Azzolini Riccardo 2019-11-06

Sincronizzazione

1 Operazione indivisibile

Definizione: Un'**operazione indivisibile** su un dato condiviso d è un'operazione che è con certezza eseguita in modo *non* concorrente rispetto ad altre operazioni su d.

Un'operazione indivisibile viene talvolta detta **atomica**.

Per definizione, le operazioni indivisibili su d non possono dar luogo a race condition su d.

2 Istruzione test and set

L'istruzione **test and set** (`TS`) ha come operando una locazione di memoria (corrispondente, ad esempio, a una variabile), ed esegue due azioni:

1. controlla se tale locazione di memoria ha valore 0, ponendo il risultato nel bit CC (Condition Code) della PSW;
2. imposta la locazione di memoria a una sequenza di 1, indipendentemente dall'esito del controllo.

`TS` è un'**istruzione indivisibile** (un'istruzione che implementa un'operazione indivisibile) sulla locazione di memoria:

- su un'architettura uniprocessore, gli interrupt non possono interrompere una singola istruzione (perché si aspetta che l'istruzione corrente termini prima di gestirli);
- su un'architettura multiprocessore, bisogna evitare che vengano eseguite contemporaneamente più `TS`, altrimenti potrebbero trovare tutte la locazione di memoria a 0: a tale scopo, durante l'esecuzione di una `TS`, viene impedito alle altre CPU di accedere alla locazione di memoria (ad esempio, disabilitando gli accessi al bus della RAM).

Quando viene eseguita una `TS` su una determinata locazione di memoria, siccome tale istruzione è indivisibile e imposta sempre il valore della locazione a una sequenza di 1, è garantito che le esecuzioni successive della `TS` sulla stessa locazione troveranno un valore diverso da 0, cioè che i test successivi falliranno.

2.1 Implementazione delle sezioni critiche

L'istruzione `TS` consente di implementare correttamente le sezioni critiche, mediante variabili di lock condivise, in quanto permette di eseguire in modo indivisibile il controllo dello stato di una variabile lock e l'assegnamento a tale variabile. Questa soluzione, però, è direttamente disponibile *solo se si programma in linguaggio macchina*.

Il meccanismo, in pseudo-codice, è:

```
entry_test  if (lock == closed) { goto entry_test; }
            lock = closed;
            {CS}
            lock = open;
```

Ad esempio, l'implementazione per IBM/370, con la convenzione `open = 0`, è:

```
LOCK        DC X '00'
ENTRY_TEST  TS LOCK
            BC 7, ENTRY_TEST
            {CS}
            MVI LOCK, X'00'
```

1. `LOCK` è una variabile inizializzata con il valore esadecimale 0 (`X '00'`, dove il prefisso `X` indica appunto un valore esadecimale), che rappresenta lo stato "aperto".
2. Si esegue la test and set sulla variabile `LOCK`: il risultato del test è memorizzato impostando i 3 bit del Condition Code a `111` se lo stato di `LOCK` era "chiuso".
3. L'istruzione di salto condizionato `BC` salta a `ENTRY_SET` se i 3 bit del CC hanno valore uguale al primo operando, cioè, in questo caso, `111` (7).
4. Viene eseguita la sezione critica (qui rappresentata da `{CS}`).
5. Infine, si assegna il valore 0 alla variabile `LOCK`, per consentire ad altri processi di accedere alla CS.

2.2 Varianti

In architetture diverse, si hanno varianti diverse della `TS`:

- A volte, l'istruzione `TS` si chiama `TSL`, *test and set lock*.
- La `TS` può avere un secondo parametro, che specifica il registro in cui memorizzare il valore del test, se quest'ultimo non viene posto nel Condition Code della PSW.

- Invece della `TS`, può essere presente un'istruzione indivisibile **swap**, che scambia il contenuto di due locazioni di memoria. Ad esempio, nelle CPU x86, la swap si chiama *exchange* (`XCHG`), e ha come secondo parametro un registro.

  Con la swap, per implementare le sezioni critiche è necessario predisporre una variabile privata (non condivisa), che:

  1. viene inizializzata al valore che rappresenta un lock chiuso (ad esempio, una sequenza di 1);
  2. dopo la swap, contiene il valore precedente del lock, che viene valutato per determinare se il processo può entrare nella CS.

```
LOCK        DC X '00'
TEMP        DC X 'FF'
ENTRY_TEST  SWAP LOCK, TEMP
            COMP TEMP, X '00'
            BC 7, ENTRY_TEST
            {CS}
            MVI LOCK, X '00'
```

3 CS nei linguaggi ad alto livello

Nei linguaggi ad alto livello, non è possibile usare direttamente l'istruzione `TS`. Esistono invece tre strategie per l'implementazione delle CS:

1. **Approccio algoritmico**: i processi (/thread) effettuano una serie di controlli per determinare se una CS è libera prima di entrarvi:
   - non servono accorgimenti a livello hardware (es. `TS`), kernel o di linguaggio;
   - è un approccio difficile da realizzare correttamente.
2. Uso di **primitive software** (system call o chiamate di libreria) per garantire la mutua esclusione.
3. Uso di costrutti ad hoc per la programmazione concorrente.

4 Approccio algoritmico

L'approccio algoritmico prevede di implementare le CS senza usare:

- istruzioni hardware ad hoc;
- system call;

- costrutti specializzati forniti dal linguaggio di programmazione.

Prima di entrare in una CS, un processo controlla alcune condizioni logiche per determinare se essa è libera:

- se il controllo è superato, il processo entra nella CS;
- se, invece, non è superato, il processo ripete il controllo, eseguendo **busy wait** finché non può entrare.

```
...
while (!ok) {
    // busy wait
}
{CS}
...
```

La busy wait sarebbe evitabile solo se il processo si facesse mettere in stato di waiting, ma ciò richiederebbe l'uso di una system call, che non è ammesso nell'approccio algoritmico.

4.1 Variabili di turno: primo tentativo errato

Un primo tentativo di implementazione dell'approccio algoritmico, già visto nella lezione precedente, è basato sull'uso di variabili di turno condivise:

- P_0 esegue il codice:

  ```
  ...
  while (true) {
      ...
      while (turn != 0) {}
      {CS}
      turn = 1;
    ...
  }
  ...
  ```

- P_1 esegue il codice:

  ```
  ...
  while (true) {
      ...
      while (turn != 1) {}
      {CS}
      turn = 0;
  ```

```
    ...
}
...
```

Questa soluzione, però, *viola la proprietà del progresso.*

4.2 Variabili di turno: secondo tentativo errato

Si usano due variabili condivise:

- `c0` vale `true` quando il processo P_0 è nella CS, altrimenti vale `false`;
- `c1` vale `true` quando P_1 è nella CS, e `false` altrimenti.

```
boolean c0 = false, c1 = false;
```

- P_0 esegue il codice:

```
...
while (true) {
    ...
    while (c1) {}
    c0 = true;
    {CS}
    c0 = false;
    ...
}
...
```

- P_1 esegue il codice:

```
...
while (true) {
    ...
    while (c0) {}
    c1 = true;
    {CS}
    c1 = false;
    ...
}
...
```

A differenza della precedente, questa soluzione rispetta la proprietà del progresso, ma *non è corretta*, cioè non garantisce la mutua esclusione. Ad esempio, se P_0 viene interrotto dopo aver controllato il valore di `c1`, ma prima di aver posto `c0` a `true`, anche P_1 può entrare nella CS.

In pratica, questa soluzione non è corretta perché il controllo di `c1` e l'assegnamento a `c0` (e viceversa) non sono indivisibili.

4.3 Variabili di turno: terzo tentativo errato

Le variabili usate sono analoghe al tentativo precedente,

```
boolean c0 = false, c1 = false;
```

ma l'assegnamento viene spostato prima del controllo:

- P_0 esegue il codice:

```
...
while (true) {
    ...
    c0 = true;
    while (c1) {}
    {CS}
    c0 = false;
    ...
}
...
```

- P_1 esegue il codice:

```
...
while (true) {
    ...
    c1 = true;
    while (c0) {}
    {CS}
    c1 = false;
    ...
}
...
```

Questa soluzione garantisce effettivamente la mutua esclusione: quando il controllo eseguito da (ad esempio) P_0 viene superato, è garantito che `c0` sia vera e `c1` sia falsa, quindi non è possibile che entrambi i processi entrino concorrentemente nelle CS.

Può succedere, però, che entrambe le variabili `c0` e `c1` siano impostate a `true` prima che vengano eseguiti i controlli: in questo caso, si i processi si bloccano a vicenda, cioè si ha una situazione di **stallo** (**deadlock**), quindi è *violata la proprietà del progresso.*

4.4 Variabili di turno: quarto tentativo errato

```
boolean c0 = false, c1 = false;
```

- P_0 esegue il codice:

```
...
while (true) {
    ...
a   c0 = true;
    while (c1) {
        c0 = false;
        goto a;
    }
    {CS}
    c0 = false;
    ...
}
...
```

- P_1 esegue il codice:

```
...
while (true) {
    ...
a   c1 = true;
    while (c0) {
        c1 = false;
        goto a;
    }
    {CS}
    c1 = false;
    ...
}
...
```

Questa soluzione garantisce la mutua esclusione (come la precedente), ed elimina la possibilità di deadlock: se (ad esempio) P_0 trova `c1 == true`, rimette `c0` a `false`, per consentire a P_1 di entrare nella sua CS (se sta eseguendo il controllo, e non è invece già entrato), e ripete da capo il procedimento di assegnamento e controllo.

Ciò nonostante, *la proprietà del progresso è comunque violata*, perché si può verificare una situazione di **livelock**, nella quale i processi si ostacolano continuamente a vicenda. Ad esempio:

P_0	P_1
`c0 = true;`	
	`c1 = true;`
`c1 == true` $\Longrightarrow$ corpo del `while`	
	`c0 == true` $\Longrightarrow$ corpo del `while`
`c0 = false;`	
`goto a;`	
	`c1 = false;`
	`goto a;`

4.5 Algoritmo di Dekker

L'**algoritmo di Dekker** è simile al tentativo precedente, ma usa in più una variabile di turno, `turn`, per decidere quale processo accede alla CS quando entrambi vogliono farlo.

```
boolean c0 = false, c1 = false;
int turn = 0;
```

- P_0 esegue il codice:

```
...
while (true) {
    ...
    c0 = true;
    while (c1) {
        if (turn == 1) {
            c0 = false;
            while (turn == 1) {}
            c0 = true;
        }
    }
    {CS}
    turn = 1;
    c0 = false;
    ...
}
...
```

- P_1 esegue il codice:

```
...
while (true) {
    ...
```

```
        c1 = true;
        while (c0) {
            if (turn == 0) {
                c1 = false;
                while (turn == 0) {}
                c1 = true;
            }
        }
        {CS}
        turn = 0;
        c1 = false;
        ...
    }
    ...
```

Questa soluzione è corretta e rispetta la proprietà del progresso, ma è *valida solo per 2 processi*. Generalizzarla a n processi è possibile, ma comporta una notevole complicazione del codice.

4.6 Algoritmo di Peterson

Un'altra soluzione valida (ancora solo per 2 processi) è l'**algoritmo di Peterson**:

- ogni processo ha una flag booleana, che usa per dichiarare di voler accedere alla CS;
- per gestire il caso in cui entrambi i processi vogliono accedere, viene usata una variabile di turno, ma, a differenza dell'algoritmo di Dekker, ciascun processo:
    1. assegna il turno al "rivale" immediantamente prima di controllare se può entrare;
    2. aspetta che il rivale finisca di eseguire la CS, oppure che gli ridia il turno (cercando di entrare).

```
boolean flag[] = {false, false};
int turn = 0;
```

- P_0 esegue il codice:

```
    ...
    while (true) {
        ...
        flag[0] = true;
        turn = 1;
```

```
    while (flag[1] && turn == 1) {}
    {CS}
    flag[0] = false;
    ...
}
...
```

- P_1 esegue il codice:

```
...
while (true) {
    ...
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0) {}
    {CS}
    flag[1] = false;
    ...
}
...
```

4.7 Algoritmi per più di due processi

Due algoritmi per l'implementazione delle CS con $n > 2$ processi sono:

- l'*algoritmo di Eisenberg e McGuire*, che estende al caso n l'algoritmo di Dekker;
- l'*algoritmo di Lamport*.

5 Semafori

Definizione: Un **semaforo** è una *variabile intera condivisa* che può assumere solo *valori non negativi* e su cui sono possibili solo le seguenti tre operazioni:

- *inizializzazione*, con un valore ≥ 0;
- operazione indivisibile (sul semaforo) **wait**:
  - se il semaforo ha valore > 0, viene decrementato;
  - se il semaforo ha valore $= 0$, il processo che esegue la wait viene "bloccato sul semaforo", cioè, di fatto, va in waiting;
- operazione indivisibile **signal**:
  - se ci sono processi bloccati sul semaforo, esattamente uno di essi viene "sbloccato", cioè, di fatto, va in ready;

  - se nessun processo è bloccato sul semaforo, il valore del semaforo viene incrementato.

Osservazioni:

- Quando ci sono processi bloccati su un semaforo, il valore di tale semaforo è sempre 0, perché:
  - se il valore è superiore a 0, un processo che fa la wait sul semaforo non si può bloccare;
  - se ci sono processi bloccati, la signal non può incrementare il valore del semaforo.
- Nella definizione, non è specificato come siano implementate wait e signal, e, in particolare, come sia garantita la loro indivisibilità.
- Nel caso in cui ci sono più processi bloccati su un semaforo e viene eseguita una signal, la definizione non specifica la politica di selezione del processo da sbloccare. Molte implementazioni adottano una politica FIFO, cioè sbloccano il processo che è bloccato da più tempo.

5.1 Implementazione delle CS con i semafori

Un semaforo può essere usato per garantire la mutua esclusione su un (altro) dato condiviso:

- il semaforo viene inizializzato a 1;

  ```
  semaphore sem = 1;
  ```

- prima di accedere alla CS, si effettua una wait sul semaforo;
- dopo aver eseguito la CS, si effettua una signal sul semaforo.

Il codice eseguito da un processo P_i è quindi:

```
...
while (true) {
    ...
    wait(sem);
    {CS}
    signal(sem);
    ...
}
...
```

Usandolo in questo modo, il semaforo `sem` assume solo valori 0 e 1:

1. quando un processo P_i vuole accedere alla CS (ipotizzando che essa sia inizialmente libera), fa la wait, e `sem` diventa così 0;
2. mentre P_i è nella CS, eventuali altri processi che vogliono accedere trovano `sem` a 0 (ciò è garantito dall'indivisibilità della wait) quindi si bloccano;
3. quando P_i esce dalla CS, fa una signal:
   - se ci sono processi in attesa, `sem` rimane 0 e uno di essi entra nella CS;
   - altrimenti, `sem` torna a 1, per indicare che la CS è libera.

Questa soluzione soddisfa le proprietà di correttezza e progresso. La bounded wait, invece, può essere garantita o meno, a seconda dell'implementazione della signal: ad esempio, se i processi vengono sbloccati nello stesso ordine in cui si sono bloccati (politica FIFO), la bounded wait è garantita.

Osservazioni:

- Se il semaforo fosse inizializzato a un valore $n > 1$, non sarebbe rispettata la proprietà di correttezza, perché nelle CS potrebbero entrare in concorrenza n processi.
- Se il semaforo fosse inizializzato a 0, nessun processo potrebbe accedere alle CS, quindi sarebbe violata la proprietà del progresso.
- Questa soluzione è molto più semplice rispetto all'approccio algoritmico, ma, in realtà, il motivo è che la "parte complicata" viene gestita dall'implementazione dei semafori.
- È comunque responsabilità del programmatore rispettare la sequenza wait–`{CS}`–signal (così come, in linguaggio macchina, è responsabilità del programmatore implementare correttamente le CS con l'istruzione `TS`).

Nota: A una variabile semaforo usata per implementare le sezioni critiche, si dà spesso il nome `mutex` (che è un'abbreviazione di "mutual exclusion").

5.2 Deadlock

L'uso dei semafori non è banale. Ad esempio, se `x` e `y` sono semafori inizializzati a 1, e

- il processo P_1 esegue:

```
...
wait(x);
wait(y);
...
signal(x);
signal(y);
```

```
...
```

- il processo P_2 esegue:

```
...
wait(y);
wait(x);
...
signal(y);
signal(x);
...
```

allora si può verificare un **deadlock**:

x	y	P_1	P_2
1	1		
0	1	`wait(x);`	
0	0		`wait(y);`
0	0		`wait(x);` $\Longrightarrow$ blocco
0	0	`wait(y);` $\Longrightarrow$ blocco	