TESTING

Parte Mandatory (`philo`)

Struttura

- Un **thread per filosofo**.
- Un **fork per filosofo**.
- Ogni fork deve essere protetto da un **mutex**.
- **Output sincronizzato**: le stampe non devono sovrapporsi.
- La morte di un filosofo deve essere gestita con un **mutex** per evitare conflitti con altre azioni (es. mangiare).

Test richiesti

- `1 800 200 200` → deve morire.
- `5 800 200 200` → nessun filosofo deve morire.
- `5 800 200 200 7` → fine quando tutti hanno mangiato 7 volte.
- `4 410 200 200` → nessun morto.
- `4 310 200 100` → uno muore.
- `2` filosofi → se la morte avviene con ritardo >10 ms → **errore**.

FUNZIONI

`pthread_create`

Crea un nuovo thread. Il thread esegue la funzione `start_routine`, ricevendo `arg` come parametro. Condivide memoria e risorse con gli altri thread del processo. Restituisce 0 in caso di successo, codice d'errore altrimenti. Il thread termina quando la funzione ritorna o viene chiamato `pthread_exit`.

`pthread_detach`

Segnala al sistema che non è necessario eseguire `pthread_join` su quel thread. Il sistema libererà automaticamente le sue risorse una volta terminato. Impedisce il leak di risorse in caso di thread non joinati.

`pthread_join`

Blocca il thread chiamante finché il thread specificato non termina. Permette di sincronizzarsi con la sua fine ed eventualmente raccogliere il valore restituito. Se usato su un thread già detached, il comportamento è indefinito.

`pthread_mutex_init`

Inizializza un mutex. Deve essere chiamata prima dell'uso. Attributi personalizzati opzionali. `NULL` applica i default.

`pthread_mutex_destroy`

Dealloca un mutex. Ammesso solo quando il mutex non è in uso da alcun thread. Chiamarlo mentre il mutex è bloccato comporta comportamento indefinito.

`pthread_mutex_lock`

Richiede il lock del mutex. Se già acquisito da un altro thread, blocca fino al rilascio. Accesso esclusivo garantito. Rischio di deadlock in assenza di gestione rigorosa.

`pthread_mutex_unlock`

Rilascia il lock. Deve essere eseguita solo dal thread che lo ha acquisito. Violazioni generano comportamento indefinito.

Mutex = meccanismo di esclusione reciproca. Evita race condition. Permette sincronizzazione tra thread in ambienti a memoria condivisa.

pthread_create

Inizializza un nuovo thread. Esegue la funzione `start_routine`, ricevendo `arg` come parametro. Condivide lo spazio di memoria del processo. Il valore di ritorno è 0 se l'operazione ha successo, un codice d'errore altrimenti. Il thread termina restituendo da `start_routine` o invocando `pthread_exit`.

pthread_detach

Marca un thread come "distaccato". Le risorse di sistema vengono rilasciate automaticamente alla sua terminazione. Impedisce l'uso di `pthread_join` sullo stesso thread. Serve a evitare la persistenza di risorse non liberate.

pthread_join

Blocca il thread chiamante fino alla terminazione del thread specificato. Se `retval` non è NULL, il valore restituito dal thread viene assegnato al puntatore fornito. Rende possibile sincronizzazione esplicita e recupero del risultato di un thread. Incompatibile con thread già distaccati.

Differenza tra processi e thread

Un **processo** è un programma in esecuzione. Ha:

- **Uno spazio di memoria indipendente** (stack, heap, data, code).
- **Risorse proprie** (file aperti, variabili ambientali, registri).
- **Isolamento**: un processo non può accedere direttamente alla memoria di un altro processo.
- **Contesto completo**: ogni processo ha il proprio contesto di esecuzione (CPU, scheduler, ecc.).

> Esempio: aprire due istanze di Firefox crea due processi separati.

Thread

Un **thread** è un'unità di esecuzione all'interno di un processo. Condivide:

- Lo **stesso spazio di memoria** del processo (heap, dati globali).
- Alcune **risorse** (file, variabili).
- Ha **stack e contesto di esecuzione proprio**, ma condivide la maggior parte dell'ambiente con gli altri thread del processo.

> Esempio: in un'app come Word, un thread può gestire l'input da tastiera, un altro il salvataggio automatico.

Lo stack è un'area di memoria che memorizza le variabili locali e le informazioni di chiamata delle funzioni(come i return address). E' organizzato secondo la regola LIFO (Last In First Out).

- Cresce e si riduce automaticamente con le chiamate e i ritorni da funzione.
- Più veloce ma limitato in dimensione
- Gestito direttamente dal sistema

Ogni volta che una funzione viene chiamata, il programma crea un nuovo stack frame, una struttura temporanea che contiene:

- il valore di ritorno
- i registri salvati
- i parametri della funzione
- le variabili locali.

Quando la funzione termina, il frame viene distrutto automaticamente. Questo significa che la memoria dello stack è a durata automatica: non serve free, né gestire puntatori.

Ogni thread ha il suo stack separato

L'heap è un'area della memoria dinamica del processo riservata all'allocazione esplicita durante il runtime.

Viene gestita dal memory allocator del sistema es. `malloc()` , `calloc()` `free()` .

A differenza dello stack, l'heap non segue una struttura LIFO e non viene gestito automaticamente.

Thread

E’ una unita’ fondamentale della CPU che rappresenta un flusso indipendente di istruzioni all’interno di un processo. (cioe’ un processo puo’ essere diviso in uno o piu filoni/istanze = thread).

Tutti i Thread dello stesso processo condividono lo stesso spazio di memoria. ma hanno un proprio stack, un proprio program counter(ovvero tiene traccia dell’istruzione corrente) e registri indipendenti.

CARATTERISTICHE PRINCIPALI

- **Condivisione della memoria**: tutti i thread di un processo vedono gli stessi dati allocati dinamicamente.
- **Esecuzione concorrente**: i thread possono essere eseguiti in parallelo su CPU diverse (se disponibili) o in modo interlacciato su una sola.
- **Contesto minimo**: passare da un thread all’altro è più veloce che tra processi, perché i thread condividono risorse.

Gli stati di un thread

- Ready = pronto ad essere eseguito
- Running = attualmente in esecuzione
- Blocked = in attesa do un evento (es: rilascio di un mutex)

RACE_CONDITION

Una race condition si verifica quando due o più thread accedono a una risorsa condivisa(memoria, file, variabile globale) senza una sincronizzazione adeguata, e almeno uno modifica lo stato della risorsa.

```
int counter = 0;
void   *increment(void *arg)
{
    int i = 0;
    while(i < 1000)
```

```
    {
        counter++;
        i++;
    }
    return NULL;
}
```

Se due thread eseguono `counter++` contemporaneamente, i cicli possono sovrapporsi, perdendo incrementi. Il risultato non sarà 2000 ma un valore inferiore, variabile a ogni esecuzione.

DEADLOCK

Un ***deadlock*** è una situazione in cui due o più thread sono **bloccati per sempre**, ciascuno in attesa di una risorsa posseduta dall'altro.

Condizioni necessarie (Coffman):

1. Mutua esclusione: almeno una risorsa non condivisibile
2. Hold and wait: un thread mantiene una risorsa mentre ne attende un'altra
3. No preemption: le risorse non possono essere forzatamente tolte
4. Attesa circolare: esiste un ciclo di attesa tra i thread

STARVATION

La ***starvation*** si verifica quando un thread non riesce mai a ottenere una risorsa o la CPU perché altri thread monopolizzano l'accesso. È una forma di iniquità di scheduling o priorità.

Cause comuni:

- Lock non equi (non FIFO)
- Thread a priorità bassa sempre preempted da altri
- Thread di gestione che monopolizzano il mutex