busy-waiting? Forse! – come scrivere un busy-wait loop in C (e C++)

Alain: *Al mondo ci sono 8 miliardi di persone, la possibilità di nascere è una su 400 bilioni, eppure tu ed io siamo qui, vuol dire che abbiamo vinto la lotteria cosmica.*

Visto che l'ultimo articolo era della serie **"Forse!"** ho deciso di battere il ferro finché è caldo e ve ne propongo un altro (non dimenticate, però, di leggere anche gli articoli della serie **"No, grazie!"** che sono, spero, interessanti: li potete trovare **qui**, **qui**, **qui**, **qui**, **qui** e **qui**). Oggi parleremo di **busy-waiting**, o meglio di **busy-wait loop**, un argomento che, per quel che ho visto in rete, è fonte di molti dubbi, e noi siamo qui per questo, per fugarli! L'argomento dell'articolo, si intona con il cinquantesimo film del Maestro **Woody Allen**, il bel **Coup de Chance**, un film sulla importanza, nella vita, del caso e della fortuna: affidarsi a un *busy-wait loop* è un po' affidarsi al caso, e quindi bisogna usarli con le dovute precauzioni… **Forse!**



…secondo me quel codice funzionava solo per caso…
Cosa sono i *busy wait-loop*? Senza girarci troppo intorno è meglio vedere un brevissimo esempio con una versione elementare. Vai col codice!

```
// faccio un busy-wait loop "elementare"
while (condizione) { // non faccio nulla: sto aspettando che la condizione diventi false
} // busy loop terminato: proseguo (e, magari, mi scuso con la povera CPU...)
...
```

Facile, no? Il codice è semplicissimo e **ben commentato**, e c'è poco da aggiungere se non che il commento finale *"…mi scuso con la povera CPU…"* anticipa un problema abbastanza grave… In realtà questo difetto è facilmente evitabile con un trucchetto (di cui ho già parlato varie volte, ad esempio **qui**), ma ne parlerò solo alla fine dell'articolo, spiegandovi anche il perché non ne parlo ora (*pazientare, prego…*).

Andiamo avanti: un *busy-wait loop* elementare come quello mostrato sopra ha vari difetti e, per sintetizzare, possiamo isolare i due più gravi (di cui il primo è quello appena anticipato):

1. È un *cpu-killer*, nel senso che durante l'attesa si mangia tutte le risorse del sistema, visto che continua a testare la condizione alla massima velocità possibile.
2. Non ha una condizione d'uscita: il *loop* potrebbe trasformarsi in infinito.

(*…apro una parentesi per fare il precisino: in alcuni ambienti di programmazione, tipicamente quelli del firmware di basso livello senza sistema operativo (il bare-metal, per gli amici) i busy-wait loop "elementari" sono usati e ammessi. Ma questa è un'altra storia…*)

E come si procede (correttamente) se abbiamo bisogno di eseguire qualcosa *"tipo un busy-wait loop"*? Ci sono varie maniere, e dipendono dal tipo di condizione da testare. Vi propongo un piccolo riassuntino di tre casi: sicuramente non è esauriente ma che credo renda bene l'idea; gli esempi forniti sono molto (ma moooolto) semplificati e servono solo a rendere un po' l'idea:

1. La condizione è l'attesa di un segnale del sistema: invece di non fare nulla nel *loop* si usa una funzione bloccante come **pause(2)** o la più moderna **sigsuspend(2)**, e questo comporta un carico nullo per la CPU. Vediamo un piccolo esempio:

   ```
   // faccio un busy-wait loop con pause(2) (o sigsuspend(2))
   while (true) { // aspetto che la pause(2) ritorni perché è arrivato un segnale pause(); // questa è bloccante e non consuma CPU break; // pause(2) è uscita: interrompo il loop
   } // busy-wait loop terminato: proseguo (e, non devo scusarmi con la CPU...)
   ...
   ```

2. La condizione è l'attesa di un evento su una variabile: useremo allora una **condition variable** attraverso la funzione **pthread_cond_wait(3)** che è bloccante (come la

pause(2) dell’esempio precedente). Vediamo l’esempio:

```
// lock mutex
pthread_mutex_lock(&my_mutex); // faccio un busy-wait loop con una condition variable
while (condizione) { // aspetto che la pthread_cond_wait(2) ritorni quando si segnala la condizione pthread_cond_wait(&my_cond, &my_mutex); // questa è bloccante e non consuma CPU // il break non è necessario: la condizione non è più valida
} // unlock mutex
pthread_mutex_unlock(&my_mutex); // busy-wait loop terminato: proseguo (e, non devo scusarmi con la CPU...)
...
```

3. La condizione è l’attesa di una evento di I/O: useremo allora una funzione bloccante (come, ad esempio, la **read(2)** o la **recv(2)**). Vediamo, di nuovo, un piccolo esempio:

```
// faccio un busy-wait loop con read(2) (o recv(2))
while (true) { // aspetto che la read(2) ritorni perché è arrivato un buffer di I/O read(my_fd, my_buf, my_count); // questa è bloccante e non consuma CPU break; // read(2) è uscita: interrompo il loop
} // busy-wait loop terminato: proseguo (e, non devo scusarmi con la CPU...)
...
```

Notare che l’esempio n.3 si poteva anche scrivere usando la nostra cara **select(2)**, che è bloccante e si sveglia quando arriva l’I/O sul canale sorvegliato… ma in questo caso non sarebbe un *busy-wait loop*! Notare anche che, per semplificare e unificare gli esempi, ho usato dei *loop* che sembrano innecessari, ma potrebbero essere utili per risolvere il difetto n.2 della lista-difetti mostrata più sopra, come vedremo più avanti.

E, come promesso all'inizio dell'articolo, vediamo quale è il trucchetto (che molti avranno già intuito) per migliorare il *busy-wait loop "elementare".* Vediamolo!

```
// faccio un busy-wait loop "elementare" migliorato
while (condizione) { // aspetto che la condizione diventi false rilasciando la CPU ad ogni ciclo sleep(1); // o nanosleep o usleep, ecc.
} // busy-wait loop terminato: proseguo (e, non devo scusarmi con la CPU...)
...
```

Con questo semplice trucco il *busy-wait loop* non è più un *cpu-killer*, perché rilascia la CPU (durante il tempo di *sleep)* permettendo ad altri *thread* e/o processi di lavorare liberamente. Ma allora perché ho lasciato questo interessante esempio per ultimo? Semplicissimo: perché un *loop* fatto così non è più un *busy-wait loop*! Infatti i cicli vengono eseguiti molte volte (magari infinite) ma visto che ogni volta viene rilasciata la CPU usare il termine *busy* è poco appropriato.

(*…e apro un'altra parentesi da precisino: i tre esempi "buoni", 1, 2 e 3 qui sopra, si possono considerare dei veri esempi di busy-waiting solo pensando che il loop è bloccato ma è anche attivo; ma c'è chi, con argomenti validi, non li considera degli esempi calzanti… va a finire che l'unico vero busy-wait loop è quello "elementare" mostrato all'inizio!…*)

Comunque questo ultimo caso ci può aiutare a risolvere il secondo dei difetti descritti sopra: *"Non ha una condizione d'uscita: il loop potrebbe trasformarsi in infinito."*. Visto che (andando in *sleep)* il *loop* si sospende per un certo tempo e poi riparte, è abbastanza semplice modificarlo per contare il numero di ripartenze e alzare un allarme o scrivere un messaggio di errore (e, magari, forzare un *break)* nel caso che si superi un certo tempo. Vediamolo!

```
// faccio un busy-wait loop "elementare" migliorato e con condizione d'uscita
int seconds = 0;
```

```
while (condizione) { // aspetto che la condizione diventi false rilasciando la CPU ad ogni ciclo sleep(1); // o nanosleep o usleep, ecc. if (++seconds > 5) { printf("impossibile rispettare la condizione in %d sec\n", seconds); break; }
} // busy-wait loop terminato: proseguo (e, non devo scusarmi con la CPU...)
...
```

Semplice ed efficace, no? Ma allora quest'ultimo esempio (ribadisco: super semplificato) è quello da usare sempre? Io direi di no: in alcuni casi è l'unico possibile e raccomandabile, ma (quando possibile) è consigliabile usare uno degli esempi (1, 2 o 3) descritti sopra, visto che non usano la **sleep(3)** (che è una fonte notevole di problemi, come scrissi **qui** e **qui**). Certo, con il codice di quegli esempi diventa un po' più complicato (ma comunque possibilissimo) gestire la condizioni d'uscita temporizzate, ma con un po' di inventiva si può fare! *Anzi, ve lo lascio come divertente attività per le vacanze di Pasqua, sempre che le facciate, ah ah ah.* E per oggi **ho detto tutto!**

Ciao, e al prossimo post!

https://italiancoders.it/busy-waiting-forse-come-scrivere-un-busy-wait-loop-in-c-e-c/