9° lezione S.O.

Due Date	@April 3, 2025
Multi-select	5 filosofi Lettori-Scrittori Scheduler
Status	Done

Problema dei cinque filosofi

Il problema dei cinque filosofi è un problema classico teorico che **modella l'accesso esclusivo a un numero limitato di risorse**, la cui soluzione può essere usata per problemi reali.

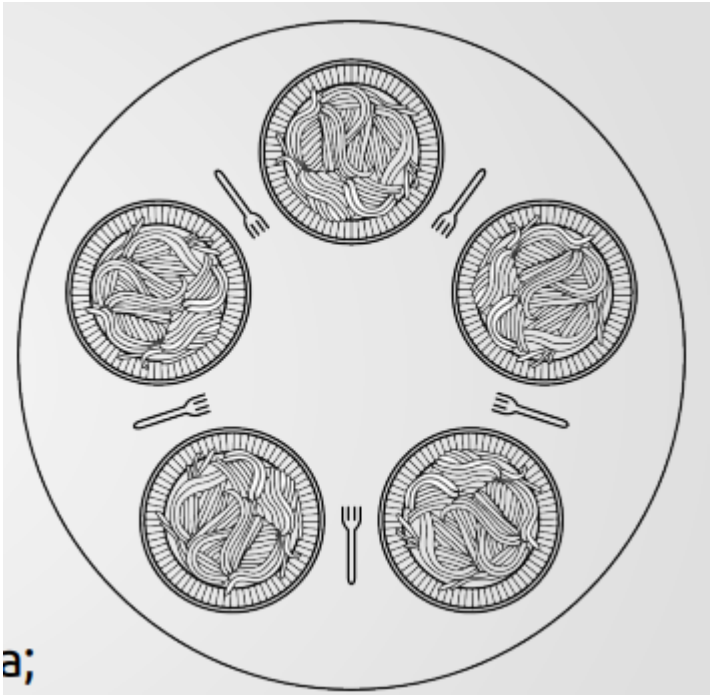
N.B.: non consideriamo 4 filosofi perché la prima soluzione che vedremo per i 5, cambiata leggermente nell'ordine in cui i filosofi prendono le forchette, è sufficiente per risolvere l'intero problema.

Descrizione del problema

Si hanno cinque filosofi seduti attorno a un tavolo circolare. Davanti a ciascun filosofo è presente un piatto, e tra ogni coppia di filosofi vi è una forchetta, per un totale di cinque forchette. Ogni filosofo **pensa** e quando ha fame **mangia**. Per mangiare, **deve prendere entrambe le forchette ai lati del piatto per poter mangiare**.

Questo scenario rappresenta un problema in cui per poter passare a una fase particolare di un task bisogna prendere

risorse esclusive, utilizzabili da un solo processo alla volta. Con cinque forchette, ci possono essere al massimo due filosofi che mangiano contemporaneamente, a
patto che acquisiscano insiemi disgiunti di forchette (ossia i due filosofi non siano vicini tra loro).



L'obiettivo è trovare una soluzione che permetta di fare mangiare i filosofi senza creare deadlock.

Prima soluzione

Un primo approccio al problema può essere il seguente:

```
int N=5
function philosopher(int i)
    think()
    take_fork(i)
    take_fork((i+1) mod N)
    eat()
    put_fork(i)
    put_fork((i+1) mod N)
```

Con la forchetta `i` e `(i + 1) mod N` indichiamo la forchetta che sta rispettivamente alla sinistra e alla destra del filosofo `i` . Le due chiamate `take_fork()` tentano di ottenere le forchette.
Il problema è che se tutti e cinque i filosofi provano a prendere le due forchette **nello stesso momento**, nessuno potrà prendere quella alla propria destra e tutti rimarranno bloccati alla chiamata `take_fork((i + 1) mod N)` (**deadlock**).

Seconda soluzione

Un secondo tentativo consiste nell'introdurre un **tempo di attesa** nel caso in cui non sia possibile prendere la seconda forchetta. L'idea è che il filosofo prenda la prima forchetta e, se non può prendere la seconda, rilascia la prima e attenda un tempo `t` prima di riprovare.
Tuttavia, anche in questo caso si può verificare un deadlock: se tutti i filosofi eseguono le operazioni nello stesso momento e con la stessa attesa `t` , il sistema rimane bloccato: prendono la propria forchetta, vedono che non possono prendere l'altra, rilasciano quella che hanno preso, aspettano e ricominciano.

Terza soluzione

Per evitare la sincronizzazione perfetta della seconda soluzione, si può introdurre un **tempo di attesa** `t` **randomico**. In questo modo, i filosofi non tenteranno di prendere le forchette nello stesso istante, evitando il blocco totale. Questa soluzione funziona, ma **non è ottimale in termini di efficienza** in quanto introduce **tempi morti dovuti** alla casualità dei tempi di attesa, e inoltre

potrebbe fallire se i numeri casuali estratti sono “sfortunati”. Non può quindi essere usata come
effettiva soluzione

Quarta soluzione — con semafori

La soluzione più robusta si basa sull’uso dei **semafori** per gestire sia l’accesso alle risorse (le
forchette) sia il controllo sullo stato di ogni filosofo. Essa è facilmente generalizzabile anche a un
numero maggiore di
5 filosofi.
Ogni filosofo può trovarsi in uno di tre stati:
`THINKING` , `HUNGRY` (vuole mangiare, ma non ha ancora
le forchette) oppure
`EATING` . Per tener traccia dello stato di ciascun filosofo, viene usato un vettore
`state` , in cui la posizione `i` rappresenta lo stato del filosofo `i` .
Per coordinare le operazioni tra i filosofi, vengono usati due strumenti fondamentali:

- Un semaforo `mutex` inizializzato a 1, che garantisce **l’accesso esclusivo al vettore** `state` ;
    - Essenziale in quanto la funzione `test()` viene anche usata per controllare lo stato dei
      filosofi adiacenti e se due filosofi facessero queste operazioni contemporaneamente,
      potrebbero succedere situazioni incoerenti.
- Un array di semafori `s`, uno per ogni filosofo, inizializzati a 0.
    - Servono a bloccare temporaneamente un filosofo quando non può prendere le forchette
      e a risvegliarlo quando le risorse diventano disponibili.

Il comportamento dei filosofi ruota attorno a tre funzioni principali: `take_forks(i)` , `put_forks(i)` e `test(i)` . Quando un filosofo vuole mangiare, chiama `take_forks(i)` :

- Entra nella sezione critica facendo `down(mutex)` , cambia il proprio stato in `HUNGRY` e chiama la

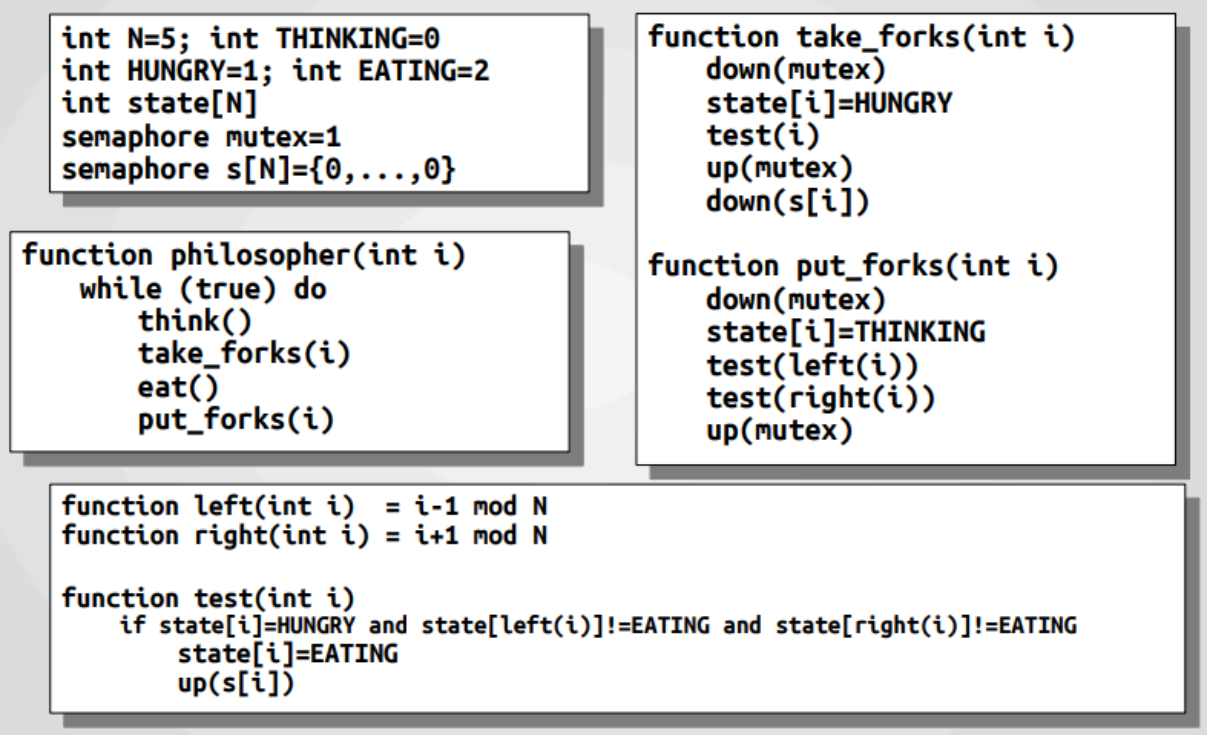
funzione
`test(i)` per verificare se ha la possibilità di mangiare

- La funzione `test(i)` controlla se i due filosofi adiacenti **non stanno mangiando**, e se il filosofo
stesso è
`HUNGRY` . Solo in questo caso può passare allo stato `EATING` , e in tal caso viene fatto
un
`up(s[i])` , così da permettergli di proseguire.
- Uscito dalla sezione critica (`up(mutex)`), il filosofo esegue `down(s[i])` : se `test` gli ha dato il via libera, prosegue subito; altrimenti si blocca finché qualcuno non lo risveglia perché `s[i]` sarà < 0
    - Potremmo pensare di fare una `down` all'interno di `test` stesso mettendo un `else` , ma il
    problema è che
    `down` potrebbe essere bloccante e quindi il mutex non verrebbe mai rilasciato (deadlock). In generale, non si devono mai fare operazioni bloccanti dentro una
    sezione critica, altrimenti si rischia di bloccare tutti. In questo modo, invece, anche se la

      `down` fosse bloccante, gli altri processi potrebbero comunque accedere alla sezione
      critica.

Quando il filosofo ha finito di mangiare, chiama `put_forks(i)` :

- Anche questa funzione accede alla sezione critica (`down(mutex)`), cambia lo stato del filosofo in `THINKING` , e chiama `test` sui due vicini. Lo scopo è verificare se, ora che lui ha liberato le
forchette, uno dei due può finalmente mangiare.
- Se sì, `test` cambierà il loro stato in `EATING` e farà `up` sul loro semaforo, sbloccandoli



Questa soluzione **evita deadlock**, poiché non esistono cicli di attesa in cui tutti i filosofi si
bloccano tra loro aspettando risorse che non si liberano mai, e
evita lo starvation, cioè il rischio
che un filosofo rimanga affamato per sempre: ogni volta che un filosofo finisce di mangiare,
attivamente prova a sbloccare i vicini, dando a ciascuno una possibilità reale di mangiare.
Infine, a differenza di altre soluzioni più semplici ma meno efficienti (dove ad esempio si blocca
l'accesso globale alle risorse e quindi può mangiare un solo filosofo per volta), questa permette

più parallelismo: se due filosofi non sono vicini, possono mangiare contemporaneamente senza
problemi, sfruttando meglio le forchette disponibili.

Soluzione con i monitor

Un'altra soluzione si può ottenere con i monitor, che risulta essere più semplice dal punto di vista
dei bloccaggi nella sezione critica, in quanto il monitor lavora di default in

maniera esclusiva.
Come prima, ogni filosofo può trovarsi in uno di tre stati: `THINKING`, `HUNGRY` o `EATING`. Anche qui si mantiene un vettore `state` per tenere traccia dello stato di ciascun filosofo, ma questa volta il

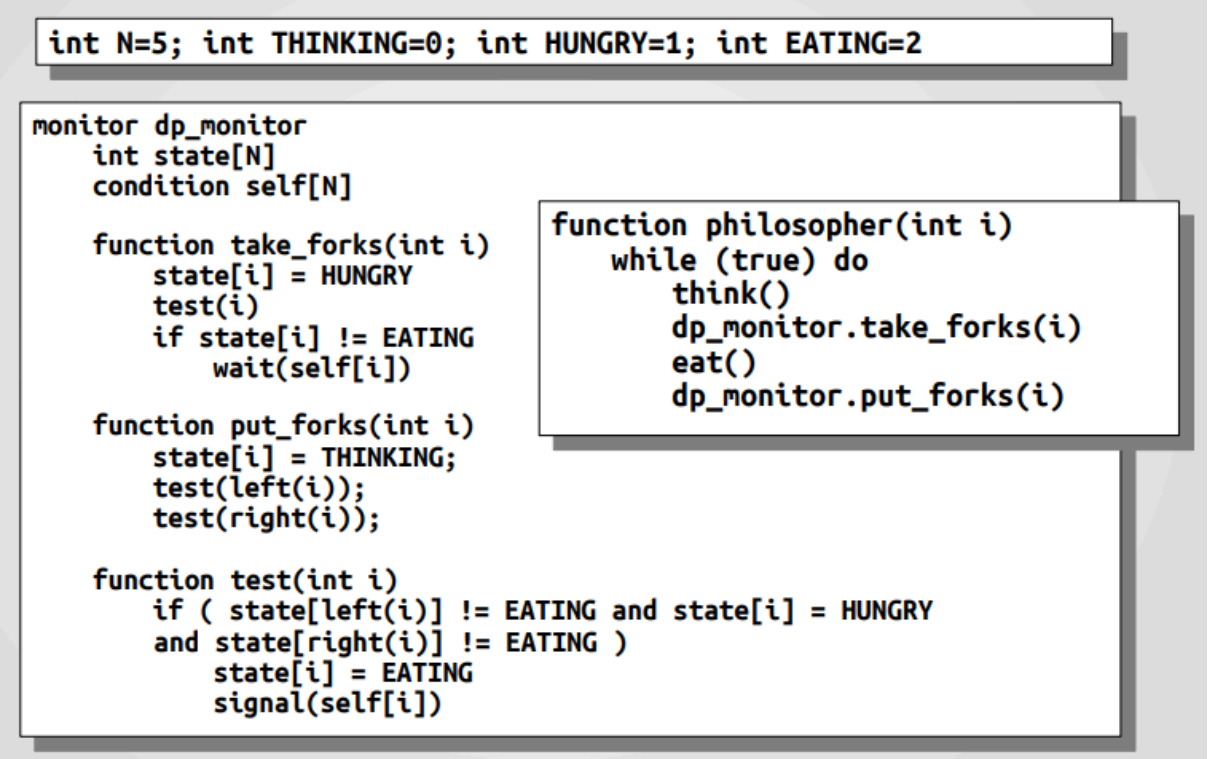
vettore è incapsulato all'interno del monitor, così come le funzioni `take_forks`, `put_forks` e `test`. Ciò garantisce che l'accesso al vettore sia sempre sicuro e coerente. Al posto di un array di semafori come nella versione precedente, qui si usa un **array di variabili di condizione** `self`, una per ogni filosofo.
Quest'ultime permettono ai filosofi di "**auto-addormentarsi**" quando non possono
mangiare, e di essere
risvegliati quando le risorse diventano disponibili

Quando un filosofo vuole mangiare, chiama `take_forks(i)` che:

- Cambia il suo stato in `HUNGRY` e chiama `test(i)` per verificare se può mangiare subito o deve
  aspettare.
- Se può mangiare (cioè se i vicini non stanno mangiando), lo stato viene impostato a `EATING`;
  altrimenti, il filosofo viene addormentato sulla sua variabile di condizione.
- La signal finale (`signal(self[i])`) è unicamente utilizzata per la chiamata test sui filosofi adiacenti.

Il metodo `put_forks(i)` viene chiamato quando il filosofo ha finito di mangiare:

- Cambia il suo stato in `THINKING` e chiama `test` sui due filosofi adiacenti.
- Se uno di questi è `HUNGRY` e le forchette sono libere, allora viene risvegliato con una `signal()`
  sulla sua variabile di condizione, permettendogli di procedere e iniziare a mangiare.



```
int N=5; int THINKING=0; int HUNGRY=1; int EATING=2

monitor dp_monitor
    int state[N]
    condition self[N]

    function take_forks(int i)
        state[i] = HUNGRY
        test(i)
        if state[i] != EATING
            wait(self[i])

    function put_forks(int i)
        state[i] = THINKING;
        test(left(i));
        test(right(i));

    function test(int i)
        if ( state[left(i)] != EATING and state[i] = HUNGRY
        and state[right(i)] != EATING )
            state[i] = EATING
            signal(self[i])

function philosopher(int i)
    while (true) do
        think()
        dp_monitor.take_forks(i)
        eat()
        dp_monitor.put_forks(i)
```

Anche qui, vengono evitati sia i **deadlock** (grazie alla struttura delle chiamate e alla gestione
ordinata delle risorse), sia lo
starvation (perché ogni filosofo ha modo di essere risvegliato
appena possibile dai vicini).

Problema dei lettori e scrittori

Il problema dei lettori e scrittori modella in modo semplice **l'accesso concorrente a una struttura
dati condivisa**
, come ad esempio un **database**, da parte di delle entità (processi o thread). È un
problema "giocattolo", ma molto realistico.

Idea di base

Questo problema sottolinea il fatto che **non tutti gli accessi concorrenti sono pericolosi**. Se **più
processi vogliono solo leggere**
 i dati, non c'è alcun problema: le letture non si disturbano tra loro
e possono avvenire in parallelo. È solo quando entra in gioco

la scrittura che bisogna fare
attenzione. Bisogna quindi garantire:

- Che più lettori possano leggere insieme
- Che quando uno scrittore vuole scrivere, **abbia l'accesso esclusivo** alla risorsa
- Che non ci siano situazioni di **deadlock** o **starvation**

Soluzione basata sui Semafori

Questa soluzione si basa su due semafori:

- `mutex` , che protegge il contatore dei lettori `rc`
  - Tiene traccia di quanti lettori ci sono **attualmente** dentro il database. Se `rc = 0` , non ci
    sono lettori, se
    `rc = 3` , ci sono tre lettori contemporaneamente nel database.
- `db` , che **regola l'accesso al database vero e proprio**, sia per lettori che per scrittori.

Il lettore

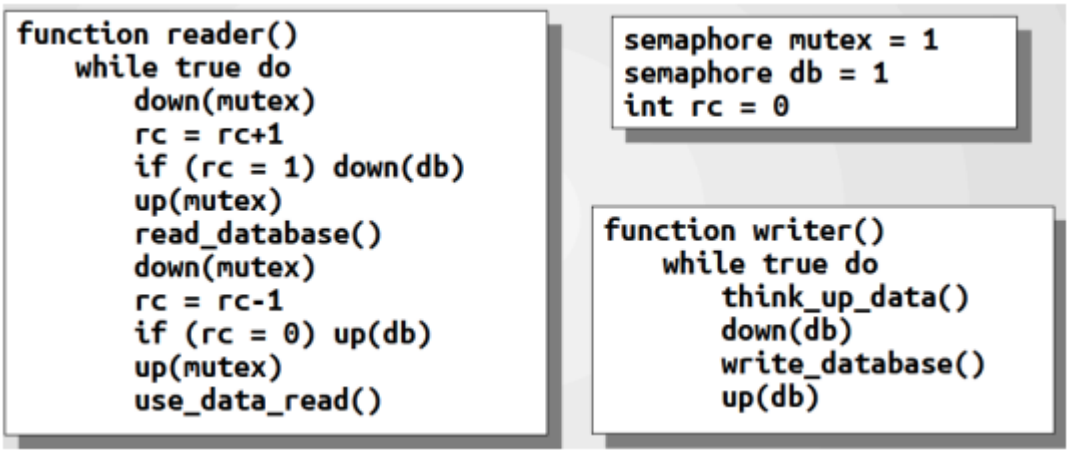
- Entra nella sezione critica protetta da `mutex` per aggiornare `rc` ;
- Se è il **primo lettore**, fa una `down(db)` bloccando l'accesso agli scrittori
  - Se il db è sotto il controllo di uno scrittore, verrà bloccato qui. Il secondo e i successivi
    lettori saranno bloccati da
    `down(mutex)` , creando una coda FIFO di processi bloccati su questo
    `mutex` .
- Esce dalla sezione critica (`up(mutex)`) e inizia a leggere
- Quando ha finito, rientra nella sezione critica (`down(mutex)`), decrementa `rc` ;
- Se è **l'ultimo lettore a uscire**, fa una `up(db)` , sbloccando gli scrittori.

Lo scrittore

- Fa una `down(db)` direttamente: vuole l'accesso esclusivo, quindi deve aspettare che il database

sia completamente libero (nessun lettore né scrittore);

- Scrive nel database
- Da una `up(db)` quando ha finito



Questa soluzione è molto **efficiente per i lettor**i: una volta che il primo è entrato, gli altri possono
seguirlo uno dopo l'altro senza aspettare lo scrittore, ma se i lettori continuano ad arrivare in modo costante, ad esempio uno ogni secondi, e ciascuno legge per secondi, lo scrittore
non entrerà mai (**starvation**).
Per prevenire questa situazione, il programma potrebbe essere scritto in modo leggermente
diverso: quando arriva un lettore e c'è uno scrittore in attesa, il lettore è sospeso dietro allo
scrittore, invece di essere ammesso immediatamente. In questo modo uno scrittore deve restare
in attesa che terminino i lettori che erano attivi quando è arrivato, ma non deve aspettare quelli
arrivati mentre era a sua volta in attesa. Lo svantaggio è che ottiene
meno concorrenza e quindi

minori prestazioni.

Soluzione con i monitor

Il database si trova fuori dal monitor per permettere l'accesso concorrente. Utilizziamo le
seguenti variabili:

- `rc` : numero di lettori attivi.
- `busy_on_write` : flag che indica se uno scrittore sta scrivendo
- `read` , `write` : variabili di condizione per lettori e scrittori.

Prima soluzione

La funzione `start_read()` fa il seguente:

- Se c'è uno scrittore attivo, il lettore si sospende su `read`
- Altrimenti, incrementa `rc` (entra nel gruppo dei lettori);
- Se è il **primo lettore risvegliato dopo uno scrittore**, si occupa di **risvegliare gli altri lettori**
  ancora in attesa (
  `signal(read)`), per permettere l'accesso simultaneo.
  - Il monitor estrae dalla coda il primo thread sospeso su quella condizione e lo risveglia.
    Quest'ultimo a sua volta farà un
    `signal(read)` , che provoca lo stesso comportamento. A cascata, i lettori verranno svegliati.

La funzione `end_read()` fa il seguente:

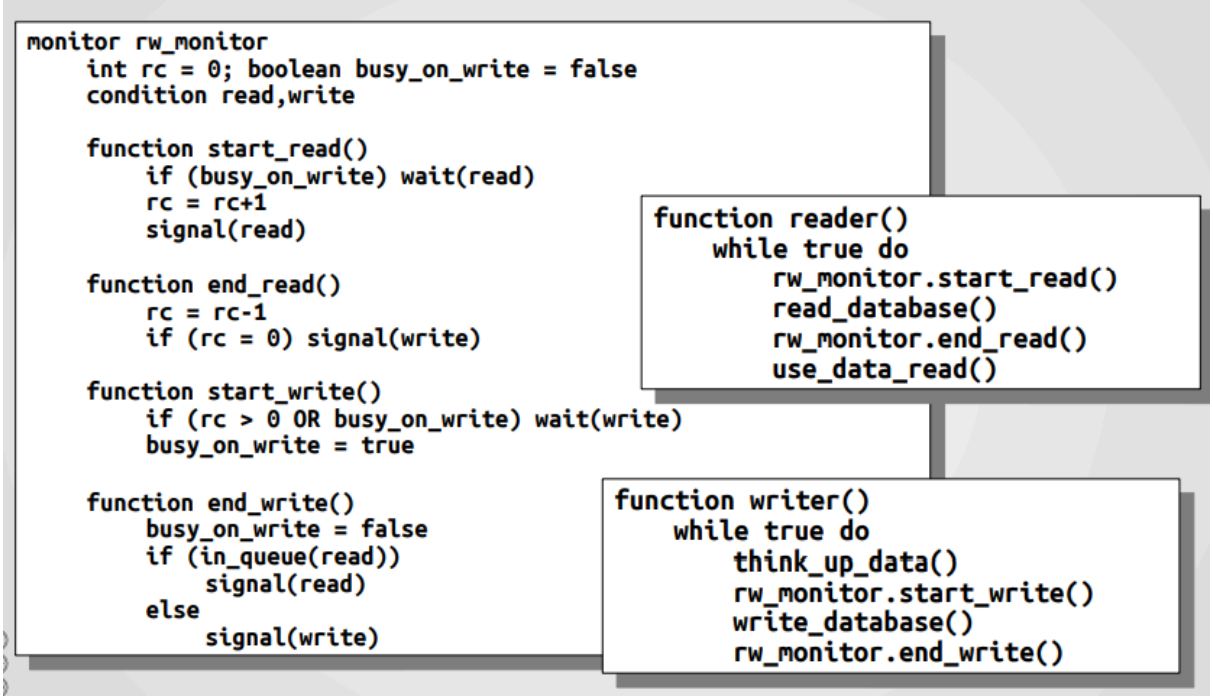
- Decrementa `rc`
- Se è **l'ultimo lettore** a uscire, risveglia **uno scrittore**, se ce ne sono in attesa (`signal(write)`)

La funzione `start_write()` fa il seguente:

- Se ci sono lettori attivi (`rc > 0`) o uno scrittore attivo, lo scrittore si sospende su `write` ;
- Altrimenti, entra nella sezione critica e imposta `busy_on_write = true` .

La funzione `end_write()` fa il seguente:

- Imposta `busy_on_write = false`
- Se ci sono lettori in attesa, **risveglia uno di essi** (il quale poi risveglierà gli altri)
- Se non ci sono lettori, risveglia un altro scrittore in attesa

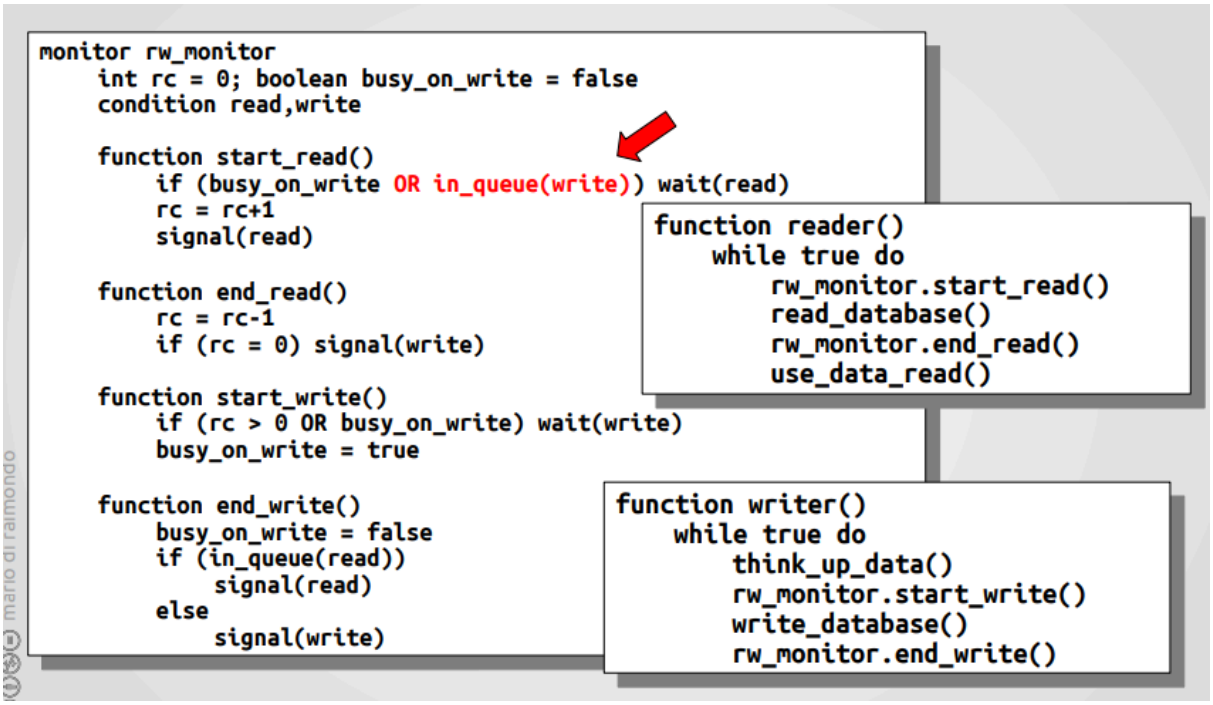


Questa implementazione **favorisce i lettori**: appena uno scrittore termina, si cerca prima di
risvegliare eventuali lettori in attesa. Ciò è **intenzionale**, a differenza della soluzione con semafori dove la scelta era più “casuale”, tuttavia, rimane il problema di **starvation degli scrittori**: se continuano ad arrivare lettori, uno scrittore potrebbe rimanere in attesa indefinitamente.

Seconda soluzione

La seconda soluzione è identica alla prima, ma con una **modifica importante** nella funzione

`start_read()` : oltre a controllare se c’è uno scrittore attivo (`busy_on_write`), si verifica anche se ci sono scrittori in attesa (`in_queue(write)`): se almeno una delle due condizioni è vera, il lettore si blocca.



Questo impedisce che nuovi lettori continuino a entrare quando ci sono scrittori in attesa,
evitando che quest'ultimi vengano continuamente rimandati. È una soluzione più equilibrata, perché riduce la discriminazione degli scrittori. Ad esempio, se un gruppo di lettori
L_1 sta leggendo e arrivano sia scrittori che un nuovo gruppo di lettori L_2, quando L_1 finisce, il primo lettore di L_2 si accorge della presenza di scrittori in attesa e si blocca, lasciando spazio allo scrittore. Dopo la scrittura, questo risveglierà uno dei lettori di L_2, che poi sbloccherà gli altri.

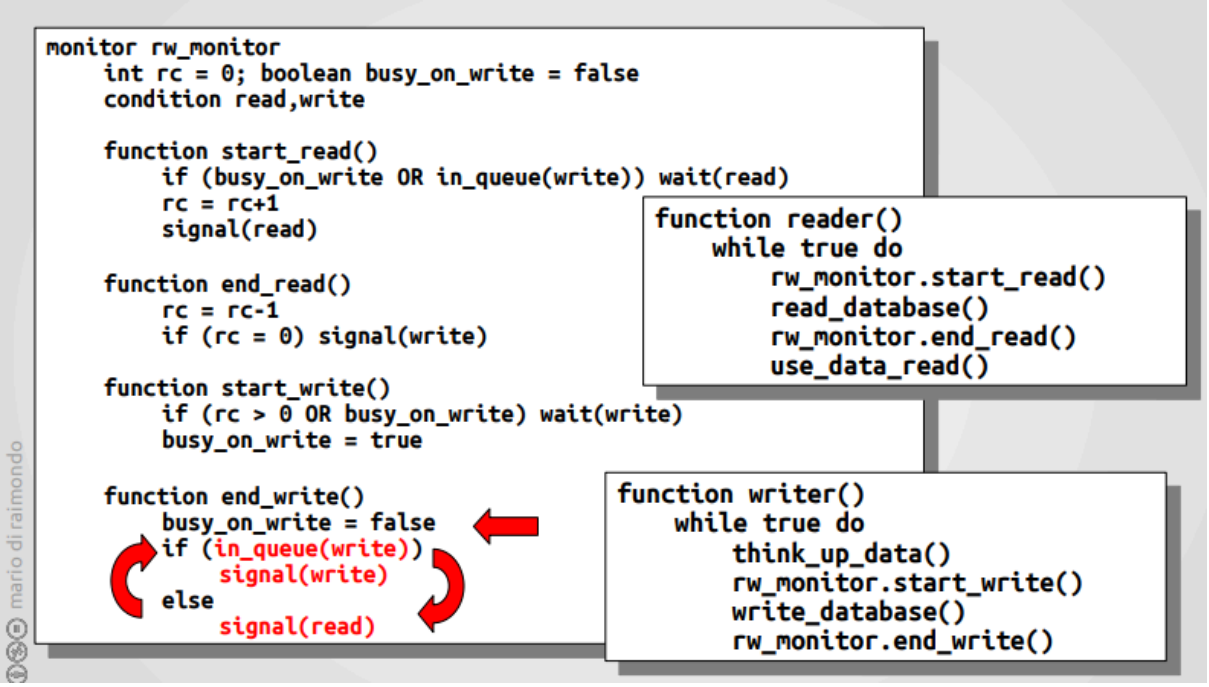
L'unica cosa che può essere **migliorata** è il fatto che **lo scrittore dovrebbe ancora aspettare troppo tempo**.

Terza soluzione

La terza soluzione è molto simile alla seconda, ma cambia il comportamento nella funzione

`end_write()` . In particolare, quando uno scrittore termina, invece di risvegliare subito un lettore (che poi sbloccherebbe gli altri), controlla prima se ci sono altri scrittori in attesa. Se sì, ne risveglia
uno, altrimenti passa ai lettori. In pratica, gli
scrittori "fanno squadra" e si **danno la precedenza a vicenda**

, cercando di ridurre il tempo d'attesa tra uno scrittore e l'altro. Questa strategia dà quindi
una leggera preferenza agli scrittori rispetto ai lettori.



Va comunque detto che anche questa soluzione, come le precedenti, non elimina del tutto il
problema della
starvation: uno scrittore potrebbe comunque trovarsi ad aspettare a lungo se continuano ad arrivare altri scrittori prima di lui.

Scheduling

Lo scheduling è quella parte del sistema operativo, gestita dal **kernel**, che si occupa di decidere
quale processo, tra quelli pronti, deve
ricevere la CPU ogni volta che questa si libera. È un
passaggio fondamentale per garantire che il sistema rimanga
reattivo ed efficiente, soprattutto
nei sistemi dove più processi sono in competizione contemporaneamente. Per fare questa scelta,
lo scheduler si basa su specifici algoritmi che variano a seconda del tipo di sistema: interattivo,
batch, real-time, ecc.

Il modo in cui un algoritmo può prendere una decisione dipende anche da **quali** processi sono
presenti nella coda, in particolare da
come utilizzano le risorse del calcolatore, ossia:

- **Uso del processore**
- **Uso di risorse di altra natura** (I/O, network, ...)

Distingueremo quindi le fasi in cui un processo usa la CPU (chiamate **CPU burst**) da quelle in cui
utilizza altre risorse.

Tipologie di processi

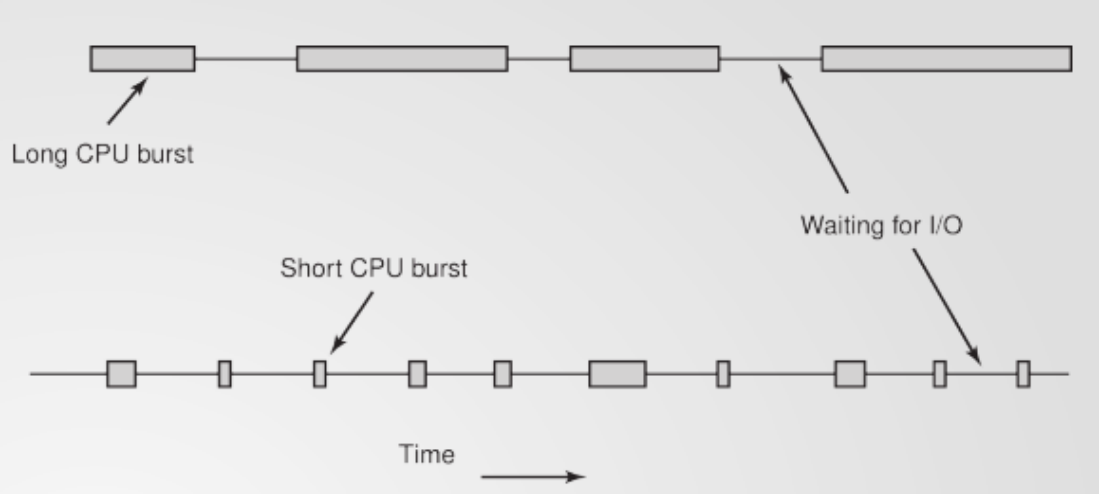
In particolare, si distinguono due famiglie di processi:

- **CPU bounded**: sono interessati per lo più all'utilizzo della CPU e hanno diversi **CPU burst**
  molto lunghi
  .
- **I/O bounded**: sono interessati più a fare operazioni di I/O e poche computazioni, quindi
  hanno
  CPU burst molto corti.

Non tutti i processi sono strettamente CPU bounder o I/O bounded, ma possono cambiare natura
nel tempo: potrebbero, ad esempio, caricare delle informazioni dalla memoria e successivamente fare diverse computazioni.

La classificazione quindi **non è assoluta**, ma può essere usata per capire come uno scheduler
(algoritmo di scelta)
deve comportarsi rispetto a queste due categorie di processi. Il SO deve quindi essere capace non solo di
identificare la natura di un processo, ma anche di capire le eventuali
evoluzioni nel tempo di esecuzione. In generale, col passare del tempo, i processi
tendono a diventare sempre più I/O bounded, perché le CPU diventano sempre più veloci e le

computazioni durano meno. Questo cambiamento ha un impatto importante sul comportamento
dello scheduler.



Strategia di scheduling

La strategia di scegliere **solo processi CPU bounded** ogni volta che ne è presente uno in coda
non è efficace, perché tenderebbe a bloccare le periferiche. È molto più efficiente
dare priorità
ai processi I/O bounded
, perché usano la CPU per poco tempo e tornano a fare I/O
(bloccandosi). Questo mantiene
attiva la CPU e anche il comparto I/O, massimizzando l'uso delle
risorse.
Con questa strategia, nella coda dei pronti rimarranno solo processi CPU bounded, ma nel
frattempo l'I/O viene sfruttato al massimo e la CPU è sempre occupata. In generale, mischiare
processi CPU bounded e I/O bounded è una strategia vincente per ottenere un sistema bilanciato
e performante.

Quando viene attivato lo scheduler

Lo scheduler entra in azione **ogni volta che la CPU può essere assegnata a un nuovo processo**.

In particolare, viene attivato nei seguenti casi:

- **Terminazione o creazione di un processo**
- **Chiamate bloccanti** (es. I/O) **e arrivo del relativo interrupt**
- **Interrupt periodici**, per esempio nei sistemi **preemptive** (con prelazione);

Collabora inoltre con il **dispatcher**, che è l'elemento incaricato di fare materialmente il cambio di
contesto (context switch).

Obiettivi dello scheduler

Gli **obiettivi di uno scheduler** variano in base al contesto in cui opera:

- **Nei sistemi batch**, dove i processi non sono interattivi:
  - Il **throughput** (numero di processi completati per unità di tempo) va **massimizzato**.
  - Il **tempo di turnaround va minimizzato**, ossia il tempo tra l'arrivo del processo nella coda
    dei pronti e la sua terminazione.
  - Il **tempo di attesa va minimizzato**, ossia parte del tempo di turnaround in cui il processo
    è in coda ma non usa la CPU. È la metrica più influenzata dall'algoritmo di scheduling.
- **Nei sistemi interattivi**, l'obiettivo principale è la **reattività**: il sistema deve rispondere
  rapidamente alle interazioni dell'utente. Quindi è fondamentale:
  - Minimizzare il **tempo di risposta** (tra input e primo output)
  - Privilegiare i **processi I/O bounded**, che usano la CPU per poco tempo e poi si bloccano
    in I/O, liberando velocemente la CPU per altri processi
- **Nei sistemi real-time**, lo scheduling deve essere **prevedibile** e rispettare **scadenze**

  temporali rigide. In questi casi si usano **algoritmi specializzati**, progettati per garantire il
  rispetto delle deadline.

In tutti i casi, un buon algoritmo dovrebbe essere anche **equo**, cioè evitare che alcuni processi
vengano continuamente trascurati. Nei sistemi
multicore, lo scheduler dovrebbe cercare di

bilanciare il carico tra i core, evitando che alcuni siano sovraccarichi mentre altri restano inattivi.