Cooperazione tra Processi

Ivan Scagnetto

Università di Udine — Facoltà di Scienze MM.FF.NN.

A.A. 2007-2008

Mutex

- I mutex sono semafori con due soli possibili valori: bloccato o non bloccato.
- Utili per implementare mutua esclusione, sincronizzazione.
- Due sono le primitive: mutex_lock e mutex_unlock.
- Le primitive sono semplici da implementare, anche in user space. Esempio:

```
mutex_lock:
        TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
        CMP REGISTER,#0             | was mutex zero?
        JZE ok                      | if it was zero, mutex was unlocked, so return
        CALL thread_yield           | mutex is busy; schedule another thread
        JMP mutex_lock              | try again later
ok:     RET | return to caller; critical region entered


mutex_unlock:
        MOVE MUTEX,#0               | store a 0 in mutex
        RET | return to caller
```

Memoria condivisa?

Implementare queste funzioni richiede una forma di memoria condivisa.

- A livello kernel: strutture come quelle usate dai semafori possono essere mantenute nel kernel, e quindi accessibili da tutti i processi (tramite le apposite system call).
- A livello utente:
  - all'interno dello stesso processo: adatto per i thread,
  - tra processi diversi: spesso i S.O. offrono la possibilità di condividere segmenti di memoria tra processi distinti (shared memory),
  - alla peggio: file su disco.

Deadlock con Semafori

- Deadlock (stallo): due o più processi sono in attesa indefinita di eventi che possono essere causati solo dai processi stessi in attesa.
- L'uso dei semafori può portare a deadlock. Esempio: siano S e Q due semafori inizializzati a 1,

P_0	P_1
$down(S)$;	$down(Q)$;
$down(Q)$;	$down(S)$;
⋮	⋮
$up(S)$;	$up(Q)$;
$up(Q)$;	$up(S)$;

- Programmare con i semafori è molto delicato e prono ad errori, difficilissimi da debuggare. Come in assembler, solo peggio, perché qui gli errori sono race condition e malfunzionamenti non riproducibili.

Monitor

- Un monitor è un tipo di dato astratto che fornisce funzionalità di mutua esclusione:
  - collezione di dati privati e funzioni/procedure per accedervi,
  - i processi possono chiamare le procedure ma non accedere alle variabili locali,
  - un solo processo alla volta può eseguire codice di un monitor.
- Il programmatore raccoglie quindi i dati condivisi e tutte le sezioni critiche relative in un monitor; questo risolve il problema della mutua esclusione.
- Possono essere implementati dal compilatore con dei costrutti per mutua esclusione (ad esempio, inserendo automaticamente `lock_mutex` e `unlock_mutex` all'inizio e fine di ogni procedura).

Monitor - esempio

Esempio della struttura di un monitor per il problema produttore consumatore:

- i dati i e c sono privati (possono essere modificati solo dalle procedure producer() e consumer(),
- un solo processo per volta può eseguire il codice delle procedure producer() e consumer().

```
monitor example
    integer i;
    condition c;

    procedure producer( );
    .
    .
    .
    end;

    procedure consumer( );
    .
    .
    .
    end;
end monitor;
```

Monitor: Controllo del flusso di esecuzione

Per sospendere e riprendere i processi, ci sono le variabili condition, simili agli eventi, con le relative operazioni

- wait(c): il processo che la esegue si blocca sulla condizione c;
- signal(c): uno dei processi in attesa su c viene risvegliato. A questo punto, chi va in esecuzione nel monitor? Due varianti:
  - chi esegue la signal(c) si sospende automaticamente (monitor di Hoare),
  - la signal(c) deve essere l'ultima istruzione di una procedura (così il processo lascia il monitor) (monitor di Brinch-Hansen),
  - i processi risvegliati possono provare ad entrare nel monitor solo dopo che il processo attuale lo ha lasciato.

  Il successivo processo ad entrare viene scelto dallo scheduler di sistema.
- I signal su una condizione senza processi in attesa vengono persi.

Produttore-consumatore con monitor

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count – 1;
        if count = N – 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

Monitor (cont.)

- I monitor semplificano molto la gestione della mutua esclusione (vi sono meno possibilità di errori).
- Sono dei veri costrutti, non funzioni di libreria $\Rightarrow$ bisogna modificare i compilatori.
- Implementati (in certe misure) in veri linguaggi.
  Esempio: i metodi `synchronized` di Java.
  - Solo un metodo `synchronized` di una classe può essere eseguito alla volta.
  - Java non ha variabili condition, ma ha wait and notify (+ o – come sleep e wakeup).
- Un problema che rimane (sia con i monitor che con i semafori): è necessario avere memoria condivisa$\Rightarrow$ questi costrutti non sono applicabili a sistemi distribuiti (reti di calcolatori) senza memoria fisica condivisa.

Passaggio 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.

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 avviene all'interno della stessa macchina, il passaggio di messaggi è sempre più lento della memoria condivisa e semafori.

Produttore-consumatore con scambio di messaggi

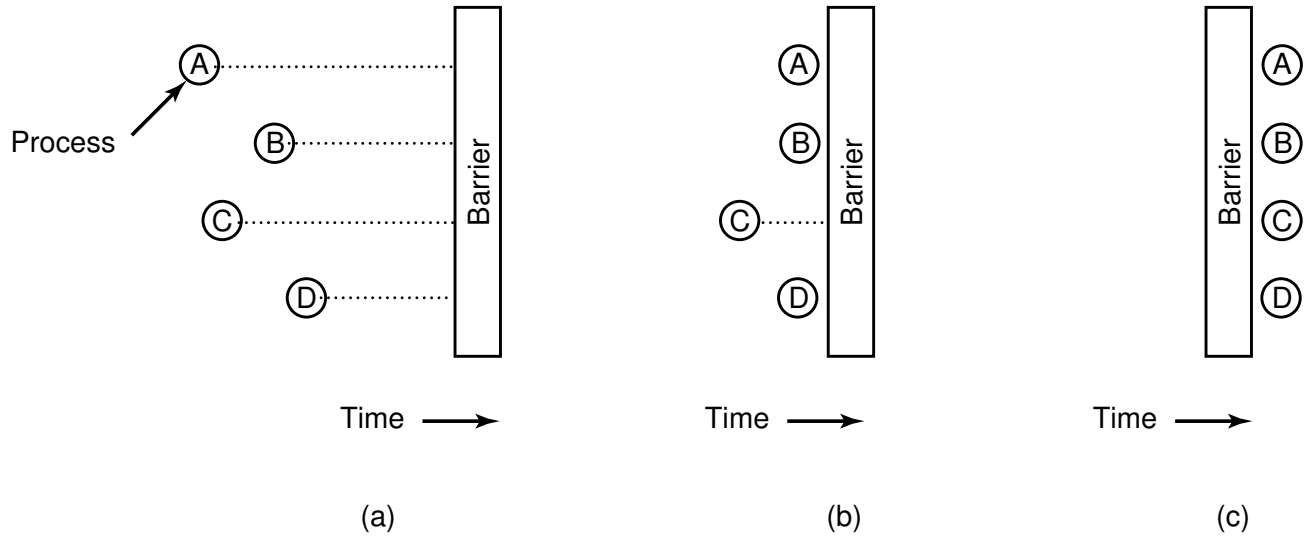
- Comunicazione asincrona:
  - I messaggi spediti ma non ancora consumati vengono automaticamente bufferizzati in una mailbox (mantenuta in spazio kernel o dalle librerie).
  - L'oggetto delle `send` e `receive` sono le mailbox.
  - La `send` si blocca se la mailbox è piena; la `receive` si blocca se la mailbox è vuota.
- Comunicazione sincrona:
  - I messaggi vengono spediti direttamente al processo destinazione.
  - L'oggetto delle `send` e `receive` sono i processi.
  - Le `send` e `receive` si bloccano fino a che la controparte non esegue la chiamata duale (rendez-vous).

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 */
     }
}
```

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).



I Grandi Classici

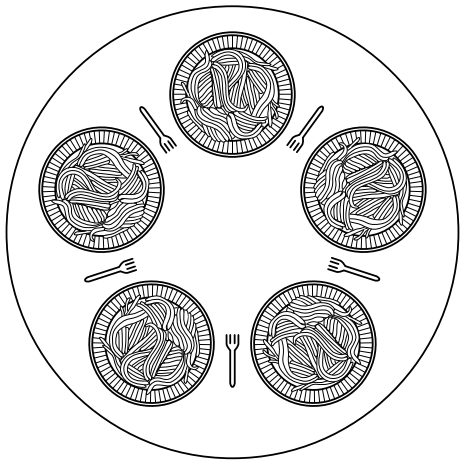
Esempi paradigmatici di programmazione concorrente. Presi come testbed per ogni primitiva di programmazione e comunicazione.

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

I Classici: 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 bacchette più vicine, una alla volta.
- Quando ha due bacchette, un filosofo mangia senza fermarsi.
- Terminato il pasto, lascia le bacchette 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.

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. . .

I Filosofi a Cena—Tentativi di correzione

- La strategia è analoga a quella precedente, ma stavolta si controlla se la forchetta dx è disponibile prima di prelevarla, altrimenti si rilascia la forchetta sx e si riprova daccapo.
  - Non c'è deadlock, ma possibilità di starvation.
- Si opera come nei casi precedenti, ma si introduce 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 situazioni mission-critical o real-time.

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), mantenuti 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.

I Filosofi a Cena—Soluzioni

```
#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 */
    }
}
```

I Filosofi a Cena—Soluzioni

```
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.

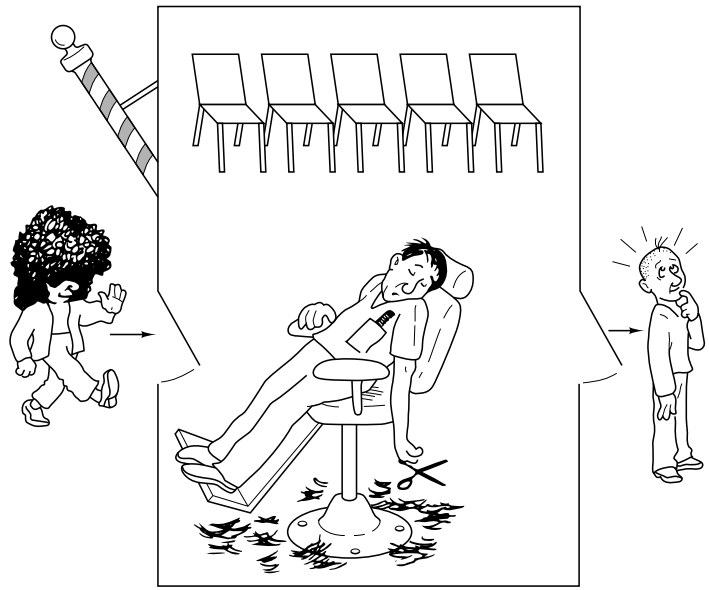
I Classici: Lettori-Scrittori

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

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


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

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.

Il Barbiere—Soluzione

- Tre semafori:
  - `customers`: i clienti in attesa (contati anche da una variabile `waiting`),
  - `barbers`: conta i barbieri in attesa,
  - `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.

Il Barbiere—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) {
        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) */
    }
}
```

Il Barbiere—Soluzione

```
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 */
        }
}
```