Esercizi pratici

6.1 Nel Paragrafo 6.4 si afferma che disabilitando le interruzioni si può influenzare l'orologio di sistema. Spiegate perché ciò può succedere e come tali effetti possono essere mitigati.

6.2 **Il problema del fumatore di sigarette.** Si consideri un sistema con tre processi *fumatore* e un processo *agente*. Ogni fumatore continua a ripetere le operazioni di arrotolarsi una sigaretta e fumarla. Per fare una sigaretta e per fumarla il fumatore ha bisogno di tre elementi: tabacco, cartina e fiammiferi. Uno dei processi fumatore ha le cartine, un altro ha il tabacco e il terzo ha i fiammiferi. L'agente ha una disponibilità infinita delle tre risorse. L'agente mette sul tavolo due dei tre ingredienti e il fumatore in possesso del terzo ingrediente si arrotola una sigaretta e la fuma, segnalando all'agente il completamento del suo incarico. L'agente mette quindi sul tavolo altri due elementi e il ciclo si ripete. Scrivete un programma per sincronizzare agente e fumatori usando la sincronizzazione in Java.

6.3 Spiegate perché Solaris, Windows XP e Linux implementano meccanismi di bloccaggio multipli. Descrivete le circostanze in cui questi sistemi operativi utilizzano spinlock, mutex, semafori, mutex adattivi e variabili condition. Spiegate, in ciascun caso, perché occorre un tale meccanismo.

6.4 Descrivete come variano in termini di costo i dispositivi di memoria volatili, non volatili e stabili.

6.5 Illustrate il compito dei checkpoint. Quanto spesso i checkpoint devono essere eseguiti? Descrivete come la frequenza di checkpoint influisca su:

- prestazioni del sistema, nel caso in cui non vi siano errori;
- tempo necessario per il ripristino in seguito a un crash del sistema;
- tempo necessario per il ripristino in seguito a un crash del disco.

6.6 Spiegate il concetto di atomicità di una transazione.

6.7 Mostrate che certe sequenze di esecuzione sono possibili con il protocollo per la gestione dei lock a due fasi, ma non con il protocollo basato su marcatura temporale, e viceversa.

Esercizi

6.8 Le race condition sono possibili in diversi sistemi. Si consideri un sistema bancario con due funzioni: `deposit(amount)` e `withdraw(amount)`. Le due funzioni ricevono in ingresso l'importo (*amount*) che deve essere depositato (*deposit*) o prelevato (*withdraw*) da un conto corrente bancario. Assumete che un conto corrente sia condiviso tra marito e moglie, e che in maniera concorrente il marito chiami la funzione `withdraw()` e la moglie la funzione `deposit()`. Descrivete com'è possibile il verificarsi di una race condition e che cosa dovrebbe essere fatto per evitarla.

6.9 L'algoritmo seguente, concepito da Dekker, è la prima soluzione programmata conosciuta del problema della sezione critica per due processi. I due processi, P_0 e P_1, condividono le seguenti variabili:

```
boolean flag[2]; /* inizialmente falsa* /
int turno;
```

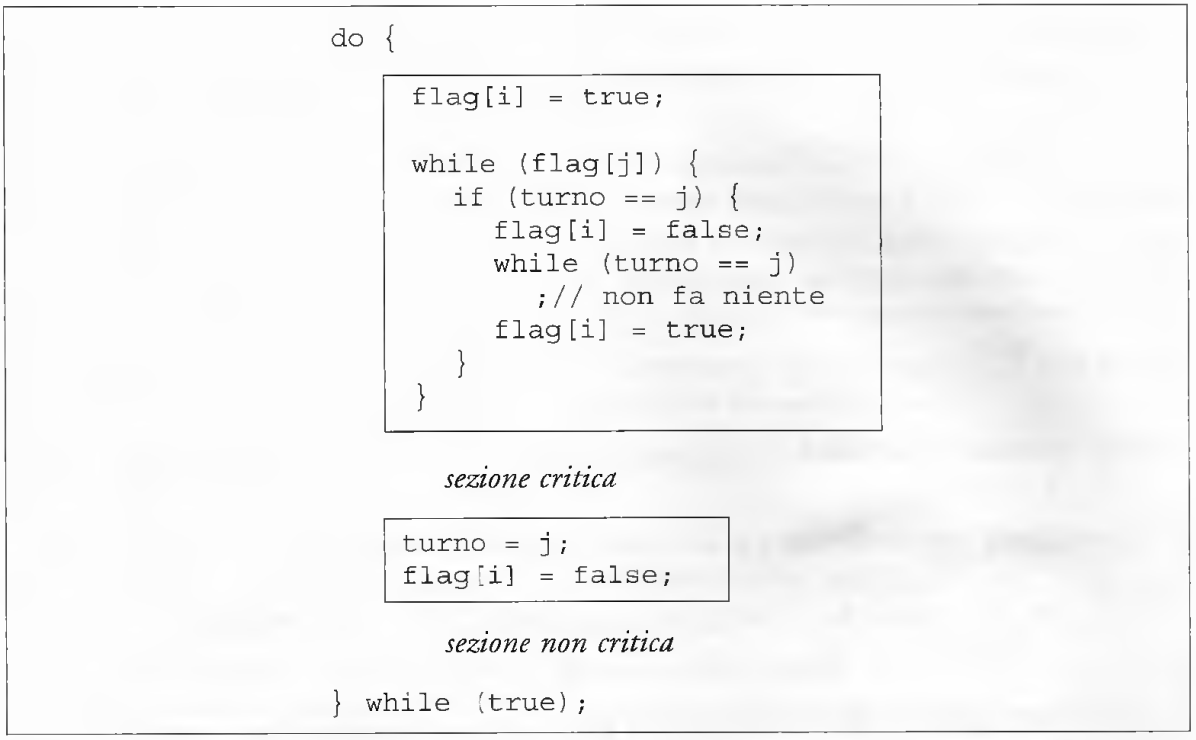

Figura 6.25 Struttura del processo P_i nell'algoritmo di Dekker.

La struttura del processo P_i (`i == 0` oppure `1`), dove P_j (`j == 1` oppure `0`) è l'altro processo, è mostrata nella Figura 6.25. Dimostrate che l'algoritmo soddisfa tutti e tre i requisiti per il problema della sezione critica.

6.10 La prima soluzione programmata corretta del problema della sezione critica per n processi con un limite di $n - 1$ turni d'attesa è stata proposta da Eisenberg e McGuire. I processi condividono le seguenti variabili:

```
enum statoP {inattivo, rich_in, in_sc};
statoP flag[n];
int turno;
```

Ciascun elemento di `pronto` è inizialmente `inattivo`; il valore iniziale di `turno` è irrilevante (compreso tra 0 e $n - 1$). La struttura del processo P_i è illustrata nella Figura 6.26. Dimostrate che tale algoritmo soddisfa tutti e tre i requisiti del problema della sezione critica.

6.11 Qual è il significato della locuzione *attesa attiva*? Esistono attese di altro genere in un sistema operativo? È possibile evitare completamente l'attesa attiva? Corredate di motivazione le vostre risposte.

6.12 Spiegate perché gli spinlock non siano adatti ai sistemi monoprocessore, ma siano spesso usati nei sistemi multiprocessore.

6.13 Chiarite perché, per implementare le primitive di sincronizzazione, non è corretto disabilitare le interruzioni di un sistema monoprocessore se le primitive stesse sono destinate a programmi utenti.

6.14 Spiegate perché le interruzioni non costituiscono un metodo appropriato per implementare le primitive di sincronizzazione nei sistemi multiprocessore.

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

      while (j != i) {
        if (flag[j] != inattivo)
          j = turno;
        else
          j = (j + 1) % n;
      }
      flag[i] = in_sc;
      j = 0;

      while ((j < n) && (j == i) || flag[i] != in_sc))
        j++;
      if ((j >= n) && (turno == i || flag[turno] == inattivo))
        break;
    }

        sezione critica

    j = (turno + 1) % n;

    while (flag[j] == inattivo)
      j = (j + 1) % n;

    turno = j;
    flag[i] = inattivo;

        sezione non critica

} while (true);
```

Figura 6.26 Struttura del processo P_i nell'algoritmo di Eisenberg e McGuire.

6.15 Descrivete due strutture dati di un kernel in cui possono verificarsi le cosidette race condition. Assicuratevi di includere una descrizione della modalità in cui queste situazioni possono verificarsi.

6.16 Descrivete come l'istruzione `swap()` consenta di ottenere una mutua esclusione che soddisfa il requisito dell'attesa limitata.

6.17 I server possono essere progettati in modo da limitare il numero di connessioni aperte. In un dato momento, per esempio, un server può ammettere solo N connessioni socket per volta; dopo questo limite, il server non accetterà alcuna connessione entrante prima che una connessione esistente sia chiusa. Spiegate come il server possa usare i semafori per limitare il numero delle connessioni concorrenti.

6.18 Si dimostri che, se le operazioni `wait()` e `signal()` dei semafori non sono eseguite in modo atomico, la mutua esclusione rischia di essere violata.

6.19 Mostrate come implementare le operazioni `wait()` e `signal()` dei semafori negli ambienti multiprocessore tramite l'istruzione `TestAndSet()`. La soluzione dovrebbe comportare un'attesa attiva minima.

6.20 L'Esercizio 4.17 richiede che il thread padre attenda che il thread figlio termini la sua esecuzione prima di visualizzare i valori calcolati. Ipotizziamo di voler permettere che il thread padre acceda ai numeri di Fibonacci non appena questi vengono calcolati dal figlio, piuttosto che attendere che il figlio termini. Spiegate quali modifiche sono necessarie alla soluzione dell'Esercizio 4.17. Implementate la soluzione modificata.

6.21 Dimostrate che monitor, sezioni critiche e semafori sono equivalenti, e che, pertanto, consentono di risolvere gli stessi problemi di sincronizzazione.

6.22 Progettate un monitor con memoria limitata, in cui i buffer siano parte del monitor stesso.

6.23 All'interno di un monitor, la mutua esclusione fa sì che il monitor con memoria limitata dell'Esercizio 6.22 sia adatto soprattutto buffer piccoli.

a. Spiegate perché questa affermazione è vera.
b. Progettate un nuovo schema idoneo a buffer grandi.

6.24 Argomentate il compromesso tra equità e produttività delle operazioni nel problema dei lettori-scrittori. Proponete un metodo per risolvere il problema dei lettori-scrittori senza che si determini attesa indefinita.

6.25 Come si differenzia l'operazione `signal()` dei monitor dalla corrispondente operazione dei semafori?

6.26 Ipotizziamo che l'istruzione `signal()` possa apparire solo per ultima in una procedura monitor. Proponete un modo per semplificare l'implementazione descritta nell Paragrafo 6.7.

6.27 Considerate un sistema formato dai processi P_1, P_2, ..., P_n, aventi priorità distinte, rappresentate da numeri interi. Scrivete un monitor grazie al quale tre identiche stampanti in linea siano assegnate a tali processi, stabilendo l'ordine di allocazione in base alle priorità.

6.28 Un file deve essere condiviso fra processi diversi, ognuno dei quali è identificato da un numero univoco. Al file possono accedere simultaneamente vari processi, a patto che osservino la seguente prescrizione: la somma degli identificatori di tutti i processi che accedono al file deve essere minore di n. Scrivete un monitor per coordinare l'accesso al file.

6.29 Se un processo invoca `signal()` al verificarsi di una condizione all'interno di un monitor, esso può continuare la propria esecuzione, oppure cedere il controllo al processo che ha ricevuto il segnale. Come cambia la soluzione al precedente esercizio in queste due circostanze?

6.30 Supponete di sostituire le operazioni `wait()` e `signal()` dei monitor con un solo costrutto, `await(B)`, dove `B` è un'espressione booleana generale, che obbliga il processo che lo esegue ad attendere finché `B` diventi vera.

a. Si metta a punto, in base a questo modello, un monitor che affronti il problema dei lettori-scrittori.
b. Chiarite che cosa impedisce di implementare efficientemente questo costrutto.

c. Quali restrizioni è necessario imporre all'istruzione `await(B)` perché possa essere implementata efficientemente? (Suggerimento: limitate la forma di `B`; si veda [Kessels 1977].)

6.31 Scrivete un monitor per realizzare una "sveglia" con cui un programma chiamante possa differire la propria esecuzione per il numero prescelto di unità di tempo ("battiti o tic"). Si può ipotizzare l'esistenza di un orologio hardware che invochi una procedura `battito` sul vostro monitor a intervalli regolari.

6.32 Perché Solaris, Linux e Windows XP usano gli spinlock come meccanismo di sincronizzazione esclusivamente sui sistemi multiprocessore, e non su quelli a processore unico?

6.33 Nei sistemi dotati di un log che mettono a disposizione le transazioni è impossibile aggiornare i dati prima che siano registrati i relativi elementi. Perché è necessaria una tale restrizione?

6.34 Dimostrate che il protocollo per la gestione dei lock a due fasi assicuri la serializzabilità dei conflitti.

6.35 Quali conseguenze derivano dall'assegnazione di una nuova marca temporale a una transazione annullata per *rollback*? In che modo il sistema gestisce le transazioni richieste successivamente al *rollback* della transazione, e caratterizzate da marche temporali precedenti rispetto a essa?

6.36 Assumete di dover gestire un numero finito di risorse, appartenenti tutte allo stesso tipo. I processi possono richiedere una parte di queste risorse, per poi restituirle quando hanno terminato. Un esempio proviene dai programmi commerciali, molti dei quali includono in un singolo pacchetto alcune **licenze**, che indicano il numero di applicazioni concorrentemente eseguibili. All'avvio di un'applicazione, il totale delle licenze a disposizione diminuisce. Se un'applicazione termina, il totale delle licenze aumenta. Quando tutte le licenze sono in uso, le nuove richieste per l'avvio di un'applicazione non avranno esito. Si ottempera a tali richieste solo nel momento in cui il proprietario di una licenza lascia libera un'applicazione e restituisce la relativa licenza.

```
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES;
```

Quando un processo mira a ottenere alcune risorse, invoca la funzione `decrease_count()`:

```
/* decrementa il numero di risorse disponibili */
/* di una quantità pari a count; restituisce 0 */
/* se vi sono risorse sufficienti, -1 altrimenti */
int decrease_count(int_count) {
   if (available_resources < count)
      return -1;
   else {
      available_resources -= count;
      return 0;
   }
}
```

Quando un processo intende restituire le risorse che possiede, chiama la funzione `increase_count()`:

```
/* incrementa il numero di risorse disponibili  */
/* di una quantità pari a count  */
int increase_count(int count) {
   available_resources += count;
   return 0;
}
```

Il programma precedente provoca un conflitto per l'accesso (ossia una *race condition*), per ovviare al quale dovrete:

a) individuare i dati interessati dal conflitto;
b) identificare, nel codice, il brano (o i brani) che danno luogo al conflitto;
c) adoperare un semaforo che regoli gli accessi, eliminando il conflitto.

6.37 La funzione `decrease_count()` dell'esercizio precedente restituisce 0 qualora vi siano sufficienti risorse a disposizione e −1 in caso contrario. Questo complica la programmazione di un processo che intenda sfruttare delle risorse:

```
while (decrease_count(count) == -1)
   ;
```

Riscrivete il segmento di codice dedicato alla gestione delle risorse, servendovi di un monitor e di variabili condizionali, in modo che la funzione `decrease_count()` sospenda il processo finché non siano disponibili risorse sufficienti. Ciò consentirà a un processo di invocare `decrease_count()` semplicemente così:

```
decrease_count(count);
```

Il processo rientra da questa chiamata solo quando vi siano risorse sufficienti disponibili.

Problemi di programmazione

6.38 Considerate il **problema del barbiere che dorme**. Un negozio di barbiere ha una sala d'attesa con n sedie e una stanza del barbiere con la sedia da lavoro. Se non ci sono clienti da servire, il barbiere dorme; se un cliente entra nel negozio e trova tutte le sedie occupate, se ne va; se il barbiere è occupato, ma ci sono sedie disponibili, il cliente si siede; se il barbiere dorme, il cliente lo sveglia. Scrivete un programma che coordini barbiere e clienti.

Progetti di programmazione

6.39 **Il problema dei produttori e dei consumatori**

Nel Paragrafo 6.6.1 abbiamo proposto una soluzione al problema dei produttori e consumatori che si basa su semafori e si avvale di un buffer limitato. In questo progetto daremo soluzione al problema del buffer limitato mediante i processi produttore e consumatore schematizzati nelle Figure 6.10 e 6.11, rispettivamente. La soluzione presentata nel Paragrafo 6.6.1 utilizza tre semafori: `vuote` e `piene`, che enume-

rano le posizioni vuote e piene del buffer, e `mutex`, un semaforo binario, o a mutua esclusione, che protegge l'inserimento nel (o l'estrazione dal) buffer. In questo progetto i semafori `vuote` e `piene` saranno semafori contatori standard, mentre, per rappresentare `mutex`, si userà un lock mutex in luogo di un semaforo binario. Il produttore e il consumatore – eseguiti come thread separati – trasferiscono gli oggetti a un buffer, oppure li prelevano da esso; il buffer è sincronizzato con le strutture `vuote`, `piene` e `mutex`. Potete scegliere di risolvere il problema con Pthreads o con la API Win32.

Buffer

Internamente, il buffer è costituito da un array di dimensione fissa avente tipo `buffer_item`, definito tramite `typefdef`. L'array di oggetti `buffer_item` sarà trattato come una coda circolare. Definizione e dimensione relative possono essere poste in un file di intestazione come il seguente:

```
/* buffer.h */
typedef int buffer_item;
#define BUFFER_SIZE 5
```

Il buffer sarà manipolato da due funzioni, `insert_item()` e `remove_item()`, richiamate, rispettivamente, dal thread del produttore e dal thread del consumatore. Tali funzioni, in un primo abbozzo schematico, appaiono nella Figura 6.27.

Le funzioni `insert_item` e `remove_item` sincronizzeranno il produttore e il consumatore usando gli algoritmi delineati nelle Figure 6.10 e 6.11. Il buffer richiederà, inoltre, una funzione per inizializzare `mutex`, come pure i semafori `vuote` e `piene`.

La funzione `main()` inizializzerà il buffer, creando due thread distinti per il produttore e il consumatore. Dopo aver creato tali thread, la funzione `main()` rimarrà inattiva per un certo tempo e, alla ripresa dell'attività, terminerà l'applicazione. La funzione `main()` riceverà tre parametri dalla riga di comando, indicanti:

```
#include "buffer.h"
/* il buffer */
buffer_item buffer[BUFFER_SIZE];
int insert_item(buffer_item item) {
   /* inserisce un elemento nel buffer
    restituisce 0 se ha successo,
    altrimenti -1 per segnalare l'errore */
}
int remove_item(buffer_item *item) {
   /* estrae un elemento dal buffer
    e lo pone in item
    restituisce 0 se ha successo,
    altrimenti -1 per segnalare l'errore */
}
```

Figura 6.27 Schema delle funzioni.

```
#include "buffer.h"
 int main(int argc, char *argv[]) {
  /* 1. Ottiene i parametri argv[0], argv[1], argv[2]
        dalla riga di comando */
  /* 2. Inizializza il buffer */
  /* 3. Crea i thread di tipo produttore */
  /* 4. Crea i thread di tipo consumatore */
  /* 5. Attende */
  /* 6. Termina */
```

Figura 6.28 Scheletro del programma.

a. il tempo di inattività prima di terminare;

b. il numero di thread produttori;

c. il numero di thread consumatori.

La struttura di questa funzione è illustrata nella Figura 6.28.

Thread produttori e consumatori

Il thread produttore alternerà l'inattività per periodi di tempo non prevedibili all'inserimento di un intero casuale nel buffer. I numeri casuali saranno generati tramite la funzione `rand()`, che dà luogo a valori interi casuali compresi tra 0 e `RAND_MAX`. Anche il consumatore rimarrà in stato di d'inattività per un intervallo di tempo casuale; tornato attivo, tenterà di estrarre un oggetto dal buffer. Lo schema dei thread produttori e consumatori è illustrato nella Figura 6.29.

Nei paragrafi successivi ci occuperemo dapprima dei dettagli relativi a Pthreads per poi passare alla API di Win32.

Creazione dei thread con Pthreads

Per la creazione di thread con la API di Pthreads rimandiamo il lettore al Capitolo 4, contenente i dettagli necessari per generare il produttore e il consumatore.

Lock mutex di Pthreads

Il brano di codice riportato nella Figura 6.30 illustra come si possano sfruttare i lock mutex forniti dalla API di Pthreads per proteggere una sezione critica.

Ptheads utilizza il tipo di dati `pthread_mutex_t` per i lock mutex. Un mutex è generato con la funzione `pthread_mutex_init(&mutex,NULL)`, in cui il primo parametro è un puntatore al mutex. Passando `NULL` come secondo parametro, il mutex ha gli attributi di default. Il mutex è attivato e disattivato con le funzioni `pthread_mutex_lock()` e `pthread_mutex_unlock()`. Se il lock mutex non è disponibile quando è invocata `pthread_mutex_lock()`, il thread chiamante rimane bloccato finché il proprietario del lock invoca `pthread_mutex_unlock()`. Tutte le funzioni mutex restituiscono il valore 0 in assenza d'errori e un valore non nullo altrimenti.

```
#include <stdlib.h> /* necessario per rand() */
#include "buffer.h"

void *producer(void *param) {
   buffer_item item;

   while (true) {
      /* resta inattivo per un intervallo di tempo casuale */
      sleep(...);
      /* genera un numero casuale */
      item = rand();
      if (insert_item(item))
         fprintf("report error condition");
      else
         printf("producer produced %d\n",item);
}

void *consumer(void *param) {
   buffer_item item;

   while (true) {
      /* resta inattivo per un intervallo di tempo casuale */
      sleep(...);
      if (remove_item(&item))
         fprintf("report error condition");
      else
         printf("consumer consumed %d\n",item);
}
```

Figura 6.29 Schema dei thread dei produttori e consumatori.

```
#include <pthread.h>
pthread_mutex_t mutex;

/* crea il lock mutex */
pthread_mutex_init(&mutex,NULL);

/* acquisisce il lock mutex */
pthread_mutex_lock(&mutex);

/*** sezione critica ***/

/* rimuove il lock mutex */
pthread_mutex_unlock(&mutex);
```

Figura 6.30 Esempio di codice.

```
#include <semaphore.h>
sem_t mutex;

/* crea il semaforo */
sem_init(&mutex, 0, 1);

/* acquisisce il semaforo */
sem_wait(&mutex);

/*** sezione critica ***/

/* restituisce il semaforo */
sem_post(&mutex);
```

Figura 6.31 Esempio di codice.

Semafori di Pthreads

Pthreads fornisce due tipi di semafori: con o senza nome. Per questo progetto, impiegheremo semafori senza nome. Il codice seguente mostra come si dà origine a un semaforo:

```
#include <semaphore.h>
sem_t sem;

/* Crea il semaforo e lo inizializza a 5 */
sem_init(&sem, 0, 5);
```

La funzione `sem_init()`, che crea e inizializza il semaforo, accetta tre parametri:

a. un puntatore al semaforo;
b. un flag indicante il livello di condivisione;
c. il valore iniziale del semaforo.

Nell'esempio precedente, passare il flag 0 significa stabilire che questo semaforo è condivisibile unicamente dai thread appartenenti al medesimo processo che ha creato il semaforo. Un valore non nullo consentirebbe l'accesso al semaforo anche da parte di altri processi. In questo esempio, il valore iniziale del semaforo è inizializzato a 5.

Nel Paragrafo 6.5 abbiamo descritto le classiche operazioni sui semafori `wait()` e `signal()`. Pthreads richiama le operazioni `wait()` e `signal()` rispettivamente nell'ordine `sem_wait()` e `sem_post()`. L'esempio di codice illustrato nella Figura 6.31 genera un semaforo binario mutex con valore iniziale 1, e ne illustra l'uso nel proteggere una sezione critica.

Lock mutex di Win32

I lock mutex sono oggetti dispatcher, come si è visto nel Paragrafo 6.8.2. Di seguito si può osservare come creare un lock mutex utilizzando la funzione `CreateMutex()`:

```
#include <windows.h>

HANDLE Mutex;
Mutex = CreateMutex(NULL, FALSE, NULL);
```

Il primo parametro si riferisce a un attributo di sicurezza per il lock mutex. Impostando a `NULL` questo attributo, i processi figli del creatore del lock mutex non ereditano il riferimento al mutex stesso. Il secondo parametro indica se il processo che ha creato il mutex sia il possessore iniziale del lock mutex: passando il valore `false`, si asserisce che il thread non è il possessore iniziale; vedremo fra breve come acquisire i lock mutex. Il terzo parametro consente di attribuire un nome al mutex. Tuttavia, poiché si è scelto un valore `NULL`, non gli attribuiremo un nome. Se `CreateMutex()` è stata eseguita senza errori, restituirà il riferimento `HANDLE` al lock; altrimenti, `NULL`.

Nel Paragrafo 6.8.2 abbiamo descritto i due possibili stati, detti *signaled* e *nonsignaled*, degli oggetti dispatcher. Quando un oggetto è nello stato *signaled* ne può entrare in possesso; una volta acquisito, l'oggetto dispatcher (per esempio, un lock mutex) passa allo stato *nonsignaled*. L'oggetto diviene *signaled* non appena risulta nuovamente disponibile.

I lock mutex si acquisiscono richiamando la funzione `WaitForSingleObject()` con due parametri: il riferimento `HANDLE` al lock, e un flag che indica la durata dell'attesa. Il codice mostra come si può acquisire il lock mutex creato in precedenza.

```
WaitForSingleObject(Mutex, INFINITE);
```

Il valore `INFINITE` significa che non vi è limite a quanto si è disposti ad attendere che il lock diventi disponibile. Con altri valori il thread chiamante potrebbe troncare l'operazione, qualora il lock non tornasse disponibile entro un tempo definito. Se il lock è in stato *signaled*, `WaitForSingleObject()` restituisce immediatamente il controllo, e il lock diviene *nonsignaled*. Per rendere disponibile un lock (ossia, per far sì che entri nello stato *signaled*) si invoca `ReleaseMutex()`:

```
ReleaseMutex(Mutex);
```

Semafori di Win32

Nella API di Win32 anche i semafori sono oggetti dispatcher e pertanto adoperano lo stesso meccanismo di segnalazione dei lock mutex. I semafori si creano tramite il codice:

```
#include <windows.h>

HANDLE Sem;
Sem = CreateSemaphore(NULL, 1, 5, NULL);
```

In analogia a quanto descritto per i lock mutex, il primo e l'ultimo parametro designano un attributo di sicurezza e un nome per il semaforo. Il secondo e il terzo parametro indicano il valore iniziale e il valore massimo del semaforo. In questo caso, il valore iniziale del semafo-

ro è 1, mentre il suo valore massimo è 5. Se termina senza errori, `CreateSemaphore()` restituisce un riferimento `HANDLE` al lock mutex; in caso contrario, restituisce `NULL`.

I semafori si acquisiscono con la stessa funzione trattata per i lock mutex, cioè `WaitForSingleObject()`. Per acquisire il semaforo Sem di questo esempio, si invoca:

```
WaitForSingleObject(Semaphore, INFINITE);
```

Se il valore del semaforo è > 0, esso si trova nello stato *signaled*, e viene quindi acquisito dal thread chiamante; altrimenti, il thread chiamante è sospeso (per un tempo imprecisato, a causa del parametro `INFINITE`) in attesa che il semaforo diventi *signaled*.

Per i semafori di Win32, l'equivalente dell'operazione `signal()`è la funzione `ReleaseSemaphore()`. Essa accetta tre parametri: (1) il riferimento `HANDLE` al semaforo, (2) il valore di cui incrementare il semaforo e (3) un puntatore al valore precedente del semaforo. Si può aumentare `Sem` di 1 nel modo seguente:

```
ReleaseSemaphore(Sem, 1, NULL);
```

`ReleaseSemaphore()` e `ReleaseMutex()` restituiscono 0 se eseguite senza errori e un valore non nullo in caso contrario.

6.11 Note bibliografiche

Il problema della mutua esclusione fu discusso per la prima volta in un classico lavoro di [Dijkstra 1965a]. L'algoritmo di Dekker (Esercizio 6.9), la prima soluzione software corretta al problema della mutua esclusione per due processi, fu sviluppato dal matematico olandese T. Dekker. L'algoritmo è anche esaminato in [Dijkstra 1965a]; una soluzione più semplice si trova in [Peterson 1981] (Figura 6.2).

[Dijkstra 1965b] contiene la prima soluzione al problema della mutua esclusione per n processi. Si tratta però di una soluzione che non garantisce un limite superiore al tempo d'attesa dei processi che richiedano l'ingresso nelle proprie sezioni critiche. [Knuth 1966] presenta il primo algoritmo con un limite pari a 2^n turni, e [deBruijn 1967] ne propone un raffinamento che riduce l'attesa a n^2 turni. [Eisenberg e McGuire 1972] riescono a portare l'attesa al limite inferiore di n-1 turni. Un altro algoritmo con questa proprietà, ma più facile da comprendere e programmare, è l'algoritmo del fornaio, proposto in [Lamport 1974]. [Burns 1978] sviluppa l'algoritmo basato sull'hardware che soddisfa il requisito dell'attesa limitata.

Trattazioni generali del problema della mutua esclusione si trovano in [Lamport 1986] e [Lamport 1991]. Una raccolta di algoritmi per la mutua esclusione è in [Raynal 1986].

Il concetto di semaforo è suggerito da [Dijkstra 1965a]. [Patil 1971] studia la misura in cui i semafori sono in grado di risolvere i problemi di sincronizzazione. [Parnas 1975] illustra alcuni errori nelle argomentazioni di Patil. [Kosoraju 1973] prosegue il lavoro di Patil esibendo un problema irrisolvibile dalle sole operazioni `wait()` e `signal()`.[Lipton 1974] analizza le varie limitazioni delle diverse primitive per la sincronizzazione.

I classici problemi del coordinamento dei processi presentati in questo capitolo fungono da paradigma per un'ampia classe di problemi di controllo della concorrenza. Il problema del buffer limitato, dei cinque filosofi e del barbiere che dorme (Esercizio 6.38) sono proposti in [Dijkstra 1965a] e [Dijkstra 1971]. Il problema dei fumatori di sigarette (Esercizio 6.2) è stato sviluppato da Patil [1971]; quello dei lettori-scrittori è dovuto a [Courtois et al. 1971]. Le questioni legate alla lettura e alla scrittura concorrenti sono discusse in [Lamport 1977]. Il problema della sincronizzazione di processi indipendenti è trattato da [Lamport 1976].

Il concetto di sezione critica è dovuto a [Hoare 1972] e [Brinch-Hansen 1972]. Il concetto di monitor è sviluppato da [Brinch-Hansen 1973], e una sua completa descrizione si trova in [Hoare 1974]. [Kessels 1977] propone un'estensione del concetto di monitor che incorpora un meccanismo automatico per le segnalazioni. I frutti dell'esperienza fatta nell'uso dei monitor in programmi concorrenti sono presentati in [Lampson e Redell 1979]. Discussioni generali sulla programmazione concorrente si trovano in [Ben-Ari 1990] e [Birrel 1989].

L'ottimizzazione delle prestazioni delle primitive per la gestione dei lock è discussa in molti lavori, fra cui [Lamport 1987], [Mellor-Crummey e Scott 1991] e [Anderson 1990]. L'impiego di oggetti condivisi che non richiede l'uso di sezioni critiche è illustrato da [Herlihy 1993], [Bershad 1993] e [Kopetz e Reisinger 1993]. Nuove istruzioni hardware utili per l'implementazione di primitive per la sincronizzazione sono discusse in lavori come [Culler et al. 1998], [Goodman et al. 1989], [Barnes 1993] e [Herlihy e Moss 1993].

Alcuni dettagli sui lock di Solaris sono presentati da [Mauro e McDougall 2007]. Si noti che i lock usati dal kernel sono implementati anche a beneficio dei thread utenti: lo stesso tipo di lock è disponibile all'interno e all'esterno del kernel. I dettagli sulla sincronizzazione in Windows 2000 si trovano in [Solomon e Russinovich 2000]. Goetz et al. [2006] presenta una trattazione dettagliata della programmazione concorrente in Java e del pacchetto `java.util.concurrent`.

Lo schema di registrazione con scrittura anticipata è introdotto per la prima volta nel [System R da Gray et al. 1981]. Il concetto di serializzabilità è formulato da [Eswaran et al. 1976], in relazione al loro lavoro sul controllo della concorrenza nel System R. Il protocollo per la gestione dei lock a due fasi è dovuto a [Eswaran et al. 1976]. Lo schema di controllo della concorrenza basato sulla marcatura temporale è in [Reed 1983]. Diversi algoritmi per il controllo della concorrenza basati sulla marcatura temporale sono spiegati in Bernstein e Goodman [1980]. Adl-Tabatabai et al. [2007] tratta il tema della memoria transazionale.