

# Thread

Laboratorio Sistemi Operativi

Giuseppe Salvi

Email: [giuseppe.salvi@uniparthenope.it](mailto:giuseppe.salvi@uniparthenope.it)

# Introduzione

- Abbiamo visto l'organizzazione dell'ambiente di un processo UNIX, la relazione tra i processi ed il modo per gestire i processi
- Andando più a fondo nello studio dei processi è possibile vedere come sia possibile usare più *thread di controllo* (o semplicemente *thread*) per eseguire compiti multipli nell'ambiente di un singolo processo
- Tutti i *thread* all'interno di un singolo processo hanno accesso alle stesse componenti del processo quali, ad esempio, i descrittori di file e la memoria

# Motivazioni

- I processi concorrenti velocizzano l'esecuzione delle applicazioni, ma ...
  - I context switch introducono un elevato overhead
- Overhead
  - Esecuzione
    - Salvataggio dello stato della CPU del processo in esecuzione
    - Caricamento dello stato della CPU del nuovo processo
  - Uso risorse
    - Commutazione del contesto del processo
      - Informazioni sulle risorse allocate al processo
      - Informazione sull'interazione con altri processi

# Motivazioni (cont.)

- Supponiamo di avere un processo  $P$  con due processi figli  $P_i$  e  $P_j$ 
  - $P_i$  e  $P_j$  ereditano il contesto del processo  $P$
  - Se  $P_i$  e  $P_j$  non hanno allocato alcuna risorse il loro contesto è identico
  - Differiscono solo per stato di CPU e stack
- Il context switch tra  $P_i$  e  $P_j$  coinvolge molte informazioni ridondanti
- I thread sfruttano tale considerazione
  - Esecuzione di un programma che usa le risorse di un processo
- I thread suddividono lo stato del processo in due parti
  - Stato delle risorse, associato al processo
  - Stato dell'esecuzione, associato ad ogni thread
- Solo gli stati di esecuzione devono essere scambiati nella commutazione tra thread
- Lo stato delle risorse è condiviso

# Perché usare i thread?

- Con thread di controllo multipli è possibile sviluppare programmi in grado di eseguire più di un compito (task) alla volta nell'ambito di un singolo processo
  - Ogni thread gestisce un compito separato
- Vantaggi
  1. Si semplifica il codice relativo alla gestione di eventi asincroni assegnando un thread differente ad ogni evento di un tipo specifico
    - Ogni thread può gestire il proprio evento mediante un modello di programmazione sincrona (molto più semplice di un modello asincrono)

# Perché usare i thread? (cont.)

2. I thread hanno accesso allo stesso spazio di indirizzi di memoria e di descrittori di file
3. E' possibile suddividere alcuni problemi in modo da migliorare il throughput complessivo del programma
  - Un processo singolo che ha più compiti da risolvere serializza implicitamente tali task poiché c'è solo un thread di controllo
  - Con più thread di controllo, l'elaborazione dei compiti indipendenti può essere intrecciata assegnando un thread separato per ogni compito
  - I programmi interattivi possono ottenere migliori tempi di risposta utilizzando thread multipli per separare parti di programma relative ad input degli utenti e ad output di altre parti del programma

# Concetto di Thread

- Un thread consiste delle informazioni necessarie per rappresentare un contesto d'esecuzione in un processo
  - Thread ID che identifica il thread in un processo
  - insieme di valori dei registri
  - stack
  - politica di scheduling e relativa priorità
  - Maschera per i segnali
  - Variabile errno
  - Dati specifici del thread
- Ogni cosa all'interno di un processo è condivisibile tra i thread di un processo
  - Testo del programma eseguibile
  - Memoria globale e di heap del programma
  - Descrittori di file

# Thread POSIX (pthread)

- POSIX definisce l'API `pthread` utilizzabile nei programmi scritti in C
  - Consiste di 60 routine che effettuano operazioni per
    - **Gestione dei thread:**
      - Creazione, recupero stato, terminazione normale o anormale, attesa per la terminazione, impostazione degli attributi di schedulazione e dimensionamento dello stack
    - **Supporto per la condivisione dei dati**
      - mutex
    - **Supporto per la sincronizzazione**
      - variabile di condizione

# Libreria Pthread

# Identificazione di thread

- Ogni thread ha un identificatore di thread (thread ID)
  - L'ID di un thread ha senso solo nel contesto del processo a cui appartiene
- Un ID di thread è rappresentato dal tipo di dato *pthread\_t*
  - Le implementazioni portabili non possono trattarli come interi (in alcune implementazioni *pthread\_t* è un puntatore ad una struttura)
  - E' necessaria una funzione per confrontare gli ID di due thread

```
#include <pthread.h>
int pthread_equal (pthread_t tid1, pthread_t tid2);

// restituisce un valore non nullo se uguali, 0
// altrimenti
```

# Identificazione di Thread

- Il fatto che `pthread_t` possa essere una struttura comporta l'impossibilità di avere un modo portabile per stampare il suo valore
- Talvolta è utile stampare l'ID di un thread durante il debugging di un programma, altrimenti non c'è necessità di farlo
  - Il peggio che si può avere è un codice di debug non portabile, ma ciò non è un grosso limite

# Identificazione dei thread

- Un thread può ottenere il proprio ID invocando la funzione `pthread_self`

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

```
pthread_t pthread_self(void)
```

```
// restituisce l'ID del thread invocante
```

- Questa funzione può essere usata con `pthread_equal` quando un thread ha bisogno di identificare le strutture dati etichettate con il proprio ID di thread
  - Un thread master può impostare dei carichi di lavoro su di una coda ed usare l'ID del thread per controllare quale job va a ciascun thread di lavoro

# Creazione di thread

- Il modello di processo UNIX tradizionale supporta solo un thread di controllo per processo
  - Concettualmente ciò è equivalente ad un modello basato su thread dove ogni processo è costituito da un solo thread
- Con i pthread, un programma inizia l'esecuzione come un singolo processo con un singolo thread di controllo
  - Con il procedere dell'esecuzione il suo comportamento è indistinguibile dal processo tradizionale fino a che esso crea più thread di controllo
- I thread possono essere creati invocando la funzione

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

```
int pthread_create(pthread_t *tidp, const pthread_attr_t *attr,  
void *(*start_rtn) (void *), void *arg)
```

```
// restituisce 0 se OK, numero di errore se fallisce
```

# Creazione di thread (2)

- Se la funzione ritorna con successo, la locazione di memoria puntata da *tidp* è impostata all'ID del nuovo thread creato
- L'argomento *attr* è utilizzato per definire vari attributi del thread (per ora consideriamolo impostato a **NULL** che corrisponde ad un thread con attributi di default)
- Il nuovo thread inizia l'esecuzione all'indirizzo della funzione *start\_rtn*
  - Tale funzione prende un solo argomento, *arg* (un puntatore senza tipo)
  - Se è necessario passare più di un argomento alla funzione *start\_rtn*, bisogna memorizzarli in una struttura e passare l'indirizzo della struttura in *arg*

# Creazione di thread (3)

- Quando è creato un thread non c'è alcuna garanzia su quale thread viene eseguito per primo: il thread appena creato o quello invocante
- Il nuovo thread ha accesso allo spazio di indirizzi del processo ed eredita l'ambiente e la maschera dei segnali
  - L'insieme dei segnali pendenti è cancellato e dunque non ereditato

# Creazione di thread (4)

- E' da osservare che le funzioni `pthread` restituiscono un codice di errore quando falliscono
  - Esse non impostano `errno` come le altre funzioni POSIX
  - Viene creata una copia di `errno` solo per compatibilità con le funzioni esistenti che la usano
- Con i thread, è più corretto restituire il codice di errore dalla funzione, restringendo di fatto l'ambito dell'errore alla funzione che lo ha causato

# Creazione di thread: esempio

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

pthread_t ntid;
void printids(const char *s)
{
    pid_t      pid;
    pthread_t   tid;
    pid = getpid();
    tid = pthread_self();
    printf("%s pid %lu tid %lu (0x%lx)\n", s, (pid_t)
    pid, (unsigned long) tid, (unsigned long) tid);
}
void * thr_fn(void *arg)
{
    printids("nuovo thread: ");
    return ((void *)0);
}
```

# Creazione di thread: esempio

```
int main(void)
{
int err;
err = pthread_create(&ntid, NULL, thr_fn, NULL);
if (err!=0){
    fprintf(stderr, "non posso creare il thread: %s \n", strerror(err));
    exit(1);
}
printids("thread principale:");
sleep(1);
exit(0);
}
```

# Creazione di thread: esempio (cont.)

- Il programma crea un thread e stampa gli ID del processo, del nuovo thread e del thread iniziale
- L'esempio ha due particolarità necessarie per gestire le race condition tra il thread principale e il nuovo thread
  1. Necessità di invocare una *sleep* nel thread principale
    - Senza la *sleep* il thread principale potrebbe uscire, terminando l'intero processo prima che il nuovo thread abbia la possibilità di iniziare l'esecuzione
    - Questo comportamento dipende dall'implementazione dei thread del sistema operativo e dagli algoritmi di scheduling

# Creazione di thread: esempio (cont.)

2. Il nuovo thread ottiene il suo ID chiamando `pthread_self` invece di leggerlo dalla memoria condivisa o riceverlo come argomento alla sua routine di inizio thread
  - Ricordiamo che `pthread_create` restituirà l'ID del nuovo thread attraverso il primo parametro (`tidp`)
  - Nell'esempio, il thread principale memorizza questo in `ntid`, ma il nuovo thread non può usarlo in maniera sicura
    - Se il nuovo thread viene eseguito prima che il thread principale ritorni dalla chiamata a `pthread_create`, allora il nuovo thread vedrà il contenuto non inizializzato di `ntid` invece dell'ID del thread

# Creazione di thread: esempio (cont.)

- Linux

```
$/a.out
```

```
thread principale: pid 17874 tid 140693894424320  
(0x7ff5d9996700)
```

```
nuovo thread:      pid 17874 tid 140693886129920  
(0x7ff5d91ad700)
```

- Gli ID dei thread sembrano puntatori, anche se sono rappresentati come interi lunghi senza segno

# Creazione thread: passaggio parametri

```
#include<pthread.h>
#include<stdio.h>
/* parametri per char_print */
struct char_print_parms
{
    char character;
    int count;
};

void* char_print (void* parameters)
{
    /* cast del puntatore al tipo corretto */
    struct char_print_parms* p = (struct char_print_parms*) parameters;
    int i;
    for (i=0; i<p->count; ++i)
        fputc(p->character,stderr);
    return NULL;
}
```

# Creazione thread: passaggio parametri (cont.)

```
int main() {
pthread_t tid1;
pthread_t tid2;
struct char_print_parms tid1_args;
struct char_print_parms tid2_args;

/* crea un thread per stampare 30000 'x' */
tid1_args.character = 'x';
tid1_args.count = 30000;
pthread_create(&tid1,NULL, char_print, (void *)&tid1_args);

/* crea un thread per stampare 20000 'y' */
tid2_args.character = 'y';
tid2_args.count = 20000;
pthread_create(&tid2,NULL, char_print, (void*)&tid2_args);

sleep(1);

return 0;
}
```

# Terminazione di thread

- Se un qualsiasi thread in un processo chiama `exit`, `_Exit` o `_exit`, allora l'intero processo termina
- Analogamente, quando l'azione di default è di terminare il processo, un segnale inviato al thread terminerà l'intero processo
- Un singolo thread può uscire in tre modi, arrestando il suo flusso di controllo, senza terminare l'intero processo
  - Il thread torna dalla routine di avvio. Il valore di ritorno è il codice di uscita del thread
  - Il thread può essere cancellato da un altro thread nello stesso processo
  - Il thread chiama `pthread_exit`

# Terminazione di thread

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

```
void pthread_exit(void *rval_ptr);
```

- `rval_ptr` è un puntatore senza tipo, simile all'argomento passato alla routine di avvio
  - Il puntatore è disponibile agli altri thread del processo chiamando la funzione `pthread_join`

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

```
int pthread_join(pthread_t thread, void **rval_ptr);
```

```
// restituisce 0 se OK, numero di errore se fallisce
```

# Terminazione di thread (cont.)

- Il thread chiamante si bloccherà fino a che il thread specificato chiama `pthread_exit`, ritorna dalla sua routine di avvio, o è cancellato
- Se il thread ritorna dalla sua routine di avvio, `rval_ptr` conterrà il codice di ritorno
- Se il thread è cancellato, la locazione di memoria specificata da `rval_ptr` è impostata a `PTHREAD_CANCELED`
- Chiamando `pthread_join`, poniamo automaticamente il thread con cui facciamo il join nello stato *distaccato* (detached) in modo che le sue risorse possono essere recuperate
  - Se il thread era già nello stato distaccato, `pthread_join` può fallire, restituendo `EINVAL`

# Terminazione di thread (cont.)

- Se non si è interessati al valore di ritorno del thread, è possibile impostare il valore di *rval\_ptr* a NULL
- In questo caso, la chiamata a *pthread\_join* ci consente di aspettare il thread specifico, ma non recupera lo stato di terminazione del thread

# Terminazione di thread: esempio

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

void *
thr_fn1(void *arg)
{
    printf("thread 1 returning\n");
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2 exiting\n");
    pthread_exit((void *)2);
}
```

# Terminazione di thread: esempio

```
int
main(void)
{
    int          err;
    pthread_t    tid1, tid2;
    void         *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_quit("can't create thread 1: %s\n", strerror(err));
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_quit("can't create thread 2: %s\n", strerror(err));
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_quit("can't join with thread 1: %s\n", strerror(err));
    printf("thread 1 exit code %d\n", (long)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_quit("can't join with thread 2: %s\n", strerror(err));
    printf("thread 2 exit code %d\n", (long)tret);
    exit(0);
}
```

# Terminazione di thread: esempio

```
$ ./a.out
```

```
thread 1 returning
```

```
thread 2 exiting
```

```
thread 1 exit code 1
```

```
thread 2 exit code 2
```

# Stato di uscita: esempio 1

```
#include <pthread.h>
#include<stdio.h>
#include<stdlib.h>
void* thread1(void *arg) {
    int error;
    error=*(int*)arg;
    printf("Sono il primo thread. Parametro = %d\n", *(int*)arg);
    pthread_exit((void*) (long)error);
}
void* thread2(void *arg) {
    static long error;
    error=*(int*)(arg);
    printf("Sono il secondo thread. Parametro = %d\n", (int)error);
    pthread_exit((void*)&error);
}
```

# Stato di uscita: esempio 1 (cont.)

```
int main()
{
pthread_t th1, th2;
int i1 = 1, i2=2;
void* uscita;
pthread_create(&th1, NULL, thread1, (void*)&i1);
pthread_create(&th2, NULL, thread2, (void*)&i2);
pthread_join(th1, &uscita);
printf("stato = %ld\n", (long)uscita);
pthread_join(th2, &uscita);
printf("stato = %ld\n", *(long*)uscita);
exit(0);
}
```

# Stato di uscita: esempio 2

```
void * compute_prime(void* arg) {
    static int candidate = 2;
    int n = *((int*) arg);
    int factor, is_prime;
    while (1) {
        is_prime = 1;
        for (factor = 2; factor < candidate; ++factor)
            if (candidate % factor == 0){
                is_prime = 0;
                break;
            }
        if (is_prime){
            if (--n == 0)
                return (void*)&candidate;
        }
        ++candidate;
    }
    return NULL;
}
```

# Stato di uscita: esempio 2 (cont.)

```
int main(){
pthread_t tid;
int which_prime = 5000;
void* prime;

pthread_create(&tid, NULL, compute_prime, (void*)&which_prime);

/* attende il calcolo del numero primo */
pthread_join(tid, &prime);
printf("The %dth prime number is %d.\n", which_prime, *(int*)prime);
exit(0);
}
```

# Joining di thread: esempio

```
#include<pthread.h>
#include<stdio.h>
/* parametri per print_function */
struct char_print_parms
{
    char character;
    int count;
};

void* char_print (void* parameters)
{
    /* cast del puntatore al tipo corretto */
    struct char_print_parms* p = (struct char_print_parms*)
parameters;
    int i;
    for (i=0; i<p->count; ++i)
        fputc(p->character, stderr);
    return NULL;
}
```

# Joining di thread: esempio (cont.)

```
int main() {
pthread_t tid1;
pthread_t tid2;
struct char_print_parms tid1_args;
struct char_print_parms tid2_args;
tid1_args.character = 'x';
tid1_args.count = 30000;
pthread_create(&tid1,NULL, char_print, &tid1_args);

tid2_args.character = 'y';
tid2_args.count = 20000;
pthread_create(&tid2,NULL, char_print, &tid2_args);

pthread_join(tid1, NULL);
pthread_join(tid2, NULL);

return 0;
}
```

# Ancora sul passaggio dei parametri

- Il puntatore senza tipo passato a `pthread_create` e `pthread_exit` può essere usato per passare più di un valore singolo
- Il puntatore può essere usato per passare l'indirizzo di una struttura contenente informazioni più complesse
  - E' necessario prestare attenzione al fatto che la memoria usata per la struttura sia ancora valida quando il chiamante ha finito
  - Se la struttura è stata allocata sullo stack del chiamante che passa un puntatore a questa struttura a `pthread_exit`, allora lo stack potrebbe essere cancellato e la sua memoria riutilizzata nel momento in cui il thread che invoca `pthread_join` cerca di usarlo
- Esempio:
  - Il programma seguente illustra tale problema usando una variabile automatica (allocata sullo stack) come argomento di `pthread_exit`

# Esempio

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

struct foo {
    int a, b, c, d;
};

void
printfoo(const char *s, const struct foo *fp)
{
    printf("%s", s);
    printf("    structure at 0x%lx\n", (unsigned long) fp);
    printf("    foo.a = %d\n", fp->a);
    printf("    foo.b = %d\n", fp->b);
    printf("    foo.c = %d\n", fp->c);
    printf("    foo.d = %d\n", fp->d);
}
```

# Esempio (cont.)

```
void *  
thr_fn1(void *arg)  
{  
    struct foo foo = {1, 2, 3, 4}; //sullo stack  
  
    printf("thread 1:\n", &foo);  
    pthread_exit((void *)&foo);  
}
```

```
void *  
thr_fn2(void *arg)  
{  
    printf("thread 2: ID is %lu\n", (unsigned long)pthread_self());  
    pthread_exit((void *)0);  
}
```

# Esempio (cont.)

```
int main(void)
{
    int                err;
    pthread_t          tid1, tid2;
    struct foo         *fp;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0):
        {printf("can't create thread 1: %s\n", strerror(err)); exit(1);}
    err = pthread_join(tid1, (void *)&fp);
    if (err != 0)
        {printf("can't join with thread 1: %s\n",strerror(err)); exit(1);}
    sleep(1);
    printf("parent starting second thread\n");
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        {printf("can't create thread 2: %s\n",strerror(err));exit(1);}
    sleep(1);
    printf("parent:\n", fp);
    exit(0);
}
```

# Esempio (cont.)

- Quando eseguiamo il programma su Linux, otteniamo

```
$ ./a.out
```

```
thread 1:
```

```
  structure at 0x7f2c83682ed0
```

```
  foo.a = 1
```

```
  foo.b = 2
```

```
  foo.c = 3
```

```
  foo.d = 4
```

```
parent starting second thread
```

```
thread 2:  ID is 139829159933696
```

```
parent:
```

```
  structure at 0x7f2c83682ed0
```

```
  foo.a = -2090321472
```

```
  foo.b = 32556
```

```
  foo.c = 1
```

```
  foo.d = 0
```

# Esempio (cont.)

- I risultati variano in base all'architettura della memoria, il compilatore e l'implementazione della libreria dei thread

- **Solaris**

```
$ ./a.out
```

```
thread 1:
```

```
    structure at 0xffffffff7f0fbf30
```

```
    foo.a = 1
```

```
    foo.b = 2
```

```
    foo.c = 3
```

```
    foo.d = 4
```

```
parent starting second thread
```

```
thread 2:  ID is 3
```

```
parent:
```

```
    structure at 0xffffffff7f0fbf30
```

```
    foo.a = -1
```

```
    foo.b = 2136969048
```

```
    foo.c = -1
```

```
    foo.d = 2138049024
```

# Esempio (cont.)

- Il contenuto della struttura (allocata sullo stack dal thread *tid1*) è cambiato nel momento in cui il thread principale può accedervi
- Osserviamo come lo stack del secondo thread (*tid2*) ha sovrascritto lo stack del primo thread
- Per risolvere questo problema è possibile usare una struttura globale o allocare la struttura usando *malloc*

# Cancellazione di thread

- Un thread può richiedere che un altro thread nello stesso processo sia cancellato mediante la funzione `pthread_cancel`

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

```
int pthread_cancel(pthread_t tid);
```

```
//restituisce 0 se OK, numero di errore se  
fallisce
```

- Per default, `pthread_cancel` fa sì che il thread specificato da *tid* si comporti come se avesse chiamato `pthread_exit`, con l'argomento di `PTHREAD_CANCELED`
- Osserviamo che `pthread_cancel` non attende che il thread termini, ma effettua solamente la richiesta

# Cancellazione di thread (2)

- Per default lo stato di terminazione di un thread è mantenuto fino a che è chiamata `pthread_join` per quel thread
- La memoria allocata sottostante un thread può essere immediatamente reclamata dopo la terminazione, se il thread è stato distaccato
  - Quando un thread è distaccato la funzione `pthread_join` non può essere usata per aspettare il suo stato di uscita

# Distacco di un thread

- Una chiamata `pthread_join`, per un thread distaccato, fallisce restituendo `EINVAL`
- Un thread si può distaccare chiamando `pthread_detach`

```
#include <pthread.h>
int pthread_detach(pthread_t tid);
//restituisce 0 se OK, numero di errore se
fallisce
```

# Thread vs processo

- Osserviamo le similitudini tra le funzioni relative ai thread e quelle relative ai processi

Primitiva di processo	Primitiva di thread	Descrizione
<code>fork</code>	<code>pthread_create</code>	Crea un nuovo flusso di controllo
<code>exit</code>	<code>pthread_exit</code>	Esce da un flusso di controllo esistente
<code>waitpid</code>	<code>pthread_join</code>	Acquisisce lo stato di uscita dal flusso di controllo
<code>getpid</code>	<code>pthread_self</code>	Determina l'id del flusso di controllo
<code>abort</code>	<code>pthread_cancel</code>	Richiede la terminazione anomala del flusso di controllo

# Attributi dei Thread

- Negli esempi visti fin qui, in cui è stata invocata `pthread_create`, abbiamo passato un puntatore nullo invece di passare un puntatore ad una struttura `pthread_attr_t`
- Si può usare `pthread_attr_t` per modificare gli attributi di default associando questi attributi ai thread che creiamo
  - E' utilizzata la funzione `pthread_attr_init` per inizializzare la struttura `pthread_attr_t`
  - Ad invocazione di `pthread_attr_init` avvenuta, `pthread_attr_t` contiene i valori di default per tutti gli attributi di thread supportati dall'implementazione

# Attributi dei Thread

```
include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);

/* Restituiscono 0 se OK, un numero di
   errore se falliscono */
```

- Per liberare la memoria della struttura `pthread_attr_t` chiamiamo `pthread_attr_destroy`
  - Se un'implementazione di `pthread_attr_init` ha allocato memoria dinamica per l'oggetto attributo, `pthread_attr_destroy` libera la memoria

# Attributi dei Thread

- La struttura `pthread_attr_t` è opaca alle applicazioni
  - Vale a dire, si suppone che le applicazioni non conoscano nulla circa la struttura interna, favorendone la portabilità
  - POSIX.1 definisce funzioni separate per interrogare ed impostare ciascun attributo

# Funzioni attributi

```
int pthread_equal(pthread_t t1, pthread_t t2);
    /* confronta i thread_id di t1 e t2 ) */

int pthread_attr_init(pthread_attr_t *attr);
    /* inizializza gli attributi con il valore di default */

int pthread_attr_destroy(pthread_attr_t *attr);
    /* distrugge gli attributi e il comportamento dell'attributo
    distrutto è indefinito */

int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate)
    /* ritorna lo stato di detach del valore dell'attributo */

int pthread_attr_setdetachstate(const pthread_attr_t *attr,
                                int detachstate)
    /* assegna lo stato di detach del valore dell'attributo */
```

- In genere per tutti gli attributi esistono le funzioni `pthread_get` e `pthread_set`

# Attributi valori assegnabili

Attributo	Valore	Significato
int scope	PTHREAD_SCOPE_PROCESS	Competizione sulle risorse all'interno di un processo
int detachstate	PTHREAD_CREATE_JOINABLE	Joinable con altri threads
void * stackaddr	NULL	Allocata dal sistema
size_t * stacksize	NULL	1 megabyte
priority	NULL	Priorità del thread
int schedpolicy	SCHED_OTHER	Determinata dal sistema
ineritsched	PTHREAD_EXPLICIT_SCHED	Attributi di scheduling stabiliti esplicitamente, es. policy

# Attributi valori assegnabili

Attributo	Valore
scope	PTHREAD_SCOPE_PROCESS PTHREAD_SCOPE_SYSTEM
detachstate	PTHREAD_CREATE_DETACHED PTHREAD_CREATE_JOINABLE
stackaddr	NULL
stacksize	NULL, PTHREAD_STACK_MIN
priority	NULL
schedpolicy	SCHED_OTHER, SCHED_FIFO, SCHED_RR
inheritsched	PTHREAD_EXPLICIT_SCHED PTHREAD_INHERIT_SCHED

# Funzioni per gli attributi

- Abbiamo visto il concetto di thread distaccato
  - Se non siamo interessati allo stato di terminazione di un thread esistente, possiamo usare `pthread_detach` per consentire al sistema operativo di reclamare le risorse del thread quando il thread esce
- Se sappiamo di non aver bisogno dello stato di terminazione del thread nel momento in cui lo creiamo, è possibile avviare il thread nello stato distaccato modificando l'attributo `detachstate` nella struttura `pthread_attr_t`
  - Possiamo usare la funzione `pthread_attr_setdetachstate` per impostare l'attributo `detachstate` del thread ad uno dei due valori
    - `PTHREAD_CREATE_DETACHED`: per avviare il thread nello stato distaccato
    - `PTHREAD_CREATE_JOINABLE`: per avviare il thread normalmente, così il suo stato di terminazione può essere recuperato dalle applicazioni

# Funzioni per gli attributi

```
#include <pthread.h>
int pthread_attr_getdetachstate(const pthread_attr_t
                               *attr, int * detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                               int detachstate);

// restituiscono 0 se OK, il numero di errore se
// falliscono
```

- `pthread_attr_getdetachstate` è usata per determinare l'attributo `detachstate` corrente
  - L'intero puntato dal secondo argomento è posto a `PTHREAD_CREATE_DETACHED` o `PTHREAD_CREATE_JOINABLE`, a seconda del valore dell'attributo nella struttura `pthread_attr_t`

# Esempio

```
#include "apue.h"
#include <pthread.h>
int
makethread(void *(*fn)(void*), void *arg)
{
    int err;
    pthread_t tid;
    pthread_attr_t attr;

    err = pthread_attr_init(&attr);
    if (err != 0)
        return(err);
    err = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    if (err == 0)
        err = pthread_create(&tid, &attr, fn, arg);
    pthread_attr_destroy(&attr);
    return(err);
}
```

# pthread\_once()

- Assicura che una routine di inizializzazione sarà eseguita solo una volta senza curarsi di quanti thread nel processo la invocano
  - Tutti i thread chiamano la routine facendo identiche chiamate alla funzione `pthread_once`. Il thread che per primo chiama la funzione `pthread_once` può eseguirla; le chiamate seguenti non eseguono la funzione

```
pthread_once_t initflag = PTHREAD_ONCE_INIT;  
int pthread_once(pthread_once_t *initflag, void (*init_routine)  
                (void));  
/* restituisce 0 se OK, numero di errore se fallisce */
```

- *initflag* deve essere una variabile non locale ed inizializzata a `PTHREAD_ONCE_INIT`
- `pthread_once` garantisce che la funzione di inizializzazione è chiamata una ed una sola volta
  - *initflag* è utilizzata per determinare se la funzione è stata chiamata in precedenza

# pthread\_once(): esempio

```
#include <pthread.h>
pthread_once_t init=PTHREAD_ONCE_INIT;
void init_funz()
{
    //varie inizializzazioni...
    printf("inizializzazione effettuata\n");
}
void* funzione(void *arg)
{
    pthread_once(&init,init_funz);
    //...istruzioni
    printf("funzione\n");
}
int main()
{
    pthread_t t1, t2;
    int r1,r2;
    r1=pthread_create(&t1, NULL, funzione), NULL);
    r2=pthread_create(&t2, NULL, funzione), NULL);
}
```

# Rientranza

- Abbiamo già discusso le funzioni rientranti e i gestori di segnali
- I thread sono simili ai gestori di segnali rispetto alla rientranza
  - Con entrambi i gestori di segnali e i thread, i thread multipli di controllo possono potenzialmente chiamare la stessa funzione nel medesimo istante
  - Se una funzione può essere richiamata da thread multipli nello stesso momento in maniera sicura, diciamo che la funzione è *thread-safe*
- Le implementazioni che supportano le funzioni thread-safe definiscono il simbolo `_POSIX_THREAD_SAFE_FUNCTIONS` in `<unistd.h>`
  - Le applicazioni possono usare anche l'argomento `_SC_THREAD_SAFE_FUNCTIONS` con `sysconf` per verificare il supporto delle funzioni thread-safe a runtime

# Thread e segnali

- Ogni thread ha una propria maschera di segnali
- La disposizione del segnale è condivisa da tutti i thread del processo
  - I thread individualmente possono bloccare i segnali, ma quando un thread modifica l'azione associata con un dato segnale, tutti i thread condividono l'azione
  - Se un thread sceglie di ignorare un dato segnale, un altro thread può annullare quella scelta, ripristinando la disposizione di default o installando un gestore di segnale

# Thread e segnali

- I segnali sono consegnati ad un singolo thread nel processo
  - Se il segnale è relativo a un errore hardware, solitamente il segnale è inviato al thread la cui azione ha causato l'evento
  - Gli altri segnali sono consegnati ad un thread arbitrario
- Per inviare un segnale ad un processo, invochiamo **kill**
  - Per inviare un segnale ad un thread, invochiamo **pthread\_kill**

```
#include <signal.h>
```

```
int pthread_kill (pthread_t thread, int signo);
```

```
//restituisce 0 se OK, numero di errore se fallisce
```

- con **signo** uguale a 0, possiamo verificare l'esistenza del thread
- Se l'azione di default per il segnale è di terminare il processo allora l'invio del segnale a un thread terminerà l'intero processo

# La chiamata clone di LINUX

- Il kernel di Linux usa la funzione **clone** per creare nuovi processi. I flag controllano quali risorse genitore e figlio condividono
- Genitore e figlio possono condividere da tutto (memoria, gestori dei segnali, file aperti,...) a niente. Mentre con la **fork** il figlio eredita le risorse del padre, con **clone** è possibile non condividere nulla
- Un programma può chiamare direttamente **clone** per produrre un programma multithread; ciò rende il programma specifico per Linux, poiché non è conforme ad alcuno standard esterno
- Le librerie per i thread utilizzano la **clone** al loro interno, e rendono trasparente gli aggiornamenti del kernel agli utenti

# La chiamata clone di LINUX

```
int clone(int (*fn) (void *), void *child_stack,  
          int flags, void *arg);
```

- Crea un nuovo processo, come `fork(2)`, ma permette al processo figlio di condividere parti del contesto di esecuzione con il processo chiamante (spazio di memoria, tabella dei file aperti, etc.)

# La chiamata clone di LINUX

Flag	Se settato ...	Se non settato ...
CLONE_VM	Crea un nuovo thread	Crea un nuovo processo
CLONE_FS	Condivide umask, radice e directory di lavoro	Non le condivide
CLONE_FILES	Condivide i descrittori di file	Copia i descrittori di file
CLONE_SIGHAND	Condivide la tabella di gestori di segnale	Copia la tabella
CLONE_PID	Il nuovo thread recupera il vecchio PID	Il nuovo thread prende il proprio PID

# Libreria Pthread

- I thread vengono supportati se in `<unistd.h>` esiste la macro `_POSIX_THREADS`. La funzione `sysconf()` con argomento `_SC_THREADS` ritorna 0 (-1) se pthreads (non) è supportata

```
long sysconf (int name)
```

- Esiste un numero massimo di threads supportati (`PTHREADS_THREADS_MAX` in `<limits.h>`). La funzione `sysconf ()` con argomento `_SC_THREADS_THREADS_MAX` in `<unistd.h>`, ci fornisce il numero massimo di thread supportati
- Per compilare un programma con i Posix thread deve essere incluso lo header file `<pthread.h>`
  - Si deve linkare la libreria pthread:
    - `host> gcc ..... programma.c -lpthread`

# Esempio 1

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

void *print_message_function( void *ptr );

int main()
{
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int  iret1, iret2;

    /* crea thread indipendenti, ciascuno dei quali eseguirà una
    funzione */

    iret1 = pthread_create(&thread1, NULL, print_message_function,
        (void*)message1);
    iret2 = pthread_create(&thread2, NULL, print_message_function,
        (void*) message2);
```

# Esempio 1 (cont.)

```
/*aspetta che i thread abbiano completato prima che il main
continui. Se non aspettiamo si potrebbe eseguire una exit che
terminerà l'intero processo e tutti i suoi thread, prima che
questi abbiano finito */
```

```
pthread_join(thread1, NULL);
```

```
pthread_join(thread2, NULL);
```

```
printf("Thread 1 returns: %d\n",iret1);
```

```
printf("Thread 2 returns: %d\n",iret2);
```

```
exit(0);
```

```
}
```

```
void *print_message_function( void *ptr )
```

```
{
```

```
    char *message;
```

```
    message = (char *) ptr;
```

```
    printf("%s \n", message);
```

```
}
```

# Esempio 2

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 8

/* Questa routine riceve in ingresso l'id del thread */

void *PrintHello(void *threadid)
{
    int *id_ptr, taskid;

    sleep(1);
    id_ptr = (int *) threadid;
    taskid = *id_ptr;
    printf("Thread %d\n", taskid);
    pthread_exit(NULL);
}
```

# Esempio 2 (cont.) errato

```
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);
}
```

# Esempio 2 (cont.) corretto

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

    for(t=0;t<NUM_THREADS;t++) {
        taskids[t] = (int *)malloc(sizeof(int)); /* indirizzo di un intero */
        *taskids[t] = t;                        /* l'intero */
        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);
}
```

# Esercizio 1

- Scrivere un programma che accetta un intero  $n$  da riga di comando, crea  $n$  thread e poi aspetta la loro terminazione
  - Ciascun thread aspetta un numero di secondi casuale tra 1 e 10, poi incrementa una variabile globale intera ed infine ne stampa il valore

# Esercizio 2

- Scrivere un programma che prende in input un intero  $n$ , il nome di un file di testo ed un carattere  $x$
- Il programma ha il compito di contare le occorrenze del carattere  $x$  nel file di testo
- Il programma esegue tale compito creando  $n$  thread, ognuno dei quali esamina una porzione diversa del file di testo
  - ad esempio, se il file è lungo 1000 bytes ed  $n=4$ , il primo thread esaminerà i primi 250 bytes, il secondo thread esaminerà i 250 bytes successivi, e così via