

Problema dei lettori e scrittori

I due processi a girano su un DB.

Il DB è da lockare nei momenti opportuni.

Ci può essere tanti lettori contemporaneamente ma basta anche un solo scrittore per rendere i dati inconsistenti.

Bisogna gestire in qualche modo l'accesso al DB.

- Tanti lettori vanno bene, basta un solo scrittore per avere problemi.

Soluzione con semafori

```
function reader()  
1 while (true)  
2   down(mutex)  
3   rc++  
4   if (rc == 1) down(db)  
5   up(mutex)  
6   read-db()  
7   down(mutex)  
8   rc--  
9   if (rc == 0) up(db)  
10  up(mutex)
```

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

```
function writer()  
while (true)  
  think()  
  down(db)  
  write-db()  
  up(db)
```

La riga 4 serve per chiudere la porta ad altri scrittori mentre la riga 9 serve per aprirla.

Il writer con down(db) e up(db) chiude e apre la porta a tutti.

Il mutex serve per gestire la variabile condivisa rc.

Svantaggio: lo scrittore in coda potrebbe aspettare un tempo indeterminato per l'accesso.

È una soluzione bilanciata a favore dei lettori.

Il lettore deve avere una down (db) dentro a una
regione critica non fa comunque danni
Sarebbe che abbiamo uno scrittore, entra un
lettore che non ha problema il down (mutex) ma
si blocca nel down (db).

Il primo lettore si blocca nel down (mutex),
in questo modo gestiamo in parte la situazione.

Soluzione con monitor

monitor rw_monitor

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

function start_read()

if (busy_on_write) wait(read)

rc++

signal(read)

function end_read()

rc--

if (rc=0) signal(write)

function start_write()

~~busy_on_write = false~~

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)

reader()

while (true)

rw.start_read()

read_db()

rw.end_read()

writer()

while (true)

rw.start_write()

write_db()

rw.end_write()

La variabile `busy_on_write` ci serve a bloccare processi che vogliono entrare mentre sta avvenendo una scrittura nel DB.

Nella `end_read` se i lettori diventano 0 rimettiamo il `water` (se c'è).

Nella `start_read` abbiamo un `signal (read)` per segnalare in questa tutti i lettori che ci sono bloccati.

La `start_write` controlla se non ci sono lettori o scrittori, nel caso si mette in attesa.

La `end_write` imposta la variabile a `false` e controlla se nella coda c'è un lettore e lo avverte, se no avverte un'altra scrittura.

Questa soluzione privilegia ancora i lettori.

Soluzione 2

Modifichiamo la `start_read`: lì il diventa così: `if (busy_on_write OR in_queue(write)) wait(read)`.

Ora la `wait(read)` avviene anche se c'è in coda un `writer`.

Così evitiamo che il lettore lo sovalchi.
La soluzione ancora sbilanciata per i lettori a causa della procedura `end_write()` che "preferisce" i lettori.



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

```
function start_read()
    if (busy_on_write OR in_queue(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()
```

Soluzione 3.

Uguale alla 2 ma moltiplichiamo la cond_write.

- if (in_queue(write)) signal(write) else signal(read)

Questa soluzione contrariamente è molto
squilibrata a favore dei lettori degli array.

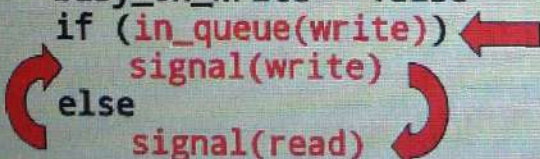

```
monitor rw_monitor
  int rc = 0; boolean busy_on_write = false
  condition read, write
```

```
function start_read()
  if (busy_on_write OR in_queue(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(write)) signal(write)
  else signal(read)
```



The diagram shows two red curved arrows forming a loop between the 'signal(write)' and 'signal(read)' lines in the 'end_write' function. A straight red arrow points from the 'in_queue(write)' condition to the 'signal(write)' line.

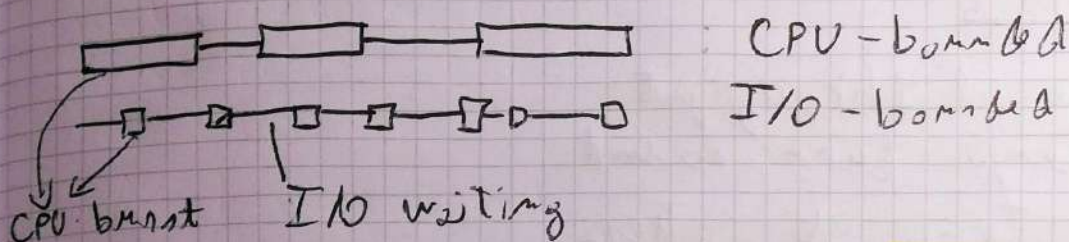
```
function reader()
  while true do
    rw_monitor.start_read()
    read_database()
    rw_monitor.end_read()
    use_data_read()
```

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


Scheduling

Per processi multiprogrammati ci sono appunto tanti processi che si "congelano" la CPU la parte del SO che si occupa della scelta del prossimo processo è lo scheduler attraverso l'algoritmo di scheduling.

Distinguiamo i processi in $\begin{cases} \text{CPU-bound} & \text{usano più CPU} \\ \text{I/O-bound} & \text{usano più I/O} \end{cases}$



Si tende a favorire i processi I/O bounded perché se si facesse il contrario la CPU verrebbe monopolizzata privilegiando l'I/O-bound per far lavorare bene entrambi i reparti: CPU e I/O.

Quando è usato lo scheduler?

- terminazione (e creazione) processi
- disparte bloccante e arrivo interrupt
- interrupt periodici (generatore non monopolizzazione CPU)
 - ↳ Sistemi non-preemptive (senza prelazione)
 - ↳ Sistemi preemptive (con prelazione)

Coi sistemi preemptive si evita che un processo monopolizzi la CPU.

Lo scheduler collabora col dispatcher che implementa la transizione tra il processo che rilascia e il processo scelto dallo scheduler.

Latency di dispatch: intervallo temporale tra inizio e fine lavoro di dispatching.

Obiettivi di un algoritmo di scheduling

Abbiamo 3 tipi di ambienti e di conseguenze 3 tipi di algoritmi.

• BATCH • INTERATTIVI • REAL-TIME

Obiettivi comuni

- equità nell'assegnazione della CPU
- bilanciamento nell'uso delle risorse

Obiettivi nei batch

- massimizzare il throughput
- minimizzare tempo turnaround
- minimizzare tempo di attesa

Obiettivi nei sistemi interattivi

- minimizzare il tempo di risposta

Obiettivi nei sistemi real-time

- rispetto scadenze
- prevedibilità

Scheduling nei Batch

(un batch lo vediamo come una insieme di Job)

• First-come First-served FCFS

ordine per ordine di arrivo

È un sistema non-preemptive che usa una coda FIFO

P	T
P ₁	25
P ₂	3
P ₃	3

$$t_{ma} = (0 + 25 + 28) / 3 = 17$$

$$t_{mc} = (25 + 28 + 30) / 3 = 27$$

• Shortest Job First SJF

ordine per brevità

È un sistema non-preemptive e presuppone di conoscere il tempo impiegato per ogni lavoro a priori.

E' ottimale solo se i lavori da svolgere sono tutti subito disponibili (Tempo di attesa nullo)

• Shortest Remaining Time Next

simile al SJF ma è preemptive per risolvere il problema legato ai tempi di attesa non nulli