27-04-2023

Scambio di messaggi fra processi

A differenza degli altri metodi, questo metodo permette di mettere in comunicazione processi fra ***macchine diverse***.

Vengono usate **2 chiamate di sistema** e non sono costrutti del linguaggio di programmazione:

- **send** ->Invia un messaggio. `send(destinazione,messaggio)`
- **receive** -> Riceve un messaggio. `receive(sorgente,messaggio)`
  - La receive è attiva se deve riceve messaggi, altrimenti non lo è.

Vantaggio di questo approccio: permette di far **comunicare**/gestire **processi fra macchine diverse** (ovviamente collegate fra loro)

Svantaggio di questo approccio: rispetto ai semafori/monitor, vengono **usate chiamate di sistema**, quindi risultano essere meno efficienti proprio per questo motivo

*Il messaggio potrebbe ***perdersi nella rete durante la communicazione***.* In questo caso si potrebbe usare un meccanismo del tipo: quando un processo riceve un messaggio, si rimanda (all'indietro) un **messaggio di conferma**. Se il messaggio di conferma non è stato ricevuto, allora il messaggio viene rispedito.

Questa soluzione non è definitiva perchè ***si può perdere anche un messaggio di conferma.*** Bisogna distinguere un messaggio ***ORIGINALE*** da un messaggio ***RITRASMESSO*** tramite un *codice identificativo*

Metodi di indirizzamento (scambio messaggio)

Diretto -> Impostazione di un id ad un messaggio e usare questo per comunicare in maniera uniovoca.
Maibox -> si usa un buffer che memorizza un numero n limitato di messaggi, e vengono usati per invio/ricezione di questi **messaggi spediti ma non ricevuti**

- se la mailbox è **piena**, il **processo viene sospeso** finchè questo buffer non viene svuotato di almeno una posizione
- Rendezvous -> evita l'uso del buffer. Destinazione e Sorgente sono collegate insieme

Lo scambio di messaggi può essere in 2 modi:

- ***ASINCRONA***: Quando un processo invia un messaggio, **continua la sua esecuzione**.
  - In questo caso, il processo invia un messaggio, anche se il destinatario non l'ha ricevuto, esso continua ad andare avanti. Chi riceve il messaggio può non cambiare nulla o l'informazione che riceve contiene delle ***informazioni non valide*** rispetto a quando è stato inviato (*visto che il mittente è andato avanti con la sua esecuzione*)
- ***SINCRONA*** (meglio nota come rendezvous): Quando viene inviato un messaggio, il mittente **si sospende**
  - In questo caso il messaggio può essere inviato **SOLO** se il ricevente è pronto a riceverlo
  - Il messaggio spedito contiene informazioni che riguardano lo stato attuale del mittente sospeso

- Non vi è necessità della presenza di un buffer visto che i messaggi non si accumulano. Dopo la spedizione di un messaggio, il processo mittente si sospende e, quindi, ***non accumula messaggi da inviare***.

Problema produttore-consumatore con scambio di messaggi

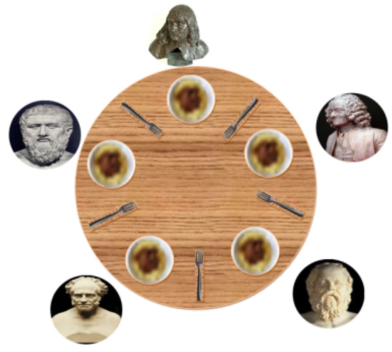
```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

Può capitare che uno dei due più veloce rispetto all'altro e quindi si potrebbero verificare letture/invii di messaggi vuoti.

Problema dei 5 filosofi a cena

RIguarda il problema della sincronizzazione:

- Si hanno 5 filosofi a tavola dove ci sono 5 piatti di spaghetti e ogni filosofo ne ha uno.
- Ognuno di loro hanno una forchetta e gli spaghetti sono scivolosi che richiedono due forchette per essere mangiati.
- Fra un piatto e un altro c'è solo una forchetta
- Quando un filosofo mangia, usa le 2 forchette ai lati e dopo aver finito le posa.
- Ogni filosofo si limita a pensare e mangiare



> *E' possibile scrivere un algoritmo per ogni filosofo in modo che non si blocchi mai?*

Prima soluzione

Per poter mangiare, un filosofo deve avere **l'uso esclusivo** di entrambe le forchette. Le risorse(forchette) sono limitate e il "prendere una forchetta" è in concorrenza con i filosofi che si hanno accanto.

```
int N=5
function philosopher(int i)
        think()
        take_fork(i) //prende la forchetta destra
        take_fork((i+1) mod N) //prende la forchetta sinistra
        eat() //mangia
        put_fork(i) //posa la forchetta
        put_fork((i+1) mod N)
```

Se tutti volessero mangiare allo stesso tempo:

- Se tutti prendono la forchetta a sinistra, quella a destra non c'è, allora posano la forchetta e tornano a pensare.
- Dopo un po' si ripete la situazione e nessuno mangia.
- Si ha uno stato di deadlock e nessuno mangia

Seconda soluzione

Si controlla se una forchetta è disponibile. Se non lo è viene rilasciata e si riprova in seguito

> *Le due soluzioni non sono corrette e portaono (o possono portare) a un deadlock.*

La prima soluzione (dove ognuno prende la forchetta sinistra allo stesso tempo per poi posarla e non mangiare) viene detta ***STARVATION***: i **programmi** vengono **eseguiti** indefinitamente **senza avanzare mai con il proprio stato**.

Terza soluzione

Ripetere l'ipotesi della prima soluzione, dove il problema era il fare l'azione nello stesso tempo, allora si inserisce un'attesa random fra le interazioni.

- un filosofo prende la forchetta sinistra, vede se c'è a destra e se non c'è la posa e aspetta un tempo random.
- Nella maggior parte dei casi un filosofo prende la forchetta in tempi diversi rispetto ad altri. Almeno un filosofo, teoricamente, dovrebbe prendere entrambe le forchette e quindi mangiare.
- C'è la **POSSIBILITA'** che un filosofo prenda 2 forchette e mangi ma non c'è **CERTEZZA**, quindi non è un approccio valido da poter usare, quindi serve una soluzioni che vada bene ***sempre***.

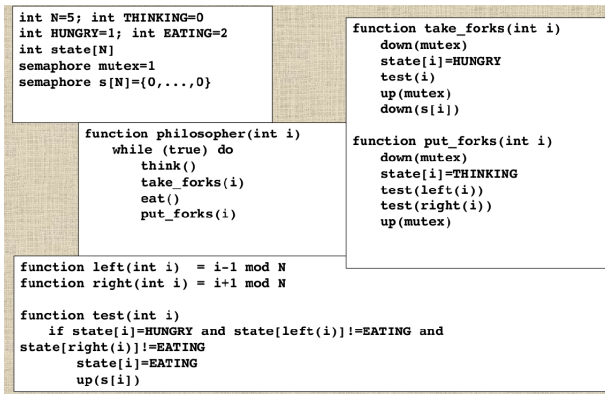
Quarta soluzione

Uso di un semaforo MUTEX:

- Un filosofo, prima di prende la forchetta, controlla un semaforo e in base al valore prende/non prende la forchetta
- Si garantisce il possesso delle forchette se sono libere e quindi può un filosofo può mangiare
- Dopo aver mangiato, il MUTEX torna allo stato iniziale

Questa soluzione non è del tutto efficiente:

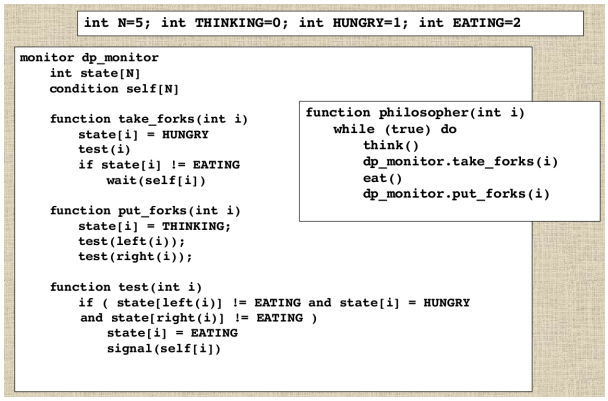
- Teoricamente può funzionare ma con l'uso del semaforo **solo 2 filosofi possono mangiare allo stesso tempo** quindi questa soluzione **NON E' OTTIMALE**



- Un filosofo può trovarsi in 3 stati: *Pensante*, *Ho Fame*, *Mangiando*
- Si usa un array che contiene lo stato dell'i-esimo filosofo e un vettore di semafori, uno per ogni filosofo

L'idea è: il filosofo è nel suo stato Pensante. Può passare allo stato Eating solo se il suo filosofo sinistro e destro non stanno mangiando

Problema dei 5 filosofi con i monitor



- Si fa uso di un array di contition che è grande quanto il numero dei filosofi

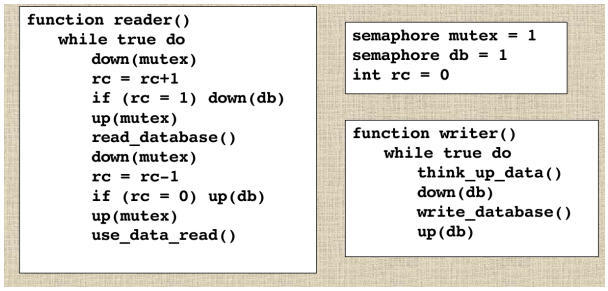
Problema dei lettori-scrittori

E' un altro **problema di sincronizzazione** che gestisce l'acesso all'interno di un database.
Quando si devono solo leggere dati dal database (*processi di lettura*), allora non ci sono problemi.
Quando qualcuno vuole scrivere sul database (*processi di scrittura*) allora possono nascere dei problemi.

- Se un processo sta scrivendo nel database, nessun'altro processo (lettura e scrittura) può accedere al database in quell'istante.
- Se ci sono processi di lettura, essi potrebbero leggere tutti allo stesso tempo.

Si può programmare un algoritmo che, quando c'è uno scrittore, nessun altro può acceder e se nessuno scritture sta scrivendo allora tutti i lettori possono leggere allo stesso tempo?

Soluzione con i semafori



```
function reader()
   while true do
      down(mutex)
      rc = rc+1
      if (rc = 1) down(db)
      up(mutex)
      read_database()
      down(mutex)
      rc = rc-1
      if (rc = 0) up(db)
      up(mutex)
      use_data_read()
```

```
semaphore mutex = 1
semaphore db = 1
int rc = 0
```

```
function writer()
   while true do
      think_up_data()
      down(db)
      write_database()
      up(db)
```

- `db` gestisce gli accessi al database
- `rc` (***variabile condivisa**, quindi modificata da tutti) si usa come contatore -> si contano il numero di processi che stanno leggendo o che vorrebbero leggere il database (*hanno fatto richiesta di lettura*)
- `mutex` gestisce gli accessi esclusivi alla variabile condivisa `rc`

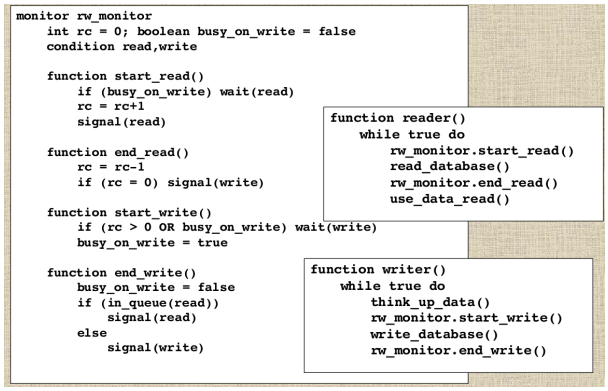
Problema

- Si sta consentendo a ***più lettori di accedere allo stesso tempo mentre lo scrittore deve aspettare che il lettore termini le sue operazioni***
  - Potrebbe arrivare un altro lettore e, visto questo codice, legge il contenuto del database. Man mano `rc` incrementa
- **All'arrivo di uno scrittore**, trova $db = 0$ e quindi non può accedere e va in sleep() e **aspetta che il lettore/i lettori escano dal database**. (potebbe attendere un tempo indefinito)
  - Quando $rc = 0$ allora nessun lettore sta leggendo il database né tantomeno ha espresso l'intenzione di farlo.

Alternativa

Lo scrittore potrebbe ***attendere solo i lettori che lo precedono*** e quindi **NON** qualsiasi lettore. In questo caso si ha lo svantaggio di avere **MINORI PRESTAZIONI**

Soluzione con monitor ma che presenta lo stesso problema: (**PRIORITA' AI LETTORI**)



```
monitor rw_monitor
    int rc = 0; boolean busy_on_write = false
    condition read,write

    function start_read()
        if (busy_on_write) wait(read)
        rc = rc+1
        signal(read)

    function end_read()
        rc = rc-1
        if (rc = 0) signal(write)

    function start_write()
        if (rc > 0 OR busy_on_write) wait(write)
        busy_on_write = true

    function end_write()
        busy_on_write = false
        if (in_queue(read))
            signal(read)
        else
            signal(write)
```

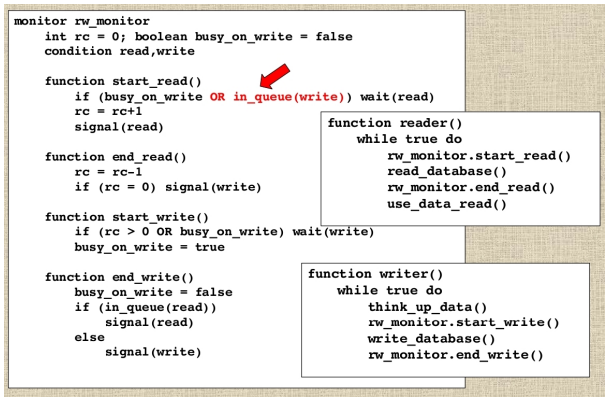
```
function reader()
    while true do
        rw_monitor.start_read()
        read_database()
        rw_monitor.end_read()
        use_data_read()
```

```
function writer()
    while true do
        think_up_data()
        rw_monitor.start_write()
        write_database()
        rw_monitor.end_write()
```

Anche in questo caso gli scrittori restano in attesa che tutti i lettori abbiano finito le proprie operazioni

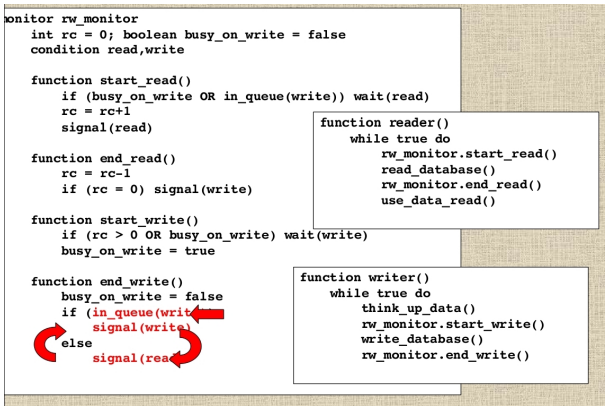
Soluzione che gestisce i successivi lettori che vengono messi in coda rispetto ad uno scrittore già presente. (**LO SCRITTORE RIMANE IN ATTESA SOLO DEI LETTORI CHE LO PRECEDONO**)
Un lettore si blocca quando il database è bloccato o quando c'è una coda di processi scrittori.

- Uno scrittore attende solo i lettori che lo precedono



Soluzione che accoda i lettori finchè c'è una coda di scrittori. (**I LETTORI ATTENDONO TUTTI I POSSIBILI SCRITTORI**)

- Quindi quando c'è una coda di ***scrittori che attendono***, allora i lettori attendono che essa si svuoti



- In questo caso si potrebbe avere una STARVATION sui lettori che attendono tutti i possibili scrittori (anche i successivi) e quindi, probabilmente, non verranno mai eseguiti

Scheduler dei processi

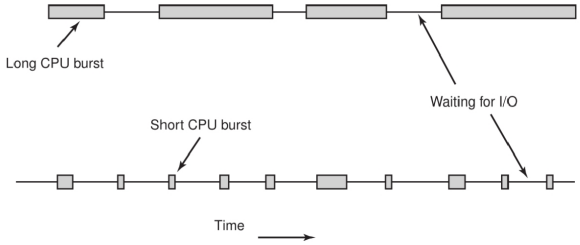
Gestire e determinare l'ordine di esecuzione dei processi.

- In un intervallo di tempo devono **avanzare più processi** e di questo lavoro se ne occupa lo ***SCHEDULER***.
- La tecnica che determina l'ordine esatto prende il nome di ***ALGORITMO DI SCHEDULING***.
- L'algoritmo di scheduling deve tener conto di alcune **caratteristiche**:
  - **Ottimizzazione le prestazioni** al fine di raggiungere l'obiettivo richiesto
  - **Uso efficiente** dell'unità di elaborazione, ovvero *non ci devono essere momenti nei quali la CPU non viene sfruttata*
  - Finalità della macchina: **INDUSTRIALE** (*eseguire più velocemente possibile le azioni richieste*), **INTERATTIVA** (*rispondere nel minor tempo possibile alle richieste dell'utente*) o sistemi **REAL-TIME** (*la risposta del processo deve essere istantanea*)

Ci sono diversi **tipologie di processi**:

- **CPU BOUNDED** -> la maggior parte dell'attività del processo è passata all'interno della CPU, quindi sono processi ***ATTIVI*** e fanno *richieste I/O molto raramente*
- **I/O BOUNDED** -> la maggior parte della loro attività la passano ***IN ATTESA*** di I/O per poi tornano allo stato precedente e consumano il loro tempo di esecuzione in un ***BREVE INTERVALLO***. Inoltre

fanno *richieste I/O molto frequentemente*



> *Dare **troppa priorità** a una tipologia di un processo oppure ad un altra tipologia di processo **non è la scelta migliore** da prendere per l'algoritmo di scheduling*.
> Invece si deve trovare un **corretto bilanciamento** fra questi due tipi di processi in modo da ottenere le prestazioni migliori.

Lo scheduler viene attivato per decidere quale processo deve essere eseguito successivamente, in particolare:

- alla **creazione/terminazione** di un processo (`fork()`)
- per la ricezione di **chiamate bloccanti** (*sezioni critiche*) e arrivo del relativo interrupt
- per il **blocco I/O**

Tipologie di sistemi in caso di **interrupt periodici**:

- **Preemptive**: lo scheduling **preleva, sospende (*di forza*)** l'esecuzione di un processo e lo sostituisce con un altro
- **Non-Preemptive**: lo scheduling lascia che il processo finisca le sue attività **senza prelevarlo forzatamente**

Un processo ha un `quanto di tempo` *assegnato* ed è sempre minore del tempo complessivo che il processo richiede per essere completato correttamente.

- L'algoritmo di assegnazione di tempo ad un processo è detto ***ALGORITMO PREEMPTIVE*** e tale processo è **obbligato a essere sospeso**. Esso hanno un **INTERRUPT DI CLOCK** che interrompono forzatamente il processo
- Un ***ALGORITMO NON-PREEMPTIVE*** è il viceversa del *PREEMPTIVE* e un processo non viene sospeso finchè non completa le sue operazioni

Il **DISPATCHER** è modulo che da il modulo della cpu al processo selezionato dallo scheduler. Si occupa del context_switch, del cambio fra esecuzione `usermode` o `kernelmode` e/o eseguire salti di riavvio di programma in `usermode`. Il Dispatcher deve essere molto veloce per ridurre i costi e i tempi.
La **latenza di dispatch** è l'intervallo di tempo del dispatcher per interrompere e successivamente avviare un nuovo processo

> - In un **contesto industriale NON** è necessario interrompere un processo ma in linea generale è importante che il processo completi la sua operazione e quindi si ***tenta di usare un algoritmo non-preemptive***
> - Viceversa accade in un **contesto interattivo**, invece, **può risultare interessante interrompere un processo** e, quindi, si tenta di usare un ***algoritmo di scheduling preemptive*** per permettere delle risposte istantanee all'utente che fa le richiest