Capitolo 6

Sincronizzazione dei processi

OBIETTIVI

- **Introduzione al problema della sezione critica, le cui soluzioni – sia hardware sia software – sono utilizzabili per assicurare la coerenza dei dati condivisi.**
- **Introduzione al concetto di transazione atomica e descrizione dei meccanismi atti ad assicurare l'atomicità.**

Un processo cooperante è un processo che può influenzarne un altro in esecuzione nel sistema o anche subirne l'influenza. I processi cooperanti possono condividere direttamente uno spazio logico di indirizzi (cioè, codice e dati) oppure condividere dati soltanto attraverso i file. Nel primo caso si fa uso dei thread, presentati nel Capitolo 4. L'accesso concorrente a dati condivisi può tuttavia causare situazioni di incoerenza degli stessi dati. In questo capitolo si trattano vari meccanismi atti ad assicurare un'ordinata esecuzione dei processi cooperanti, che condividono uno spazio logico di indirizzi, così da mantenere la coerenza dei dati.

6.1 Introduzione

Nel Capitolo 3 è stato descritto un modello di sistema costituito da un certo numero di processi sequenziali cooperanti o thread, tutti in esecuzione asincrona e con la possibilità di condividere dati. Tale modello è illustrato attraverso l'esempio del produttore/consumatore, che ben rappresenta molte situazioni che riguardano i sistemi operativi. Nel Paragrafo 3.4.1, in particolare, si è descritto come un buffer limitato sia utilizzabile per permettere ai processi la condivisione della memoria.

Torniamo al concetto di buffer limitato. Come è stato sottolineato, la nostra soluzione consente la presenza contemporanea di non più di `DIM_BUFFER – 1` elementi. Si supponga di voler modificare l'algoritmo per rimediare a questa carenza. Una possibilità consiste nell'aggiungere una variabile intera, `contatore`, inizializzata a `0`, che si incrementa ogniqualvolta s'inserisce un nuovo elemento nel buffer e si decrementa ogniqualvolta si preleva un elemento dal buffer. Il codice per il processo produttore si può modificare come segue:

```
while (true)
{
   /* produce un elemento in appena_Prodotto */
```

```
    while (contatore == DIM_BUFFER)
        ; /* non fa niente */
    buffer[inserisci] = appena_Prodotto;
    inserisci = (inserisci + 1) % DIM_BUFFER;
    contatore++;
}
```

Il codice per il processo consumatore si può modificare come segue:

```
while (true)
{
    while (contatore == 0)
        ; /* non fa niente */
    da_Consumare = buffer[preleva];
    preleva = (preleva + 1) % DIM_BUFFER;
    contatore--;
    /* consuma un elemento in da_Consumare */
}
```

Sebbene siano corrette se si considerano separatamente, le procedure del produttore e del consumatore possono non funzionare altrettanto correttamente se si eseguono in modo concorrente. Si supponga per esempio che il valore della variabile `contatore` sia attualmente `5`, e che i processi produttore e consumatore eseguano le istruzioni `contatore++` e `contatore--` in modo concorrente. Terminata l'esecuzione delle due istruzioni, il valore della variabile `contatore` potrebbe essere `4`, `5` o `6`! Il solo risultato corretto è `contatore == 5`, che si ottiene se si eseguono separatamente il produttore e il consumatore.

Si può dimostrare che il valore di `contatore` può essere scorretto: l'istruzione `contatore++` si può codificare in un tipico linguaggio macchina, come

```
registro1 := contatore
registro1 := registro1 + 1
contatore := registro1
```

dove $registro_1$ è un registro locale della CPU. Analogamente, l'istruzione `contatore--` si può codificare come

```
registro2 := contatore
registro2 := registro2 - 1
contatore := registro2
```

dove $registro_2$ è un registro locale della CPU. Anche se $registro_1$ e $registro_2$ possono essere lo stesso registro fisico, per esempio un accumulatore, occorre ricordare che il contenuto di questo registro viene salvato e recuperato dal gestore dei segnali d'interruzione (Paragrafo 1.2.3).

L'esecuzione concorrente delle istruzioni `contatore++` e `contatore--` equivale a un'esecuzione sequenziale delle istruzioni del linguaggio macchina introdotte precedentemente, intercalate (*interleaved*) in una qualunque sequenza che però conservi l'ordine interno di ogni singola istruzione di alto livello. Una di queste sequenze è

T_0:	*produttore*	esegue	`registro1 := contatore`	`{registro1 = 5}`
T_1:	*produttore*	esegue	`registro1 := registro1 + 1`	`{registro1 = 6}`
T_2:	*consumatore*	esegue	`registro2 := contatore`	`{registro2 = 5}`
T_3:	*consumatore*	esegue	`registro2 := registro2 – 1`	`{registro2 = 4}`
T_4:	*produttore*	esegue	`contatore := registro1`	`{contatore = 6}`
T_5:	*consumatore*	esegue	`contatore := registro2`	`{contatore = 4}`

e conduce al risultato errato in cui `contatore == 4`; si registra la presenza di 4 elementi nel buffer, mentre in realtà gli elementi sono 5. Se si invertisse l'ordine delle istruzioni in T_4 e T_5 si giungerebbe allo stato errato in cui `contatore == 6`.

Per evitare le situazioni di questo tipo, in cui più processi accedono e modificano gli stessi dati in modo concorrente e i risultati dipendono dall'ordine degli accessi (le cosiddette **race condition**) occorre assicurare che un solo processo alla volta possa modificare la variabile `contatore`. Questa condizione richiede una forma di sincronizzazione dei processi.

Tali situazioni si verificano spesso nei sistemi operativi, nei quali diversi componenti compiono operazioni su risorse condivise. Inoltre, con la diffusione dei sistemi multicore si dà sempre più importanza allo sviluppo di applicazioni multithread in cui diversi thread, che possono anche condividere dei dati, sono in esecuzione in parallelo su unità di calcolo distinte. Ovviamente tali operazioni non devono interferire reciprocamente in modi indesiderati. Data l'importanza della questione, la maggior parte di questo capitolo è dedicata ai problemi della **sincronizzazione** e **coordinazione dei processi**.

6.2 Problema della sezione critica

Si consideri un sistema composto di n processi $\{P_0, P_1, \ldots, P_{n-1}\}$ ciascuno avente un segmento di codice, chiamato **sezione critica** (detto anche *regione critica*), in cui il processo può modificare variabili comuni, aggiornare una tabella, scrivere in un file e così via. Quando un processo è in esecuzione nella propria sezione critica, non si deve consentire a nessun altro processo di essere in esecuzione nella propria sezione critica. Quindi, l'esecuzione delle sezioni critiche da parte dei processi è *mutuamente esclusiva* nel tempo. Il problema della *sezione critica* si affronta progettando un protocollo che i processi possono usare per cooperare. Ogni processo deve chiedere il permesso per entrare nella propria sezione critica. La sezione di codice che realizza questa richiesta è la **sezione d'ingresso**. La sezione critica può essere seguita da una **sezione d'uscita**, e la restante parte del codice è detta **sezione non critica**. La Figura 6.1 mostra la struttura generale di un tipico processo P_i. La sezione d'ingresso e quella d'uscita sono state inserite nei riquadri per evidenziare questi importanti segmenti di codice.

Una soluzione del problema della sezione critica deve soddisfare i tre seguenti requisiti.

1. **Mutua esclusione.** Se il processo P_i è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
2. **Progresso.** Se nessun processo è in esecuzione nella sua sezione critica e qualche processo desidera entrare nella propria sezione critica, solo i processi che si trovano fuori delle rispettive sezioni non critiche possono partecipare alla decisione riguardante la scelta del processo che può entrare per primo nella propria sezione critica; questa scelta non si può rimandare indefinitamente.

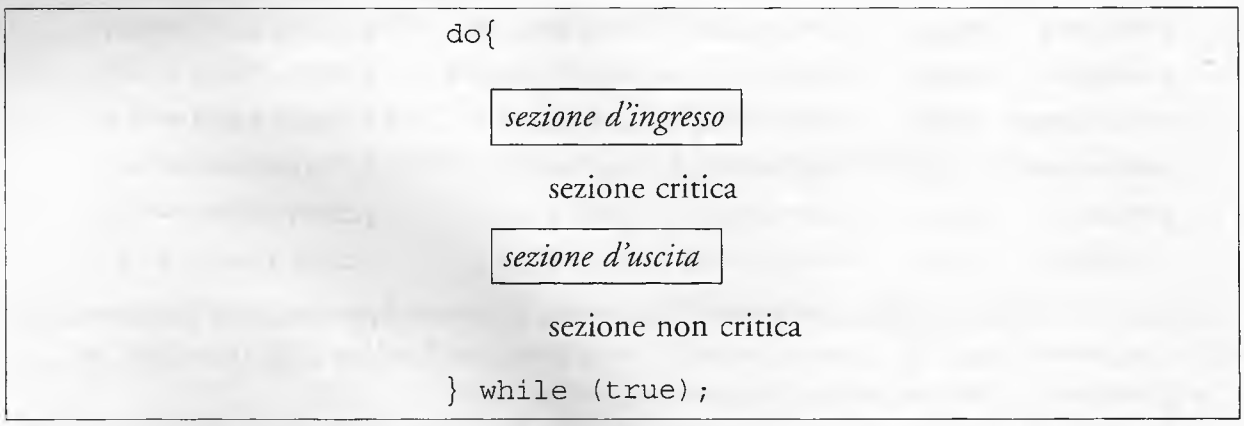

Figura 6.1 Struttura generale di un tipico processo P_i.

3. **Attesa limitata.** Se un processo ha già richiesto l'ingresso nella sua sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che si accordi la richiesta del primo processo.

Si suppone che ogni processo sia eseguito a una velocità diversa da zero. Tuttavia, non si può fare alcuna ipotesi sulla **velocità relativa** degli *n* processi.

In un dato momento, numerosi processi in modalità utente possono essere attivi nel sistema operativo. Se ciò si verifica, il codice del kernel, che implementa il sistema operativo, si trova a dover regolare gli accessi ai dati condivisi. Si consideri per esempio una struttura dati del kernel che mantenga una lista di tutti i file aperti nel sistema. Tale lista deve essere modificata quando un nuovo file è aperto, e quindi aggiunto all'elenco, oppure chiuso, e quindi tolto dall'elenco. Due o più processi che dovessero aprire dei file, ognuno per proprio conto e simultaneamente, potrebbero ingenerare nel sistema una cosiddetta *race condition* legata ai necessari aggiornamenti della lista dei file aperti. Altre strutture dati del kernel soggette a problemi analoghi sono quelle per l'allocazione della memoria, per la gestione delle interruzioni e le liste dei processi. La responsabilità di preservare il sistema operativo da simili problemi relativi all'ordine degli accessi compete a chi sviluppa il kernel.

La due strategie principali per la gestione delle sezioni critiche nei sistemi operativi prevedono l'impiego di: (1) **kernel con diritto di prelazione** e (2) **kernel senza diritto di prelazione.** Un kernel con diritto di prelazione consente che un processo funzionante in modalità di sistema sia sottoposto a prelazione, rinviandone in tal modo l'esecuzione. Un kernel senza diritto di prelazione non consente di applicare la prelazione a un processo attivo in modalità di sistema: l'esecuzione di questo processo seguiterà finché lo stesso esca da tale modalità, si blocchi o ceda volontariamente il controllo della CPU. In sostanza, i kernel senza diritto di prelazione sono immuni dai problemi legati all'ordine degli accessi alle strutture dati del kernel, visto che un solo processo per volta impegna il kernel. Altrettanto non si può dire dei kernel con diritto di prelazione, motivo per cui bisogna avere cura, nella progettazione, di mantenerli al riparo dai problemi insiti nell'ordine degli accessi. I kernel con diritto di prelazione rivelano particolari difficoltà di progettazione quando sono destinati ad architetture SMP, poiché in tali ambienti due processi nella modalità di sistema possono essere eseguiti in contemporanea su processori differenti.

Perché, allora, i kernel con diritto di prelazione dovrebbero essere preferiti a quelli senza diritto di prelazione? I kernel con diritto di prelazione sono più adatti alla programmazione real-time, dal momento che permette ai processi in tempo reale di far valere il loro diritto di precedenza nei confronti di un processo attivo nel kernel. Inoltre, i kernel con dirit-

to di prelazione possono vantare una maggiore prontezza nelle risposte, data la loro scarsa propensione a eseguire i processi in modalità di sistema per un tempo eccessivamente lungo, prima di liberare la CPU per i processi in attesa. Naturalmente, il fatto che questo effetto sia notevole o trascurabile dipende dai dettagli del codice del kernel. Vedremo più avanti come diversi sistemi operativi usino la prelazione all'interno del kernel.

6.3 Soluzione di Peterson

Illustriamo adesso una classica soluzione software al problema della sezione critica, nota come **soluzione di Peterson**. A causa del modo in cui i moderni elaboratori eseguono le istruzioni elementari del linguaggio macchina, quali `load` e `store`, non è affatto certo che la soluzione di Peterson funzioni correttamente su tali sistemi. Tuttavia si è scelto di presentarla ugualmente perché rappresenta un buon algoritmo per il problema della sezione critica che illustra alcune insidie legate alla progettazione di programmi che soddisfino i tre requisiti di mutua esclusione, progresso e attesa limitata.

La soluzione di Peterson si applica a due processi, P_0 e P_1, ognuno dei quali esegue alternativamente la propria sezione critica e la sezione rimanente. Per il seguito, è utile convenire che se P_i denota uno dei due processi, P_j denoti l'altro; ossia, che $j = 1 - i$.

La soluzione di Peterson richiede che i processi condividano i seguenti dati:

```
int turno;
boolean flag[2];
```

La variabile `turno` segnala, per l'appunto, di chi sia il turno d'accesso alla sezione critica; quindi, se `turno == i`, il processo P_i è autorizzato a eseguire la propria sezione critica. L'array `flag`, invece, indica se un processo *sia pronto* a entrare nella propria sezione critica. Per esempio, se `flag[i]` è `true`, P_i lo è. In base a queste delucidazioni, possiamo ora analizzare l'algoritmo descritto nella Figura 6.2.

Per accedere alla sezione critica, il processo P_i assegna innanzitutto a `flag[i]` il valore `true`; quindi attribuisce a `turno` il valore `j`, conferendo così all'altro processo la facoltà di entrare nella sezione critica. Qualora entrambi i processi tentino l'accesso contemporaneo, all'incirca nello stesso momento sarà assegnato a `turno` sia il valore `i` sia il valore `j`. Soltanto uno dei due permane: l'altro sarà immediatamente sovrascritto. Il valore definitivo

```
do {
    flag[i]= true;
    turno = j;
    while (flag[j] && turno == j);

        sezione critica

    flag[i] = false;

        sezione non critica

} while (true);
```

Figura 6.2 Struttura del processo P_i nella soluzione di Peterson.

di `turno` stabilisce quale dei due processi sia autorizzato a entrare per primo nella propria sezione critica.

Dimostriamo ora la correttezza di questa soluzione. Dobbiamo provare che:

1. la mutua esclusione è preservata;
2. il requisito del progresso è soddisfatto;
3. il requisito dell'attesa limitata è rispettato.

Per dimostrare la proprietà 1, si osservi come ogni P_i acceda alla propria sezione critica solo se `flag[j] == false` oppure `turno == i`. Si noti anche che, se entrambi i processi sono eseguibili in concomitanza nelle rispettive sezioni critiche, allora `flag[0] == flag[1] == true`. Si desume da queste due osservazioni che P_0 e P_1 sono impossibilitati a eseguire con successo le rispettive istruzioni `while` approssimativamente nello stesso momento: `turno`, infatti, può valere 0 o 1, ma non entrambi. Pertanto, uno dei processi – poniamo P_i – deve aver eseguito con successo l'istruzione `while`, mentre P_j aveva da eseguire almeno un'istruzione aggiuntiva ("`turno == j`"). Tuttavia, poiché da quel momento, e fino al termine della permanenza di P_j nella propria sezione critica, restano valide le asserzioni `flag[j] == true` e `turno == j`, ne consegue che la mutua esclusione è preservata.

Per dimostrare le proprietà 2 e 3, osserviamo come l'ingresso di un processo P_i nella propria sezione critica possa essere impedito solo se il processo è bloccato nella sua iterazione `while`, con le condizioni `flag[j] == true` e `turno == j`; questa è l'unica possibilità. Qualora P_j non sia pronto a entrare nella sezione critica, `flag[j] == false`, e P_i può accedere alla propria sezione critica. Se P_j ha impostato `flag[j]` a `true` e sta eseguendo il proprio ciclo `while`, `turno == i`, oppure `turno == j`. Se `turno == i`, P_i entrerà nella propria sezione critica. Se `turno == j`, P_j entrerà nella propria sezione critica. Tuttavia, al momento di uscire dalla propria sezione critica, P_j reimposta `flag[j]` a `false`, consentendo a P_i di entrarvi. Se P_j imposta `flag[j]` a `true`, deve anche attribuire alla variabile `turno` il valore `i`. Poiché tuttavia P_i non modifica il valore della variabile `turno` durante l'esecuzione dell'istruzione `while`, P_i entrerà nella sezione critica (progresso) dopo che P_j abbia effettuato non più di un ingresso (attesa limitata).

6.4 Hardware per la sincronizzazione

Abbiamo appena descritto una soluzione software al problema della sezione critica. In generale, si può affermare che qualunque soluzione al problema richiede l'uso si un semplice strumento detto **lock** (*lucchetto*). Il corretto ordine degli accessi alle strutture dati del kernel è garantito dal fatto che le sezioni critiche sono protette da lock. In altri termini, per accedere alla propria sezione critica un processo deve acquisire il possesso di un lock, che restituirà al momento della sua uscita. Si veda in proposito la Figura 6.3.

Nelle pagine seguenti esploreremo altre soluzioni al problema della sezione critica che sfruttano tecniche diverse, dai meccanismi hardware alle API disponibili per i programmatori. Tali soluzioni si basano tutte sul concetto di *lock*; come si vedrà, la progettazione di un lock può divenire complessa.

Iniziamo presentando alcune semplici istruzioni hardware disponibili in molti sistemi e mostrando come queste possano essere efficacemente utilizzate per risolvere il problema della sezione critica. Le funzionalità hardware possono rendere più facile il compito del programmatore e migliorare l'efficienza del sistema.

In un sistema dotato di una singola CPU tale problema si potrebbe risolvere semplicemente se si potessero interdire le interruzioni mentre si modificano le variabili condivise. In

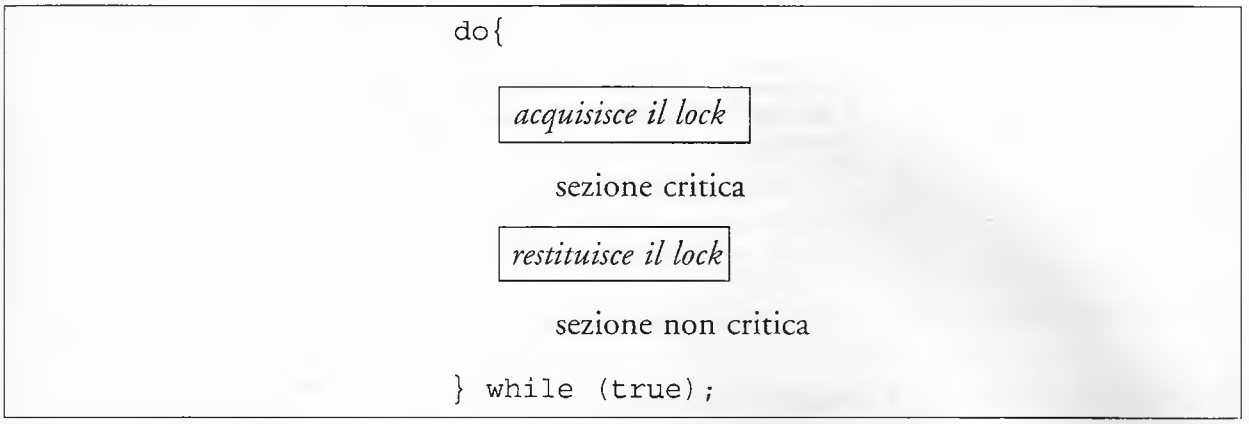

Figura 6.3 Soluzione al problema della sezione critica tramite lock.

questo modo si assicurerebbe un'esecuzione ordinata e senza possibilità di prelazione della corrente sequenza di istruzioni; non si potrebbe eseguire nessun'altra istruzione, quindi non si potrebbe apportare alcuna modifica inaspettata alle variabili condivise. È questo l'approccio seguito dai kernel senza diritto di prelazione.

Sfortunatamente questa soluzione non è sempre praticabile; la disabilitazione delle interruzioni nei sistemi multiprocessore può comportare sprechi di tempo dovuti alla necessità di trasmettere la richiesta di disabilitazione delle interruzioni a tutte le unità d'elaborazione. Tale trasmissione ritarda l'accesso a ogni sezione critica determinando una diminuzione dell'efficienza. Si considerino, inoltre, gli effetti su un orologio di sistema aggiornato tramite le interruzioni.

Per questo motivo molte delle moderne architetture offrono particolari istruzioni che permettono di controllare e modificare il contenuto di una parola di memoria, oppure di scambiare il contenuto di due parole di memoria, in modo **atomico** – cioè come un'unità non interrompibile. Queste speciali istruzioni sono utilizzabili per risolvere il problema della sezione critica in modo relativamente semplice. Anziché discutere una specifica istruzione di una particolare architettura, è preferibile astrarre i concetti principali che stanno alla base di queste istruzioni.

L'istruzione `TestAndSet()` si può definire com'è illustrato nella Figura 6.4. Questa istruzione è eseguita **atomicamente**, cioè come un'unità non soggetta a interruzioni; quindi, se si eseguono contemporaneamente due istruzioni `TestAndSet()`, ciascuna in un'unità d'elaborazione diversa, queste vengono eseguite in modo sequenziale in un ordine arbitrario. Se si dispone dell'istruzione `TestAndSet()`, si può realizzare la mutua esclusione dichiarando una variabile booleana globale `lock`, inizializzata a `false`. La struttura del processo P_i è illustrata nella Figura 6.5.

L'istruzione `Swap()`, definita nella Figura 6.6, agisce sul contenuto di due parole di memoria; come l'istruzione `TestAndSet()`, è anch'essa eseguita atomicamente. Se si dispo-

```
boolean TestAndSet(boolean *obiettivo) {
    boolean valore = *obiettivo;
    *obiettivo = true;
    return valore;
}
```

Figura 6.4 Definizione dell'istruzione `TestAndSet()`.

```
do {
    while (TestAndSet(&lock));
        sezione critica
    lock = false;
        sezione non critica
} while (true);
```

Figura 6.5 Realizzazione di mutua esclusione con `TestAndSet()`.

ne dell'istruzione `Swap()`, la mutua esclusione si garantisce dichiarando e inizializzando al valore `false` una variabile booleana globale `lock`. Inoltre, ogni processo possiede anche una variabile booleana locale `chiave`. La struttura del processo P_i è illustrata nella Figura 6.7.

Questi algoritmi soddisfano il requisito della mutua esclusione, ma non quello dell'attesa limitata. La Figura 6.8 mostra un altro algoritmo che sfrutta l'istruzione `TestAndSet()` per soddisfare tutti e tre i requisiti desiderati. Le strutture dati condivise sono:

```
boolean attesa[n];
boolean lock;
```

e sono inizializzate al valore `false`. Per dimostrare che l'algoritmo soddisfa il requisito di mutua esclusione, si considera che il processo P_i possa entrare nella propria sezione critica solo se `attesa[i] == false` oppure `chiave == false`. Il valore di `chiave` può diventare `false` solo se si esegue `TestAndSet()`. Il primo processo che esegue `TestAndSet()` trova `chiave == false`; tutti gli altri devono attendere. La variabile `attesa[i]` può diventare `false` solo se un altro processo esce dalla propria sezione critica; solo una variabile `attesa[i]` vale `false`, il che consente di rispettare il requisito di mutua esclusione.

Per dimostrare che l'algoritmo soddisfa il requisito di progresso, basta osservare che le argomentazioni fatte per la mutua esclusione valgono anche in questo caso; infatti un processo che esce dalla sezione critica imposta `lock` al valore `false` oppure `attesa[j]` al valore `false`; entrambe consentono a un processo in attesa l'ingresso nella propria sezione critica.

Per dimostrare che l'algoritmo soddisfa il requisito di attesa limitata occorre osservare che un processo, quando lascia la propria sezione critica, scandisce il vettore `attesa` nell'ordinamento ciclico ($i + 1, i + 2, ..., n - 1, 0, ..., i - 1$) e designa il primo processo in questo ordinamento presente nella sezione d'ingresso (`attesa[j] == true`) come il primo

```
void Swap(boolean *a, boolean *b) {
   boolean temp = *a;
   *a = *b;
   *b = temp;
}
```

Figura 6.6 Definizione dell'istruzione `Swap()`.

```
do {

    chiave = true;
    while (chiave == true)
       Swap(&lock, &chiave);

       sezione critica

    lock = false;

       sezione non critica

} while (true);
```

Figura 6.7 Realizzazione di mutua esclusione con `Swap()`.

processo che deve entrare nella propria sezione critica. Qualsiasi processo che attende l'ingresso nella propria sezione critica può farlo entro $n - 1$ turni.

Per gli sviluppatori hardware la progettazione delle istruzioni atomiche `TestAndSet()` per sistemi multiprocessore non è certo un compito banale. Quest'argomento è trattato nei testi di architetture dei calcolatori.

```
do {

    attesa[i] = true;
    chiave = true;
    while (attesa[i] && chiave)
       chiave = TestAndSet(&lock);
    attesa[i] = false;

       sezione critica

    j = (i + 1) % n;
    while ((j != i) && !attesa[i])
      j = (j + 1) % n;

    if (j == i)
      lock = false;
    else
      attesa[j] = false;

       sezione non critica

} while (true);
```

Figura 6.8 Mutua esclusione con attesa limitata con `TestAndSet()`.

6.5 Semafori

Le varie soluzioni hardware al problema della sezione critica presentate nel Paragrafo 6.4, basate su istruzioni quali `TestAndSet()` e `Swap()`, complicano l'attività dei programmatori di applicazioni. Per superare questa difficoltà si può usare uno strumento di sincronizzazione chiamato semaforo.

Un **semaforo** `S` è una variabile intera cui si può accedere, escludendo l'inizializzazione, solo tramite due operazioni atomiche predefinite: `wait()` e `signal()`. Queste operazioni erano originariamente chiamate `P` (per `wait()`; dall'olandese *proberen*, verificare) e `V` (per `signal()`; da *verhogen*, incrementare). La definizione classica di `wait()` in pseudocodice è la seguente:

```
wait(S) {
   while(S <= 0)
      ;//non-op
   S--;
}
```

La definizione classica di `signal()` in pseudocodice è la seguente:

```
signal(S) {
   S++;
}
```

Tutte le modifiche al valore del semaforo contenute nelle operazioni `wait()` e `signal()` si devono eseguire in modo indivisibile: mentre un processo cambia il valore del semaforo, nessun altro processo può contemporaneamente modificare quello stesso valore. Inoltre, nel caso della `wait(S)` si devono eseguire senza interruzione anche la verifica del valore intero di `S` (`S` ≤ 0) e la sua possibile modifica (`S--`). Nel Paragrafo 6.5.2 si spiega come si possono realizzare queste operazioni.

6.5.1 Uso dei semafori

Si usa distinguere tra **semafori contatore**, il cui valore numerico è illimitato, e i **semafori binari**, il cui valore è 0 o 1. In relazione a certi sistemi i semafori binari sono anche detti **lock mutex** (*mutex locks*), perché fungono da "lock" che garantiscono la mutua esclusione (dall'inglese *mu*tual *ex*clusion).

I semafori sono utilizzabili per risolvere il problema della sezione critica con n processi. Gli n processi condividono un semaforo comune, `mutex`, inizializzato a `1`. Ogni processo P_i è strutturato com'è illustrato nella Figura 6.9.

I semafori contatore trovano applicazione nel controllo dell'accesso a una data risorsa presente in un numero finito di esemplari. Il semaforo è inizialmente impostato al numero di esemplari disponibili. I processi che desiderino utilizzare un esemplare della risorsa invocano `wait()` sul semaforo, decrementandone così il valore; i processi che restituiscono un esemplare della risorsa, invece, invocano `signal()` sul semaforo, incrementandone il valore. Quando il semaforo vale 0, vengono allocati tutti gli esemplari della risorsa, e i processi che ne richiedano l'uso dovranno bloccarsi fino a che il semaforo non ritorni positivo.

I semafori sono utilizzabili anche per risolvere diversi problemi di sincronizzazione. Si considerino, per esempio, due processi in esecuzione concorrente: P_1 con un'istruzione S_1 e P_2 con un'istruzione S_2. Si supponga di voler eseguire S_2 solo dopo che S_1 è terminata. Que-

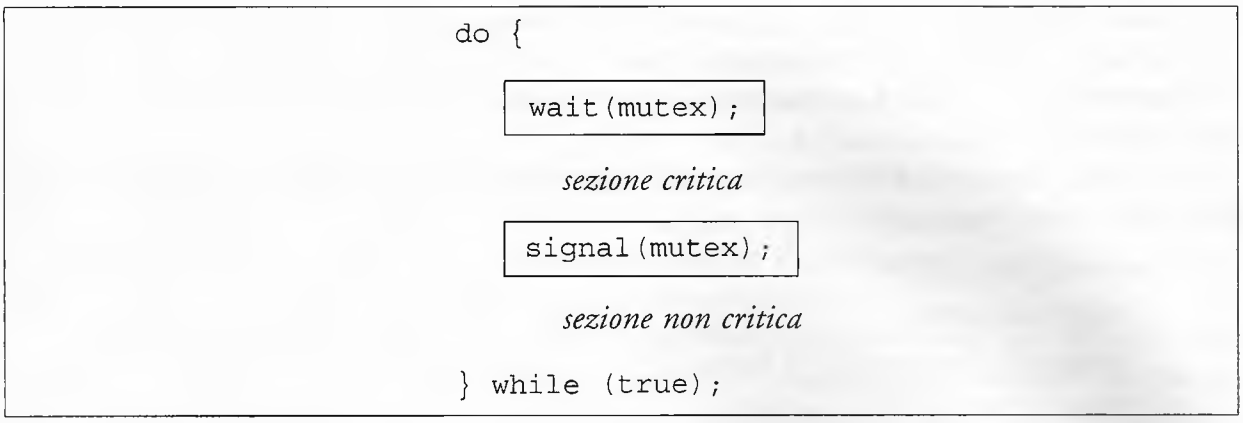

Figura 6.9 Realizzazione di mutua esclusione con semafori.

sto schema si può prontamente realizzare facendo condividere a P_1 e P_2 un semaforo comune, `sincronizzazione`, inizializzato a `0`, e inserendo nel processo P_1 le istruzioni

```
S1;
signal(sincronizzazione);
```

e nel processo P_2 le istruzioni

```
wait(sincronizzazione);
S2;
```

Poiché `sincronizzazione` è inizializzato a 0, P_2 esegue S_2 solo dopo che P_1 ha eseguito `signal(sincronizzazione)`, che si trova dopo S_1.

6.5.2 Realizzazione

Il principale svantaggio della definizione di semaforo è che richiede una condizione di **attesa attiva** (*busy waiting*). Mentre un processo si trova nella propria sezione critica, qualsiasi altro processo che tenti di entrarvi si trova sempre nel ciclo del codice della sezione d'ingresso. Chiaramente questa soluzione costituisce un problema per un sistema con multiprogrammazione, poiché la condizione d'attesa attiva spreca cicli della CPU che un altro processo potrebbe sfruttare in modo produttivo. Questo tipo di semaforo è anche detto **spinlock**, perché i processi "girano" (*spin*) mentre attendono al semaforo. (I semafori spinlock hanno però il vantaggio di non richiedere cambio di contesto nel caso in cui un processo sia fermo in attesa. Tali cambi di contesto possono essere piuttosto costosi, in termini di tempo. Ne consegue che i semafori spinlock sono utili quando i lock sono applicati per brevi intervalli di tempo: infatti, trovano frequente applicazione nei sistemi multiprocessore, dove un processo gira su un processore mentre un altro thread esegue la propria sezione critica su un altro processore.)

Per superare la necessità dell'attesa attiva, si possono modificare le definizioni delle operazioni `wait()` e `signal()`: quando un processo invoca l'operazione `wait()` e trova che il valore del semaforo non è positivo, deve attendere, ma anziché restare nell'attesa attiva può *bloccare* se stesso. L'operazione di *bloccaggio* pone il processo in una coda d'attesa associata al semaforo e cambia lo stato del processo nello stato d'attesa. Quindi, si trasferisce il controllo allo scheduler della CPU che sceglie un altro processo pronto per l'esecuzione.

Un processo bloccato, che attende a un semaforo `S`, sarà riavviato in seguito all'esecuzione di un'operazione `signal()` su `S` da parte di qualche altro processo. Il processo si riav-

via tramite un'operazione `wakeup()`, che modifica lo stato del processo da attesa a pronto. Il processo entra nella coda dei processi pronti. (L'uso della CPU può essere o non essere commutato dal processo in esecuzione al processo appena divenuto pronto, a seconda del criterio di scheduling.)

Per realizzare i semafori secondo quel che s'è detto si può definire il semaforo come una struttura del linguaggio C:

```
typedef struct {
   int valore;
   struct processo *lista;
} semaforo;
```

A ogni semaforo sono associati un `valore` intero e una `lista` di processi, contenente i processi in attesa a un semaforo; l'operazione `signal()` preleva un processo da tale lista e lo attiva.

L'operazione `wait()` del semaforo si può definire come segue:

```
wait(semaforo *S) {
        S->valore--;
        if (S->valore < 0) {
                aggiungi questo processo a S->lista;
                block();
        }
}
```

L'operazione `signal()` del semaforo si può definire come segue:

```
signal(semaforo *S) {
        S->valore++;
        if (S->valore <= 0) {
                togli un processo P da S->lista;
                wakeup(P);
        }
}
```

L'operazione `block()` sospende il processo che la invoca; l'operazione `wakeup(P)` pone in stato di pronto per l'esecuzione un processo `P` bloccato. Queste due operazioni sono fornite dal sistema operativo come chiamate di sistema di base.

Occorre notare che, mentre la definizione classica di semaforo ad attesa attiva è tale che il valore del semaforo non è mai negativo, tale definizione può condurre a valori negativi. Se il valore del semaforo è negativo, la sua dimensione è data dal numero dei processi che attendono a quel semaforo. Ciò avviene a causa dell'inversione dell'ordine del decremento e della verifica nel codice dell'operazione `wait()`.

La lista dei processi che attendono a un semaforo si può facilmente realizzare inserendo un campo puntatore in ciascun blocco di controllo del processo (PCB). Ogni semaforo contiene un valore intero e un puntatore a una lista di PCB. Per aggiungere e togliere processi dalla lista assicurando un'attesa limitata si può usare una coda FIFO, della quale il semaforo contiene i puntatori al primo e all'ultimo elemento. In generale si può usare *qualsiasi* criterio d'accodamento; il corretto uso dei semafori non dipende dal particolare criterio adottato.

I semafori devono essere eseguiti in modo atomico. Si deve garantire che nessuno dei due processi possa eseguire operazioni `wait()` e `signal()`contemporaneamente sullo stesso semaforo. Si tratta di un problema di accesso alla sezione critica, e in un contesto monoprocessore lo si può risolvere semplicemente inibendo le interruzioni durante l'esecuzione di `signal()` e `wait()`. Nei sistemi con una sola CPU, infatti, le interruzioni sono i soli elementi di disturbo: non vi sono istruzioni eseguite da altri processori. Finché non si riattivino le interruzioni, dando la possibilità allo scheduler di riprendere il controllo della CPU, il processo corrente continua indisturbato la sua esecuzione.

Nei sistemi multiprocessore è necessario disabilitare le interruzioni di tutti i processori, perché altrimenti le istruzioni dei diversi processi in esecuzione su processori distinti potrebbero interferire fra loro. Tuttavia, disabilitare le interruzioni di tutti i processori può non essere cosa semplice, e causare un notevole calo delle prestazioni. È per questo che – per garantire l'esecuzione atomica di `wait()` e `signal()` – i sistemi SMP devono mettere a disposizione altre tecniche di realizzazione dei lock (per esempio, gli spinlock).

È importante rilevare che questa definizione delle operazioni `wait()` e `signal()` non consente di eliminare completamente l'attesa attiva, ma piuttosto di rimuoverla dalle sezioni d'ingresso dei programmi applicativi. Inoltre, l'attesa attiva si limita alle sezioni critiche delle operazioni `wait()` e `signal()`, che sono abbastanza brevi; se sono convenientemente codificate, non sono più lunghe di 10 istruzioni. Quindi, la sezione critica non è quasi mai occupata e l'attesa attiva si presenta raramente e per breve tempo. Una situazione completamente diversa si verifica con i programmi applicativi le cui sezioni critiche possono essere lunghe minuti o anche ore, oppure occupate spesso. In questi casi l'attesa attiva è assai inefficiente.

6.5.3 Stallo e attesa indefinita

La realizzazione di un semaforo con coda d'attesa può condurre a situazioni in cui ciascun processo di un insieme di processi attende indefinitamente un evento – l'esecuzione di un'operazione `signal()` – che può essere causato solo da uno dei processi dello stesso insieme. Quando si verifica una situazione di questo tipo si dice che i processi sono in **stallo** (*deadlocked*).

Per illustrare questo fenomeno si consideri un insieme di due processi, P_0 e P_1, ciascuno dei quali ha accesso a due semafori, `S` e `Q`, impostati al valore `1`:

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
.	.
.	.
.	.
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Si supponga che P_0 esegua `wait(S)` e quindi P_1 esegua `wait(Q)`; eseguita `wait(Q)`, P_0 deve attendere che P_1 esegua `signal(Q)`; analogamente, quando P_1 esegue `wait(S)`, deve attendere che P_0 esegua `signal(S)`. Poiché queste operazioni `signal()` non si possono eseguire, P_0 e P_1 sono in stallo.

INVERSIONE DI PRIORITÀ E IL MARS PATHFINDER

L'inversione di priorità può essere più di un semplice inconveniente nello scheduling. Su sistemi con vincoli temporali molto restrittivi (come i sistemi real-time, si veda il Capitolo 19) l'inversione di priorità può far sì che un processo non venga eseguito nei tempi richiesti. Quando ciò succede possono innescarsi errori a cascata in grado di provocare un guasto del sistema.

Si consideri il caso del Mars Pathfinder, una sonda spaziale della NASA che nel 1997 portò il robot Sojourner rover su Marte per condurre un esperimento. Poco dopo che il Sojourner ebbe iniziato il suo lavoro, cominciarono ad aver luogo numerosi reset del sistema. Ciascun reset inizializzava di nuovo sia hardware che software, inclusi gli strumenti preposti alla comunicazione. Se il problema non fosse stato risolto, il Sojourner avrebbe fallito la sua missione.

Il problema era causato dal fatto che un processo ad alta priorità, di nome "bc_dist", impiegava più tempo del dovuto a portare a termine il suo compito. Questo processo era forzatamente in attesa di una risorsa condivisa utilizzata da un processo a priorità inferiore, denominato "ASI/MET", a sua volta prelazionato da diversi processi di priorità media. Il processo "bc_dist" andava quindi in stallo in attesa della risorsa condivisa, e il processo "bc_sched", rilevando il problema, eseguiva il reset. Il Sojourner soffriva dunque di un tipico caso di inversione di priorità.

Il sistema operativo installato sul Sojourner era VxWorks (si veda il Paragrafo 19.6), che disponeva di una variabile globale per abilitare l'eriditarietà delle priorità su tutti i semafori. Dopo alcuni test, il valore della variabile del Sojourner (su Marte!) fu impostato correttamente, e il problema fu risolto.

Un resoconto completo del problema, della sua scoperta, e della sua soluzione è stato scritto dal responsabile del team software ed è disponibile all'indirizzo
http://research.microsoft.com/enus/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html.

Un insieme di processi è in stallo se ciascun processo dell'insieme attende un evento che può essere causato solo da un altro processo dell'insieme. In questo contesto si considerano principalmente gli *eventi di acquisizione e rilascio di risorse*, tuttavia anche altri tipi di eventi possono produrre situazioni di stallo (si veda il Capitolo 7, che descrive anche i meccanismi che servono ad affrontare questo tipo di problema).

Un'altra questione connessa alle situazioni di stallo è quella dell'**attesa indefinita** (nota anche col termine *starvation*). Con tale termine si definisce una situazione d'attesa indefinita nella coda di un semaforo, che si può per esempio presentare se i processi si aggiungono e si rimuovono dalla lista associata a un semaforo secondo un criterio LIFO (last-in, first-out).

6.5.4 Inversione di priorità

Nello scheduling dei processi si possono incontrare difficoltà ogniqualvolta un processo a priorità più alta abbia bisogno di leggere o modificare dati a livello kernel utilizzati da un processo, o da una catena di processi, a priorità più bassa. Visto che i dati a livello kernel sono tipicamente protetti da un lock, il processo a priorità maggiore dovrà attendere finché il processo a priorità minore non avrà finito di utilizzare le risorse. La situazione si complica ulteriormente se il processo a priorità più bassa ha dovere di prelazione su un processo a priorità più alta. Assumiamo, ad esempio, che vi siano tre processi L, M e H, le cui priorità seguono l'ordine $L < M < H$. Assumiamo che il processo H richieda la risorsa R alla quale sta accedendo il processo L. Usualmente il processo H resterebbe in attesa che L liberi la risorsa R. Supponiamo però che M diventi eseguibile, con prelazione sul processo L. Avviene indi-

rettamente che un processo con priorità più bassa, il processo *M*, influenzi il tempo che *H* attenderà in attesa della risorsa *R*.

Questo problema è noto come **inversione della priorità**. Dato che l'inversione di priorità si verifica solo su sistemi con più di due priorità, una delle soluzioni è limitare a due il numero di priorità. Tuttavia, questa soluzione non è accettabile nella maggior parte dei sistemi a uso generale. Solitamente questi sistemi risolvono il problema implementando un **protocollo di ereditarietà delle priorità**, secondo il quale tutti i processi che stanno accedendo a risorse di cui hanno bisogno processi con priorità maggiore ereditano la priorità più alta finché non finiscono di utilizzare le risorse in questione. Quando hanno terminato, la loro priorità ritorna al valore originale. Nell'esempio discusso in precedenza, un protocollo di ereditarietà delle priorità avrebbe permesso al processo *L* di ereditare temporaneamente la priorità di *H*, impedendo così al processo *M* di prelazionare la sua esecuzione. In un tale caso, una volta che il processo *H* avrà terminato con la risorsa *R*, rinuncerà alla priorità ereditata da *H* assumendo di nuovo la priorità originale. Poiché *R* sarà a questo punto disponibile, il processo *H*, e non il processo *M*, sarà il successivo processo eseguito.

6.6 Problemi tipici di sincronizzazione

In questo paragrafo s'illustrano diversi problemi di sincronizzazione come esempi di una vasta classe di problemi connessi al controllo della concorrenza. Questi problemi sono utili per verificare quasi tutte le nuove proposte di schemi di sincronizzazione. Nelle soluzioni che proponiamo ai problemi s'impiegano i semafori.

6.6.1 Produttori e consumatori con memoria limitata

Il problema dei *produttori e consumatori con memoria limitata*, trattato anche nel Paragrafo 6.1, si usa generalmente per illustrare la potenza delle primitive di sincronizzazione. In questa sede si presenta uno schema generale di soluzione, senza far riferimento a nessuna realizzazione particolare. In conclusione del capitolo proponiamo al riguardo un progetto di programmazione.

Si supponga di disporre di una certa quantità di memoria rappresentata da un buffer con *n* posizioni, ciascuna capace di contenere un elemento. Il semaforo `mutex` garantisce la mutua esclusione degli accessi al buffer ed è inizializzato al valore `1`. I semafori `vuote` e `piene` conteggiano rispettivamente il numero di posizioni vuote e il numero di posizioni piene nel buffer. Il semaforo `vuote` si inizializza al valore n; il semaforo `piene` si inizializza al valore `0`.

La Figura 6.10 riporta la struttura generale del processo produttore, la Figura 6.11 quella del processo consumatore. È interessante notare la simmetria esistente tra il produttore e il consumatore. Il codice si può interpretare nel senso di produzione, da parte del produttore, di posizioni piene per il consumatore; oppure di produzione, da parte del consumatore, di posizioni vuote per il produttore.

6.6.2 Problema dei lettori-scrittori

Si supponga che una base di dati da condividere tra numerosi processi concorrenti. Alcuni processi possono richiedere solo la lettura del contenuto dell'oggetto condiviso, mentre altri possono richiedere un aggiornamento, vale a dire una lettura e una scrittura, dello stesso oggetto. Questi due processi sono distinti, e si indicano chiamando **lettori** quelli interessati alla sola lettura e **scrittori** gli altri. Naturalmente, se due lettori accedono nello stesso mo-

```
do {
   . . .
   produce un elemento in appena_Prodotto
   . . .
   wait(vuote);
   wait(mutex);
   . . .
   inserisci in buffer l'elemento in appena_Prodotto
   . . .
   signal(mutex);
   signal(piene);
} while (true);
```

Figura 6.10 Struttura generale del processo produttore.

mento all'insieme di dati condiviso, non si ha alcun effetto negativo; viceversa, se uno scrittore e un altro processo (lettore o scrittore) accedono contemporaneamente alla stessa base di dati, ne può derivare il caos.

Per impedire l'insorgere di difficoltà di questo tipo è necessario che gli scrittori abbiano un accesso esclusivo alla base di dati condivisa. Questo problema di sincronizzazione è conosciuto come **problema dei lettori-scrittori**. Da quando tale problema fu enunciato, è stato usato per verificare quasi tutte le nuove primitive di sincronizzazione. Il problema dei lettori-scrittori ha diverse varianti, che implicano tutte l'esistenza di priorità; la più semplice, cui si fa riferimento come al *primo* problema dei lettori-scrittori, richiede che nessun lettore attenda, a meno che uno scrittore abbia già ottenuto il permesso di usare l'insieme di dati condiviso. In altre parole, nessun lettore deve attendere che altri lettori terminino l'operazione solo perché uno scrittore attende l'accesso ai dati. Il *secondo* problema dei lettori-scrittori si fonda sul presupposto che uno scrittore, una volta pronto, esegua il proprio compito di scrittura al più presto. In altre parole, se uno scrittore attende l'accesso all'insieme di dati, nessun nuovo lettore deve iniziare la lettura.

La soluzione del primo problema e quella del secondo possono condurre a uno stato d'attesa indefinita (*starvation*), degli scrittori, nel primo caso; dei lettori, nel secondo. Per questo motivo sono state proposte altre varianti. In questo paragrafo si presenta una solu-

```
do {
   wait(piene);
   wait(mutex);
   . . .
   rimuovi un elemento da buffer e mettilo in da_Consumare
   . . .
   signal(mutex);
   signal(vuote);
   . . .
   consuma l'elemento contenuto in da_Consumare
   . . .
} while (true);
```

Figura 6.11 Struttura generale del processo consumatore.

zione del primo problema dei lettori-scrittori; indicazioni attinenti a soluzioni immuni all'attesa indefinita si trovano nelle Note bibliografiche.

La soluzione del primo problema dei lettori-scrittori prevede dunque la condivisione da parte dei processi lettori delle seguenti strutture dati:

```
semaforo mutex, scrittura;
int numlettori;
```

I semafori `mutex` e `scrittura` sono inizializzati a `1`; `numlettori` è inizializzato a `0`. Il semaforo `scrittura` è comune a entrambi i tipi di processi (lettura e scrittura). Il semaforo `mutex` si usa per assicurare la mutua esclusione al momento dell'aggiornamento di `numlettori`. La variabile `numlettori` contiene il numero dei processi che stanno attualmente leggendo l'insieme di dati. Il semaforo `scrittura` funziona come semaforo di mutua esclusione per gli scrittori e serve anche al primo o all'ultimo lettore che entra o esce dalla sezione critica. Non serve, invece, ai lettori che entrano o escono mentre altri lettori si trovano nelle rispettive sezioni critiche.

La Figura 6.12 illustra la struttura generale di un processo scrittore; la Figura 6.13 presenta la struttura generale di un processo lettore. Occorre notare che se uno scrittore si trova nella sezione critica e n lettori attendono di entrarvi, si accoda un lettore a `scrittura` e $n - 1$ lettori a `mutex`. Inoltre, se uno scrittore esegue `signal(scrittura)` si può riprendere l'esecuzione dei lettori in attesa, oppure di un singolo scrittore in attesa. La scelta è fatta dallo scheduler.

Le soluzioni al problema dei lettori-scrittori sono state generalizzate su alcuni sistemi in modo da fornire **lock di lettura-scrittura**. Per acquisire un tale lock, è necessario specificare la modalità di scrittura o di lettura: se il processo desidera solo leggere i dati condivisi, richiede un lock di lettura-scrittura in modalità lettura; se invece desidera anche modificare i dati, lo richiede in modalità scrittura. È permesso a più processi di acquisire lock di lettura-scrittura in modalità lettura, ma solo un processo alla volta può avere il lock di lettura-scrittura in modalità scrittura, visto che nel caso della scrittura è necessario garantire l'accesso esclusivo.

I lock di lettura-scrittura sono massimamente utili nelle situazioni seguenti.

- Nelle applicazioni in cui è facile identificare i processi che si limitano alla lettura di dati condivisi e quelli che si limitano alla scrittura di dati condivisi.
- Nelle applicazioni che prevedono più lettori che scrittori. Infatti, i lock di lettura-scrittura comportano in genere un carico di lavoro aggiuntivo rispetto ai semafori o ai lock mutex, compensato però dalla possibilità di eseguire molti lettori in concorrenza.

```
do {
  wait(scrittura);
     . . .
     esegui l'operazione di scrittura
     . . .
  signal(scrittura);
}while (true);
```

Figura 6.12 Struttura generale di un processo scrittore.

```
do {
  wait(mutex);
  numlettori++;
  if (numlettori == 1)
    wait(scrittura);
  signal(mutex);
     . . .
     esegui l'operazione di lettura
     . . .
  wait(mutex);
  numlettori--;
  if (numlettori == 0)
    signal(scrittura);
  signal(mutex);
}while (true);
```

Figura 6.13 Struttura generale di un processo lettore.

6.6.3 Problema dei cinque filosofi

Si considerino cinque filosofi che trascorrono la loro esistenza pensando e mangiando. I filosofi condividono un tavolo rotondo circondato da cinque sedie, una per ciascun filosofo. Al centro del tavolo si trova una zuppiera colma di riso, e la tavola è apparecchiata con cinque bacchette (Figura 6.14). Quando un filosofo pensa, non interagisce con i colleghi; quando gli viene fame, tenta di prendere le bacchette più vicine: quelle che si trovano tra lui e i commensali alla sua destra e alla sua sinistra. Un filosofo può prendere una bacchetta alla volta e non può prendere una bacchetta che si trova già nelle mani di un suo vicino. Quando un filosofo affamato tiene in mano due bacchette contemporaneamente, mangia senza lasciare le bacchette. Terminato il pasto, le posa e riprende a pensare.

Il *problema dei cinque filosofi* è considerato un classico problema di sincronizzazione, non certo per la sua importanza pratica, e neanche per antipatia verso i filosofi da parte degli informatici, ma perché rappresenta una vasta classe di problemi di controllo della con-

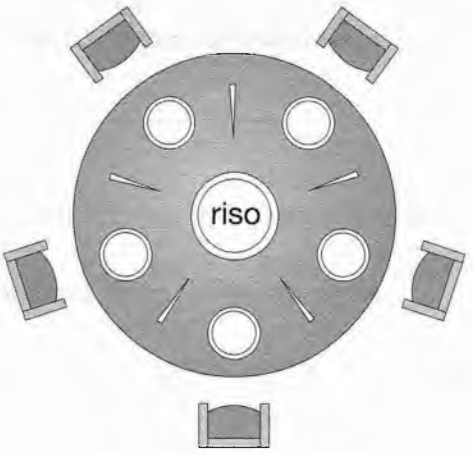

Figura 6.14 Situazione dei cinque filosofi.

```
do {
    wait(bacchetta[i]);
    wait(bacchetta[(i + 1) % 5]);

       . . .
       mangia
       . . .

    signal(bacchetta[i]);
    signal(bacchetta[(i + 1) % 5]);

       . . .
       pensa
       . . .
} while (true);
```

Figura 6.15 Struttura del filosofo *i*.

correnza, in particolare i problemi caratterizzati dalla necessità di assegnare varie risorse a diversi processi evitando situazioni di stallo e d'attesa indefinita.

Una semplice soluzione consiste nel rappresentare ogni bacchetta con un semaforo: un filosofo tenta di afferrare ciascuna bacchetta eseguendo un'operazione `wait()` su quel semaforo e la posa eseguendo operazioni `signal()` sui semafori appropriati. Quindi, i dati condivisi sono

```
semaforo bacchetta[5];
```

dove tutti gli elementi `bacchetta` sono inizializzati a `1`. La struttura del filosofo *i* è illustrata nella Figura 6.15.

Questa soluzione garantisce che due vicini non mangino contemporaneamente, ma è insufficiente poiché non esclude la possibilità che si abbia una situazione di stallo. Si supponga che tutti e cinque i filosofi abbiano fame contemporaneamente e che ciascuno tenti di afferrare la bacchetta di sinistra; tutti gli elementi di `bacchetta` diventano uguali a zero, perciò ogni filosofo che tenta di afferrare la bacchetta di destra entra in stallo. Di seguito sono elencate diverse possibili soluzioni per tali situazioni di stallo:

- solo quattro filosofi possono stare contemporaneamente a tavola;
- un filosofo può prendere le sue bacchette solo se sono entrambe disponibili (occorre notare che quest'operazione si deve eseguire in una sezione critica);
- si adotta una soluzione asimmetrica: un filosofo dispari prende prima la bacchetta di sinistra e poi quella di destra, invece un filosofo pari prende prima la bacchetta di destra e poi quella di sinistra.

Nel Paragrafo 6.7 presentiamo una soluzione al problema dei cinque filosofi che assicura l'assenza di situazioni di stallo. Si noti che qualsiasi soluzione soddisfacente per il problema dei cinque filosofi deve escludere la possibilità di situazioni d'attesa indefinita, in altre parole che uno dei filosofi muoia di fame (da qui il termine *starvation*) – una soluzione immune alle situazioni di stallo non esclude necessariamente la possibilità di situazioni d'attesa indefinita.

6.7 Monitor

Benché i semafori costituiscano un meccanismo pratico ed efficace per la sincronizzazione dei processi, il loro uso scorretto può generare errori difficili da individuare, in quanto si manifestano solo in presenza di particolari sequenze di esecuzione che non si verificano sempre.

L'impiego di contatori nell'ambito della soluzione al problema dei produttori e consumatori rappresenta un esempio di tali errori (Paragrafo 6.1). In quella circostanza, il problema di sincronizzazione appariva solo sporadicamente, e anche il valore del contatore si manteneva entro limiti ragionevoli, essendo sfasato tutt'al più di 1. Ciononostante, le soluzioni di questo tipo restano inaccettabili, ed è per ottenere soluzioni soddisfacenti che sono stati inventati i semafori.

Neanche l'uso dei semafori, purtroppo, esclude la possibilità che si verifichi qualche errore di sincronizzazione. Per capire perché, analizziamo la soluzione al problema della sezione critica. Tutti i processi condividono una variabile semaforo `mutex`, inizializzata a 1. Ogni processo deve eseguire `wait(mutex)` prima di entrare nella sezione critica e `signal(mutex)` al momento di uscirne. Se questa sequenza non è rispettata, può accadere che due processi occupino simultaneamente le rispettive sezioni critiche. Esaminiamo le difficoltà che possono insorgere. (Si noti che tali difficoltà possono insorgere anche nel caso che *un solo* processo abbia delle pecche. L'inconveniente può nascere da un involontario errore di programmazione o essere causato dalla negligenza del programmatore.)

- Supponiamo che un processo capovolga l'ordine in cui sono eseguite le istruzioni `wait()` e `signal()`, in questo modo:

```
signal(mutex);
        . . .
    sezione critica
        . . .
wait(mutex);
```

  In questa situazione, numerosi processi possono eseguire le proprie sezioni critiche allo stesso tempo, infrangendo il requisito della mutua esclusione. Questo errore può essere scoperto solo qualora diversi processi siano attivi simultaneamente nelle rispettive sezioni critiche. Si osservi che tale situazione potrebbe non essere sempre riproducibile.

- Ipotizziamo che un processo sostituisca `signal(mutex)` con `wait(mutex)`, cioè che esegua

```
wait(mutex);
        . . .
    sezione critica
        . . .
wait(mutex);
```

  Si genera, in questo caso, uno stallo (*deadlock*).

- Si supponga che un processo ometta `wait(mutex)`, `signal(mutex)`, o entrambi. In questo caso si viola la mutua esclusione oppure si genera uno stallo.

Questi esempi chiariscono come sia facile incorrere in errori allorché i programmatori utilizzino i semafori in maniera scorretta, nel tentativo di risolvere il problema delle sezioni critiche. Problemi di natura simile possono insorgere negli altri modelli di sincronizzazione, esaminati nel Paragrafo 6.6.

Per rimediare a questi errori, i ricercatori hanno sviluppato costrutti con un linguaggio ad alto livello. Un costrutto fondamentale di sincronizzazione ad alto livello – il tipo **monitor** – è descritto nel paragrafo successivo.

6.7.1 Uso del costrutto monitor

Un tipo, o tipo di dato astratto, incapsula i dati privati mettendo a disposizione dei metodi pubblici per operare su tali dati. Il tipo monitor presenta un insieme di operazioni definite dal programmatore che, all'interno del monitor, sono contraddistinte dalla mutua esclusione. Il tipo monitor contiene anche la dichiarazione delle variabili i cui valori definiscono lo stato di un'istanza del tipo, oltre ai corpi delle procedure o funzioni che operano su tali variabili. La sintassi di un monitor è mostrata nella Figura 6.16. La rappresentazione di un tipo monitor non può essere usata direttamente dai vari processi. Pertanto, una procedura definita all'interno di un monitor ha accesso unicamente alle variabili dichiarate localmente, situate nel monitor, e ai relativi parametri formali. In modo analogo, alle variabili locali di un monitor possono accedere solo le procedure locali.

Il costrutto monitor assicura che all'interno di un monitor possa essere attivo un solo processo alla volta, sicché non si deve codificare esplicitamente il vincolo di mutua esclusione (Figura 6.17). Tale definizione di monitor non è abbastanza potente per esprimere alcuni schemi di sincronizzazione, sono perciò necessari ulteriori meccanismi che, in questo caso, sono forniti dal costrutto `condition`.

```
monitor nome_monitor
{
    dichiarazioni di variabili condivise

    procedure P1 ( . . . ) {
        . . .
    }
    procedure P2 ( . . . ) {
        . . .
    }
        .
        .
        .
    procedure Pn ( . . . ) {
        . . .
    }
        codice d'inizializzazione ( . . . ) {
        . . .
    }
}
```

Figura 6.16 Sintassi di un monitor.

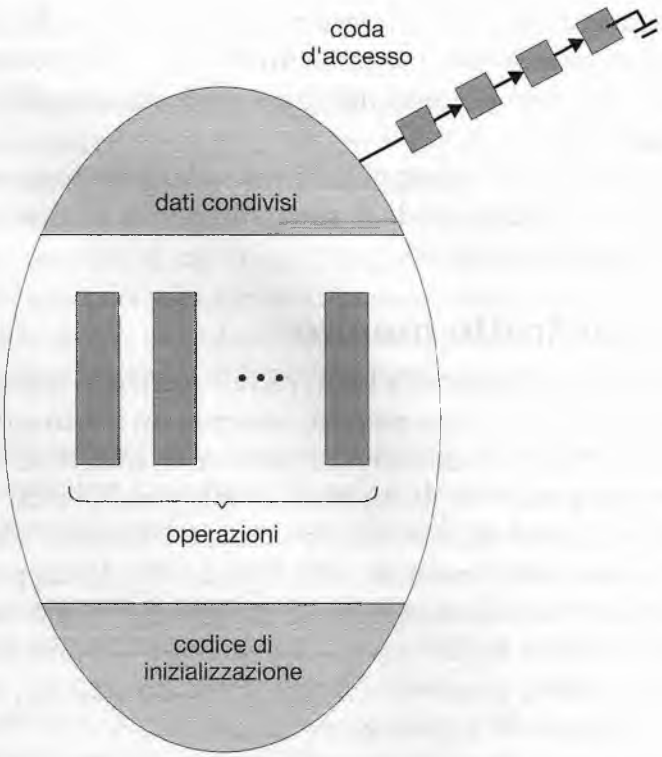

Figura 6.17 Schema di un monitor.

Un programmatore che deve scrivere un proprio schema di sincronizzazione può definire una o più variabili condizionali:

```
condition x, y;
```

Le uniche operazioni eseguibili su una variabile `condition` sono `wait()` e `signal()`. L'operazione

```
x.wait();
```

implica che il processo che la invoca rimanga sospeso finché un altro processo non invochi l'operazione

```
x.signal();
```

che risveglia esattamente un processo sospeso. Se non esistono processi sospesi l'operazione `signal()` non ha alcun effetto, vale a dire che lo stato di `x` resta immutato, come se l'operazione non fosse stata eseguita; la situazione è descritta nella Figura 6.18. Tutto ciò contrasta con l'operazione `signal()` associata ai semafori, poiché questa influisce sempre sullo stato del semaforo.

Si supponga, per esempio, che quando un processo P invoca l'operazione `x.signal()`, esista un processo sospeso Q associato alla variabile `x` di tipo `condition`. Chiaramente, se al processo sospeso Q si permette di riprendere l'esecuzione, il processo segnalante P è costretto ad attendere, altrimenti P e Q sarebbero contemporaneamente attivi all'interno del monitor. Occorre in ogni modo notare che, concettualmente, entrambi i processi possono continuare l'esecuzione. Sussistono quindi due possibilità:

1. **segnalare e attendere**. P attende che Q lasci il monitor o attenda su un'altra variabile `condition`;
2. **segnalare e proseguire**. Q attende che P lasci il monitor o attenda su un'altra variabile `condition`.

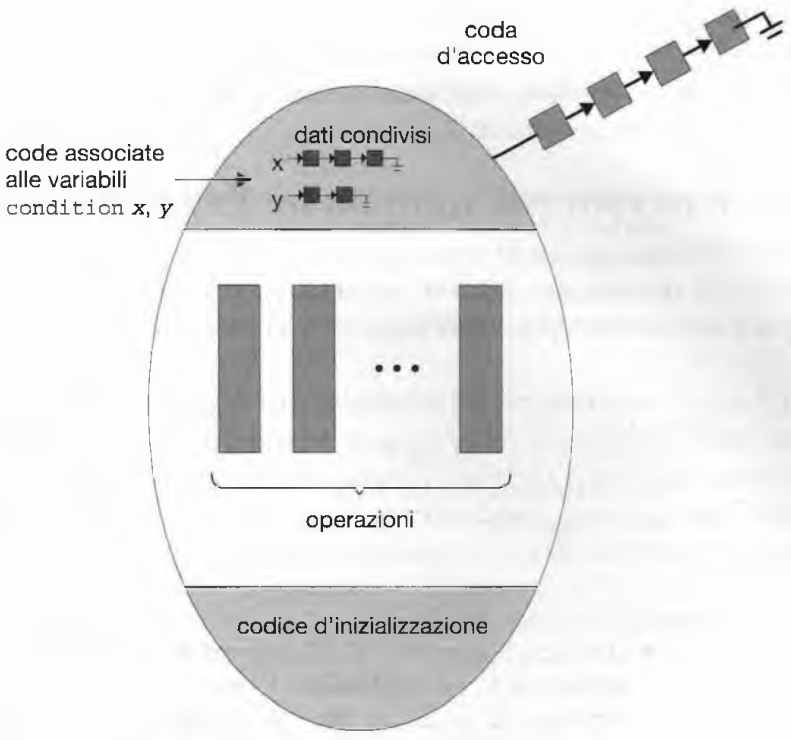

Figura 6.18 Monitor con variabili `condition`.

Si possono fornire argomenti ragionevoli a favore dell'uno o dell'altra opzione. Da un lato, visto che *P* era già in esecuzione all'interno del monitor, il secondo metodo appare più ragionevole. D'altro canto, se si lascia proseguire il thread *P*, la condizione attesa da *Q* potrebbe non valere più al momento in cui quest'ultimo riprende l'esecuzione. Il linguaggio Concurrent Pascal ha scelto un compromesso: quando il thread *P* esegue l'operazione `signal()`, lascia subito il monitor; pertanto, *Q* riprende immediatamente l'esecuzione.

Molti linguaggi di programmazione incorporano l'idea di monitor descritta in questo paragrafo. Tra questi, il Concurrent Pascal, Mesa, C# (da leggersi "C-sharp") e Java. Altri linguaggi, come Erlang, forniscono alcune tipologie di supporto alla concorrenza usando un meccanismo simile.

6.7.2 Soluzione al problema dei cinque filosofi per mezzo di monitor

Illustriamo quindi i concetti relativi al costrutto monitor presentando una soluzione esente da stallo del problema dei cinque filosofi. La soluzione impone però il vincolo che un filosofo posso prendere le sue bacchette solo quando siano entrambe disponibili. Per codificare questa soluzione si devono distinguere i tre diversi stati in cui può trovarsi un filosofo. A tale scopo si introduce la seguente struttura dati:

```
enum {pensa, affamato, mangia} stato[5];
```

Il filosofo *i* può impostare la variabile `stato[i] = mangia` solo se i suoi due vicini non stanno mangiando:

```
((stato[(i + 4) % 5] != mangia) && (stato[(i + 1) % 5] != mangia)).
```

```
monitor fc
{
  enum {pensa, affamato, mangia} stato[5];
  condition auto[5];

  prende(int i) {
    stato[i] = affamato;
    verifica(i);
    if (stato[i] != mangia)
      auto[i].wait();
  }

  posa(int i) {
    stato[i] = pensa;
    verifica((i + 4) % 5);
    verifica((i + 1) % 5);
  }

  verifica(int i) {
    if((stato[(i + 4) % 5] =! mangia) &&
      (stato[i] == affamato) &&
      (stato[(i + 1) % 5] != mangia)) {
        stato[i] = mangia;
        auto[i].signal();
    }
  }

  codice di inizializzazione() {
    for (int i = 0; i < 5; i++)
      stato[i] = pensa;
  }
}
```

Figura 6.19 Una soluzione con monitor al problema dei cinque filosofi.

Inoltre, occorre impiegare la seguente struttura dati:

```
condition auto[5];
```

dove il filosofo *i* può ritardare se stesso quando ha fame, ma non riesce a ottenere le bacchette di cui ha bisogno.

A questo punto si può descrivere la soluzione al problema dei cinque filosofi. La distribuzione delle bacchette è controllata dal monitor `fc` (Figura 6.19). Ciascun filosofo, prima di cominciare a mangiare, deve invocare l'operazione `prende()`; ciò può determinare la sospensione del processo filosofo. Completata con successo l'operazione, il filosofo può mangiare; in seguito, il filosofo invoca l'operazione `posa()` e comincia a pensare. Il filosofo *i* deve quindi chiamare le operazioni `prende()` e `posa()` nella seguente sequenza:

```
fc.prende(i);
   ...
```

È facile dimostrare che questa soluzione assicura che due vicini non mangino contemporaneamente e che non si verifichino situazioni di stallo. Occorre però notare che un filosofo può attendere indefinitamente. La soluzione di questo problema è lasciata come esercizio per il lettore.

6.7.3 Realizzazione di un monitor per mezzo di semafori

A questo punto si considera la possibilità di realizzare il meccanismo del monitor usando i semafori. A ogni monitor si associa un semaforo `mutex`, inizializzato a `1`; un processo deve eseguire `wait(mutex)` prima di entrare nel monitor, e `signal(mutex)` dopo aver lasciato il monitor.

Poiché un processo che esegue una `signal()` deve attendere finché il processo risvegliato si metta in attesa o lasci il monitor, si introduce un altro semaforo, `prossimo`, inizializzato a `0`, a cui i processi che eseguono una `signal()` possono autosospendersi. Per contare i processi sospesi al semaforo `prossimo`, si usa una variabile intera `prossimo_contatore`. Quindi, ogni procedura esterna di monitor `F` si sostituisce col seguente codice:

```
wait(mutex);
   ...
   corpo di F
   ...
if (prossimo_contatore > 0)
   signal(prossimo);
else
   signal(mutex);
```

In questo modo si assicura la mutua esclusione all'interno del monitor.

A questo punto si può descrivere la realizzazione delle variabili `condition`. Per ogni variabile `x` di tipo `condition` si introducono un semaforo `x_sem` e una variabile intera `x_contatore`, entrambi inizializzati a `0`. L'operazione `x.wait()` si può realizzare come segue:

```
x_contatore++;
if (prossimo_contatore > 0)
  signal(prossimo);
else
  signal(mutex);
wait(x_sem);
x_contatore--;
```

L'operazione `x.signal()` si può realizzare come segue:

```
if (x_contatore > 0) {
  prossimo_contatore++;
  signal(x_sem);
  wait(prossimo);
  prossimo_contatore--;
}
```

Questa soluzione è applicabile alle definizioni di monitor date da Hoare e Brinch-Hansen. In alcuni casi, tuttavia, questo livello di generalità della codifica non è necessario, e si possono apportare notevoli miglioramenti all'efficienza. La soluzione a questo problema è lasciata al lettore nell'Esercizio 6.27.

6.7.4 Ripresa dei processi all'interno di un monitor

A questo punto si discute il problema dell'ordine di ripresa dei processi all'interno di un monitor. Se più processi sono sospesi alla condizione `x`, e se qualche processo esegue l'operazione `x.signal()`, è necessario stabilire quale tra i processi sospesi si debba riattivare per primo. Una semplice soluzione consiste nell'usare un ordinamento FCFS, secondo cui il processo che attende da più tempo viene ripreso per primo. Tuttavia, in molti casi uno schema di scheduling di questo tipo non risulta adeguato; in questi casi si può usare un costrutto di **attesa condizionale** della forma

```
x.wait(c);
```

dove con `c` si indica un'espressione intera che si valuta al momento dell'esecuzione dell'operazione `wait()`. Il valore di `c`, chiamato **numero di priorità**, viene poi memorizzato col nome del processo sospeso. Quando si esegue `x.signal()`, si riprende il processo cui è associato il numero di priorità più basso.

Per comprendere questo nuovo meccanismo, si consideri il monitor illustrato nella Figura 6.20; tale monitor ha il compito di assegnare una particolare risorsa a processi in competizione. Quando richiede l'assegnazione di una delle sue risorse, ogni processo specifica il tempo massimo per il quale prevede di usare la risorsa. Il monitor assegna la risorsa al processo con la richiesta di assegnazione più breve.

```
monitor assegnazione_risorse
{
    boolean occupato;
    condition x;

    acquisizione(int tempo) {
       if (occupato)
          x.wait(tempo);
       occupato = true;
    }

    rilascio() {
       occupato = false;
       x.signal();
    }

    codice di inizializzazione() {
       occupato = false;
    }
}
```

Figura 6.20 Un monitor per l'assegnazione di una singola risorsa.

Per accedere alla risorsa in questione il processo deve rispettare la sequenza

```
R.acquisizione(t);
    ...
    accesso alla risorsa;
    ...
R.rilascio();
```

dove `R` è un'istanza di tipo `assegnazione_risorse`.

Sfortunatamente il concetto di monitor non può garantire che la precedente sequenza d'accesso sia rispettata. In particolare, può accadere quanto segue:

- un processo può accedere alla risorsa senza prima ottenere il permesso d'accesso;
- una volta che ne ha ottenuto l'accesso, un processo può non rilasciare più la risorsa;
- un processo può tentare di rilasciare una risorsa che non ha mai richiesto;
- un processo può richiedere due volte la stessa risorsa, senza rilasciarla prima della seconda richiesta.

Le stesse difficoltà si sono incontrate col costrutto della sezione critica e sono, in realtà, simili alle difficoltà che condussero allo sviluppo dei costrutti di sezione critica e di monitor. In precedenza ci si è preoccupati del corretto uso dei semafori, ora ci si deve preoccupare del corretto uso delle operazioni ad alto livello definite dal programmatore, senza poter avere, a questo livello, l'assistenza del compilatore.

Una possibile soluzione del problema precedente prevede l'inclusione delle operazioni d'accesso alle risorse all'interno del monitor `assegnazione_risorse`. Tuttavia, adottando questa soluzione, per lo scheduling delle risorse si userebbe l'algoritmo di scheduling del monitor anziché quello desiderato.

Per garantire che i processi rispettino le sequenze appropriate, è necessario controllare tutti i programmi che usano il monitor `assegnazione_risorse` e la risorsa da esso gestita. Per stabilire la correttezza del sistema è necessario verificare le seguenti due condizioni: la prima, che i processi utenti devono sempre impiegare il monitor secondo una sequenza corretta; la seconda, che è necessario assicurare che un processo non cooperante non cerchi di aggirare la mutua esclusione offerta dal monitor, e tenti di accedere direttamente alla risorsa condivisa senza usare i protocolli d'accesso. Soltanto se si assicurano queste due condizioni, si può garantire l'assenza di errori di sincronizzazione e che l'algoritmo di scheduling sia rispettato.

Questo controllo è possibile per sistemi statici di piccole dimensioni, mentre non è ragionevolmente applicabile a sistemi di grandi dimensioni o a sistemi dinamici. Questo problema di controllo dell'accesso si può risolvere solo introducendo ulteriori meccanismi, descritti nel Capitolo 14. Molti linguaggi di programmazione hanno accolto il concetto di monitor descritto in questo paragrafo, tra cui Concurrent Pascal, Mesa, C# (si pronuncia *C-sharp*) e Java, mentre altri, come Erlang, forniscono meccanismi simili di supporto alla concorrenza.

6.8 Esempi di sincronizzazione

Si descrivono ora i meccanismi di sincronizzazione forniti dai sistemi operativi Solaris, Windows XP e Linux, e le API Pthreads. Abbiamo prescelto questi tre sistemi perché offrono esempi validi di approcci differenti rispetto alla sincronizzazione del kernel; le API Pthreads

MONITOR IN JAVA

Per la sincronizzazione dei thread Java fornisce un meccanismo affine a quello del monitor. Ciascun oggetto, in Java, ha associato un singolo lock. Quando si dichiara un metodo `synchronized`, per invocare il metodo su un oggetto occorre possedere il lock dell'oggetto.

Si dichiara `synchronized` un metodo inserendo la parola chiave nella definizione del metodo. Il codice che segue, per esempio, dichiara `metodoSicuro()` come `synchronized`:

```
public class SempliceClasse {
  ...
  public synchronized void metodoSicuro() {
   /* Implementazione di metodoSicuro() */
  ...
  }
}
```

Si supponga di creare un'istanza di `SempliceClasse()`, nel modo seguente:

```
SempliceClasse sc = new SempliceClasse();
```

Per richiamare il metodo `sc.metodoSicuro()` è necessario possedere il lock dell'oggetto istanza `sc`. Se il lock è già proprietà di un thread diverso, il thread che invoca il metodo dichiarato `synchronized` si blocca ed è collocato nella lista d'attesa per il lock. La lista d'attesa è formata dall'insieme di thread che attendono la disponibilità del lock. Se, al momento dell'invocazione di un metodo `synchronized` dell'istanza, il lock è disponibile, il thread chiamante diviene il proprietario del lock dell'oggetto e può accedere al metodo. Il lock ritorna disponibile quando il thread termina l'esecuzione del metodo; un thread in lista d'attesa, quindi, è selezionato come nuovo proprietario del lock.

Java offre inoltre i metodi `wait()` e `notify()`, che funzionano analogamente alle istruzioni `wait()` e `signal()` per i monitor. Nella versione 1.5 la macchina virtuale Java comprende (tra gli altri meccanismi per la concorrenza) la API del package `java.util.concurrent` che mette a disposizione semafori, variabili condizionali e semafori mutex.

sono state incluse nella trattazione perché ampiamente utilizzate dagli sviluppatori per la creazione e la sincronizzazione dei thread su piattaforme UNIX e Linux. Come si vedrà nel corso del paragrafo i metodi di sincronizzazione messi a disposizione dai tre differenti sistemi variano in modo sottile ma significativo.

6.8.1 Sincronizzazione in Solaris

Per regolare l'accesso alle sezioni critiche, Solaris mette a disposizione semafori mutex adattivi, variabili condizionali, semafori, lock di lettura-scrittura e i cosiddetti tornelli (*turnstiles*). Si vedano i Paragrafi 6.5 e 6.6 per la descrizione di questi argomenti.

Un **mutex adattivo** (*adaptive mutex*) protegge l'accesso a ogni elemento critico di dati; in un sistema multiprocessore si attiva come un semaforo ordinario realizzato come uno spinlock. Se i dati sono soggetti a lock e quindi già in uso, nel mutex adattivo si possono verificare due situazioni: se i dati sono posseduti da un thread correntemente in esecuzione in un'altra unità d'elaborazione, il thread che ha fatto la nuova richiesta d'accesso entra in uno stato d'attesa attiva (*spinlock*), mentre aspetta la rimozione del lock, poiché è probabile che il thread in possesso dei dati termini la propria elaborazione in breve tempo; viceversa, se quest'ultimo non si trova nello stato d'esecuzione, il thread richiedente si sospende nello sta-

to d'attesa fino alla rimozione del lock. In questo modo si evita il ciclo d'attesa attiva, poiché probabilmente i dati non saranno rilasciati in un tempo ragionevolmente breve. Si ha questa situazione, per esempio, quando il thread che ha bloccato l'accesso ai dati è a sua volta nello stato d'attesa. Poiché un sistema dotato di una singola CPU può eseguire un solo thread alla volta, il thread che possiede il lock non è mai in esecuzione nell'istante in cui un altro thread verifica la presenza del lock. Quindi, in un sistema con una CPU, un thread che incontra un lock sospende la propria esecuzione anziché persistere nel ciclo d'attesa attiva.

Solaris adotta il metodo del mutex adattivo per proteggere soltanto i dati cui si accede da segmenti di codice molto corti; in pratica si usa solo se un lock permane per meno di qualche centinaio di istruzioni. Se il segmento di codice fosse più lungo, l'uso dei cicli d'attesa sarebbe inefficiente. Per segmenti di codice più lunghi il sistema ricorre all'impiego di semafori o variabili condizionali. Se l'oggetto richiesto è già posseduto da altri thread, il thread richiedente invoca una `wait()` e si sospende. Quando il thread che possiede il dato ne rilascia il controllo, invia un segnale al successivo thread presente nella coda d'attesa per quel dato. L'ulteriore costo richiesto dalla sospensione e dalla successiva riattivazione del thread, compresi i relativi cambi di contesto, è sicuramente minore di quello dovuto alle parecchie centinaia di istruzioni che si sprecherebbero nel ciclo d'attesa del semaforo ad attesa attiva.

I lock di lettura-scrittura si usano per proteggere i dati cui si accede spesso, e di solito per la sola lettura. In tali circostanze essi sono più efficienti dei semafori, poiché più thread possono leggere i dati in modo concorrente, mentre un semaforo avrebbe imposto la serializzazione di questi accessi. Poiché la sua realizzazione introduce un costo aggiuntivo, anche i lock di lettura-scrittura si applicano solo alle sezioni di codice lunghe.

Solaris utilizza i cosidetti tornelli per ordinare la lista dei thread che attendono di ottenere un mutex adattivo o un lock di lettura-scrittura. Un **tornello** (*turnstile*) è una struttura a coda contenente i thread che attendono il rilascio di un lock. Ad esempio, se un thread possiede il lock di un oggetto sincronizzato, tutti gli altri thread che cercano di acquisirlo si bloccano ed entrano nel tornello relativo a quel lock. Quando il lock è rimosso, il kernel seleziona un thread dal tornello per concedergli la proprietà del lock. Ogni oggetto sincronizzato con almeno un thread che attende di acquisire il lock richiede un tornello separato. Tuttavia, anziché associare un tornello a ciascun oggetto sincronizzato, il sistema operativo assegna un tornello a ogni thread a livello kernel.

Il tornello del primo thread che si blocca su un oggetto sincronizzato diventa il tornello per l'oggetto stesso, i thread successivi si aggiungono allo stesso tornello. Quando il thread iniziale infine rilascia il lock, acquisisce un nuovo tornello da una lista di tornelli liberi mantenuta dal kernel. Per prevenire un'**inversione delle priorità**, i tornelli sono organizzati secondo un **protocollo di ereditarietà delle priorità**. Ciò significa che, se un thread detiene il lock di un oggetto di cui è in attesa un thread a priorità maggiore, il thread a priorità minore eredita temporaneamente la priorità del thread a priorità maggiore. Al rilascio del lock, il thread riassume la priorità originaria.

Si noti che i meccanismi di gestione dei lock usati dal kernel sono disponibili anche per i thread a livello utente, sicché gli stessi tipi di lock sono disponibili sia all'interno sia all'esterno del kernel. Una differenza cruciale nella loro realizzazione è il protocollo di ereditarietà delle priorità: i lock del kernel adottano i metodi di ereditarietà delle priorità del kernel usati dallo scheduler (Paragrafo 19.4); quelli dei thread a livello utente non offrono questa funzionalità.

Per ottimizzare le prestazioni di Solaris, gli sviluppatori hanno via via perfezionato e calibrato l'implementazione dei lock. Poiché i lock si usano frequentemente, e spesso per funzioni cruciali del kernel, la loro ottimizzazione può condurre a notevoli incrementi delle prestazioni.

6.8.2 Sincronizzazione in Windows XP

Il sistema operativo Windows XP ha un kernel multithread che offre anche la gestione di applicazioni per le elaborazioni in tempo reale e di architetture multiprocessore. Quando il kernel di Windows XP accede a una risorsa globale in un sistema con singola CPU, disabilita temporaneamente le interruzioni aventi procedure di gestione che potrebbero accedere alla stessa risorsa globale. In un sistema multiprocessore, si protegge l'accesso alle risorse globali con i semafori ad attesa attiva (*spinlock*). Proprio come nel caso di Solaris, il kernel usa i semafori ad attesa attiva solo per proteggere segmenti di codice brevi. Inoltre, per ragioni di efficienza, il kernel impedisce che un thread sia sottoposto a prelazione mentre detiene un semaforo ad attesa attiva.

Per la sincronizzazione fuori dal kernel, il sistema operativo offre gli **oggetti dispatcher**, che permettono ai thread di sincronizzarsi servendosi di diversi meccanismi, inclusi mutex, semafori, eventi e timer. I dati condivisi si possono proteggere richiedendo che un thread entri in possesso di un mutex prima di potervi accedere, e rilasci il mutex al completamento dell'elaborazione di quei dati. Il comportamento dei semafori è illustrato nel Paragrafo 6.5. Gli **eventi** sono un meccanismo di sincronizzazione utilizzabile in modo simile alle variabili condizionali; cioè, possono notificare il verificarsi di una determinata condizione a un thread che l'attendeva. Infine i timer sono usati per informare un thread (o più di uno) della scadenza di uno specifico periodo di tempo.

Gli oggetti dispatcher possono essere nello stato *signaled* o nello stato *nonsignaled*. Uno **stato signaled** indica che l'oggetto è disponibile e che un thread che tentasse di accedere all'oggetto non sarebbe bloccato; uno **stato nonsignaled** indica che l'oggetto non è disponibile e che qualsiasi thread che tentasse di accedervi sarebbe bloccato. La Figura 6.21 illustra le transizioni di stato di un oggetto dispatcher di tipo lock mutex.

C'è una relazione tra lo stato di un *oggetto dispatcher* e lo stato di un thread. Se un thread si blocca a un *oggetto dispatcher* nello stato *nonsignaled*, il suo stato cambia da pronto per l'esecuzione ad attesa, e il thread viene messo nella coda d'attesa per quell'oggetto. Quando lo stato dell'*oggetto dispatcher* diventa *signaled*, il kernel verifica che non ci sia alcun thread che attende l'oggetto, altrimenti, ne fa passare uno, o più d'uno, dallo stato di attesa allo stato di pronto per l'esecuzione, dal quale può riprendere l'esecuzione. Il numero dei thread che il kernel seleziona dalla coda d'attesa dipende dal tipo di *oggetto dispatcher* al quale attendono. Il kernel selezionerebbe un solo thread dalla coda d'attesa nel caso di un mutex, poiché un oggetto mutex può essere posseduto da un solo thread. Nel caso di un oggetto evento, il kernel selezionerebbe tutti i thread che attendono l'evento stesso.

Consideriamo un lock mutex come esempio per illustrare gli *oggetti dispatcher* e gli stati dei thread. Se cercasse di acquisire un *oggetto dispatcher* di tipo mutex che sia nello stato *nonsignaled*, un thread sarebbe sospeso e messo in una coda d'attesa per l'oggetto mutex. Se il mutex passasse allo stato *signaled* (il risultato del rilascio del lock mutex da parte di un al-

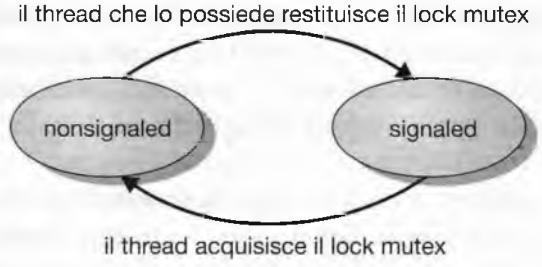

Figura 6.21 Oggetto dispatcher di tipo mutex.

tro thread), il thread in attesa in testa alla coda del mutex passerebbe dallo stato d'attesa allo stato di pronto per l'esecuzione e acquisirebbe il mutex.

Il progetto di programmazione rispetto al problema del produttore/consumatore alla fine di questo capitolo è espressamente dedicato ai lock mutex e ai semafori nella API di Win32.

6.8.3 Sincronizzazione dei processi in Linux

Prima della versione 2.6, Linux adoperava un kernel senza prelazione; ciò significa che non consentiva di applicare la prelazione ai processi eseguiti in modalità di sistema – neppure nel caso che processi con priorità più alta fossero pronti per l'esecuzione. Ora, per contro, il kernel di Linux ha adottato compiutamente il procedimento della prelazione, cosicché i task attivi nel kernel possono essere sottoposti a prelazione.

Il kernel di Linux si serve di spinlock e semafori (nonché della variante lettore-scrittore di questi due meccanismi) per implementare i lock a livello kernel. Sulle macchine SMP, il meccanismo fondamentale è lo spinlock; il kernel è progettato in modo da mantenere attivi gli spinlock solo per brevi periodi di tempo. Sulle macchine monoprocessore, gli spinlock sono inadatti, e si ricorre all'abilitazione e inibizione del diritto di prelazione nel kernel. Su tali macchine, in pratica, anziché attivare uno spinlock, il kernel inibisce la prelazione; anziché rimuovere lo spinlock, abilita la prelazione. In sintesi:

monoprocessore	multiprocessore
Inibisce la prelazione a livello kernel.	Attiva spinlock.
Abilita la prelazione a livello kernel.	Rimuove spinlock.

Il modo impiegato da Linux per abilitare e inibire il diritto di prelazione nel kernel è di particolare interesse; si basa su due semplici chiamate di sistema, `preempt_disable()` e `preempt_enable()`. Va detto che non è possibile sottoporre il kernel a prelazione se un task attivo nella modalità di sistema possiede un lock. Per aggirare l'ostacolo, ogni task nel sistema possiede una struttura, `thread_info`, in cui un contatore, `preempt_count`, indica il numero dei lock attivi nel sistema. Quando un lock entra in funzione, `preempt_count` aumenta di uno, mentre diminuisce di uno quando ne viene rimosso. Qualora il valore di `preempt_count` per il task in esecuzione sia maggiore di zero, sarebbe rischioso sottoporre a prelazione il kernel, dato che il task possiede un lock. Se il valore è zero, il kernel può subire l'interruzione (assumendo che non vi siano chiamate in sospeso a `preempt_disable()`).

Gli spinlock, insieme all'abilitazione e inibizione della prelazione, sono utilizzati nel kernel solo quando si ricorre per breve tempo a un lock (o all'inibizione della prelazione del kernel). Quando vi sia necessità di mantenere un lock attivo più a lungo, è opportuno utilizzare i semafori.

6.8.4 Sincronizzazione in Pthreads

La API Pthreads fornisce lock mutex, variabili condizionali e lock di lettura-scrittura per la sincronizzazione dei thread; è a disposizione dei programmatori e non fa parte di questo o quel kernel. I lock mutex rappresentano, in ambiente Pthreads, la tecnica di sincronizzazione fondamentale. La loro finalità è di proteggere le sezioni critiche del codice: un thread che sia in procinto di entrare in una sezione critica si appropria del lock, quindi, al momento di uscirne, lo rimuove. Le variabili condizionali si comportano in Pthreads in maniera molto

simile a quanto descritto nel Paragrafo 6.7. I lock di lettura-scrittura hanno un funzionamento analogo al meccanismo descritto al Paragrafo 6.6.2. Molti sistemi predisposti per l'uso di Pthreads, inoltre, offrono semafori, sebbene non rientrino nello standard Pthreads; appartengono, invece, all'estensione POSIX SEM. Tra le altre estensioni della API Pthreads figurano gli spinlock, malgrado non tutte le estensioni siano ritenute portabili da un'implementazione a un'altra. Vedremo alla fine del capitolo, nel progetto di programmazione, come impiegare lock mutex e semafori in Pthreads.

6.9 Transazioni atomiche

La mutua esclusione nelle sezioni critiche assicura che siano eseguite in modo atomico. In altri termini, se due sezioni critiche sono eseguite in modo concorrente, il risultato che si ottiene coincide esattamente con quello che si otterrebbe dall'esecuzione sequenziale delle due sezioni critiche in un qualsiasi ordine. Pur essendo utile in molti ambiti, questa proprietà non è più sufficiente nei molti casi in cui si vuole avere la certezza che una sezione critica costituisca una singola unità logica di lavoro, caratterizzata dal fatto di essere eseguita nella sua totalità o non essere eseguita per niente. Un tipico esempio è costituito dal trasferimento di fondi; un'operazione che comporta l'addebito della valuta da un conto e il contemporaneo accredito su un altro. Chiaramente, affinché i dati mantengano la loro coerenza, o si portano a termine con successo entrambe le operazioni oppure non devono avvenire né l'addebito né l'accredito.

La discussione contenuta nel resto del paragrafo è strettamente correlata alla gestione delle **basi di dati**, a cui sono correlati i problemi di archiviazione, recupero e coerenza dei dati. Recentemente si è riscontrato un notevole aumento d'interesse circa l'uso di tecniche proprie delle basi di dati all'interno dei sistemi operativi. I sistemi operativi si possono considerare sistemi per la manipolazione di dati; in questa veste, possono sicuramente trarre vantaggio da tecniche e modelli ottenuti nella ricerca sulle basi di dati. Ad esempio, la flessibilità e la potenza di molti metodi per la gestione dei file usati nei sistemi operativi potrebbero migliorare se si sostituissero con metodi più formali propri delle basi di dati. Dal Paragrafo 6.9.2 al Paragrafo 6.9.4 saranno descritte alcune di queste tecniche e verrà illustrato il loro impiego da parte dei sistemi operativi. Prima di tutto, però, ci occupiamo dell'atomicità delle transazioni in senso generale, il concetto centrale per le tecniche di gestione delle basi di dati.

6.9.1 Modello di sistema

Un insieme di istruzioni (operazioni) che esegue una singola funzione logica prende il nome di **transazione**. Uno dei principali motivi per cui si fa ricorso alle transazioni è conservare l'atomicità malgrado la possibilità che si verifichino situazioni anomale all'interno del sistema.

Una transazione è un'unità di programma che accede a elementi contenuti in file residenti nella memoria secondaria ed eventualmente li aggiorna. Per gli scopi della presente trattazione è sufficiente considerare una transazione come una sequenza di operazioni `read` e `write`, terminate da un'operazione `commit` o da `abort`. L'operazione `commit` indica che la transazione è terminata con successo, mentre l'operazione `abort` significa che la transazione è fallita, a causa di qualche errore logico o di un guasto del sistema.

In entrambi i casi, poiché una transazione fallita può aver già alterato i dati ai quali ha avuto accesso, il loro stato potrebbe non coincidere con quello in cui si sarebbero trovati se la transazione fosse stata eseguita in modo atomico. Al fine di assicurare la proprietà di ato-

MEMORIA TRANSAZIONALE

Con l'emergere dei sistemi multicore si è visto un incremento della pressione per lo sviluppo di applicazioni multithread che traggano vantaggio dalle unità di calcolo multiple. Tuttavia, le applicazioni multithread presentato un rischio maggiore di race condition e situazioni di stallo. Per risolvere questi problemi sono state usate tecniche tradizionali come lock, semafori e monitor. Le **memorie transazionali** forniscono una strategia alternativa per lo sviluppo di applicazioni concorrenti sicure.

Una **transazione di memoria** è una sequenza di operazioni atomiche di lettura-scrittura. Se tutte le operazioni di una transazione sono eseguite, la transazione di memoria viene completata, altrimenti le operazioni devono essere annullate e deve essere ripristinata la situazione precedente l'inizio della transazione. I vantaggi della memoria transazionale possono essere sfruttati mediante l'aggiunta di nuove funzionalità a un linguaggio di programmazione.

Si consideri il seguente esempio. Si supponga di avere a disposizione una funzione `update()` che modifica dati condivisi. Questa funzione potrebbe essere scritta in maniera tradizionale usando i lock:

```
update () {
   acquire();
   /* modifica dati condivisi */
   release();
}
```

Tuttavia, l'utilizzo di meccanismi di sincronizzazione come lock e semafori implica molti potenziali problemi, incluso lo stallo dei processi. Inoltre, in seguito alla crescita del numero di thread, i meccanismi tradizionali basati sui lock non si adattano al meglio.

Un'alternativa consiste nell'aggiungere ai linguaggi di programmazione nuove funzionalità che sfruttano il vantaggio dato dalla memoria transazionale. Nel nostro esempio, supponiamo di aggiungere il costrutto `atomic{S}` che assicura che le operazioni in `S` siano eseguite come transazione. Potremo riscrivere il metodo `update()` così:

```
update () {
   atomic {
   /* modifica dati condivisi */
   }
}
```

Il vantaggio di utilizzare tale meccanismo al posto dei lock sta nel fatto che è il sistema di memoria transazionale, e non il programmatore, a garantire l'atomicità. Inoltre, il sistema è in grado di identificare le istruzioni nei blocchi atomici che possono essere eseguite in concorrenza, come ad esempio accessi concorrenti in lettura a una variabile condivisa. È comunque certamente possibile per un programmatore identificare queste situazioni e utilizzare i lock di lettura-scrittura, ma il compito si complica al crescere del numero di thread in un'applicazione.

La memoria transazionale può essere implementata via software oppure via hardware. La memoria transazionale software (STM), come il nome suggerisce, implementa la memoria transazionale esclusivamente via software, senza la necessità di hardware particolare. In questo schema si inserisce del codice ausiliario nelle transazioni. Esso è prodotto e inserito da un compilatore e gestisce ciascuna transazione esaminando quali istruzioni possono essere eseguite in concorrenza, e quando sono necessari dei lock a basso livello. La memoria transazionale hardware (HTM piccola) utilizza gerarchie di cache hardware e protocolli di coerenza della cache per gestire e risolvere conflitti riguardanti dati condivisi residenti in memorie cache di processori distinti. HTM non richiede una particolare strumentazione software e ha quindi un minor overhead rispetto a STM. Tuttavia, richiede che le gerarchie di cache esistenti e i protocolli di coerenza della cache siano modificati per il supporto di memorie transazionali.

Le memorie transazionali esistono da diversi anni, ma per diverso tempo non hanno conosciuto una vasta diffusione. Soltanto di recente il crescente utilizzo dei sistemi multicore e la maggior enfasi sulla programmazione concorrente hanno focalizzato molta ricerca su questo settore, sia da parte delle istituzioni accademiche sia da parte dei produttori di hardware, inclusi Intel e Sun Microsystems.

micità, la terminazione anomala di una transazione non deve produrre alcun effetto sullo stato dei dati che questa ha già modificato.

Quindi, è necessario ripristinare lo stato dei dati adoperati dalla transazione fallita, riportandolo a quello che li caratterizzava appena prima dell'inizio della transazione (**roll back**). Il rispetto di questa proprietà deve essere garantito dal sistema.

Per stabilire il modo in cui un sistema deve garantire l'atomicità delle proprie transazioni, è necessario identificare le proprietà dei dispositivi che si usano per memorizzare i dati ai quali esse accedono. I diversi tipi di dispositivi di memorizzazione si possono caratterizzare secondo capacità, velocità e attitudine al recupero dai guasti.

- **Memorie volatili.** Le informazioni registrate nelle memorie volatili, per esempio la memoria centrale o le cache, di solito non sopravvivono ai crolli del sistema. L'accesso a questo tipo di dispositivi è molto rapido, grazie sia alla velocità intrinseca degli accessi alla memoria sia alla possibilità di accedere in modo diretto ai dati.
- **Memorie non volatili.** Le informazioni registrate in memorie non volatili, per esempio dischi e nastri magnetici, di solito sopravvivono ai crolli del sistema. I dischi sono più affidabili della memoria centrale, ma meno dei nastri magnetici. Sia i dischi sia i nastri sono soggetti a guasti che possono causare anche la perdita dei dati. Poiché dischi e nastri sono dispositivi elettromeccanici che richiedono movimenti fisici per accedere ai dati, attualmente i tempi d'accesso alle memorie non volatili superano di diversi ordini di grandezza quelli alle memorie volatili.
- **Memorie stabili.** Le informazioni contenute nelle memorie stabili per definizione non si perdono *mai* (data l'impossibilità teorica di garantire questa proprietà, è però indispensabile considerare quest'affermazione con una certa cautela). Per realizzare un'approssimazione di questi dispositivi è necessario duplicare le informazioni in più memorie non volatili (di solito dischi) con tipi di guasto indipendenti, e aggiornare i dati in maniera controllata (si veda in proposito il Paragrafo 12.8).

Ci limitiamo qui a descrivere come sia possibile assicurare l'atomicità di una transazione in un ambiente in cui eventuali guasti comporterebbero la perdita delle informazioni contenute in memorie volatili.

6.9.2 Ripristino basato sulla registrazione delle modifiche

Un modo per assicurare l'atomicità è registrare in memorie stabili le informazioni che descrivono tutte le modifiche che la transazione ha apportato ai dati a cui ha avuto accesso. In tal senso, il metodo più largamente usato è quello della **registrazione con scrittura anticipata** (*write-ahead logging*); il sistema mantiene, nella memoria stabile, una struttura dati chiamata **log**, in cui ciascun elemento descrive una singola operazione `write` eseguita dalla transazione ed è composto dei seguenti campi:

- **nome della transazione.** Il nome, unico, della transazione che ha richiesto l'operazione `write`;
- **nome del dato modificato.** Il nome, unico, del dato scritto dall'operazione `write`;
- **valore precedente.** Il valore posseduto dall'elemento prima dell'operazione `write`;
- **nuovo valore.** Il valore che l'elemento avrà una volta terminata l'operazione `write`.

Esistono altri elementi speciali del log, usati per registrare gli eventi significativi che si possono verificare durante l'elaborazione della transazione, per esempio avvio, successo o fallimento della transazione.

Prima dell'avvio di una transazione T_i si registra l'elemento <T_i `start`> nel log; durante l'esecuzione, ciascuna operazione `write` di T_i è *preceduta* dalla scrittura dell'apposito nuovo elemento nel log. Il successo di T_i è sancito dalla registrazione dell'elemento <T_i `commit`> nel log.

Poiché le informazioni contenute nel log servono per la ricostruzione dello stato delle strutture dati alle quali le diverse transazioni hanno avuto accesso, non è ammissibile che l'effettivo aggiornamento di un componente di queste strutture avvenga prima che il corrispondente elemento del log sia stato registrato nell'apposito dispositivo di memoria stabile. Quindi, il requisito fondamentale affinché questo metodo abbia successo è che l'elemento relativo a un componente `x` sia registrato nella memoria stabile prima dell'esecuzione dell'operazione `write(x)`.

Naturalmente, questo metodo determina una penalizzazione delle prestazioni, poiché ogni `write` logica richiede in realtà l'esecuzione di due `write` fisiche. Inoltre, aumenta la quantità di memoria secondaria usata poiché, oltre allo spazio necessario al contenimento dei dati, si deve prevedere lo spazio necessario al log. In situazioni in cui i dati sono molto importanti e si deve disporre di una rapida funzione di ripristino, tale onere merita di essere sostenuto.

Mediante l'uso dei log il sistema può gestire qualsiasi malfunzionamento, purché non sia una perdita delle informazioni contenute nella memoria non volatile. L'algoritmo di ripristino impiega le due seguenti procedure:

- `undo(`T_i`)`, per ripristinare il precedente valore di tutti i dati modificati dalla transazione T_i;
- `redo(`T_i`)`, per assegnare il nuovo valore a tutti i dati modificati dalla transazione T_i.

L'insieme dei dati aggiornati da T_i, con i valori vecchi e nuovi associati, è reperibile nel log.

Per garantirne un corretto comportamento anche se si dovesse verificare un guasto durante il procedimento di ripristino, le operazioni `undo` (annullamento) e `redo` (ripetizione) devono essere idempotenti (in altri termini, le esecuzioni multiple di un'operazione devono produrre i medesimi risultati di una singola esecuzione).

Se una transazione T_i termina in modo anormale (fallisce), per ripristinare lo stato dei dati modificati è sufficiente eseguire l'operazione `undo(`T_i`)`. Se si verifica un guasto nel sistema, il ripristino di uno stato corretto comporta la consultazione del log per determinare quali transazioni annullare e quali ripetere. Tale classificazione delle transazioni avviene secondo i seguenti parametri:

- la transazione T_i deve essere annullata se il log contiene l'elemento <T_i `start`> ma non l'elemento <T_i `commit`>;
- la transazione T_i deve essere ripetuta se il log contiene sia l'elemento <T_i `start`> sia l'elemento <T_i `commit`>.

6.9.3 Punti di verifica

Quando si verifica un guasto nel sistema è necessario consultare il log per determinare quali transazioni annullare e quali ripetere. Questo metodo richiede la scansione dell'intero log e presenta principalmente due inconvenienti.

1. La ricerca può richiedere un tempo piuttosto lungo.
2. La maggior parte delle transazioni, che secondo questo algoritmo deve essere ripetuta, ha già aggiornato con successo dati che, secondo il log, si dovrebbero ancora modificare. Benché la ripetizione delle modifiche non causi alcun danno (per l'idempotenza dell'operazione `redo`), il processo di ripristino richiederà senza dubbio più tempo.

Per ridurre questo genere di sprechi si introduce il concetto di **punto di verifica** (*checkpoint*). Durante l'esecuzione il sistema esegue la registrazione con scrittura anticipata e registrazioni periodiche, che costituiscono ciascun punto di verifica, definite dall'esecuzione della seguente sequenza di azioni:

1. registrazione in una memoria stabile di tutti gli elementi del log correntemente residenti in memorie volatili (di solito la memoria centrale);
2. registrazione in una memoria stabile di tutti i dati modificati correntemente residenti in memorie volatili;
3. registrazione dell'elemento `<checkpoint>` nel log residente nella memoria stabile.

La presenza dell'elemento `<checkpoint>` consente al sistema di rendere più efficiente la propria procedura di ripristino. Si consideri una transazione T_i terminata con successo (*committed*) prima dell'ultimo punto di verifica; l'elemento $<T_i$ `commit>` precede nel log l'elemento `<checkpoint>`; quindi, qualsiasi modifica apportata da T_i è stata sicuramente registrata nella memoria stabile prima del punto di verifica o come parte del punto di verifica stesso; quindi durante un eventuale ripristino sarebbe inutile eseguire l'operazione `redo` sulla transazione T_i.

Quest'osservazione consente di migliorare il precedente algoritmo di ripristino: in seguito a un malfunzionamento, la procedura di ripristino esamina il log per determinare la più recente transazione T_i che ha iniziato la propria esecuzione prima del più recente punto di verifica; scandisce a ritroso il log fino al primo elemento `<checkpoint>` e quindi fino al primo elemento $<T_i$ `start>`.

Una volta identificata la transazione T_i, le operazioni `redo` e `undo` si devono applicare solamente a T_i e a tutte le transazioni T_j cominciate dopo T_i, ignorando il resto del log. Detto T l'insieme di queste transazioni, le operazioni necessarie al completamento del ripristino sono le seguenti:

- esecuzione dell'operazione `redo(`T_k`)` per tutte le transazioni T_k appartenenti a T tali che il log contiene l'elemento $<T_k$ `commit>`;
- esecuzione dell'operazione `undo(`T_k`)` per tutte le transazioni T_k appartenenti a T tali che il log non contiene l'elemento $<T_k$ `commit>`.

6.9.4 Transazioni atomiche concorrenti

Abbiamo finora considerato un ambiente in cui sia eseguibile solo una transazione per volta. Passiamo ora al caso in cui transazioni multiple siano attive concorrentemente. Poiché le transazioni sono atomiche, il risultato dell'esecuzione concorrente di più transazioni deve essere equivalente a quello che si otterrebbe eseguendo le transazioni in una sequenza arbitraria. Questa caratteristica di **serializzabilità** si può rispettare semplicemente eseguendo ciascuna transazione all'interno di una sezione critica. In altre parole, tutte le transazioni condividono un semaforo `mutex` inizializzato a `1`; la prima operazione che una transazione esegue è `wait(mutex)`, mentre l'ultima operazione che si esegue dopo il successo o il fallimento della transazione è `signal(mutex)`.

Questo schema assicura l'atomicità di tutte le transazioni che si eseguono in modo concorrente, ma è troppo restrittivo; in molte situazioni si può consentire la sovrapposizione dell'esecuzione di più transazioni, pur rispettando la proprietà di serializzabilità; nel seguito si descrivono alcuni **algoritmi di controllo della concorrenza** che assicurano la serializzabilità.

T_0	T_1
`read(`A`)`	
`write(`A`)`	
`read(`B`)`	
`write(`B`)`	
	`read(`A`)`
	`write(`A`)`
	`read(`B`)`
	`write(`B`)`

Figura 6.22 Sequenza d'esecuzione 1: sequenza d'esecuzione seriale in cui T_0 è seguita da T_1.

6.9.4.1 Serializzabilità

Si consideri un sistema contenente due oggetti A e B, entrambi letti e modificati da due transazioni T_0 e T_1, e si supponga che le due transazioni siano eseguite in modo atomico, nell'ordine T_0, T_1. Questa **sequenza d'esecuzione** (*schedule*) è rappresentata nella Figura 6.22. Nella sequenza d'esecuzione 1 le istruzioni rispettano un ordinamento cronologico dall'alto verso il basso, con le istruzioni di T_0 disposte nella colonna di sinistra e quelle di T_1 nella colonna di destra.

Una sequenza d'esecuzione in cui ciascuna transazione sia eseguita in modo atomico si chiama **sequenza d'esecuzione seriale**; è composta da una sequenza di istruzioni appartenenti a transazioni diverse, caratterizzata dal fatto che tutte le istruzioni appartenenti a una singola transazione sono raggruppate. Quindi, dato un insieme di n transazioni, esistono $n!$ differenti sequenze d'esecuzione seriali valide. Ogni sequenza d'esecuzione seriale è valida, poiché equivale all'esecuzione atomica in un ordine qualsiasi delle transazioni in essa contenute.

Se si consente a due transazioni di sovrapporre le proprie esecuzioni, la sequenza d'esecuzione risultante non è più seriale. Il che non implica necessariamente che l'esecuzione risultante sia scorretta (cioè non equivalente a una sequenza d'esecuzione seriale). Per dimostrare quest'affermazione è necessario definire la nozione di **operazioni conflittuali**.

Si consideri una sequenza d'esecuzione S in cui compaiono in successione due operazioni O_i e O_j appartenenti rispettivamente alle transazioni T_i e T_j. Se tentano di accedere agli stessi dati e se almeno una tra le due è una `write`, le operazioni O_i e O_j sono *conflittuali*. Al fine di illustrare meglio il concetto di operazioni conflittuali, si consideri la sequenza d'esecuzione 2, non seriale, nella Figura 6.23. L'operazione `write(`A`)` di T_0 è in conflitto con

T_0	T_1
`read(`A`)`	
`write(`A`)`	
	`read(`A`)`
	`write(`A`)`
`read(`B`)`	
`write(`B`)`	
	`read(`B`)`
	`write(`B`)`

Figura 6.23 Sequenza d'esecuzione 2: sequenza d'esecuzione concorrente serializzabile.

`read(`A`)` di T_1; viceversa, l'operazione `write(`A`)` di T_1 non è in conflitto con `read(`B`)` di T_0, poiché le due operazioni accedono a elementi diversi.

Si supponga che O_i e O_j siano due operazioni concatenate all'interno di una sequenza d'esecuzione S; se O_i e O_j appartengono a transazioni diverse e non conflittuali si può invertirne l'ordine d'esecuzione, producendo una nuova sequenza d'esecuzione S'. Ci si aspetta che S ed S' siano equivalenti, poiché tutte le operazioni rispettano il medesimo ordine in entrambe le sequenze d'esecuzione, a eccezione delle sole O_i e O_j il cui ordine non è rilevante ai fini del risultato finale.

Per illustrare meglio il concetto di inversione dell'ordine delle operazioni si consideri la sequenza d'esecuzione 2 della Figura 6.23. Poiché l'operazione `write(`A`)` di T_1 non è in conflitto con l'operazione `read(`B`)` di T_0, dallo scambio della loro posizione si ottiene una sequenza d'esecuzione equivalente: indipendentemente dallo stato iniziale, entrambe le sequenze d'esecuzione producono il medesimo risultato. Proseguendo nello scambio delle operazioni non conflittuali si ottiene ciò che segue:

- scambio dell'operazione `read(`B`)` di T_0 con l'operazione `read(`A`)` di T_1;
- scambio dell'operazione `write(`B`)` di T_0 con l'operazione `write(`A`)` di T_1;
- scambio dell'operazione `write(`B`)` di T_0 con l'operazione `read(`A`)` di T_1.

Il risultato finale di questo procedimento è la sequenza d'esecuzione 1 della Figura 6.22, una sequenza d'esecuzione seriale. Questo ragionamento ha dimostrato come la sequenza d'esecuzione 2 sia equivalente a una sequenza d'esecuzione seriale e implica che, indipendentemente dallo stato iniziale del sistema, la sequenza d'esecuzione 2 produce il medesimo stato finale determinato da una qualsiasi sequenza d'esecuzione seriale equivalente.

Se una sequenza d'esecuzione S si può trasformare in una sequenza d'esecuzione seriale S' con una serie di scambi tra operazioni non conflittuali, si dice che S è in **conflitto serializzabile**. Quindi, la sequenza d'esecuzione 2 è in conflitto serializzabile, poiché si può trasformare nella sequenza d'esecuzione seriale 1.

6.9.4.2 Protocolli per la gestione dei lock

Uno dei metodi che si usano per garantire la serializzabilità consiste nell'associare un lock a ciascun dato, e richiedere che ogni transazione rispetti il **protocollo per la gestione dei lock** (*locking protocol*), che governa l'acquisizione e il rilascio dei lock. Si può applicare un lock a un dato in in diversi modi, due dei quali sono i seguenti.

- **Condiviso.** Se una transazione T_i ottiene il lock di Q in modo condiviso, T_i può leggere, ma non scrivere nell'elemento (tale forma si denota con S).
- **Esclusivo.** Se una transazione T_i ottiene il lock di Q in modo esclusivo, T_i può leggere e scrivere nell'elemento (tale forma si denota con X).

Ogni transazione deve richiedere il lock di un elemento Q nel modo adeguato al tipo di operazione che deve eseguire su di esso.

Per accedere a un elemento Q, una transazione T_i deve innanzitutto eseguire il lock di Q in modo adeguato. Se il lock di Q è attualmente disponibile, la sua acquisizione da parte di T_i è garantita; se però il lock di Q è posseduto da un'altra transazione, T_i dovrà attenderne il rilascio. In particolare, se T_i richiede un lock esclusivo per Q, deve attendere che la transazione che ne è attualmente in possesso lo rilasci. Se, invece, il lock richiesto è di tipo condiviso, T_i deve aspettare il rilascio della risorsa solo se Q è attualmente soggetto a un lock esclusivo; altrimenti ottiene anch'essa il diritto di accedere a Q. Si noti la somiglianza tra questo schema e l'algoritmo dei lettori-scrittori trattato nel Paragrafo 6.6.2.

Una transazione può rilasciare il lock di un elemento precedentemente acquisito, anche se deve in ogni caso mantenerlo per tutto il periodo in cui accede all'elemento. Inoltre, non è sempre auspicabile che una transazione rilasci il lock di un elemento immediatamente dopo il suo ultimo accesso, poiché in questo modo la serializzabilità della transazione potrebbe non essere garantita.

Un protocollo che assicura la serializzabilità è il cosiddetto **protocollo per la gestione dei lock a due fasi**, che esige che ogni transazione richieda l'esecuzione delle operazioni di lock e di rilascio (*unlock*) in due fasi distinte:

- **fase di crescita.** Una transazione può ottenere nuovi lock sui dati, ma non rilasciarne alcuno in suo possesso;
- **fase di riduzione.** Una transazione può rilasciare lock sui dati di cui è in possesso, ma non ottenerne di nuovi.

Inizialmente, una transazione si trova nella fase di crescita e acquisisce i lock sui dati necessari; nel rilasciarne uno, la transazione entra nella fase di riduzione e non ne può richiedere di nuovi.

Il protocollo per la gestione dei lock a due fasi garantisce la serializzabilità dei conflitti (Esercizio 6.35), ma non elimina la possibilità di situazioni di stallo. Inoltre può accadere che, dato un insieme di transazioni, esistano sequenze d'esecuzione in conflitto serializzabile che tuttavia non si possono ottenere attraverso il protocollo per la gestione dei lock a due fasi. A ogni modo, per migliorare le prestazioni di questo protocollo è indispensabile disporre di ulteriori informazioni relative alle transazioni, oppure imporre una qualche struttura o un ordinamento dell'insieme dei dati.

6.9.4.3 Protocolli basati sulla marcatura temporale

Nei precedenti protocolli per la gestione dei lock l'ordinamento seguito da ogni coppia di transazioni conflittuali è determinato nella fase d'esecuzione dal primo lock sui dati richiesto da entrambe e che comporta modi incompatibili. Un altro metodo per determinare l'ordine di serializzabilità consiste nella scelta anticipata di un ordinamento delle transazioni. Il metodo più comunemente adottato consiste nell'usare uno schema con **ordinamento a marche temporali** (*timestamp ordering*).

A ciascuna transazione T_i nel sistema si associa una marca temporale (*timestamp*) unica individuata da $\text{TS}(T_i)$; il sistema assegna la marca temporale prima che la transazione T_i inizi la propria esecuzione. Se una transazione T_i riceve una marca temporale $\text{TS}(T_i)$ e si presenta una nuova transazione T_j, allora $\text{TS}(T_i) < \text{TS}(T_j)$. Per realizzare questo schema esistono due semplici metodi, che sono i seguenti.

- Adoperare come marca temporale il valore dell'orologio di sistema; in altre parole, la marca temporale di una transazione equivale al valore dell'orologio nell'istante in cui si presenta nel sistema. Questo metodo non funziona per transazioni che avvengano in sistemi separati o con unità d'elaborazione che non condividono lo stesso orologio.
- Adoperare come marca temporale un contatore logico; in altri termini, la marca temporale di una transazione equivale al valore del contatore nell'istante in cui essa si presenta nel sistema. Il contatore s'incrementa dopo l'assegnazione di una nuova marca temporale.

Le marche temporali delle transazioni determinano l'ordine di serializzabilità. Quindi, se $\text{TS}(T_i) < \text{TS}(T_j)$, il sistema deve garantire che la sequenza d'esecuzione generata sia equivalente alla sequenza d'esecuzione seriale in cui la transazione T_i appare prima della transazione T_j.

Per realizzare questo schema a ogni elemento Q si associano due valori di marche temporali:

- **R-timestamp**(Q), che denota la maggiore tra le marche temporali di tutte le transazioni che hanno completato con successo un'operazione `read(`Q`)`.
- **W-timestamp**(Q), che denota la maggiore tra le marche temporali di tutte le transazioni che hanno completato con successo un'operazione `write(`Q`)`.

Queste marche temporali si aggiornano dopo l'esecuzione di ogni nuova istruzione `read(`Q`)` o `write(`Q`)`.

Il protocollo a ordinamento di marche temporali garantisce che l'esecuzione di tutte le operazioni conflittuali `read` e `write` rispetti l'ordinamento stabilito dalle marche temporali e funziona nel modo seguente.

- Si supponga che la transazione T_i sottoponga una `read(`Q`)`.
  - Se TS(T_i) < W-timestamp(Q), questo stato implica che T_i deve leggere un valore di Q che è già stato sovrascritto; quindi, si rifiuta l'operazione `read` e si annulla T_i ripristinando lo stato iniziale (*rollback*).
  - Se TS(T_i) ≥ W-timestamp(Q), si esegue l'operazione `read` e a R-timestamp(Q) si assegna il massimo tra R-timestamp(Q) e TS(T_i).
- Si supponga che la transazione T_i sottoponga una `write(`Q`)`.
  - Se TS(T_i) < R-timestamp(Q), questo stato implica che il valore di Q che T_i sta producendo era necessario in precedenza e che T_i aveva supposto che tale valore non sarebbe mai stato prodotto. Quindi si rifiuta l'operazione `write` e si annulla T_i ripristinando lo stato iniziale.
  - Se TS(T_i) < W-timestamp(Q), questo stato implica che T_i sta cercando di scrivere un valore di Q obsoleto. Quindi si rifiuta l'operazione `write` e si annulla T_i ripristinando lo stato iniziale.
  - Altrimenti, si esegue l'operazione `write`.

Ogni transazione T_i per cui lo schema di controllo della concorrenza ripristina lo stato iniziale come risultato della sottomissione di un'operazione `read` o `write`, riceve una nuova marca temporale e viene riavviata.

Per illustrare questo protocollo si consideri la sequenza d'esecuzione 3 di due transazioni T_2 e T_3 proposta nella Figura 6.24. Si assume che ciascuna transazione riceva la propria marca temporale immediatamente prima dell'esecuzione della sua prima istruzione. Quindi, nella sequenza d'esecuzione 3, TS(T_2) < TS(T_3), e la presente sequenza d'esecuzione è consentita dal protocollo basato sulla marcatura temporale.

T_2	T_3
`read(`B`)`	
	`read(`B`)`
	`write(`B`)`
`read(`A`)`	
	`read(`A`)`
	`write(`A`)`

Figura 6.24 Sequenza d'esecuzione 3: sequenza d'esecuzione possibile con il protocollo basato sulla marcatura temporale.

Questa sequenza d'esecuzione può essere generata anche dal protocollo per la gestione dei lock a due fasi; in ogni caso, certe sequenze d'esecuzione sono possibili con tale protocollo, ma non col protocollo basato sulla marcatura temporale, e viceversa.

Il protocollo a ordinamento di marche temporali garantisce la serializzabilità del conflitto. Questa proprietà deriva dal fatto che le operazioni conflittuali si elaborano secondo l'ordine stabilito dalle marche temporali. Il protocollo assicura inoltre l'assenza di situazioni di stallo, poiché nessuna transazione rimane in attesa.

6.10 Sommario

Dato un gruppo di processi sequenziali cooperanti che condividono dati, è necessario garantirne la mutua esclusione. Si tratta di assicurare che una sezione critica di codice sia utilizzabile da un solo processo o thread alla volta. Di solito l'hardware di un calcolatore fornisce diverse operazioni che assicurano la mutua esclusione, ma per la maggior parte dei programmatori queste soluzioni hardware sono troppo complicate da utilizzare. I semafori rappresentano una soluzione a questo problema. I semafori consentono di superare questa difficoltà; sono utilizzabili per risolvere diversi problemi di sincronizzazione e sono realizzabili in modo efficiente, soprattutto se è disponibile un'architettura che permette le operazioni atomiche.

Sono stati presentati diversi problemi di sincronizzazione – come il problema dei produttori e consumatori con memoria limitata, dei lettori-scrittori e dei cinque filosofi – che costituiscono esempi rappresentativi di una vasta classe di problemi di controllo della concorrenza. Questi problemi sono stati usati per verificare quasi tutti gli schemi di sincronizzazione proposti.

Il sistema operativo deve fornire mezzi di protezione contro gli errori di sincronizzazione. A tal fine sono stati proposti parecchi costrutti di linguaggio. I monitor offrono un meccanismo di sincronizzazione per la condivisione di tipi di dati astratti. Una variabile condizionale consente a una procedura di monitor di sospendere la propria esecuzione finché non riceve un segnale.

Solaris, Windows XP e Linux sono esempi di sistemi operativi moderni che offrono vari meccanismi come semafori, mutex, spinlock e variabili condizionali per il controllo dell'accesso ai dati condivisi. La API Pthreads fornisce supporto a mutex e variabili condizionali.

Una transazione è un'unità di programma da eseguire in modo atomico, cioè le operazioni a essa associate vanno eseguite nella loro totalità o non si devono eseguire per niente. Per assicurare la proprietà di atomicità anche nel caso di malfunzionamenti, si può usare la registrazione con scrittura anticipata. Si registrano tutti gli aggiornamenti nel log, che si mantiene in una memoria stabile. Se si verifica un crollo del sistema, si usano le informazioni conservate nel log per ripristinare lo stato dei dati aggiornati; tale ripristino si esegue usando le operazioni `undo` e `redo`. Per ridurre il carico dovuto alla ricerca nel log dopo un malfunzionamento del sistema è utilizzabile un metodo basato su punti di verifica.

Per assicurare una corretta esecuzione si deve usare uno schema di controllo della concorrenza che garantisca la serializzabilità. Esistono diversi schemi di controllo della concorrenza che assicurano la serializzabilità differendo un'operazione o arrestando la transazione che ha richiesto l'operazione. I più diffusi sono i protocolli per la gestione dei lock e gli schemi d'ordinamento a marche temporali.