Updated:
22/04/2023

Operating Systems
Progetto Multithreading

Job paralleli

Da consegnare entro il 2?.0?.2023@23:59

Student Code Ethics

Students are expected to maintain the highest standards of academic integrity. Work that is not of the student's own creation will receive no credit, and disciplinary actions can follow. Remember that you cannot give or receive unauthorized aid on any assignment, quiz, or exam. A student cannot use the ideas of another and declare it as his or her own.

Submission

Upload a file `main.c` on OLE.

Objectives

Analyze an existing code base in C. Write standard C code that satisfies a specification using specific existing functions. Implement a solution to have multiple concurrent threads complete a shared list of jobs and report back to the user. Use Pthread mutexes to ensure a sound synchronization and avoid race conditions. Write correct and clean code.

Evaluation criteria

The grade will reflect the correctness, elegance, and overall quality of the code in the file `main.c`. Any academic misconduct will lead to a failing grade.

Further remarks

- This project description may evolve; specially to correct mistakes or ambiguities.
- Signal any possible ambiguity in this description.
- Allocate only the memory necessary.
- Free memory, close files, destroy mutexes, etc, as soon as possible.
- Eliminate race conditions, but do not abuse mutual exclusion. Only use mutex locks around critical sections. Release mutex locks as soon as possible.
- Test your implementation carefully; make sure that no race conditions occur; make sure that there are no memory leaks.
- Printf is buffered, and it may **hide** race conditions!

Instructions

Si richiede di scrivere un programma C all'interno del file sorgente "`main.c`" che dimostri la capacità dello studente nel gestire la sincronizzazione fra diversi thread. Il programma dovrà leggere da un file di testo dei task (detti "job") che devono essere eseguiti in parallelo da dei thread. Un esempio di file di testo è il seguente:

```
s
1
m
3*4
f
34024135
p
18287
```

Ogni job è definito da due righe. La prima contiene un carattere che indica il job da svolgere mentre la seconda i parametri del job. I job possibili sono 4:

1. **`s`** (`sleep'): in questo caso il thread deve solo "dormire" per il numero di secondi indicato dal suo parametro;
2. **`m`** (`math'): il thread dovrà leggere i parametri e calcolare il risultato di un'operazione matematica fra due operandi che puoi essere '`+`', '`-`', '`*`', '`/`', stampandolo a terminale;
3. **`f`** (`factorization'): il thread dovrà calcolare la scomposizione in fattori primi del numero dato come parametro e stampare tale lista di numeri;
4. **`p`** (`prime'): il thread dovrà calcolare l'n-esimo numero primo (dove n è il parametro) e stamparlo. Ad esempio "`p 1`" dovrà stampare `2` (2 è il primo numero primo) mentre "`p 3`" dovrà stampare `5`.

L'implementazione di questi metodi vi viene già fornita, poiché lo scopo del progetto è dimostrare la capacità di lavorare con i thread e di sincronizzarli. Trovate questi metodi all'interno della libreria "`jobs.{h,c}`" su OLE. Questo vale anche per le funzioni di lettura del file ("`filereader.{h,c}`" su OLE). Inoltre vi viene fornito un file "`main.c`" con alcuni metodi e definizioni già presenti. Le parti che richiedono implementazione sono indicate dal seguente commento:

`// STUDENT CODE GOES HERE`

I metodi da implementare sono i seguenti:

- **`main()`**: il programma principale;
- **`thread()`**: la funzione svolta da ogni thread;
- i metodi **`run_sleep()`**, **`run_math()`**, **`run_factorization()`** e **`run_prime()`**: questi metodi vengono invocati da un metodo già implementato (**`run_job()`**, che potete sfruttare nel vostro codice) e devono semplicemente estrarre i parametri del job da una stringa (letta dal file) e invocare l'operazione corrispondente a tale job all'interno della libreria "`jobs.h`". Questi metodi si occupano anche di stampare informazioni come il job che viene eseguito, quale thread lo sta eseguendo, il risultato del job, etc.

Nello sviluppare il progetto dovrete far si che ogni thread legga i job dal file. **NON È CONSENTITO** far leggere tutti i job al thread principale e poi passare i job ai vari thread.

Dovrete quindi **sincronizzare l'accesso al file** tramite un semaforo per evitare letture simultanee. In aggiunta dovrete **sincronizzare la stampa a terminale**, altrimenti l'output dei thread potrebbe mescolarsi rendendo illeggibile l'output. Nella sincronizzazione, ridurre al minimo il tempo in cui un thread tiene il semaforo bloccato, in modo che l'esecuzione dei job sia in parallelo. Evitate quindi un thread di questo tipo

```
acquisisci_semaforo()
leggi_file()
esegui_job()
stampa_risultato()
rilascia_semaforo()
```

poiché bloccherebbe gli altri thread durante l'esecuzione del job.

Altre considerazioni:

- il file dal quale caricare i jobs **dovrà chiamarsi** "`jobs.txt`" e dovrà trovarsi nella stessa cartella nella quale si trovano i file sorgenti e il programma compilato e non in altre sotto-cartelle. Tale nome è definito tramite una costante all'interno di "`main.c`". Il file dovrà quindi venire aperto invocando "`open_file(FILENAME)`". La funzione restituisce un puntatore a una variabile di tipo "`FILE`", la quale viene utilizzata per far riferimento a tale file durante le operazioni di lettura. Terminato il suo utilizzo, il file deve essere chiuso passando tale puntatore alla funzione "`close_file()`";
- per quanto riguarda il numero di thread, deve dipendere da una costante definita in "`main.c`" (`N_THREADS`);
- il parametro passato a ciascun thread dovrà essere un puntatore ad una `struct` definita come sotto. Tale `struct` include l'indice del thread (utilizzato nella stampa dell'output per indicare quale thread sta svolgendo cosa), un puntatore al file (che dovrà essere aperto all'interno della funzione `main()` prima di creare i thread) e un puntatore al mutex utilizzato per la sincronizzazione. (I thread si sincronizzano con un solo mutex.) Quest'ultimo quindi **NON DOVRÀ** essere definito come una variabile globale, ma come una variabile locale del metodo `main()`;

```
struct thread_params {
    // index of the thread
    int thread_id;
    // pointer to the file from which jobs should be read
    FILE *jobs_file;
    // pointer to the mutex used to synchronize the threads
    pthread_mutex_t *mutex;
};
```

- per ottenere il prossimo job da eseguire vi viene fornita una funzione già implementata e così definita: **`char* get_next_job(struct thread_params *params, char* job_type)`**. Questa funzione **non contiene codice di sincronizzazione** ed è quindi compito dell'utente garantire che solo un thread alla volta la utilizzi. I parametri sono i seguenti:
    - `params`: puntatore alla struct contenente i parametri passati al thread, come descritto al punto precedente

    - `job_type`: puntatore a un **singolo** carattere all'interno del quale la funzione scrive il tipo di job da eseguire. Questo verrà impostato a `'s'`, `'m'`, `'f'` o `'p'` nel caso di operazioni valide, a `0` se non vi sono altri job all'interno del file, oppure a `-1` se il job letto dal file non è uno di quelli elencati sopra;
    - la funzione restituisce una stringa con i parametri del job (seconda riga del file job). Questa stringa viene **allocata dinamicamente** e la sua memoria deve essere liberata dall'utente quando questa non è più necessaria.
- il metodo "`job_factorization()`" di "`jobs.h`" restituisce un array allocato dinamicamente contenente i fattori primi del numero da fattorizzare. È compito dell'utente liberare la memoria riservata una volta che questa non viene più utilizzata (ossia all'interno della funzione `run_factorization`);
- per eseguire il parsing dei parametri di un job (parametro `*job_parameters`), utilizzate la funzione "**`sscanf(`**`job_parameters`**`)`**" ("string `scanf`"). Funziona esattamente come "`scanf`" con la differenza che questa esegue la ricerca all'interno di una stringa invece che prendere l'input dalla tastiera. In aggiunta, se dovete leggere più valori dalla stessa stringa, dovete farlo con una sola invocazione mettendo più modificatori all'interno della stringa di controllo, e non invocando la funzione più volte sulla stessa stringa. I parametri di **`sscanf(char *string, char *format, …)`** sono i seguenti:
    - **`string`**: stringa all'interno della quale cercare i valori da convertire
    - **`format`**: stringa di controllo (con `%d`, `%c`, `%f`, … come in `scanf`)
    - …: elenco di puntatori alle variabili all'interno delle quali scrivere i valori ottenuti dalla conversione (come in `scanf`)

  Un esempio di utilizzo per il job `sleep' è il seguente:

```
int seconds;
sscanf(job_parameters, "%d", &seconds);
```

Un esempio di output del programma, dato il file "`jobs.txt`" ad inizio documento, potrebbe essere il seguente:

```
Thread 1: Running sleep(1)
Thread 2: Running factorization(34024135)
Thread 3: Running math(3.000000 * 4.000000)
Thread 3: Job terminated. Result: 12.000000
Thread 3: Running prime(18287)
Thread 2: Job terminated. Result: 5 1871 3637
Thread 2: No further jobs to be processed. Terminating
Thread 3: Job terminated. Result: 203807
Thread 3: No further jobs to be processed. Terminating
Thread 1: Job terminated
Thread 1: No further jobs to be processed. Terminating
```

Assumiamo che il file contenga un numero pari di righe. Ma ci possono essere dei jobs mal specificati.

```
p
2
m
22.1-p
s
5
q
45678
m
3.8*32.8
f
1340135
s
-
m
3/6.4
```

Un esempio di output del programma, dato il file "`jobs.txt`" sopra, potrebbe essere il seguente:

```
Thread 1: Running prime(2)
Thread 1: Job terminated. Result: 3
Thread 1: Running sleep(5)
Thread 2: Error getting next job
Thread 3: Error parsing math job parameters
Thread 2: Running math(3.800000 * 32.799999)
Thread 2: Job terminated. Result: 124.639992
Thread 2: Error parsing sleep job parameters
Thread 3: Running factorization(1340135)
Thread 2: Running math(3.000000 / 6.400000)
Thread 2: Job terminated. Result: 0.468750
Thread 2: No further jobs to be processed. Terminating
Thread 3: Job terminated. Result: 5 433 619
Thread 3: No further jobs to be processed. Terminating
Thread 1: Job terminated
Thread 1: No further jobs to be processed. Terminating
```