

Meccanismi di sincronizzazione

Scambio di messaggi

1

Scambio di messaggi

- Comunicazione non basata su memoria condivisa con controllo di accesso.
- Basato su due primitive (chiamate di sistema o funzioni di libreria)
 - `send(destinazione, messaggio)`: spedisce un messaggio ad una certa destinazione; solitamente non bloccante.
 - `receive(sorgente, &messaggio)`: riceve un messaggio da una sorgente; solitamente bloccante (fino a che il messaggio non è disponibile).
- Meccanismo più astratto e generale della memoria condivisa e semafori
- Si presta ad una implementazione su macchine distribuite

2

Problematiche dello scambio di messaggi

- Affidabilità: i canali possono essere inaffidabili (es: reti). Bisogna implementare appositi protocolli fault-tolerant (basati su acknowledgment e timestamping).
- Autenticazione: come autenticare i due partner?
- Sicurezza: i canali utilizzati possono essere intercettati
- Efficienza: se prende luogo sulla stessa macchina, il passaggio di messaggi è sempre più lento della memoria condivisa e semafori.

3

Produttore-consumatore con scambio di messaggi

- Comunicazione *asincrona*
 - I messaggi spediti ma non ancora consumati vengono automaticamente bufferizzati in una *mailbox* (mantenuto in kernel o dalle librerie)
 - L'oggetto delle `send` e `receive` sono le mailbox
 - La `send` si blocca se la mailbox è piena
 - La `receive` si blocca se la mailbox è vuota.

4

```

#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 */
    }
}

```

Prod.-cons. con scambio di messaggi asincrono

- Il consumatore manda N messaggi vuoti (che verranno allocati in un buffer di sistema)
- Ogni qualvolta il produttore produce un item, prende un messaggio vuoto spedito dal consumatore, lo riempie e lo manda indietro
- Il numero di elementi del "buffer" rimane costante (=N)
- Il produttore si blocca (assumiamo che la receive sia bloccante) se non ci sono messaggi vuoti (il buffer per la comunicazione Prod-Cons è pieno)
- Il consumatore si blocca se non arrivano messaggi dal consumatore (il buffer per la comunicazione è vuoto)

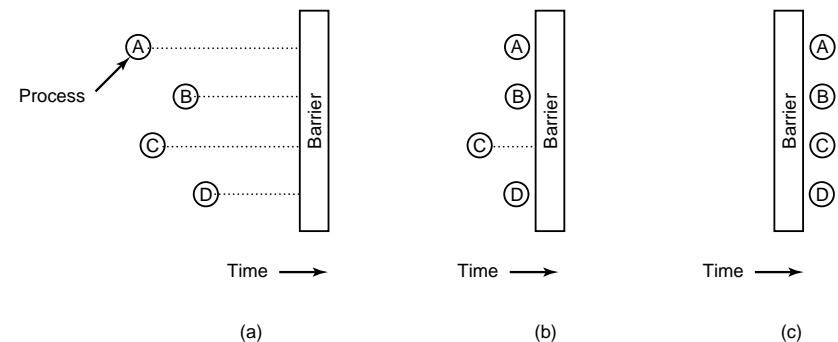
5

Meccanismi di sincronizzazione

Barriere

Barriere

- Meccanismo di sincronizzazione per *gruppi* di processi, specialmente per calcolo parallelo a memoria condivisa (es. SMP, NUMA)
 - Ogni processo alla fine della sua computazione, chiama la funzione *barrier* e si sospende.
 - Quando tutti i processi hanno raggiunto la barriera, la superano *tutti assieme* (si sbloccano).



6

7

I Grandi Classici

Esempi paradigmatici di programmazione concorrente. Presi come testbed per ogni primitiva di programmazione e comunicazione.

(E buoni esempi didattici!)

- Produttore-Consumatore a buffer limitato (già visto)
- I Filosofi a Cena
- Lettori-Scrittori
- Il Barbiere che Dorme

8

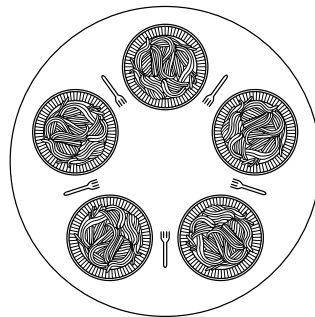
Problemi classici della concorrenza

9

I Filosofi a Cena (Dijkstra, 1965)

n filosofi seduti attorno ad un tavolo rotondo con n piatti di spaghetti e n forchette (bastoncini). (nell'esempio, $n = 5$)

- Mentre pensa, un filosofo non interagisce con nessuno
- Quando gli viene fame, cerca di prendere le forchette più vicine, una alla volta.
- Quando ha due forchette, un filosofo mangia senza fermarsi.
- Terminato il pasto, lascia le forchette e torna a pensare.



Problema: programmare i filosofi in modo da garantire

- assenza di deadlock: non si verificano mai blocchi
- assenza di starvation: un filosofo che vuole mangiare, prima o poi mangia.

10

I Filosofi a Cena—Una non-soluzione

```
#define N 5                                /* number of philosophers */

void philosopher(int i)                    /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think();                          /* philosopher is thinking */
        take_fork(i);                      /* take left fork */
        take_fork((i+1) % N);              /* take right fork; % is modulo operator */
        eat();                             /* yum-yum, spaghetti */
        put_fork(i);                       /* put left fork back on the table */
        put_fork((i+1) % N);               /* put right fork back on the table */
    }
}
```

Possibilità di deadlock: se tutti i processi prendono contemporaneamente la forchetta alla loro sinistra...

11

I Filosofi a Cena—Tentativi di correzione

- Come prima, ma controllare se la forchetta dx è disponibile prima di prelevarla, altrimenti rilasciare la forchetta sx e riprovare daccapo.
 - Non c'è deadlock, ma possibilità di starvation.
- Come sopra, ma introdurre un ritardo casuale prima della ripetizione del tentativo.
 - Non c'è deadlock, la possibilità di starvation viene ridotta ma non azzerata. Applicato in molti protocolli di accesso (CSMA/CD, es. Ethernet). Inadatto in situazione mission-critical o real-time.

12

I Filosofi a Cena—Soluzioni

- Introdurre un semaforo `mutex` per proteggere la sezione critica (dalla prima `take_fork` all'ultima `put_fork`):
 - Funziona, ma solo un filosofo per volta può mangiare, mentre in teoria $\lfloor n/2 \rfloor$ possono mangiare contemporaneamente.
- Tenere traccia dell'*intenzione* di un filosofo di mangiare. Un filosofo ha tre stati (THINKING, HUNGRY, EATING), mantenuto in un vettore `state`. Un filosofo può entrare nello stato EATING solo se è HUNGRY e i vicini non sono EATING.
 - Funziona, e consente il massimo parallelismo.

13

```
#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] = {0,...,0}; /* 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();             /* eat your spaghetti */
        put_forks(i);     /* put both forks back on table */
    }
}
```

```
void take_forks(int i)    /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);         /* enter critical region */
    state[i] = HUNGRY;    /* record fact that philosopher i is hungry */
    test(i);              /* try to acquire 2 forks */
    up(&mutex);           /* exit critical region */
    down(&s[i]);           /* block if forks were not acquired */
}

void put_forks(i)         /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);         /* enter critical region */
    state[i] = THINKING; /* philosopher has finished eating */
    test(LEFT);          /* see if left neighbor can now eat */
    test(RIGHT);         /* see if right neighbor can now eat */
    up(&mutex);           /* exit critical region */
}

void test(i)              /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

I Classici: Lettori-Scrittori

Un insieme di dati (es. un file, un database, dei record), deve essere condiviso da processi *lettori* e *scrittori*

- Due o più lettori possono accedere contemporaneamente ai dati
- Ogni scrittore deve accedere ai dati in modo esclusivo.

Implementazione con i semafori:

- Tenere conto dei lettori in una variabile condivisa, e fino a che ci sono lettori, gli scrittori non possono accedere.
- Dà maggiore priorità ai lettori che agli scrittori.

14

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

/* use your imagination */
/* controls access to 'rc' */
/* controls access to the database */
/* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {
        /* repeat forever */
        down(&mutex);
        /* get exclusive access to 'rc' */
        /* one reader more now */
        rc = rc + 1;
        /* if this is the first reader ... */
        if (rc == 1) down(&db);
        /* release exclusive access to 'rc' */
        up(&mutex);
        /* access the data */
        read_data_base();
        /* get exclusive access to 'rc' */
        down(&mutex);
        /* one reader fewer now */
        rc = rc - 1;
        /* if this is the last reader ... */
        if (rc == 0) up(&db);
        /* release exclusive access to 'rc' */
        up(&mutex);
        /* noncritical region */
        use_data_read();
    }
}

void writer(void)
{
    while (TRUE) {
        /* repeat forever */
        /* noncritical region */
        think_up_data();
        /* get exclusive access */
        down(&db);
        /* update the data */
        write_data_base();
        /* release exclusive access */
        up(&db);
    }
}
```

I Classici: Il Barbiere che Dorme

In un negozio c'è un solo barbiere, una sedia da barbiere e n sedie per l'attesa.

- Quando non ci sono clienti, il barbiere dorme sulla sedia.
- Quando arriva un cliente, questo sveglia il barbiere se sta dormendo.
- Se la sedia è libera e ci sono clienti, il barbiere fa sedere un cliente e lo serve.
- Se un cliente arriva e il barbiere sta già servendo un cliente, si siede su una sedia di attesa se ce ne sono di libere, altrimenti se ne va.



Problema: programmare il barbiere e i clienti filosofi in modo da garantire assenza di deadlock e di starvation.

15

Il Barbiere—Soluzione

- Tre semafori:
 - **customers**: i clienti in attesa (contati anche da una variabile *waiting*)
 - **barbers**: conta i barbieri in attesa (0 o 1)
 - **mutex**: per mutua esclusione
- Il barbiere esegue una procedura che lo blocca se non ci sono clienti; quando si sveglia, serve un cliente e ripete.
- Ogni cliente prima di entrare nel negozio controlla se ci sono sedie libere; altrimenti se ne va.
- Un cliente, quando entra nel negozio, sveglia il barbiere se sta dormendo.

16

```

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

```

```

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

```