TANKERINO.com

Lezione 175

https://www.tankerino.com/it/corsi/10/tepi-4/lezioni/175/operazioni-tipiche-sui-thread

Operazioni Tipiche sui Thread

Una delle operazioni più comuni sui thread è la loro creazione. In molti linguaggi di programmazione, come Java o C#, creare un thread è piuttosto semplice. Quando un nuovo thread viene creato, viene associato a un pezzo di codice che deve eseguire. Per esempio, in Java, possiamo creare un thread definendo una classe che implementa l'interfaccia Runnable o estendendo la classe Thread.

In Java, possiamo creare un thread usando il seguente codice:

```
Thread thread = new Thread(new MyRunnable());
```

Questo creerà un thread che eseguirà il codice definito nella classe MyRunnable.

Avviare un thread significa dire al sistema operativo che quel thread è pronto per essere eseguito. Questo viene fatto chiamando il metodo start() (o un metodo simile, a seconda del linguaggio). Una volta avviato, il thread entra nello stato di esecuzione.

La creazione di un thread implica assegnare un'unità di lavoro specifica al thread. Avviarlo significa renderlo pronto per l'esecuzione.

Quando crei più thread all'interno di un programma, ognuno di essi può essere eseguito in parallelo, eseguendo il proprio compito indipendentemente dagli altri. Tuttavia, gestire questi thread concorrenti può essere complicato, specialmente quando devono condividere risorse.

Sincronizzazione dei Thread

Quando più thread devono accedere a risorse condivise, come variabili o oggetti in memoria, può verificarsi un problema chiamato race condition. Questo accade quando due o più thread tentano di accedere contemporaneamente alla stessa risorsa e l'ordine di esecuzione influisce sul risultato finale.

Per evitare queste situazioni, è necessario sincronizzare i thread. La sincronizzazione assicura che solo un thread alla volta possa accedere a una risorsa condivisa. In Java, questo viene fatto utilizzando

la parola chiave synchronized, che blocca l'accesso a un oggetto o a un blocco di codice.

In Java, possiamo sincronizzare un metodo così:

public synchronized void metodoSincronizzato() { // codice che solo un thread può eseguire alla volta}

Questo assicura che solo un thread possa eseguire questo metodo in un dato momento.

La sincronizzazione permette di evitare conflitti tra thread concorrenti che accedono alle stesse risorse, garantendo che solo un thread alla volta possa eseguire determinate operazioni.

Sebbene la sincronizzazione risolva il problema dei race condition, può introdurre un altro problema chiamato deadlock. Questo accade quando due thread si bloccano a vicenda perché ognuno aspetta che l'altro rilasci una risorsa. La gestione dei deadlock richiede attenzione nella progettazione del codice multithread.

Attesa e Risveglio dei Thread

Un'altra operazione tipica sui thread è l'attesa. A volte un thread deve aspettare che un altro thread completi un'operazione prima di poter proseguire. Questo viene spesso fatto usando metodi come wait() e notify() (o equivalenti, a seconda del linguaggio di programmazione).

Il metodo wait() sospende l'esecuzione del thread fino a quando non viene risvegliato da un altro thread con il metodo notify(). Questo è utile quando un thread deve attendere che una certa condizione venga soddisfatta prima di continuare.

Il metodo wait() sospende l'esecuzione di un thread, mentre notify() risveglia un thread sospeso quando la condizione necessaria è soddisfatta.

Ad esempio, in un programma in cui un thread produce dati e un altro thread li consuma, il thread consumatore potrebbe aspettare fino a quando il thread produttore ha generato nuovi dati.

In un'implementazione tipica del produttore-consumatore, il thread consumatore chiamerà wait() finché non ci sono nuovi dati, e il thread produttore chiamerà notify() quando i dati sono pronti.

Terminazione dei Thread

Un thread termina quando ha completato il proprio compito o quando viene esplicitamente interrotto. In molti linguaggi, i thread terminano automaticamente quando raggiungono la fine del loro metodo principale. Tuttavia, ci sono casi in cui un thread deve essere terminato manualmente, ad esempio quando una condizione cambia e non è più necessario continuare l'elaborazione.

In Java, non è considerata una buona pratica interrompere forzatamente un thread, poiché può lasciare il programma in uno stato inconsistente. Invece, è preferibile che il thread verifichi periodicamente una condizione e si interrompa in modo sicuro quando necessario.

Un thread può essere terminato in modo sicuro verificando una variabile booleana. Ad esempio:

```
while (!terminato) { // esegui il lavoro}
```

Se la variabile terminato viene impostata a true, il thread si interromperà in modo sicuro.

La terminazione di un thread deve essere gestita in modo sicuro, evitando interruzioni improvvise che potrebbero causare stati inconsistenti nel programma.

Una gestione accurata della terminazione è fondamentale per evitare che il thread lasci risorse aperte o dati parzialmente elaborati.

Priorità dei Thread

Non tutti i thread hanno la stessa importanza all'interno di un programma. A volte, è utile assegnare una priorità ai thread per indicare quali thread dovrebbero essere eseguiti prima degli altri. In molti sistemi operativi, i thread con priorità più alta ricevono più tempo di esecuzione rispetto ai thread con priorità più bassa.

In Java, ad esempio, possiamo impostare la priorità di un thread utilizzando il metodo setPriority(). Tuttavia, la priorità è solo un suggerimento per il sistema operativo, che può scegliere di ignorarla se necessario per mantenere l'equilibrio delle risorse.

La priorità di un thread indica al sistema operativo quali thread dovrebbero ricevere più attenzione, ma non garantisce un ordine esatto di esecuzione.

Usare correttamente le priorità dei thread è importante nei sistemi dove alcune operazioni devono essere eseguite più velocemente di altre. Tuttavia, l'uso improprio delle priorità può portare a problemi come la starvation, dove thread a bassa priorità non ricevono abbastanza tempo di

esecuzione.

Un thread che gestisce l'interfaccia grafica di un'applicazione potrebbe avere una priorità più alta rispetto a un thread che esegue un compito in background, in modo da mantenere l'interfaccia reattiva.

Questa lezione e' stata realizzata grazie al contributo di:

Risorse per la scuola
https://www.baobab.school

Siti web a Varese
https://www.francescobelloni.it