- Su molte versioni di Unix è disponibile la libreria dei POSIX threads (Pthreads in breve), cioè una libreria per la programmazione mediante threads che fa parte dello standard di Unix
- Le diverse implementazioni (su Linux, su Solaris etc.) possono supportare soltanto una parte delle funzionalità definite nello standard
- La libreria non specifica se deve trattarsi di una implementazione dei threads a livello kernel o no
- Libro di riferimento:
  - Butenhof, "Programming with POSIX threads" (Addison-Wesley 1997)
- In questo corso vedremo soltanto gli aspetti di base più semplici che ci permettono di attivare diversi threads in un processo, e gli aspetti relativi alla sincronizzazione

Posix Thread: funzioni

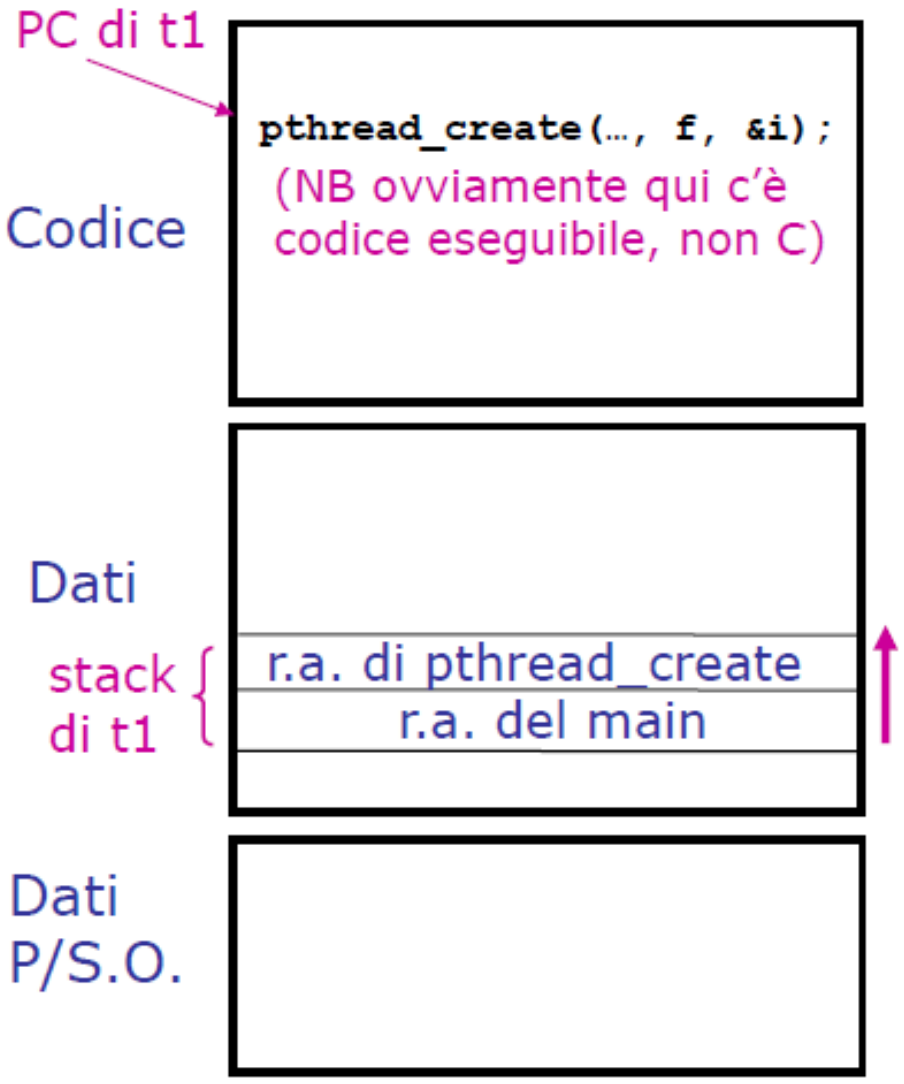
- La libreria prevede:
  - una funzione *pthread_create* che permette di creare, all'interno del processo corrente, un nuovo thread che esegue la funzione il cui puntatore viene passato come argomento alla create;
  - una funzione *pthread_exit* che permette ad un thread di terminare (ma il thread termina anche quando esce dalla funzione passata alla create che lo ha generato);
  - una funzione *pthread_join* che permette ad un thread di attendere la terminazione di un altro thread.
- Vedere nel man l'opzione per il linking della libreria pthread
- Considerato il flusso di controllo, queste funzioni sono l'analogo di ciò che in programmi con processi multipli è rappresentato da:
  - la combinazione f*ork-exec*;
  - *exit* (o l'uscita dal main);
  - *wait*

Thread: creazione

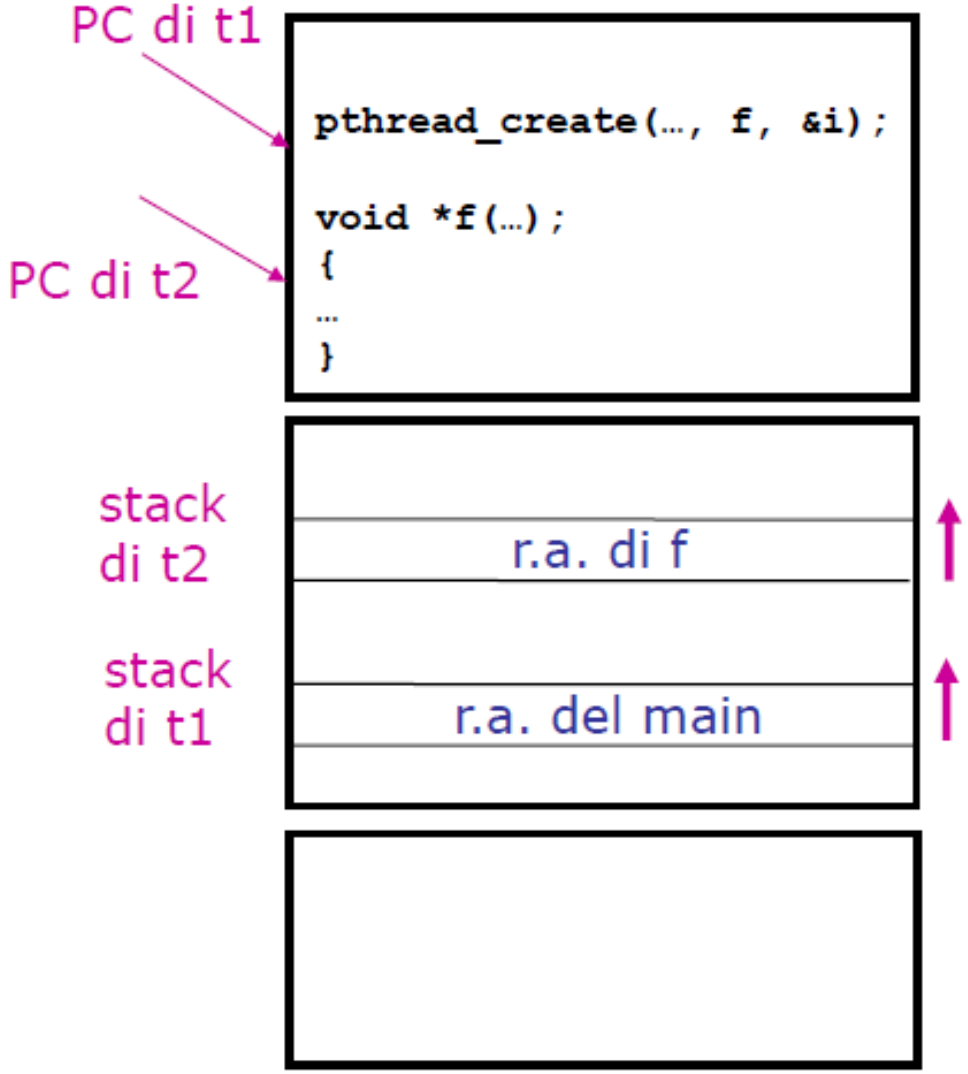
- Consideriamo un processo P (con codice sorgente in C) che inizia l'esecuzione. All'inizio P comprende un unico thread, chiamiamolo t1
- Se la funzione main (eseguita da t1) chiama:
  - pthread_create(…, f, &i);
- lo stack di t1 comprende il "record di attivazione" (che contiene una copia di parametri, variabili locali e altro) della funzione *main* e quello della funzione *pthread_create*
- L'effetto è la creazione di un nuovo thread t2, allocando spazio per un nuovo stack, sulla quale avviene la chiamata di f(&i)
- La memoria accessibile al programma è automaticamente condivisa fra i vari threads. Esiste – ma non la trattiamo - la possibilità di allocare dati a cui un solo thread può accedere
- Ogni thread ha degli attributi (es. dimensione della stack), con un valore di default, la pthread_create può specificare valori diversi

Thread: effetto della creazione

Alla chiamata di *pthread_create()*



All'uscita di *pthread_create()*



pthread_create()

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

- Esempio di invocazione

```
#include <pthread.h>

void *tbody(void *arg)
{...}

pthread_t t;
   . . .
pthread_create(&t, NULL, tbody, NULL);
```

- Compilazione e linking con libreria -pthread.

Scheduling delle thread

- Lo scheduling dei threads nella libreria può dipendere dalle scelte implementative, inoltre gli attributi possono definire proprietà relative allo scheduling (ma le implementazioni non necessariamente supportano tutti i tipi di scheduling)
- In generale lo scheduling deve garantire che se un processo P ha almeno un thread T pronto per l'esecuzione, T deve essere preso in considerazione dallo scheduling
- Ad es. se un thread T1 di un processo P compie una operazione sospensiva, e P ha altri thread pronti, questi devono poter andare avanti
- Questo deve avvenire se l'operazione sospensiva è una lettura da file e ancora più ovviamente se si tratta di una operazione di sincronizzazione con altri threads (T1 attende una segnalazione da altri thread, che devono poter girare per effettuarla)

Scheduling delle thread

- A differenza del caso dei processi in sistemi interattivi, non è sempre indispensabile che avvenga il timesharing fra threads di uno stesso processo in quanto:
  - diversi processi si trovano di solito (tranne nel caso di una applicazione costituita da processi multipli che cooperano) per caso a condividere le risorse del sistema e bisogna che non ne soffrano troppo: bisogna evitare che uno monopolizzi la CPU a danno degli altri
  - una applicazione costituita da diversi thread è probabilmente progettata come un tutto unico, e in generale basta che almeno un thread vada avanti; se si deve sincronizzare con altri (deve attendere qualcosa dagli altri), chiamerà una primitiva sospensiva, e potranno andare avanti solo gli altri
  - i threads esistono per avere a costo basso (rispetto ai processi) la commutazione di contesto, quindi non costa moltissimo fare il timesharing

Esercizi in appunti parte 3

- Compilare il programma t1.c
  - Esercizio 3.1: verificare che i thread condividono le variabili globali. Inserendo una variabile a cui si assegna un valore prima della pthread_create, e che viene modificata dai due thread quando girano in pseudoparallelo

- Esaminare i programmi t2.c e t2a.c
  - Quali differenze vi sono nel loro comportamento, in particolare nell'ordine di schedulazione delle thread?

- Eseguire il programma race.c
  - Incrementando considerevolmente il valore passato da linea di comando, cosa capita? Perchè?

Esercizio

```
#include <sys/types.h>
#include <wait.h>
#include "errors.h"

int main(int argc, int argv[]){
  int status;
  char line[128];
  int seconds;
  pid_t pid;
  char message[64];

  while(1){
    printf("Allarme >");

    if(fgets(line, sizeof(line), stdin)==NULL) exit(0);
    if(strlen(line)<=1) continue;

    if(sscanf(line, "%d %64[^\n]", &seconds,
message)<2) {
      fprintf(stderr, "Comando sconosciuto\n");
    } else {
      pid = fork();
      if(pid==(pid_t)-1){
        printf("Errore nella fork\n");
        exit(-1);
      }
      if(pid == (pid_t) 0){    // FIGLIO
        sleep(seconds);
        printf("(%d) %s\n", seconds, message);
        exit(0);
       }
      }
     }
   }
}
```

Esercizio

- Implementare il programma precedente (basato su processi) usando le thread
  - Dovete passare alle thread due argomenti:
    - il numero di secondi da attendere
    - Il messaggio
  - Come potete fare?