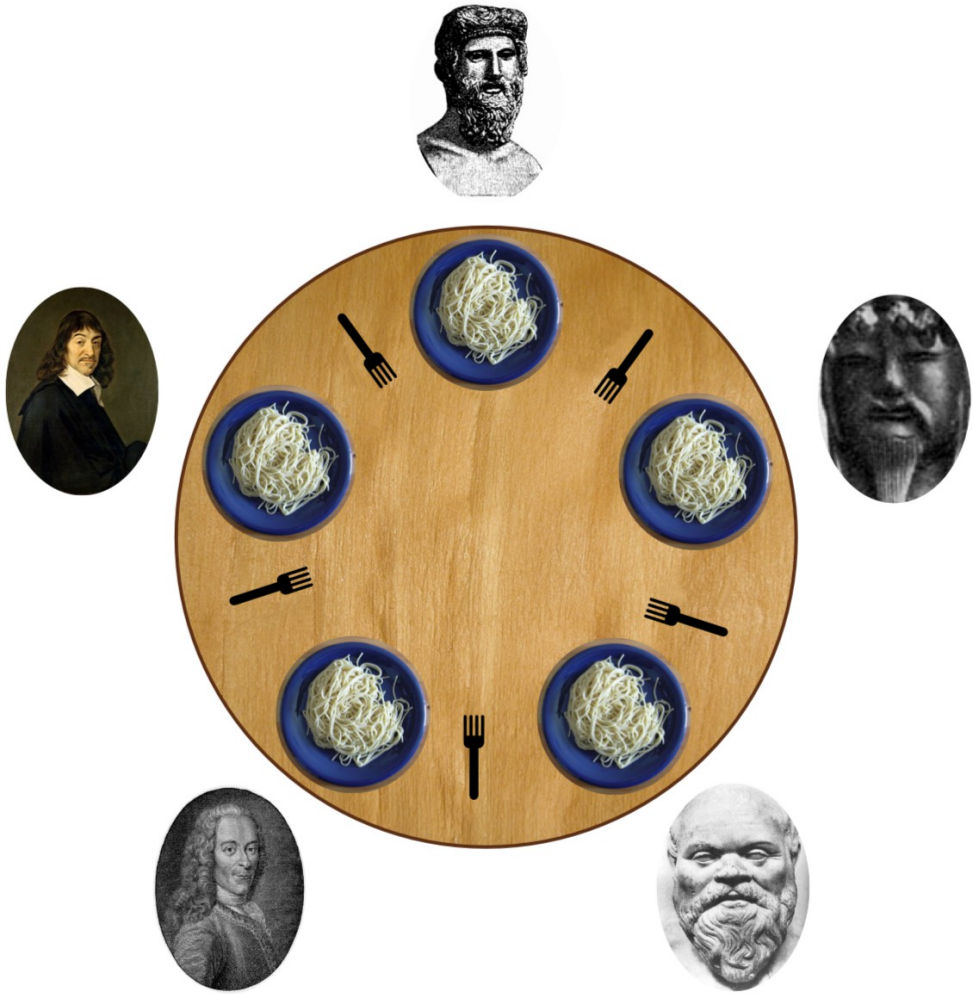
Sistemi Operativi: Problemi classici

Amos Brocco, Ricercatore, DTI / ISIN

Basato su:
[STA09] “Operating Systems: Internals and Design Principles”, 6/E, William Stallings, Prentice Hall, 2009
[TAN01] “Modern Operating Systems”, 2/E, Andrew S. Tanenbaum, Prentice Hall, 2001
[TAN09] “Modern Operating Systems”, 3/E, Andrew S. Tanenbaum, Prentice Hall, 2009

Il problema dei filosofi a cena



Il problema dei filosofi a cena

Il problema: la pasta è scivolosa e quindi a ogni filosofo, per mangiare servono due forchette.

I filosofi concordano di condividere le forchette: ogni forchetta è condivisa fra i due filosofi vicini (soluzione poco igienica, ma funzionale).

Il problema per l'informatico

Simulare il comportamento dei filosofi con un programma:

- Ogni filosofo rappresenta un thread
  - Ogni filosofo vuole il massimo di autonomia di comportamento, alternando periodi ragionevoli durante i quali pensa o mangia.
  - Se pensa all'infinito non pone problemi agli altri, ma muore di fame
  - Se mangia all'infinito fa morire di fame i vicini.

- Le forchette rappresentano le risorse condivise

Filosofi a cena: sviluppo della soluzione

```
thread filosofo(int i) {

  while(TRUE) {
        pensa (per un certo tempo),
        afferra la forchetta di sinistra,
        afferra la forchetta destra,
        mangia (per un certo tempo),
        depone la forchetta sinistra
        depone la forchetta destra
  }
}
```

Nota: I tempi (pensa e mangia) possono essere scelti in modo casuale

Filosofi a cena: sviluppo della soluzione

Idea: se ogni forchetta è una risorsa condivisa, associamo ad ogni forchetta un mutex

```
thread filosofo(int i) {

 while(TRUE) {
    pensa (per un certo tempo)
    blocca mutex(i)         // afferra forchetta sinistra
    blocca mutex[(i+1)%N]   // afferra forchetta destra
    mangia (per un certo tempo)
    sblocca mutex(i)        //depone forchetta sinistra
    sblocca mutex[(i+1)%N] //depone forchetta destra
 }
}
```

...una soluzione apparentemente corretta... oppure no?

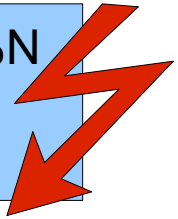
Filosofi a cena: sviluppo della soluzione

Idea: se ogni forchetta è una risorsa condivisa, associamo ad ogni forchetta un mutex

```
thread filosofo(int i) {

 while(TRUE) {
    pensa (per un certo tempo)
    blocca mutex(i)         // afferra forchetta sinistra
    blocca mutex[(i+1)%N]   // afferra forchetta destra
    mangia (per un certo tempo)
    sblocca mutex(i)        //depone forchetta sinistra
    sblocca mutex[(i+1)%N] //depone forchetta destra
 }
}
```

Se il filosofo i blocca il mutex i e non riesce a bloccare il mutex (i+1)%N poiché nel frattempo è stato bloccato dal filosofo (i+1)%N
e questo si ripete per tutti i filosofi si ha un deadlock!



Situazioni critiche

- Se tutti i 5 filosofi afferrano nello stesso istante la forchetta alla loro sinistra (situazione poco probabile ma possibile su un sistema multiprocessore), si genera un deadlock.

- Se un filosofo afferra la forchetta di sinistra e deve aspettare all'infinito la forchetta di destra, fa morire di fame il collega a sinistra.

Filosofi a cena: sviluppo della soluzione

Come si comporta il filosofo se una forchetta è occupata ?

- **Aspetta**
  - Può produrre dei dead lock
- **Rinuncia e torna a pensare**
  - Può fare morire di fame un filosofo (starvation)

Filosofi a cena: sviluppo della soluzione

Cerchiamo una soluzione ottimale che garantisca la massima indipendenza fra i filosofi, e che eviti deadlock e starvation!

Osservazioni:

- Tutti possono pensare contemporaneamente
- Con 5 filosofi, e 5 forchette, fino a 2 filosofi possono mangiare contemporaneamente

Filosofi a cena: sviluppo della soluzione

```
thread filosofo(int i) {

  while(TRUE) {
        pensa (per un certo tempo),
        afferra le forchette,
        mangia (per un certo tempo),
        depone le forchette.
  }
}
```

Raggruppiamo l'acquisizione e il rilascio delle forchette in due funzioni

Filosofi a cena: sviluppo della soluzione

La procedura per afferrare le forchette deve essere migliorata:

- **può servire un semaforo (per ogni filosofo)?**

```
procedura afferra_forchette(i) { // i si riferisce al filosofo
    test(i) // verifica se  le due forchette sono libere
            // (senza  bloccarsi all'interno di una sezione
            //  critica)

    aspetta semaforo(i)  //  se è necessario aspettare, l'attesa avviene
                         //  fuori dalla sezione critica.
                         //  il semaforo sarà disponibile quando i
                         //  filosofi vicini avranno deposto le forchette
                         //  che interessano al filosofo i
}
```

Il semaforo dovrà essere incrementato solo se l'interessato sta aspettando, altrimenti c'è il rischio che il semaforo venga incrementato inutilmente!

Filosofi a cena: sviluppo della soluzione

Come testare se le forchette dei vicini sono libere ?

- Le forchette si possono prendere se i vicini non stanno mangiando
  - Quando un filosofo mangia deve mettersi nello **stato=MANGIA**

```
#define sinistro  (i+ N-1)%N
#define destro    (i+1)%N

procedura test(i) {
          Se  stato(sinistro) != MANGIA e  stato(destro)  != MANGIA) {
               stato(i) = MANGIA // se non mangiano i vicini,mangio io
               incrementa semaforo(i)
           }
}
```

Filosofi a cena: sviluppo della soluzione

Come faccio a capire se i vicini sono in attesa anche loro di qualche forchetta?

- Un filosofo vuole prendere le forchette se è affamato, e deve segnalarlo mettendosi nello **stato=AFFAMATO**

```
procedura afferra_forchette(i) {
    stato(i)=AFFAMATO          // Segnala che è affamato
    test(i)                    // verifica se le due forchette sono libere
    aspetta semaforo(i)        // aspetta se non ha le forchette
}
```

Filosofi a cena: sviluppo della soluzione

La stessa funzione di test potrà essere usata anche dai vicini quando deporranno le forchette per incrementare il semaforo.

- Siccome il semaforo *i* si può incrementare a condizione che il filosofo *i* sia **AFFAMATO**, è necessario completare il test come segue:

```
procedura test(i) {
          Se stato(i) = AFFAMATO
          e stato(sinistro) != MANGIA
          e  stato(destro)  != MANGIA {
              stato(i) = MANGIA
              incrementa semaforo(i) // Questo permetterà di non aspettare
                                     // in afferra_forchette
         }
}
```

Filosofi a cena: sviluppo della soluzione

Esistono momenti dove ogni filosofo è in uno stato diverso da "AFFAMATO" o "MANGIA" , cioè quando pensa: introduciamo perciò anche lo stato "PENSA"

- Questo stato viene assunto quando il filosofo depone le forchette
- La funzione depone_forchette deve preoccuparsi di eventualmente aprire i semafori dei vicini. Questo si realizza chiamando la stessa funzione **test().**

```
procedura depone_forchette(i) {
        stato(i) = PENSA
        test(sinistra) // Il vicino a sinistra mangia, se era affamato e ha le forchette
        test(destra) // Il vicino a destra mangia, se era affamato e ha le forchette
}
```

Filosofi a cena: sviluppo della soluzione

```
thread filosofo(int i) {

 while(TRUE) {
    pensa (per un certo tempo)
    afferra_forchette(i)
    mangia (per un certo tempo)
    depone_forchette(i)
 }
}
```

Abbiamo introdotto altre variabili condivise oltre alle forchette, come gli stati... non manca qualcosa?

Filosofi a cena: sviluppo della soluzione

```
procedura depone_forchette(i) {
      blocca mutex
      stato(i) = PENSA
      test(sinistra)   // i test non si
                       // bloccano, nessun
                       // pericolo di deadlock
      test(destra)
      sblocca mutex
}
```

Durante il test le variabili non devono essere modificate da altri

Filosofi a cena: sviluppo della soluzione

```
procedura afferra_forchette(i) {
    blocca mutex
    stato(i)=AFFAMATO
    test(i)
    sblocca mutex
    aspetta semaforo(i)  // l'attesa, se
                         // necessaria, avviene
                         // fuori dalla sezione
                         // critica, nessun
                         // pericolo di deadlock
}
```

Filosofi a cena: sviluppo della soluzione

- Le varabile **stato[N]** sono lette e scritte da una sola funzione per volta
- Le due funzioni non possono mai bloccarsi

La soluzione completa

```
#define N 5
#define SINISTRA (i+N-1)%N
#define DESTRA   (i+1)%N
#define PENSA    0
#define AFFAMATO 1
#define MANGIA   2

#define up(s) sem_post(s)
#define down(s) sem_wait(s)

int stato[N];
pthread_mutex_t mutex; // Il mutex deve essere sbloccato
all'inizio
pthread_sem_t semaforo_filosofo[N];
```

La soluzione completa

```
void filosofo (int i)
{
    while (TRUE) {
        pensa();
        afferra_forchette(i);
        mangia();
        deponi_forchette(i);
    }
}
```

La soluzione completa

```
void afferra_forchette(int i)
{
    pthread_mutex_lock(&mutex);
    stato[i] = AFFAMATO;
    test(i);
    pthread_mutex_unlock(&mutex);
    down(&semaforo_filosofo[i]);
}

void deponi_forchette(i)
{
    pthread_mutex_lock(&mutex);
    stato[i] = PENSA;
    test(SINISTRA);
    test(DESTRA);
```

La soluzione completa

```
void test(i)
{
  if (stato[i] == AFFAMATO
      && stato[SINISTRA] != MANGIA
      && stato[DESTRA] != MANGIA) {
    stato[i] = MANGIA;
    up(&semaforo_filosofo[i]);
  }
}
```

Il monitor

Concetto informatico introdotto C.A.R.Hoare (1974) e P.B.Hansen (1975)

- **synchronized** in Java

Il *monitor* è un insieme di

- Variabili
- Procedure
- Sequenza di inizializzazione (dei dati)

Le variabili sono accessibili solo tramite le procedure del monitor

Si dice che una *thread* (o un processo) **entra** nel monitor, quando invoca una sua procedura

Solo una *thread* (un processo) per volta può entrare nel *monitor*

Una procedura può invocare altre procedure del monitor non direttamente accessibile alle *thread*

Cinque filosofi e un monitor...

- Nel nostro esempio, appartengono al monitor
  - Le variabili stato(i)
  - Le funzioni afferra_forchette(i) e deponi_forchette(i)
- Ma non la funzione test(i), visto che non è invocata direttamente dalle thread, ma dalle funzioni del monitor!

```
while(TRUE) {
    pensa
    afferra_forchette(i) // funzione del monitor
    aspetta semaforo(i)
    mangia
    depone_forchette(i) // funzione del monitor
}
```

Lettori e scrittori

- Problema che riproduce una situazione in cui ci sono più processi concorrenti che cercano di leggere e scrivere in una memoria (es. database)
  - Più processi di lettura (lettori) possono tranquillamente accedere in maniera concorrente
  - Se un processo di scrittura (scrittore) sta modificando i dati, nessun altro processo deve poter accedere, nemmeno i lettori (che potrebbero leggere dei dati inconsistenti)

Lettori e scrittori: soluzione favorevole ai lettori

- Più processi di lettura (lettori) possono tranquillamente accedere in maniera concorrente
  - *il primo lettore blocca l'accesso agli scrittori*
  - *l'ultimo lettore sblocca l'accesso agli scrittori*
- Se un processo di scrittura (scrittore) sta modificando i dati, nessun altro processo deve poter accedere, nemmeno i lettori (che potrebbero leggere dei dati inconsistenti)

Lettori e scrittori: soluzione favorevole ai lettori

```
semaphore nlm = 1; /* Protegge numlet */
semaphore db = 1; /* Protegge dati */
int numlet = 0;

thread lettore(void)
{
    while (TRUE) {
        down(&nlm);
        numlet = numlet + 1;
        if (numlet == 1)
down(&db);
        up(&nlm);
        leggi();
        down(&nlm);
        numlet = numlet - 1;
        if (numlet == 0) up(&db);
        up(&nlm);
        processa_dati();
    }
}
```

```
thread scrittore(void)
{
    while (TRUE) {
        genera_dati();
        down(&db);
        scrivi();
        up(&db);
    }
}
```

Lettori e scrittori: soluzione favorevole ai lettori

```
semaphore nlm = 1;
semaphore db = 1;
int numlet = 0;

thread lettore(void)
{
    while (TRUE) {
        down(&nlm);
        numlet = numlet + 1;
        if (numlet == 1)
                down(&db);
        up(&nlm);
        leggi();
        down(&nlm);
        numlet = numlet - 1;
        if (numlet == 0) up(&db);
        up(&nlm);
        processa_dati();
    }
}
```

Il primo lettore blocca i dati in modo da non lasciar entrare uno scrittore

L'ultimo lettore sblocca i dati in modo da lasciar entrare uno scrittore (se era in attesa)

Lettori e scrittori: soluzione favorevole ai lettori

Lo scrittore deve aspettare che tutti i lettori abbiano finito! Poi blocca l'accesso a altri thread (lettori / scrittori)

```
thread scrittore(void)
{
    while (TRUE) {
        genera_dati();
        down(&db);
        scrivi();
        up(&db);
    }
}
```

Favorire i lettori

- Se ci sono sempre lettori, lo scrittore potrebbe non aver mai l'occasione di scrivere:
  - **starvation**!
- Per evitare questa situazione potremmo modificare il programma come segue:
  - quando un lettore arriva e c'è uno scrittore in attesa, il lettore deve aspettare dietro lo scrittore invece di permettergli di entrare subito a leggere
    - “Concurrent Control with "Readers" and "Writers" P.J. Courtois,* F. H, 1971”

Lettori e scrittori: costruzione di una soluzione favorevole agli scrittori

- Più processi di lettura (lettori) possono tranquillamente accedere in maniera concorrente
- Se un processo di scrittura (scrittore) sta modificando i dati, nessun altro processo deve poter accedere, nemmeno i lettori (che potrebbero leggere dei dati inconsistenti)
  - *il primo scrittore blocca l'accesso ai lettori*
  - *l'ultimo scrittore sblocca l'accesso ai lettori*

Lettori e scrittori: costruzione di una soluzione favorevole agli scrittori

```
semaphore fs = 1; /* Flag scrittori: down quando entra primo scrittore
*/
semaphore nsm = 1;
```

```
thread scrittore(void)
{
    while (TRUE) {
        genera_dati();
        down(&db);
        scrivi();
        up(&db);
    }
}
```

```
down(&nsm);
numscr = numscr + 1;
if (numscr == 1) {
   down(&fs);
}
up(&nsm);
```

```
down(&nsm);
numscr = numscr - 1;
if (numscr == 0) {
   up(&fs);
}
up(&nsm);
```

"quando un lettore arriva e c'è uno scrittore in attesa, il lettore deve aspettare"

Lettori e scrittori: costruzione di una soluzione favorevole agli scrittori

```
semaphore nlm = 1;
semaphore db = 1;
int numlet = 0;

thread lettore(void)
{
    while (TRUE) {
        down(&nlm);
        numlet = numlet + 1;
        if (numlet == 1)
down(&db);
        up(&nlm);
        leggi();
        down(&nlm);
        numlet = numlet - 1;
        if (numlet == 0) up(&db);
        up(&nlm);
        processa_dati();
    }
}
```

down(&fs);

up(&fs);

Nota: sblocco subito il semaforo fs per permettere la lettura concorrente di più lettori

"quando un lettore arriva e c'è uno scrittore in attesa, il lettore deve aspettare"

Lettori e scrittori: costruzione di una soluzione favorevole agli scrittori

```
semaphore nlm, nsm = 1;
semaphore fs = 1;
semaphore db = 1;
int numlet = 0;
int numscr = 0;

thread lettore(void)
{
    while (TRUE) {
         down(&fs);
         down(&nlm);
        numlet = numlet + 1;
        if (numlet == 1) down(&db);
        up(&nlm);
        up(&fs);

        leggi();

        down(&nlm);
        numlet = numlet - 1;
        if (numlet == 0) up(&db);
        up(&nlm);
        processa_dati();
    }
}
```

```
thread scrittore(void)
{
    while (TRUE) {
        genera_dati();
         down(&nsm);
         numscr = numscr + 1;
         if (numscr == 1) down(&fs);
         up(&nsm);
        down(&db);
        scrivi();
        up(&db);

         down(&nsm);
         numscr = numscr - 1;
         if (numscr == 0) up(&fs);
         up(&nsm);
    }
}
```

... manca ancora qualcosa: se sia dei lettori che degli scrittori sono in attesa su down(&fs), non viene data la precedenza agli scrittori!

Lettori e scrittori: soluzione favorevole agli scrittori

```
semaphore mutex, nlm, msm = 1;
semaphore fs = 1; /* Flag scrittori */
semaphore db = 1;
int numlet = 0;
int numscr = 0;

thread lettore(void)
{
    while (TRUE) {
        down(&mutex);
         down(&fs);
        down(&nlm);
        numlet = numlet + 1;
        if (numlet == 1) down(&db);
        up(&nlm);
        up(&fs);
        up(&mutex);

        leggi();

        down(&nlm);
        numlet = numlet - 1;
        if (numlet == 0) up(&db);
        up(&nlm);
        processa_dati();
    }
}
```

```
thread scrittore(void)
{
    while (TRUE) {
        genera_dati();
         down(&nsm);
        numscr = numscr + 1;
        if (numscr == 1) down(&fs);
        up(&nsm);
        down(&db);
        scrivi();
        up(&db);

        down(&nsm);
        numscr = numscr - 1;
        if (numscr == 0) up(&fs);
        up(&nsm);
    }
}
```

Con mutex non posso avere più lettori in attesa su down(&fs), quindi quando faccio up(&fs) se sblocco qualcuno in attesa può trattarsi solo di uno scrittore

Lettori e scrittori: soluzione favorevole agli scrittori

```
semaphore mutex, nlm, msm = 1;
semaphore fs = 1; /* Flag scrittori */
semaphore db = 1;
int numlet = 0;
int numscr = 0;

thread lettore(void)
{
    while (TRUE) {
        down(&mutex);
         down(&fs);
         down(&nlm);
        numlet = numlet + 1;
        if (numlet == 1) down(&db);
        up(&nlm);
        up(&fs);
        up(&mutex);

        leggi();

        down(&nlm);
        numlet = numlet - 1;
        if (numlet == 0) up(&db);
        up(&nlm);
        processa_dati();
    }
}
```

Vogliamo garantire che ci sia al massimo un lettore in attesa su '&l'

Se non ci sono scrittori qui non devo aspettare, altrimenti loro hanno la precedenza

Mi metto in coda, se sono il primo impedisco agli scrittori di continuare

Lascio passare il prossimo

Se sono l'ultimo lettore, gli scrittori possono passare

Lettori e scrittori: soluzione favorevole agli scrittori

Mi metto in coda, se sono il primo impedisco ai lettori di mettersi in coda (aspetteranno prima)

Aspetto la possibilità di scrivere (i.e. che esca il lettore corrente), poi posso scrivere

Ho finito, sveglio un lettore che era in attesa

```
thread scrittore(void)
{
    while (TRUE) {
        genera_dati();
        down(&nsm);
        numscr = numscr + 1;
        if (numscr == 1) down(&fs);
        up(&nsm);
        down(&db);
           scrivi();
        up(&db);

        down(&nsm);
        numscr = numscr - 1;
        if (numscr == 0) up(&fs);
        up(&nsm);
    }
}
```

Favorire gli scrittori

- Se ci sono sempre scrittori, un lettore potrebbe non aver mai l'occasione di leggere:
  - **starvation**!

Lettori e scrittori: soluzione equa

- Più processi di lettura (lettori) possono tranquillamente accedere in maniera concorrente
  - *il primo lettore blocca l'accesso agli scrittori*
  - *l'ultimo lettore sblocca l'accesso agli scrittori*
- Se un processo di scrittura (scrittore) sta modificando i dati, nessun altro processo deve poter accedere, nemmeno i lettori (che potrebbero leggere dei dati inconsistenti)

- Viene stabilito l'ordine di arrivo
  - se arriva uno scrittore, aspetta finché tutti i lettori correntemente in esecuzione (non in attesa!) finiscono, e poi ha la precedenza

Soluzione equa

```
semaphore nlm, msm = 1;
semaphore ordine = 1;
semaphore db = 1;
int numlet = 0;

thread lettore(void)
{
    while (TRUE) {
         down(&ordine);
        down(&nlm);
        numlet = numlet + 1;
        if (numlet == 1) down(&db);
        up(&ordine);
        up(&nlm);

        leggi();

        down(&nlm);
        numlet = numlet - 1;
        if (numlet == 0) up(&db);
        up(&nlm);
        processa_dati();
    }
}
```

```
thread scrittore(void)
{
    while (TRUE) {
        genera_dati();
         down(&ordine);
        down(&db);
        up(&ordine);
        scrivi();
        up(&db);
    }
}
```

Presupponiamo che i thread in attesa sul semaforo ordine vengano risvegliati nell'ordine corretto

Soluzione equa

```
semaphore nlm, msm = 1;
semaphore ordine = 1;
semaphore db = 1;
int numlet = 0;

thread lettore(void)
{
    while (TRUE) {
         down(&ordine);
        down(&nlm);
        numlet = numlet + 1;
        if (numlet == 1) down(&db);
        up(&ordine);
        up(&nlm);

        leggi();

        down(&nlm);
        numlet = numlet - 1;
        if (numlet == 0) up(&db);
        up(&nlm);
        processa_dati();
    }
}
```

Mi metto in coda

Sono fuori dalla coda, pronto per leggere

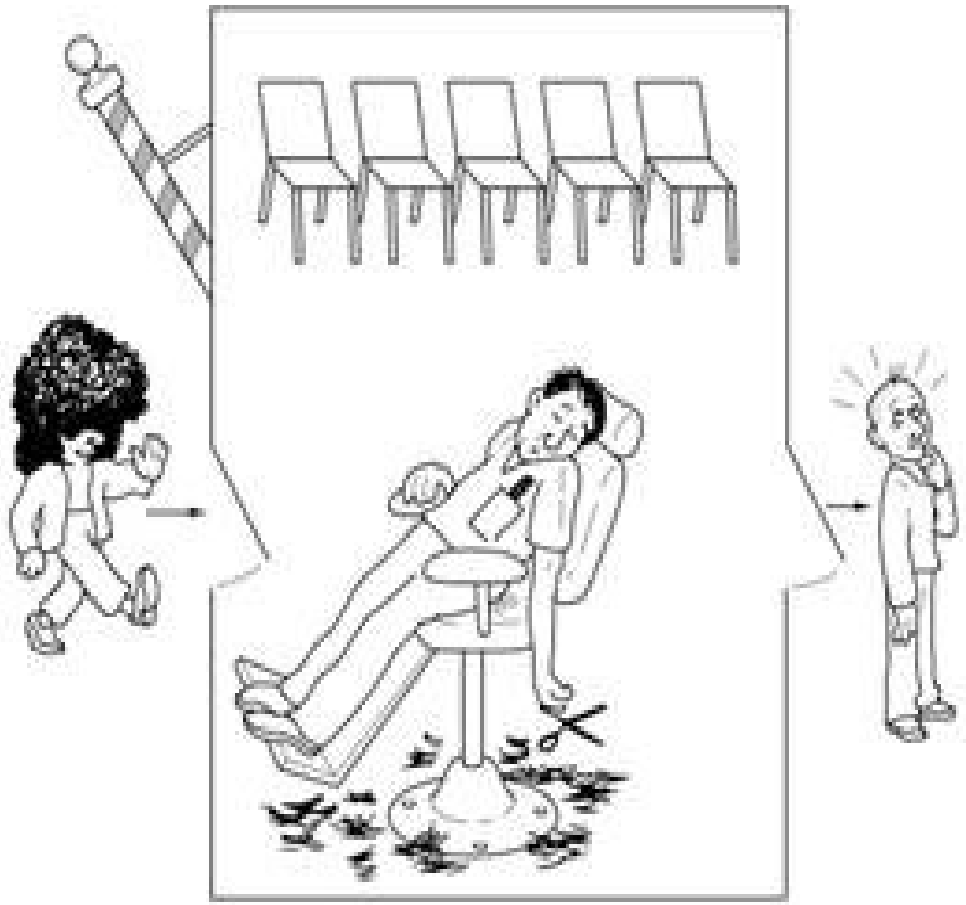
Il semaforo 'db' blocca gli scrittori fintanto che ci sono lettori in esecuzione...

Soluzione equa


```
thread scrittore(void)
{
    while (TRUE) {
        genera_dati();
        down(&ordine);
        down(&db);
        up(&ordine);
        scrivi();
        up(&db);
    }
}
```

Appena uno scrittore viene risvegliato si mette in attesa su db... finché non ottiene l'accesso ogni altro thread dovrà aspettare su 'ordine': i lettori quindi non potranno più aumentare

Il problema del barbiere sonnolento (Sleeping Barber, E.Dijkstra – 1965)



da Modern Operating Systems, 2nd Ed., A. Tanenbaum

Il problema del barbiere sonnolento (Sleeping Barber, E.Dijkstra – 1965)

- Il barbiere ha una sedia per tagliare i capelli e per dormire e una sala d'aspetto con N sedie per i clienti.
- Il barbiere, quando ha terminato un servizio, guarda nella sala d'aspetto se ci sono altri clienti. Se sì, fa accomodare il prossimo sulla sua sedia, altrimenti si mette lui a dormire.
- Il cliente, quando arriva, guarda cosa sta facendo il barbiere. Se dorme, lo sveglia. Se sta già lavorando, va nella sala d'aspetto e, se la sala non è piena, attende il suo turno, altrimenti se ne va.

Il barbiere sonnolento: soluzione ingenua

```
semaforo_clienti = 0
semaforo_barbiere = 0
stato = dorme

thread barbiere {

  while (TRUE) {
    if (clienti > 0) {
       stato=lavora
     /* Fai accomodare cliente */
       up(semaforo_clienti)
       clienti=clienti - 1
       esegue servizio
    }
    else {
       stato=dorme
       down(semaforo_barbiere)
    }
  }
}

thread cliente {

  if (stato==dorme)/* Sveglia! */
     up(semaforo_barbiere)
  if (clienti < N) {
     /* Si mette in coda */
     clienti = clienti+1
     down(semaforo_clienti)
     riceve servizio
   }
   else
      se ne va
}
```

Il barbiere sonnolento: soluzione ingenua

- Attenzione ai **deadlock** !
  - Il barbiere guarda nella sala d'aspetto. Non c'è nessuno (clienti=0), va sulla sua sedia e dorme (stato=dorme).
  - Dopo aver visto "clienti=0" e prima di porre stato=dorme, arriva un cliente che vede il barbiere ancora stato=lavora. Va nella sala sala d'aspetto e attende l'incremento del semaforo_clienti, che non avverrà, a meno che non arrivi un altro cliente.

Il barbiere sonnolento: soluzione corretta

```
thread barbiere {

  while (TRUE) {
    blocca mutex
    if (clienti > 0) {
       stato=lavora
       up(semaforo_clienti)
       clienti=clienti - 1
       sblocca mutex
       esegue servizio
    }
    else {
       stato=dorme
       sblocca mutex
       down(semaforo_barbiere)
    }
  }
}
```

```
semaforo_clienti = 0
semaforo_barbiere = 0
stato = dorme

thread cliente {
 blocca  mutex
 if (stato==dorme)
     up(semaforo_barbiere)
  if (clienti < N) {
     clienti = clienti+1
     sblocca  mutex
     down(semaforo_clienti)
     riceve servizio
   }
   else {
      sblocca mutex
      se ne va
   }
}
```