Programmazione di sistema

Anno accademico 2018-2019

Esercitazione 4

Si realizzi un programma per la simulazione di un gestore di attività con politica di scheduling Round-Robin. Nel simulatore, le attività vengono rappresentate sotto forma di oggetti (Job) caratterizzati da alcuni parametri fondamentali:

- id, identificativo univoco di tipo intero, assegnato sequenzialmente all'atto della creazione dell'oggetto;
- duration, indicatore della durata del compito da svolgere in millisecondi;
- execution_time, che indica quanto tempo è già stato dedicato dallo schedulatore all'attività corrente;
- start_time, valore numerico che rappresenta l'istante di tempo in cui potrà iniziare l'elaborazione dell'attività;
- wait_time, valore numerico che indica il tempo cumulato di attesa, ovvero la somma di tutti gli intervalli di tempo in cui l'attività, pur risultando eseguibile, non ha potuto essere eseguita per mancanza di risorse;
- completion_time, indica l'istante di tempo in cui il gestore ha terminato l'elaborazione dell'attività.

Il gestore, rappresentato da una istanza della classe JobScheduler, mantiene al proprio interno diverse strutture dati: una coda di Job in attesa di poter iniziare la propria esecuzione, ordinata per start_time; una coda di attività in corso di svolgimento, di tipo FIFO, un vettore di attività terminate. Inoltre, il JobScheduler gestisce un gruppo di thread, responsabili della esecuzione delle attività in corso; tali thread simulano i diversi core di una ipotetica CPU responsabile di eseguire le attività. I thread di tale gruppo estraggono, se presente, un Job dalla seconda coda e ne simulano la (parziale) esecuzione attendendo una quantità di tempo pari al minimo tra la durata residua dell'attività e il quanto di tempo impostato sul simulatore. Terminato tale intervallo il thread aggiorna il tempo di esecuzione, il tempo di attesa ed eventualmente il tempo di completamento; inoltre, se il compito non risulta ancora completato, lo rimette in coda.
Ogni volta che un thread estrae una nuova attività, provvede a stamparne i relativi parametri caratterizzanti. Al termine dell'esecuzione vengono stampati in output il tempo di turnaround medio (differenza tra completion_time e start_time), il tempo di attesa medio e la percentuale di occupazione della CPU.

La classe **JobScheduler** mantiene come membri privati un pool di thread (pari al massimo tra 2 e il numero di thread hardware a disposizione), un coda a priorità di attività in attesa di partire, una coda thread-safe di attività in corso di esecuzione, un vettore thread-safe di attività terminate ed una costante che rappresenta il quanto di tempo di esecuzione per ogni round. JobScheduler offre le seguenti funzioni membro pubbliche:

- **void submit**(Job j) – riceve come parametro un nuovo Job e lo inserisce nella coda dei processi in attesa di partire
- **void start**() – dà inizio alla simulazione: innanzitutto crea tutti i thread del pool; allo scadere del tempo di avvio di ciascun elemento posto nella coda di ingresso, lo sposta alla coda dei processi in esecuzione; si mette in attesa che tutti i thread del pool terminino.
  I singoli thread devono attendere che sia presente almeno un elemento nella coda dei processi attivi, estrarlo, simulare l'esecuzione aggiornandone i parametri caratteristici e inviarlo alla sua corretta destinazione (la cosa stessa o il vettore delle attività terminate). Se sia la coda dei processi attivi che quella in attesa di avvio sono vuote, i thread terminano la propria esecuzione.
- **~JobScheduler**() – distruttore; si occupa di verificare che tutti i thread del pool abbiano terminato l'esecuzione e stampa le statistiche finali

La classe **Job** rappresenta un processo, caratterizzato da id, tempo di inizio, tempo di attesa cumulato e durata.

Si faccia ad evitare forme di busy-loop, ovvero, ad eseguire cicli che consumano attivamente la CPU dell'elaboratore in attesa che si verifichino le condizioni necessarie alla prosecuzione del programma

Esempio di uso

```
int main() {

    Scheduler p{}; // RR, quanto di tempo = 3000, poolsize = 8
    p.submit(Job(1, 0, 15000));  // Job(int id, int start_time, int duration)
    p.submit(Job(2, 0, 6000));
    p.submit(Job(3, 1000, 9000));
    p.submit(Job(4, 2000, 12000));
    p.submit(Job(5, 3000, 16000));
    p.submit(Job(6, 3000, 5000));
    p.submit(Job(7, 4000, 7000));
    p.submit(Job(8, 4000, 6000));
    p.submit(Job(9, 5000, 9000));

    p.start();

}
```

Output

```
thread 139751705626368 running job 1 at t = 0
thread 139751697233664 running job 2 at t = 0
thread 139751688840960 running job 3 at t = 1000
thread 139751680448256 running job 4 at t = 2000
```

```
...
job 2 done at t = 5000
thread 139751680448256 running job 4 at t = 6000
...

avg turnaround time: 9.78s
avg waiting time: 0.33s
cpu usage: 55.92%
exec time: 10.62s
compl time: 19s
```

Competenze da acquisire

- Uso della programmazione concorrente
- Conoscenza e comprensione dei meccanismi di sincronizzazione ed attesa
- Realizzazione di strutture dati thread-safe