

Problemi di sincronizzazione

- problema dei cinque filosofi
- problema dei lettori e degli scrittori
- problema del barbiere sonnolento

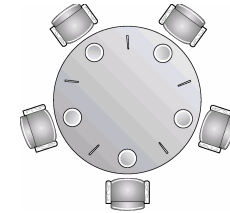
7bis. Esercitazione su sincronizzazione di processi

1

marco lapegna

Problema dei cinque filosofi

- **Cinque filosofi** passano la vita pensando e mangiando, attorno ad una tavola rotonda. Al centro della tavola vi è una zuppiera di riso e la tavola è apparecchiata con **cinque bacchette**.
- Quando un filosofo pensa non interagisce con i colleghi. Quando gli viene fame tenta di impossessarsi delle **bacchette che stanno alla sua destra ed alla sua sinistra**. Il filosofo può prendere una sola bacchetta per volta.
- Quando un filosofo affamato possiede due bacchette contemporaneamente, mangia. Terminato il pasto, appoggia le bacchette e ricomincia a pensare.



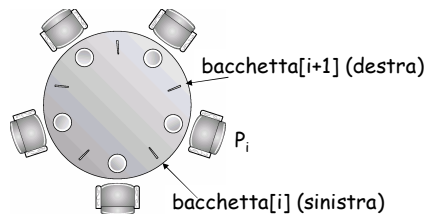
7bis. Esercitazione su sincronizzazione di processi

2

marco lapegna

Problema dei cinque filosofi

- Si può rappresentare **ciascuna bacchetta con un semaforo**.
- Ogni filosofo tenta di **afferrare** una bacchetta con un'operazione di **wait** e la **rilascia** eseguendo **signal**



7bis. Esercitazione su sincronizzazione di processi

3

marco lapegna

Problema dei cinque filosofi : soluzione 1

- Variabili condivise:
`semaphore bacchetta[5]; // tutti gli elementi inizializzati a 1`
- Filosofo i :

Sezione critica

```
do {  
    wait(bacchetta[ i ]) ← Prende bacch. sinistra  
    wait(bacchetta[(i+1) % 5]) ← Prende bacch. destra  
    ...  
    mangia  
    ...  
    signal(bacchetta[ i ]) ← lascia bacch. sinistra  
    signal(bacchetta[(i+1) % 5]) ← lascia bacch. destra  
    ...  
    pensa  
    ...  
} while (1);
```

7bis. Esercitazione su sincronizzazione di processi

4

marco lapegna

Soluzione 1: DEADLOCK

se tutti i filosofi hanno fame contemporaneamente e prendono la bacchetta alla loro sinistra **nessuno puo' prendere la bacchetta alla sua destra** (DEADLOCK)

Possibile soluzione

Ogni filosofo, dopo aver preso la prima forchetta, verifica se la seconda e' disponibile. In caso contrario posa anche la prima

soluzione 2:

```
do {
    while(not.entrambe){
        prendi( i );
        if (b[(i+1) % 5] > 0) {
            prendi( (i+1) % 5 );
            entrambe = true
        } else{
            posa( i );
        }
        ...
        mangia
        ...
        posa( i );
        posa( (i+1) % 5 );
        entrambe = false;
        ...
        pensa
        ...
    } while (1);
}
```

```
void prendi(int i){
    wait( &mutex );
    wait( &bacchetta[ i ] )
    b[ i ]--;
    signal( &mutex);
}

void posa(int i){
    wait( &mutex );
    b[ i ]++;
    signal( &bacchetta[ i ] );
    signal( &mutex );
}
```

b[5] array di interi condiviso che "replica" il contenuto di bacchetta[5]

soluzione 2: STARVATION

c'e il rischio che tutti i filosofi prendono la bacchetta sinistra, **vedono che la destra e' occupata e posano di nuovo la sinistra.** (STARVATION)

Possibile soluzione

Ogni filosofo, prima di mangiare si assicura di aver preso entrambe le bacchette

Deadlock e starvation

Deadlock: insieme di **processi bloccati** (su una istruzione wait), ognuno dei quali in attesa che si libera una risorsa assegnata in uso esclusivo di un altro processo

Starvation: insieme di **processi in esecuzione** che tentano ripetutamente senza successo di accedere ad una risorsa assegnata in uso esclusivo di un altro processo

Soluzione 3: INEFFICIENTE

```
do {
    ...
    pensa
    ...
    wait(&mutex);
    wait(bacchetta[ i ])
    wait(bacchetta[(i+1) % 5])
    ...
    mangia
    ...
    signal(bacchetta[ i ]);
    signal(bacchetta[(i+1) % 5]);
    signal(&mutex);
    ...
    pensa
    ...
} while (1);
```

Proteggere le
"istruzioni critiche"



1 solo filosofo alla
volta puo' mangiare!!

Soluzione 4

```
#define N 5 /* number of philosophers */
#define LEFT (i+N-1)%N /* number of i's left neighbor */
#define RIGHT (i+1)%N /* number of i's right neighbor */
#define THINKING 0 /* philosopher is thinking */
#define HUNGRY 1 /* philosopher is trying to get forks */
#define EATING 2 /* philosopher is eating */
typedef int semaphore; /* semaphores are a special kind of int */
int state[N]; /* array to keep track of everyone's state */
semaphore mutex = 1; /* mutual exclusion for critical regions */
semaphore s[N]; /* one semaphore per philosopher */

void philosopher(int i) /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) { /* repeat forever */
        think(); /* philosopher is thinking */
        take_forks(i); /* acquire two forks or block */
        eat(); /* yum-yum, spaghetti */
        put_forks(i); /* put both forks back on table */
    }
}
```

Soluzione 3 (cont.): OK

```
void test(int i) {
    if(state[i]==HUNGRY && state[LEFT]=EATING && state[RIGHT]=EATING){
        state[i] = EATING; // comunico agli altri che inizio a mangiare
        signal( &s[i] ); // posso mangiare: semaforo +1
    }
}

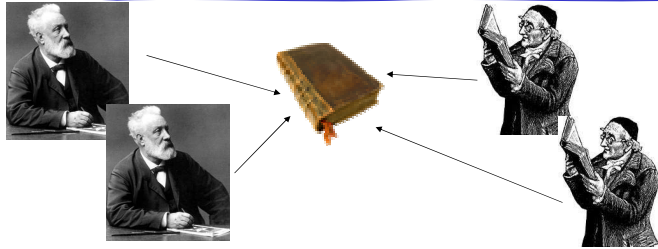
void take_forks(int i) {
    wait( &mutex );
    state[ i ]=HUNGRY; // sto per prendere le forchette
    test(i);
    signal( &mutex );
    wait( &s[ i ] ); // inizio a mangiare.
}

void put_forks(int i) {
    wait( &mutex );
    state[ i ] = THINKING; // comunico che ho finito di mangiare
    test(LEFT); // semaforo a sinistra +1
    test(RIGHT); // semaforo a destra +1
    signal( &mutex );
}
```

osservazioni

- $state[i]=0$ $i=0,\dots,4$ $s[i]=0$ $i=0,\dots,4$
- Il semaforo e' relativo al filosofo e non alla forchetta
- La funzione test vede se i vicini stanno mangiando. In caso contrario dichiara che inizia a mangiare e alza il semaforo (abilita il filosofo a mangiare)
- Un filosofo in attesa sulla wait in take_forks, sara' sbloccato da un filosofo vicino che esegue test in put_forks

Problema dei lettori e degli scrittori



- Un **insieme di dati** (ad es. un file) deve essere **condiviso** da più processi concorrenti che possono richiedere la **sola lettura** del contenuto, o **anche un aggiornamento**.
- Se **due lettori** accedono contemporaneamente ai dati condivisi non ha luogo **alcun effetto negativo**
- Se **uno scrittore** accede simultaneamente ad un altro processo si può avere **incoerenza dell'informazione**

2 tipi di problemi

- **1° problema dei lettori-scrittori**: nessun processo lettore deve attendere, salvo che uno scrittore abbia già ottenuto l'accesso ai dati condivisi (precedenza ai lettori)



possibilità di starvation per gli scrittori

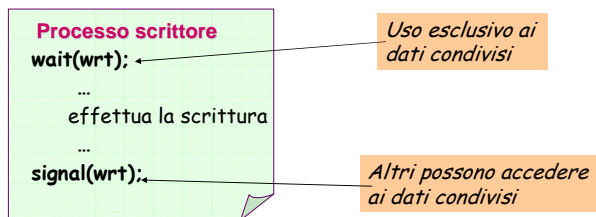
- **2° problema dei lettori-scrittori**: nessun processo scrittore deve attendere, salvo che uno scrittore abbia già ottenuto l'accesso ai dati condivisi (precedenza agli scrittori)



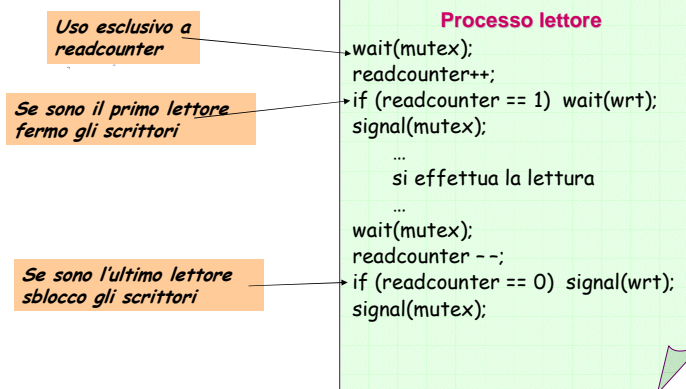
possibilità di starvation per gli lettori.

soluzione al primo problema scrittori/lettori

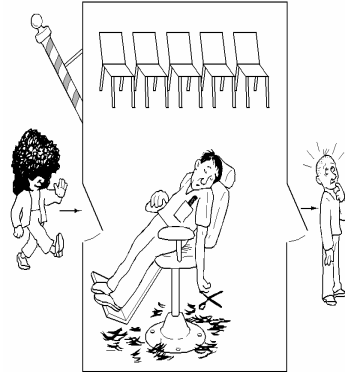
- Variabili condivise:
 - semaphore **mutex, wrt**;
 - int **readcounter**;
 - // inizialmente **mutex = 1, wrt = 1, readcounter = 0**



soluzione al primo problema scrittori/lettori



il barbiere sonnolento



in un negozio di barbiere ci sono

- una sedia per lavorare
- n sedie per far attendere i clienti
- il barbiere di solito dorme
- quando arriva un cliente, questi lo sveglia e si fa servire
- se nel frattempo arrivano altri clienti, si accomodano sulle sedie oppure vanno via se sono tutte occupate

7bis. Esercitazione su sincronizzazione di processi

17

marco lapegna

soluzione

```
#define CHAIRS 5 /* # chairs for waiting customers */
typedef int semaphore; /* use your imagination */

semaphore customers = 0; /* # of customers waiting for service */
semaphore barbers = 0; /* # of barbers waiting for customers */
semaphore mutex = 1; /* for mutual exclusion */
int waiting = 0; /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        wait (&customers); /* go to sleep if # of customers is 0 */
        wait (&mutex); /* acquire access to 'waiting' */
        waiting = waiting - 1; /* decrement count of waiting customers */
        signal (&barbers); /* one barber is now ready to cut hair */
        signal (&mutex); /* release 'waiting' */
        cut_hair(); /* cut hair (outside critical region) */
    }
}

void customer(void)
{
    wait (&mutex); /* enter critical region */
    if (waiting < CHAIRS) { /* if there are no free chairs, leave */
        waiting = waiting + 1; /* increment count of waiting customers */
        signal (&customers); /* wake up barber if necessary */
        signal (&mutex); /* release access to 'waiting' */
        wait (&barbers); /* go to sleep if # of free barbers is 0 */
        get_haircut(); /* be seated and be serviced */
    } else {
        signal (&mutex); /* shop is full; do not wait */
    }
}
```

7bis. Esercitaz

marco lapegna