

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
#define MAXPAROLA 30
#define MAXRIGA 80
```

```
int main(int argc, char *argv[])
```

```
{
    int freq[MAXPAROLA]; /* vettore di contatori
    delle frequenze delle lunghezze delle parole */
    char riga[MAXRIGA];
    int i, inizio, lunghezza;
    FILE *f;
```

```
for(i=0; i<MAXPAROLA; i++)
    freq[i]=0;
```

```
if(argc != 2)
```

```
{
    fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
    exit(1);
}
```

```
f = fopen(argv[1], "r");
if(f==NULL)
```

```
{
    fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
    exit(1);
}
```

```
while( fgets( riga, MAXRIGA, f ) != NULL )
```



Thread

La libreria Pthread

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

Librerie di thread

- ❖ Una libreria sui thread, fornisce l'interfaccia per effettuare la gestione dei thread da parte del programmatore
- ❖ Le librerie di thread più usate sono

- Thread POSIX
- C11
- Windows 32/64
- Java
- C++

Implementata a livello utente e a livello kernel

Realizzata a livello kernel

In genere implementata tramite una libreria thread del sistema ospitante Java (ovvero Pthread POSIX o Windows 32/64)

Pthreads

❖ POSIX threads o Pthreads

- È la libreria standard UNIX per la gestione di thread
 - POSIX 1003.1c del 1995
 - Rivista con la IEEE POSIX 1003.1 2004 Edition
- È definita per il C UNIX, ma è disponibile anche per
 - Altre piattaforme (Linux, Windows, MacOS, BSD, Solaris, etc.)
 - Per altri linguaggi (e.g., FORTRAN)
- La sua implementazione dipende dalla piattaforma
 - Su Linux usa normalmente NPTL (Native POSIX Thread Library) con threading-model 1x1

1 T utente → 1 T kernel

Pthreads

❖ Attraverso Pthreads

- Il thread è una **funzione** che viene eseguita in maniera indipendente dal resto del programma

Processo concorrente costituito da più thread = insieme di funzioni in esecuzione indipendente che condividono le risorse del processo

Pthreads

- ❖ Pthreads permette di
 - Creare e manipolare thread
 - Sincronizzare thread
 - Proteggere le risorse comuni ai thread
 - Schedulare thread
 - Distruggere thread
- ❖ Definisce più di **60** funzioni di gestione
 - Tutte le funzioni hanno nome **pthread_***
 - pthread_equal, pthread_self, pthread_create, pthread_exit, pthread_join, pthread_cancel, pthread_detach

Libreria e compilazione

- ❖ Le funzioni di Pthread sono definite in
 - `pthread.h`
- ❖ Occorre ricordarsi di
 - Inserire nei file **.c** la riga
 - `#include <pthread.h>`
 - Compilare i programmi con l'opzione o includendo la libreria **pthread**

```
gcc -Wall -g -o <exeName> -pthread <file.c>  
gcc -Wall -g -o <exeName> <file.c> -lpthread
```

Thread Identifier

- ❖ Un thread è identificato in maniera univoca
 - Da un identificatore di tipo **pthread_t**
 - Simile al PID di un processo (pid_t)
 - Il tipo **pthread_t** è opaco
 - La sua definizione dipende dall'implementazione
 - Occorre farne accesso/uso solo mediante funzioni appositamente definite in Pthreads
 - Non è possibile confrontare due identificatori direttamente o stamparne il valore
 - Ha significato solo all'interno del processo in cui il thread è stato eseguito
 - Si ricordi che il PID è globale all'interno del sistema

System call pthread_equal

```
int pthread_equal (  
    pthread_t tid1,  
    pthread_t tid2  
);
```

- ❖ Confronta due identificatori di thread
- ❖ Parametri
 - Due identificatori di thread
- ❖ Valore di ritorno
 - Diverso da 0, se i thread sono uguali
 - Uguale a 0, altrimenti

System call pthread_self

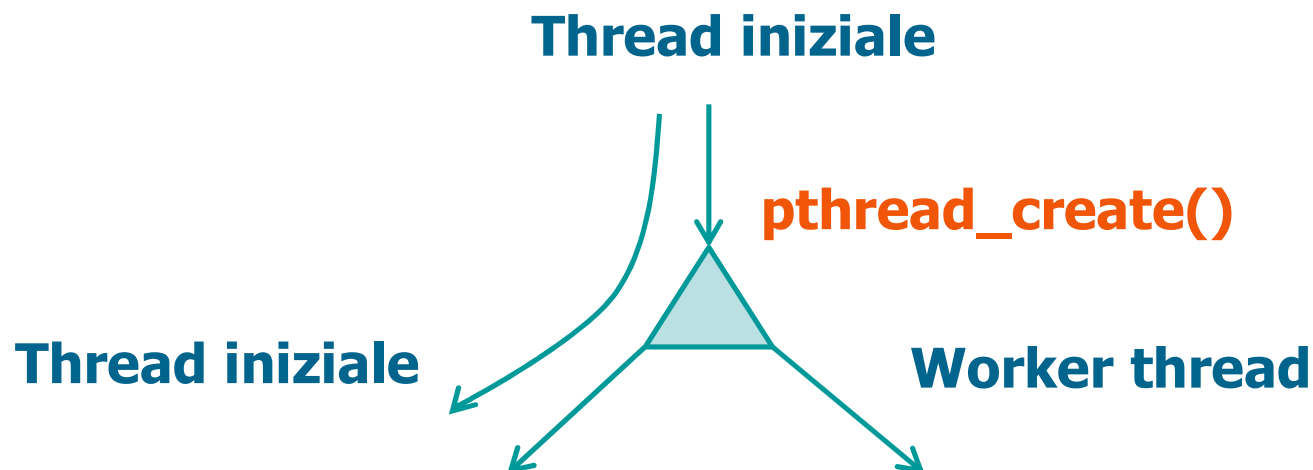
```
pthread_t pthread_self (  
    void  
);
```

- ❖ Ritorna l'identificatore di thread del thread chiamante
 - Può essere utilizzata da un thread (insieme a **pthread_equal**) per auto-identificarsi

Auto-identificarsi può essere importante per accedere correttamente ai propri dati personali

System call `pthread_create`

- ❖ All'esecuzione, ogni programma include un solo processo e un solo thread
- ❖ Per creare un nuovo thread si utilizza **`pthread_create`**
 - Il numero massimo di chiamate a `pthread_create` è indefinito e dipende dall'implementazione



System call pthread_create

```
int pthread_create (  
    pthread_t *tid,  
    const pthread_attr_t *attr,  
    void *(*startRoutine) (void *),  
    void *arg  
);
```

❖ Parametri

- Identificatore del thread generato (tid)
- Attributi del thread (attr)
 - NULL è l'attributo di default
- Routine C eseguita dal thread (startRoutine)
- Argomento passato alla routine di inizio (arg)
 - NULL se non c'è argomento

Un solo argomento!

System call pthread_create

❖ Valore di ritorno

- Il valore 0, in caso di successo
- Un codice di errore, in caso di fallimento

```
int pthread_create (  
    pthread_t *tid,  
    const pthread_attr_t *attr,  
    void *(*startRoutine) (void *),  
    void *arg  
);
```

System call `pthread_exit`

- ❖ Un intero processo (con tutti i suoi thread) termina se
 - Un suo thread effettua una **exit** (**_exit** o **_Exit**)
 - Il main effettua una **return**
 - Un suo thread riceve un segnale la cui azione è terminare
- ❖ Un singolo thread può terminare
 - Effettuando un **return** dalla sua funzione di inizio
 - Eseguendo una **pthread_exit**
 - Ricevendo una **pthread_cancel** da un altro thread

System call `pthread_exit`

```
void pthread_exit (  
    void *valuePtr  
);
```

- ❖ Permette a un thread di terminare restituendo il suo stato di terminazione
- ❖ Parametri
 - Il valore `valuePtr` è mantenuto dal SO sino a quando un thread fa una **`pthread_join`**
 - Tale valore risulta disponibile al thread che effettua una **`pthread_join`**

Esempio

Attivazione di
1 thread senza
parametri

```
void *tF () {  
    ...  
    pthread_exit (NULL);  
}
```

Attributi

Parametri

```
pthread_t tid;  
int rc;  
rc = pthread_create (&tid, NULL, tF, NULL);  
if (rc) {  
    // Error ...  
    exit (-1);  
}  
...  
pthread_exit (NULL);  
// exit (0);  
// return (0); (nel main)
```

OK

Termina il solo
thread originario

Termina l'intero
processo
(tutti i thread)

Esempio ...

Attivazione di
N thread con 1
parametro

```
void *tF (void *par) {  
    int *tidP, tid;  
    ...  
    tidP = (int *) par;  
    tid = *tidP;  
    ...  
    pthread_exit (NULL);  
}
```

Collezione i tid

```
pthread_t t[NUM_THREADS];  
int rc, i;
```

```
for (i=0; i<NUM_THREADS; i++) {  
    rc = pthread_create (&t[i], NULL, tF,  
        (void *) &i);  
    if (rc) {...}  
}  
pthread_exit(NULL);
```

Buggy

Parametro puntatore
a intero

... Esempio

Attivazione di
N thread con 1
parametro

I thread possono
essere attivati quando
i è mutato

```
void *tF (void *par) {  
    int *tidP, tid;  
    ...  
    tidP = (int *) par;  
    tid = *tidP;  
    ...  
    pthread_exit (NULL);  
}
```

Buggy

```
pthread_t t[NUM_THREADS];  
int rc, i;  
  
for (i=0; i<NUM_THREADS; i++) {  
    rc = pthread_create (&t[i], NULL, tF,  
        (void *) &i);  
    if (rc) {...}  
}  
pthread_exit(NULL);
```

ERRORE

i è un puntatore e il
suo valore cambia

Esempio

Attivazione di
N thread con 1
parametro

Cast di dato singolo
void * \leftrightarrow **long int**

```
void *tF (void *par) {  
    long int tid;  
    ...  
    tid = (long int) par;  
    ...  
    pthread_exit(NULL);  
}
```

Dirty

```
pthread_t t[NUM_THREADS];  
int rc; long int i;
```

```
for (i=0; i<NUM_THREADS; i++) {  
    rc = pthread_create (&t[i], NULL, tF,  
        (void *) i);  
    if (rc) { ... }  
}  
pthread_exit (NULL);
```

Parametro long int
con cast esplicito a
un puntatore

Esempio

Attivazione di
N thread con 1
parametro

Cast di vettore di
puntatori
void * \leftrightarrow **int**

```
void *tF (void *par) {  
    int *tid, taskid;  
    ...  
    tid = (int *) par;  
    taskid = *tid;  
    ...  
    pthread_exit(NULL);  
}
```

OK

```
int tA[NUM_THREADS];  
for (i=0; i<NUM_THREADS; i++) {  
    tA[i] = i;  
    rc = pthread_create (&t[i], NULL, tF,  
        (void *) &tA[i]);  
    if (rc) { ... }  
}  
pthread_exit (NULL);
```

Se il parametro è in
un vettore posso
passare il puntatore

Esempio

Attivazione di
N thread con 1
struct

```
struct tS {  
    int tid;  
    char str[N];  
};
```

```
void *tF (void *par) {  
    struct tS *tD;  
    int tid; char str[L];
```

Cast di vettore di
strutture

```
    tD = (struct tS *) par;  
    tid = tD->tid; strcpy (str, tD->str);  
    ...
```

```
pthread_t t[NUM_THREADS];  
struct tS v[NUM_THREADS];  
...  
for (i=0; i<NUM_THREADS; i++) {  
    v[i].tid = i;  
    strcpy (v[i].str, str);  
    rc = pthread_create (&t[i], NULL, tF, (void *) &v[i]);  
    ...  
}  
...
```

OK

Puntatore a struct
convertito in void *

System call `pthread_join`

❖ Alla sua creazione un thread può essere dichiarato

➤ **Joinable**

- Un altro thread può effettuare una "wait" (`pthread_join`) su di lui e può quindi acquisire il suo stato di uscita

➤ **Detached**

- Non si può attendere esplicitamente la sua terminazione (non è joinable)

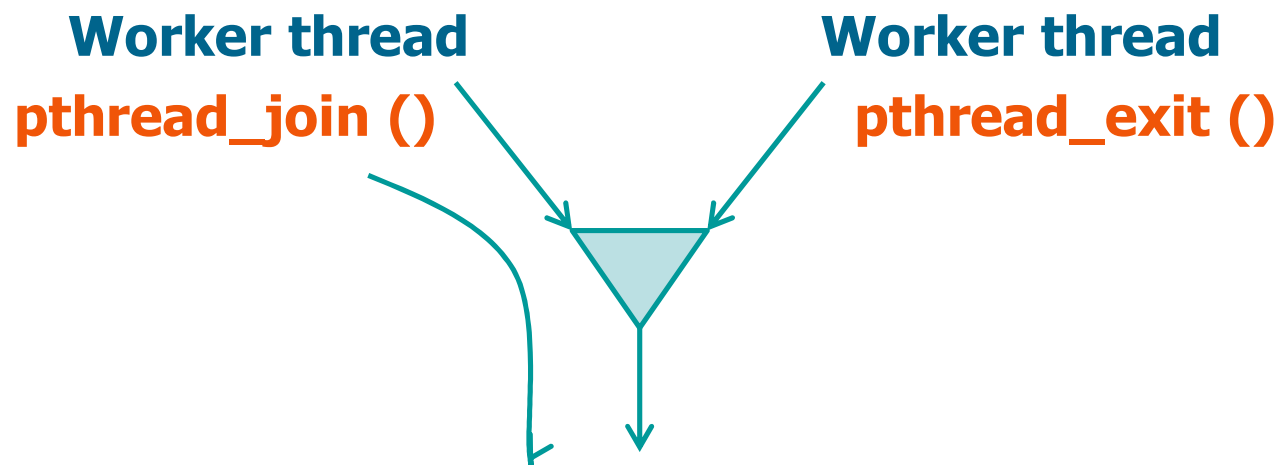
System call **pthread_join**

- ❖ Se il thread
 - È joinable il suo stato di terminazione viene mantenuto sino a quando un altro thread esegue una **pthread_join** per quel thread
 - È detached il suo stato di terminazione viene subito rilasciato
- ❖ In ogni caso
 - Il thread che richiama la **pthread_join** rimane bloccato sino a quando il thread richiesto non effettua una **pthread_exit**

System call pthread_join

```
int pthread_join (  
    pthread_t tid,  
    void **valuePtr  
);
```

- ❖ Usata da un thread per attendere (wait) un'altro specifico thread



System call pthread_join

❖ Parametri

- Identificatore del thread atteso (tid)
- Il puntatore (senza tipo) valuePtr riferirà il valore ritornato dal thread **tid**
 - Ritornato dalla pthread_exit
 - Ritornato con una return
 - PTHREAD_CANCELED se il thread è stato cancellato

valuePtr può essere NULL qualora non si voglia recuperare tale valore

```
int pthread_join (pthread_t tid, void **valuePtr);
```


System call `pthread_join`

❖ Valore di ritorno

- Il valore 0, in caso di successo
- Un codice di errore, in caso di fallimento
 - Se il thread era nello stato detached la **`pthread_join`** dovrebbe fallire
 - A seconda del SO e del timing può anche terminare correttamente
 - Se fallisce ritorna la costante `EINVAL` oppure `ESRCH`

```
int pthread_join (pthread_t tid, void **valuePtr);
```

Esempio

Utilizzo della
pthread_join

Ritorno lo stato di exit
(il pid in questo caso)

```
void *tF (void *par) {  
    long int tid;  
    ...  
    tid = (long int) par;  
    ...  
    pthread_exit ((void *) tid);  
}
```

```
void *status;  
long int pid;  
...  
/* Wait for threads */  
for (i=0; i<NUM_THREADS; i++) {  
    rc = pthread_join (t[i], &status);  
    pid = (long int ) status;  
    if (rc) { ... }  
}  
...
```

t[i] collezionava i tid

Attesa dei thread e
cattura del loro stato
in status

Esempio

❖ Utilizzo di una variabile globale comune a più thread

```
int myglobal;

void *threadF (void *arg) {
    int *argc = (int *) arg;
    int i, j;
    for (i=0; i<20; i++) {
        j = myglobal;
        j = j + 1;
        printf ("t");
        if (*argc > 1) sleep (1);
        myglobal = j;
    }
    printf ("(T:myglobal=%d)", myglobal);
    return NULL;
}
```

La variabile globale viene incrementata tramite copia in j

Il thread può attendere oppure no

Esempio

```
int main (int argc, char *argv[]) {
    pthread_t mythread;
    int i;
    pthread_create (&mythread, NULL, threadF, &argc);
    for (i=0; i<20; i++) {
        myglobal = myglobal + 1;
        printf ("m");
        sleep (1);
    }
    pthread_join (mythread, NULL);
    printf ("(M:myglobal=%d)", myglobal);
    exit (0);
}
```


System call `pthread_cancel`

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

- ❖ Termina il thread indicato
 - È come se tale thread eseguisse una `pthread_exit` con parametro `PTHREAD_CANCELED`
- ❖ Il thread che fa la richiesta non attende la terminazione del thread (effettua la richiesta e continua)

System call pthread_cancel

❖ Parametri

- Identificatore del thread (tid) da terminare

❖ Valore di ritorno

- Il valore 0, in caso di successo
- Un codice di errore, in caso di fallimento

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

System call `pthread_detach`

```
int pthread_detach (  
    pthread_t tid  
);
```

L'attributo della `pthread_create` permette una strada alternativa per dichiarare un thread detached

- ❖ Dichiarare il thread **tid** come detached
 - La memoria del thread che termina sarà restituita al SO e non mantenuta per una join
 - Non sarà più possibile effettuare un join con tale thread
 - Chiamate alla **pthread_join** dovrebbero fallire con il codice di errore `EINVAL` o `ESRCH`

System call pthread_detach

❖ Parametro

- Identificatore del thread (tid)

❖ Valore di ritorno

- Il valore 0, in caso di successo
- Un codice di errore, in caso di fallimento

```
int pthread_detach (pthread_t tid);
```

Esempio

❖ Creare un thread e poi renderlo detached

```
pthread_t tid;
int rc;
void *status;

rc = pthread_create (&tid, NULL, PrintHello, NULL);
if (rc) { ... }

pthread_detach (tid);

rc = pthread_join (tid, &status);
if (rc) {
    // Error
    exit (-1);
}

pthread_exit (NULL);
```

Detach a thread

Errore se si attende

Esempio

- ❖ Creare un thread "detached" utilizzando il parametro attributo di `pthread_create`

```
pthread_attr_t attr;  
void *status;  
  
pthread_attr_init (&attr);  
pthread_attr_setdetachstate (&attr,  
    PTHREAD_CREATE_DETACHED);  
//PTHREAD_CREATE_JOINABLE);  
  
rc = pthread_create (&t[t], &attr, tF, NULL);  
if (rc) {...}  
  
pthread_attr_destroy (&attr);  
  
rc = pthread_join (thread[t], &status);  
if (rc) {  
    // Error  
    exit (-1);  
}
```

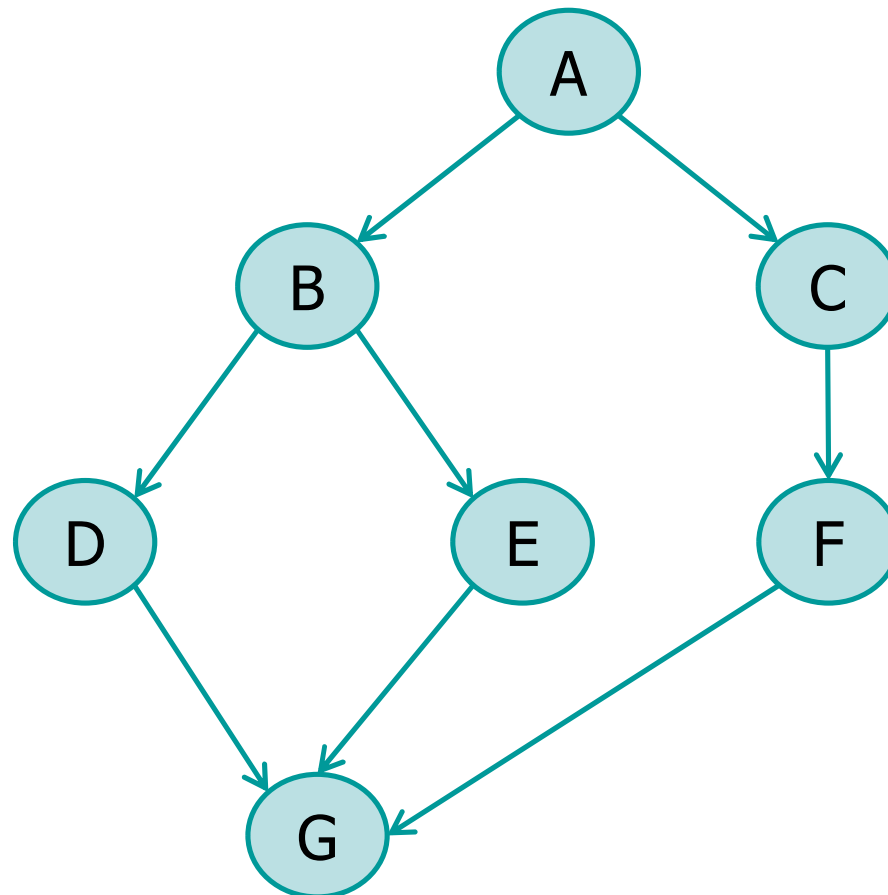
Crea un thread come detached

Distrugge l'attributo

Errore se si attende

Esercizio proposto

- ❖ Realizzare, tramite l'utilizzo di thread, il seguente grafo di precedenza

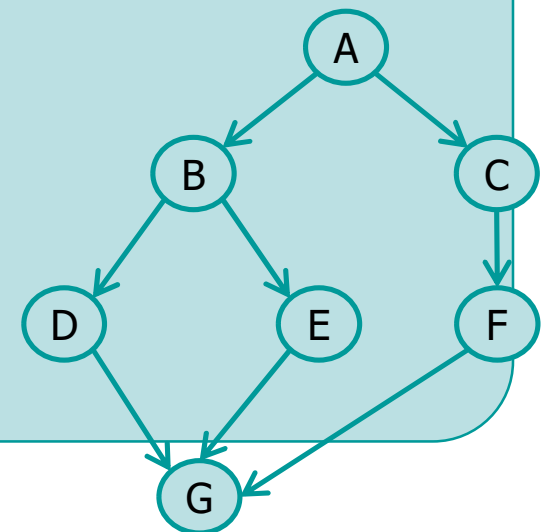


Soluzione

```
void waitRandomTime (int max){  
    sleep ((int)(rand() % max) + 1);  
}
```

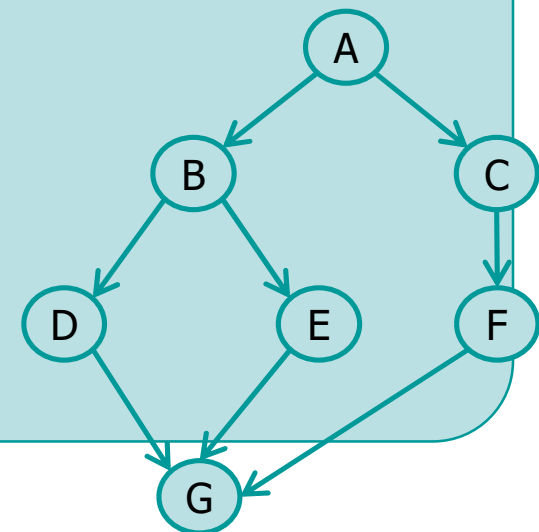
```
int main (void) {  
    pthread_t th_cf, th_e;  
    void *retval;
```

```
    srand (getpid());  
    waitRandomTime (10);  
    printf ("A\n");
```



Soluzione

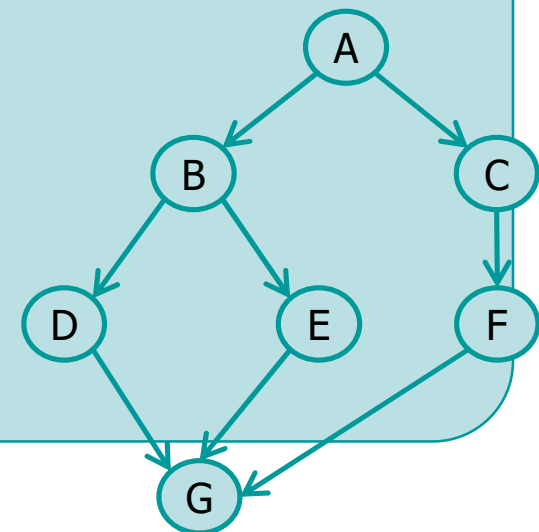
```
waitRandomTime (10);  
pthread_create (&th_cf, NULL, CF, NULL);  
waitRandomTime (10);  
printf ("B\n");  
waitRandomTime (10);  
pthread_create (&th_e, NULL, E, NULL);  
waitRandomTime (10);  
printf ("D\n");  
pthread_join (th_e, &retval);  
pthread_join (th_cf, &retval);  
waitRandomTime (10);  
printf ("G\n");  
return 0;  
}
```



Soluzione

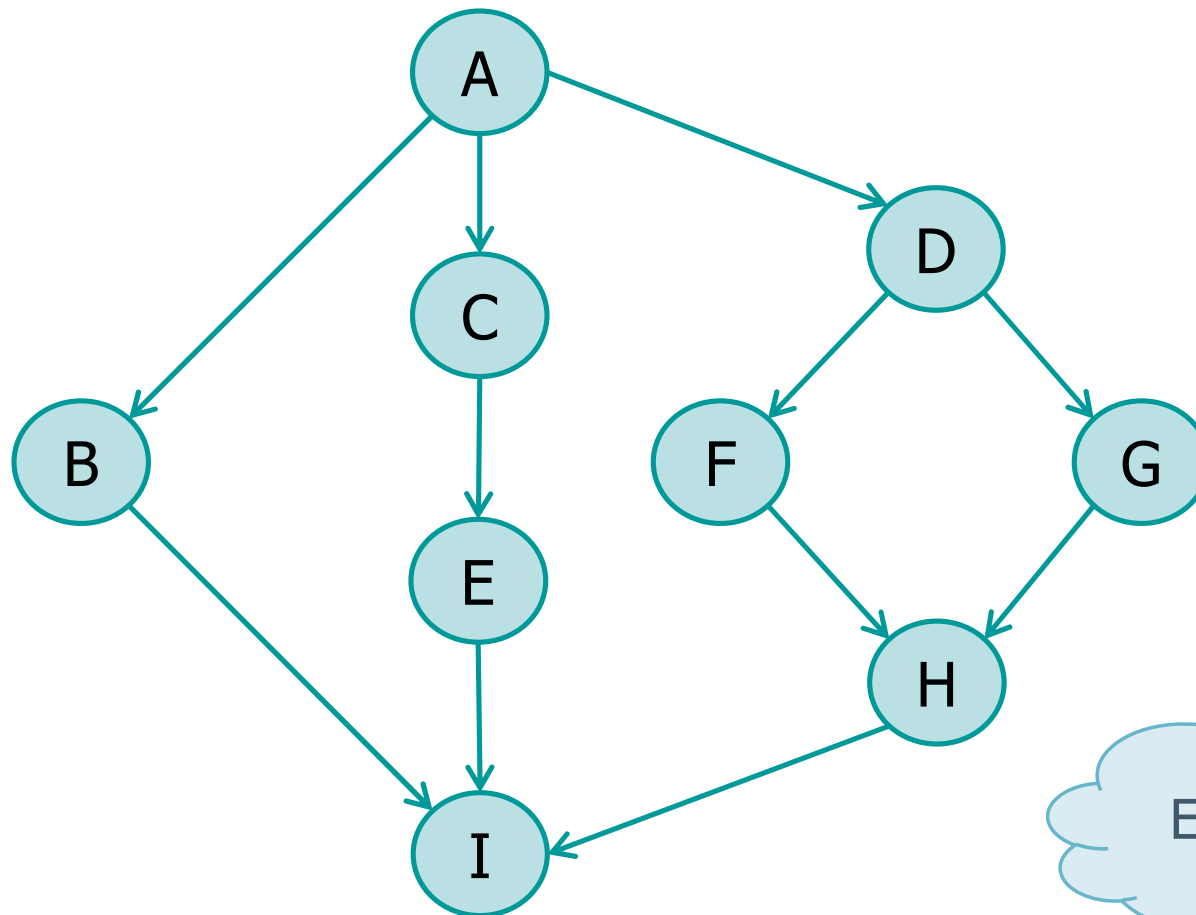
```
static void *CF () {  
    waitRandomTime (10);  
    printf ("C\n");  
    waitRandomTime (10);  
    printf ("F\n");  
    return ((void *) 1); // Return code  
}
```

```
static void *E () {  
    waitRandomTime (10);  
    printf ("E\n");  
    return ((void *) 2); // Return code  
}
```



Esercizio

- ❖ Realizzare, tramite l'utilizzo di thread, il seguente grafo di precedenza



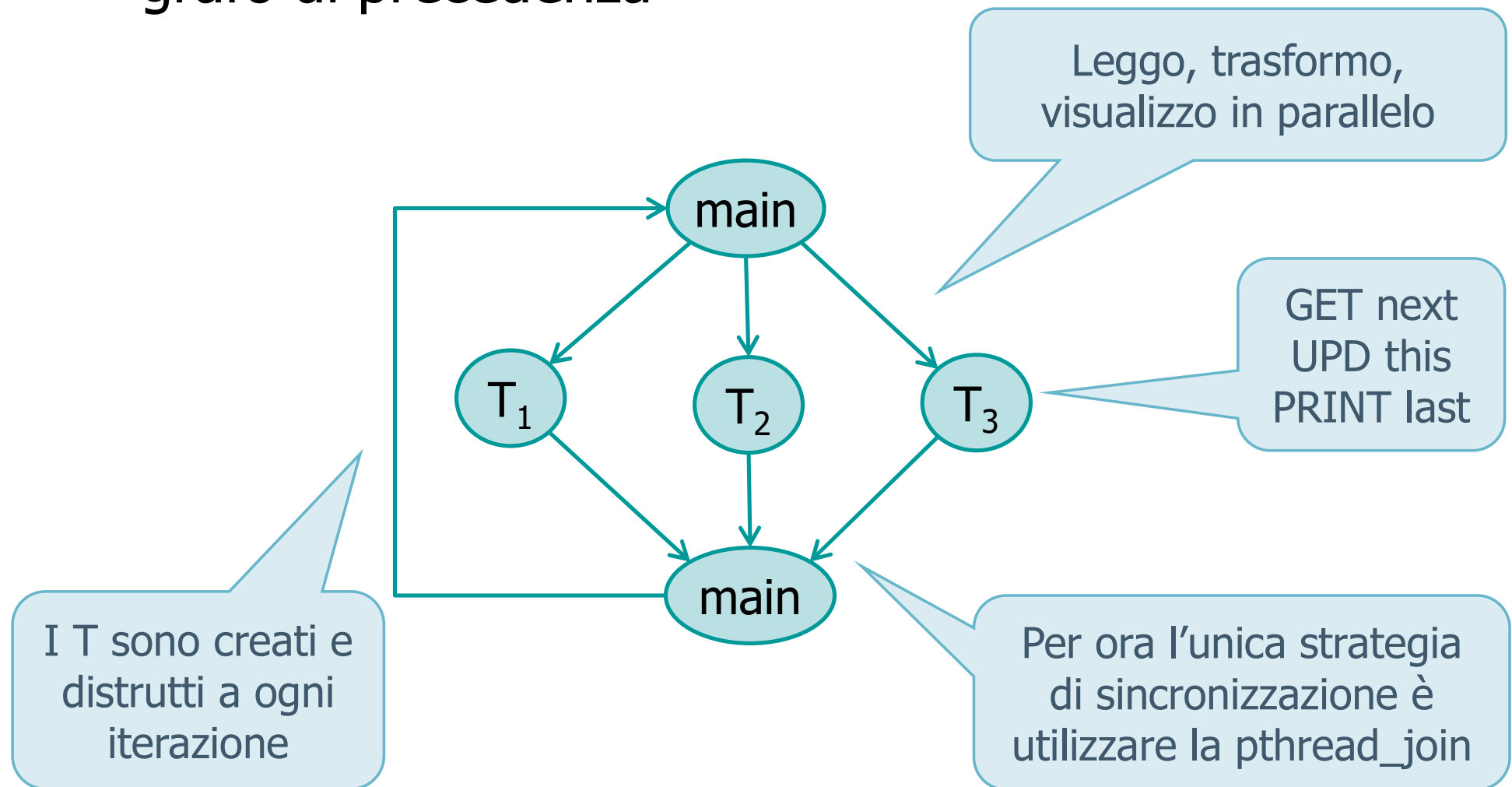
Esercizio ...

Esercizio

- ❖ Un file contiene un numero di caratteri indefinito
- ❖ Realizzare un programma con thread concorrenti in cui tre thread (**T₁**, **T₂**, **T₃**) lavorino in pipeline per gestire il file
 - **T₁**: Legge dal file un carattere alla volta
 - **T₂**: Trasforma il carattere letto da **T₁** in maiuscolo
 - **T₃**: Visualizza il carattere prodotto da **T₂** su standard output

Soluzione

- ❖ Realizzare, tramite l'utilizzo di thread, il seguente grafo di precedenza



Soluzione

```
static void *GET (void *arg) {
    char *c = (char *) arg;
    *c = fgetc (fg);
    return NULL;
}

static void *UPD (void *arg) {
    char *c = (char *) arg;
    *c = toupper (*c);
    return NULL;
}

static void *PRINT (void *arg) {
    char *c = (char *) arg;
    putchar (*c);
    return NULL;
}
```

Soluzione

```
FILE *fg;

int main (int argc, char ** argv) {
    char next, this, last;
    int retC;
    pthread_t tGet, tUpd, tPrint;
    void *retV;

    if ((fg = fopen(argv[1], "r")) == NULL) {
        perror ("Errore fopen\n");
        exit (0);
    }
    this = ' ';
    last = ' ';
    next = ' ';
```

Soluzione

È possibile gestire separatamente i primo due caratteri

```
while (next != EOF) {
    retC = pthread_create (&tGet,NULL,GET,(void *)&next);
    if (retC != 0) fprintf (stderr, ...);
    retC = pthread_create (&tUpd,NULL,UPD,(void *)&this);
    if (retC != 0) fprintf (stderr, ...);
    retC = pthread_create (&tPrint,NULL,PRINT,(void *)&last);
    if (retcode != 0) fprintf (stderr, ...);
    retC = pthread_join (tGet, &retV);
    if (retC != 0) fprintf (stderr, ...);
    retC = pthread_join (tUpd, &retV);
    if (retC != 0) fprintf (stderr, ...);
    retC = pthread_join (tPrint, &retV);
    if (retC != 0) fprintf (stderr, ...);
    last = this;
    this = next;
}
```

Soluzione

Gestione degli ultimi
due caratteri (coda)

```
// Last two chars processing

retC = pthread_create(&tUpd, NULL, UPD, (void *)&this);
if (retC != 0) fprintf (stderr, ...);
retC = pthread_create(&tPrint, NULL, PRINT, (void *)&last);
if (retC != 0) fprintf (stderr, ...);
retC = pthread_join (tUpd, &retV);
if (retC != 0) fprintf (stderr, ...);
retC = pthread_join (tPrint, &retV);
if (retC != 0) fprintf (stderr, ...);
retC = pthread_create(&tPrint, NULL, PRINT, (void *)&this);
if (retC != 0) fprintf (stderr, ...);
return 0;
}
```

Considerazione

- ❖ La soluzione precedente è efficiente?
 - No, perché i thread della pipeline sono creati ex-novo a ogni iterazione
 - Soluzione alternativa
 - T_1 , T_2 e T_3 possono essere thread ciclici in grado di sincronizzarsi tra di loro a ogni iterazione

