogni thread nell'applicazione è necessario chiamare o **pthread_detach** o **pthread_join**.
- Nel modello a processi si può ottenere lo stesso comportamento eseguendo una "**fork** su **fork**".
- Una volta terminato il thread, *pthread_id* può essere riutilizzato

Esempio 2 (1/2)

```
#include <pthread.h>

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

#define QUIT   0

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

main()
{
   pthread_t   tid;
   int         return_val;
   int         cmd;

   while (1) {

        /* Get the user's command */
        cmd = get_user_cmd();
        if (cmd == QUIT)
                break;

        /* Create a new thread */
        return_val = pthread_create(&tid, (pthread_attr_t *)NULL,
                                    (void *(*)())thread_func,
                                    (void *)cmd);
        check_error(return_val, "pthread_create()");
```

Esempio 2 (2/2)

```
        /* Now detach the thread */
        return_val = pthread_detach(tid);

        check_error(return_val, "pthread_detach()");
    }

    /* Say Good-bye */
    printf("Program Terminating...\n");
    exit(0);
}


/* Function to process user's command */
void thread_func(int cmd)
{
    /* Real application processing code goes here... */
    printf("Processing command %d\n", cmd);

    pthread_exit((void *)NULL);
}


int get_user_cmd()
{
    static int  i = 5;

    /* Replace this with a "real" get_user_cmd()... */
    return(--i);
}
```

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_prt
        );
```

```
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_prt
        );
```

- *value_prt* 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_prt* che punta ad una struttura dati contenente informazioni dettagliate.
- Se un thread ha chiamato **pthread_cancel** per un altro thread, questo può liberare le risorse usando **pthread_cleanup_push** e **pthread_cleanup_pop**

Esempio 3 (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 *(*)())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 *(*)())thread2_func,
   (void *)NULL);
   check_error(return_val, "pthread_create() - 2");
```

Esempio 3 (2/3)

```
   /* Create thread #3 */
   return_val = pthread_create(&tid[2], (pthread_attr_t *)NULL,
                               (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);
}
```

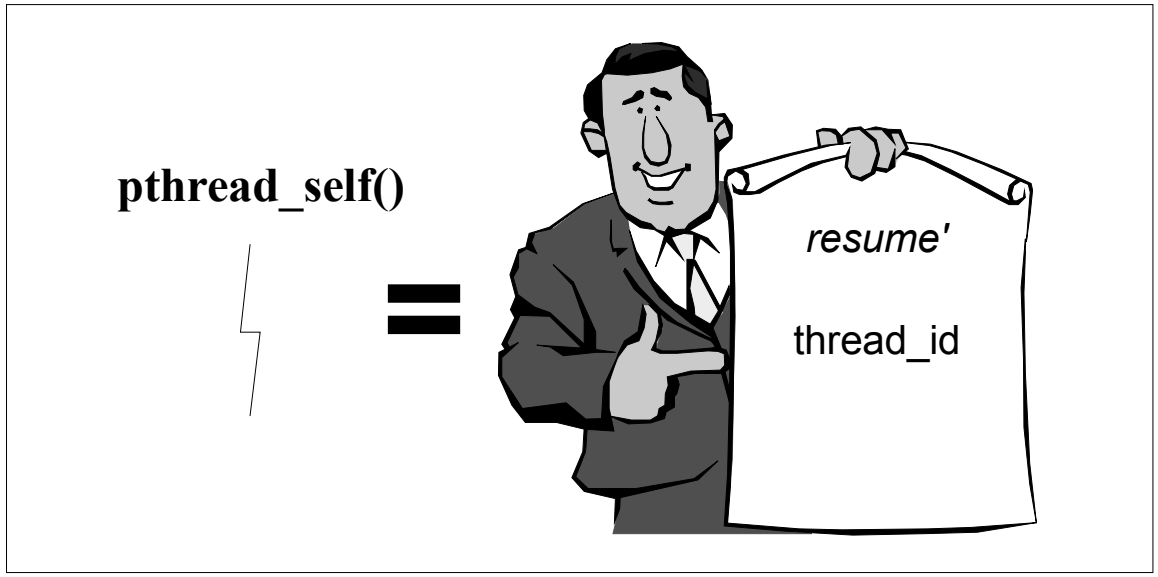
Esempio 3 (3/3)

```
/* 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);
}
```

Ricerca e confronto

- L’identificativo del processo è ritornato da **getpid** ed è di tipo **pid_t**; ciascun thread è individuato univocamente da un identificativo di tipo *pthread_t.*
- Per ottenere l’identificativo del thread corrente si usa:

```
pthread_t pthread_self( void );
```



Identificativo di thread

- L'identificativo del processo è ritornato da **getpid** ed è di tipo **pid_t**; ciascun thread è individuato univocamente da un identificativo di tipo *pthread_t.*
- Per ottenere l'ID del thread corrente si usa:

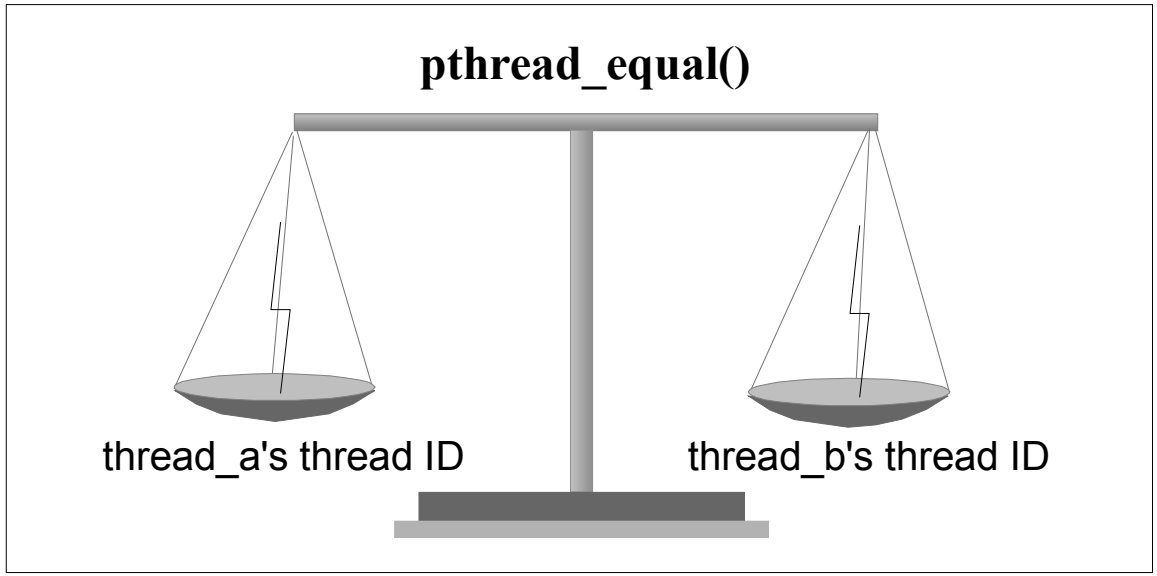
```
pthread_t pthread_self(void);
```

- Gli ID sono unici all'interno di ogni singolo processo.
- Nel modello a processi, **pid_t** può essere un **long** e si può usare un operatore di confronto **==**. Nei thread, **pthread_t** è un tipo *opaco*, e potrebbe essere una struttura.

Confronto

- Per confrontare gli ID di due thread bisogna usare:

```
int pthread_equal(
        pthread_t       thread1,
        pthread_t       thread2);
```



Esempio 4 (1/2)

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

extern void    fatal_error(int err_num, char *function);
void           thread_func();
pthread_t      main_thread, child_thread;

main()
{
   int         return_val;

   /* Who are we? */
   main_thread = pthread_self();

   /* Create a new thread to execute thread1_func(). */
   return_val = pthread_create(&child_thread,
                           (pthread_attr_t *)NULL,
                           (void *(*)())thread_func,
                           (void *)NULL);
   check_error(return_val, "pthread_create()");

   /* Call the same function as the child thread. This function
    * will call pthread_exit() for us (we never return). */
   thread_func();
}
```

Esempio 4 (2/2)

```
void
thread_func()
{
   pthread_t  calling_thread;

   calling_thread = pthread_self();

   if (pthread_equal(calling_thread, main_thread) == 0) {
       printf("Main  thread in thread_func()\n");
   } else {
       printf("Child thread in thread_func()\n");
   }

   pthread_exit((void *)NULL);
}
```

Inizializzazione dinamica

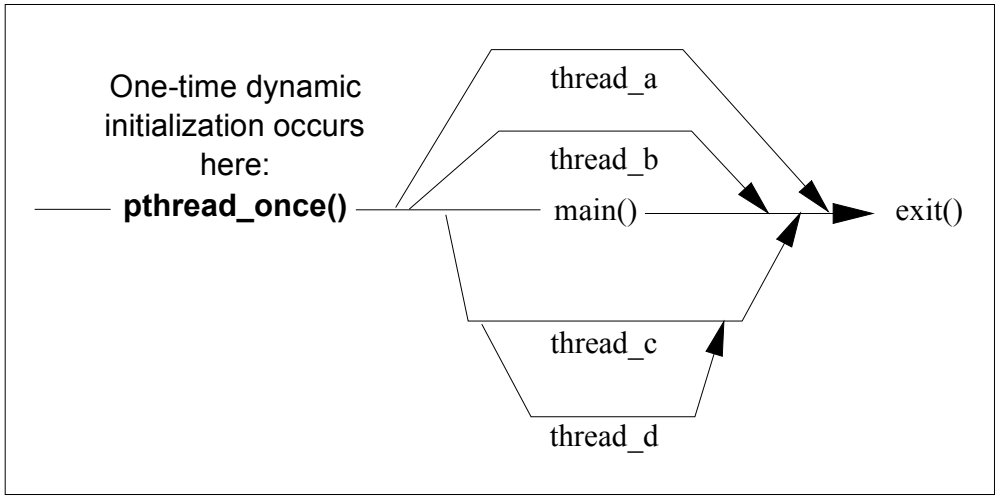
- Molte routine di libreria e applicazioni sono inizializzate dinamicamente. L'inizializzazione di un modulo avviene automaticamente quando esso viene chiamato la prima volta.
- L'inizializzazione dinamica viene utilizzata di solito per moduli che possono non andare in esecuzione. Ad esempio si può usare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

```
static int func_x_inizialized = 0;
void func_x_init( )
{
        /* Qui va il codice di inizializzazione */
        …
}
void func_x( )
{
        if (func_x_inizialized == 0){
                func_x_init ()
                func_x_inizialized = 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_inizialized == 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)
);
```



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_inizialized == 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 funzionedi 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**

Esempio

```
#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 *(*)())func_x, (void *)NULL);
   check_error(return_val, "pthread_create() - 1");

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

   func_x();
}
```

Esercizi

- Completare gli esempi con le parti di codice mancante.
- Scrivere un programma multithreaded produttore-consumatore a buffer limitato che legge linee dallo standard output e le stampa a terminale ogni N secondi.
- Riscrivere gli esempi proposti in Java.