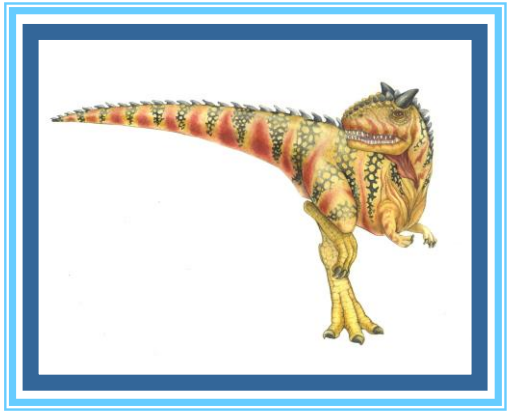
Esempi di sincronizzazione



Obiettivi

- Illustrare i problemi classici di sincronizzazione — produttore–consumatore con buffer limitato, lettori–scrittori, e filosofi a cena
- Descrivere gli strumenti implementati da Linux per la sincronizzazione
- Illustrare come utilizzare POSIX per risolvere i problemi di sincronizzazione fra processi

Sommario

- Problemi classici di sincronizzazione
- Sincronizzazione in Linux
- Sincronizzazione POSIX
- Approcci alternativi

Problemi classici di sincronizzazione

- Problema del produttore–consumatore con buffer limitato
- Problema dei lettori–scrittori
- Problema dei cinque filosofi
  - Esempi di una vasta classe di problemi di controllo della concorrenza

Problema del buffer limitato – 1

- Variabili condivise:

```
int n;
semaphore full, empty, mutex;
// inizialmente full = 0, empty = n, mutex = 1
```

- Il buffer ha n posizioni, ciascuna in grado di contenere un elemento
- Il semaforo **`mutex`** garantisce la mutua esclusione sugli accessi al buffer
- I semafori **`empty`** e **`full`** contano, rispettivamente, il numero di posizioni vuote ed il numero di posizioni piene nel buffer

Problema del buffer limitato – 2

- Struttura del processo produttore

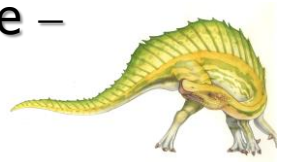
```
while(true) {
     ...
  /* produce un elemento in next_produced */
     ...
   wait(empty);
   wait(mutex);
     ...
  /* inserisce next_produced nel buffer */
     ...
   signal(mutex);
   signal(full);
}
```

Problema del buffer limitato – 3

- Struttura del processo consumatore

```
while(true) {
        wait(full);
        wait(mutex);
            ...
     /* sposta un elemento dal buffer in next_consumed */
            ...
        signal(mutex);
        signal(empty);
            ...
    /* consuma un elemento in next_consumed */
            ...
        }
```

Da notare la perfetta simmetria fra i due processi: il produttore – che produce posizioni piene per il consumatore – e il consumatore – che produce posizioni vuote per il produttore



Problema dei lettori–scrittori – 1

- Un insieme di dati (ad es., un file, una base di dati, etc.) deve essere condiviso tra più processi concorrenti che possono:
  - richiedere la sola lettura del contenuto…
  - o l'accesso ai dati sia in lettura che in scrittura
- Problema: permettere a più lettori di accedere ai dati contemporaneamente; solo uno scrittore, viceversa, può accedere ai dati condivisi (accesso esclusivo)
- Variabili condivise (oltre ai dati):

```
semaphore mutex, rw_mutex;
int read_count;
// inizialmente mutex=1, rw_mutex=1, read_count=0
```

Problema dei lettori–scrittori – 2

```
while(true) {
     wait(rw_mutex);
            ...
        /* scrittura */
            ...
     signal(rw_mutex);
}
```

Scrittore

```
while(true) {
     wait(mutex);
     read_count++;
     if (read_count == 1)
          wait(rw_mutex);
     signal(mutex);
            ...
        /* lettura */
            ...
     wait(mutex);
     read_count--;
     if (read_count == 0)
          signal(rw_mutex);
     signal(mutex);
}
```

Lettore

Problema dei lettori–scrittori – 3

- Il semaforo **`rw_mutex`** è comune ad entrambi i tipi di processi
- Il semaforo **`mutex`** serve per assicurare la mutua esclusione all'atto dell'aggiornamento (o del controllo del valore) di **`read_count`**
- La variabile **`read_count`** contiene il numero dei processi che stanno attualmente leggendo l'insieme di dati
- Il semaforo **`rw_mutex`** realizza la mutua esclusione per gli scrittori e serve anche al primo e all'ultimo lettore che entra/esce dalla sezione critica
  - Non serve ai lettori che accedono alla lettura in presenza di altri lettori

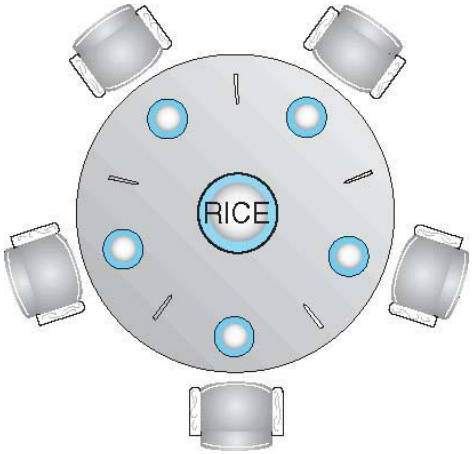
Problema dei lettori–scrittori – 4

- Diverse varianti: la soluzione proposta riguarda il primo problema dei lettori–scrittori, in cui si richiede che nessun lettore attenda, almeno che uno scrittore abbia già ottenuto l'accesso ai dati condivisi
  - Priorità ai lettori ⇨ gli scrittori possono essere soggetti a starvation
- Il secondo problema dei lettori–scrittori richiede invece che uno scrittore, se 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
  - Priorità agli scrittori ⇨ i lettori possono essere soggetti a starvation

Lock di lettura–scrittura

- Alcuni sistemi dispongono di lock di lettura–scrittura
  - In Linux, sia i lock mutex che i semafori hanno una variante per lettura/scrittura: **`rwlock_t`** e **`struct rw_semaphore`**
- È permesso a più processi di ottenere il lock in modalità lettura, ma un solo processo alla volta può acquisire il lock in modalità scrittura
- Lock di lettura–scrittura utili:
  - nelle applicazioni in cui è facile identificare processi che si limitano alla lettura dei dati condivisi e processi che si limitano alla scrittura degli stessi
  - nelle applicazioni che prevedono più lettori che scrittori; i lock di lettura–scrittura sono più "pesanti" di lock mutex e semafori, ma forniscono la possibilità di eseguire molti lettori in concorrenza

Problema dei cinque filosofi – 1



- Cinque filosofi passano la vita pensando e mangiando, attorno ad una tavola rotonda
- Non interagiscono con i loro vicini; occasionalmente tentano di procurarsi due bacchette per mangiare
- Al centro della tavola vi è una zuppiera di riso e la tavola è apparecchiata con cinque bacchette

Problema dei cinque filosofi – 2

- Quindi… quando un filosofo pensa, non interagisce con i colleghi, quando gli viene fame, tenta di impossessarsi delle bacchette che stanno alla sua destra ed alla sua sinistra
- Il filosofo può appropriarsi di una sola bacchetta alla volta e non può strapparla dalle mani di un vicino
- Quando un filosofo affamato possiede due bacchette contemporaneamente, mangia; terminato il pasto, appoggia le bacchette e ricomincia a pensare
- Variabili condivise:
  - La zuppiera di riso (l'insieme dei dati)
  - Cinque semafori, `bacchetta[5]`, inizializzati ad 1

Problema dei cinque filosofi – 3

- Codice per l'*i*–esimo filosofo

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

                 //  mangia

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

                 //  pensa

}
```

- Soluzione che garantisce che due vicini non mangino contemporaneamente, ma non riesce ad evitare lo stallo

Problema dei cinque filosofi – 4

- Stallo: i filosofi hanno fame simultaneamente ed ognuno si impossessa della bacchetta alla propria sinistra
- Soluzioni
  - Solo quattro filosofi possono stare a tavola contemporaneamente
  - Un filosofo può prendere le bacchette solo se sono entrambe disponibili (operazione da eseguire all'interno di una sezione critica)
  - I filosofi di posto "pari" raccolgono prima la bacchetta alla loro sinistra, quelli di posto "dispari" la bacchetta alla loro destra

Problema dei cinque filosofi – 5

```
monitor filosofo
 {
  enum{PENSA, AFFAMATO, MANGIA} stato[5];
  condition self[5];

  void prende (int i) {
       stato[i] = AFFAMATO;
       test(i);
       if (stato[i] != MANGIA) self[i].wait;
  }

  void posa (int i) {
       stato[i] = PENSA;
       // controlla i vicini a sinistra e a destra
       test((i+4) % 5);
       test((i+1) % 5);
  }
```

Problema dei cinque filosofi – 6

```
   void test (int i) {
           if ( (stato[(i+4) % 5] != MANGIA) &&
           (stato[i] == AFFAMATO) &&
           (stato[(i+1) % 5] != MANGIA) ) {
                stato[i] = MANGIA;
                self[i].signal();
            }
   }

   codice_iniz() {
          for (int i = 0; i < 5; i++)
          stato[i] = PENSA;
   }
}
```

Problema dei cinque filosofi – 7

- Il filosofo *i*–esimo invocherà le operazioni **`prende()`** e **`posa()`** nell'ordine:

```
filosofo.prende(i)
        ... ...
      mangia
        ... ...
filosofo.posa(i)
```

- La soluzione non provoca deadlock, ma la starvation è ancora possibile!

Esempio 1

- La seguente coppia di processi condivide l'insieme di variabili **`counter`**, **`tempA`** e **`tempB`**:

```
Processo A
…
A1: tempA = counter + 1;
A2: counter = tempA;
…
```

```
Processo B
…
B1: tempB = counter + 2;
B2: counter = tempB;
…
```

con **`counter`** inizializzato al valore 10. Quanti sono i possibili valori assunti da **`counter`** al termine dell'esecuzione di entrambi i processi? Modificare i processi **`A`** e **`B`**, inserendo opportune istruzioni **`wait()`**/**`signal()`** sul semaforo **`synch`** per ottenere il solo valore **`counter=13`**. Quale deve essere il valore iniziale di **`synch`**?

Esempio 1 (cont.)

- Si ottengono tre diversi valori per **`counter`**, in base all'ordine di esecuzione delle istruzioni, in particolare:

```
Processo A
…
A1: tempA = counter + 1;
A2: counter = tempA;
…
```

```
Processo B
…
B1: tempB = counter + 2;
B2: counter = tempB;
…
```

```
A1 A2 B1 B2: counter = 13
A1 B1 A2 B2: counter = 12
A1 B1 B2 A2: counter = 11
B1 A1 B2 A2: counter = 11
B1 A1 A2 B2: counter = 12
B1 B2 A1 A2: counter = 13
```

Esempio 1 (cont.)

- Considerando:

```
semaphore synch = 1
```

```
Processo A
…
     wait(synch);
A1:  tempA = counter + 1;
A2:  counter = tempA;
     signal(synch);
…
```

```
Processo B
…
     wait(synch);
B1:  tempB = counter + 2;
B2:  counter = tempB;
     signal(synch);
…
```

si ottiene che i processi **A** e **B** vengano eseguiti in sequenza (in ordine qualsiasi), producendo in stampa il solo valore 13

Esempio 2

- Selvaggi a cena – Una tribù di **N** selvaggi mangia in comune da una pentola che può contenere fino ad **M** (con **M<N**) porzioni di stufato di missionario; quando un selvaggio ha fame controlla la pentola: se non ci sono porzioni, sveglia il cuoco ed attende che questo abbia riempito di nuovo la pentola; se la pentola contiene almeno un pasto, se ne appropria; il cuoco controlla che ci siano delle porzioni e, se ci sono, si addormenta, altrimenti cuoce **M** porzioni e le serve ai cannibali in attesa
- Si descriva una soluzione al problema che impieghi i semafori
- Siano:

```
semaphore  mutex=1, pieno=0, vuoto=0;
int porzioni=M;
```

le risorse condivise

Esempio 2 (cont.)

- I processi Cuoco e Cannibale sono:

Processo Cuoco

```
{
   …
   while(1)
   {
      wait(vuoto);
      <riempi la pentola>
      porzioni = M;
      signal(pieno);
   }
}
```

Processo Cannibale

```
{
   …
   wait(mutex);
   if (porzioni==0)
      {
         signal(vuoto);
         wait(pieno);
      }
   porzioni--;
   signal(mutex);
   <mangia porzione>
   …
}
```

Sincronizzazione in Linux – 1

- Prima della versione 2.6, Linux adoperava un kernel senza diritto di prelazione
  - Gli interrupt venivano disabilitati per ottenere sezioni critiche di breve durata
- Attualmente, Linux fornisce diversi meccanismi per la sincronizzazione nel kernel
  - Lock mutex: una variabile booleana che indica se il lock è disponibile, modificata tramite le chiamate di sistema **`mutex_lock()`** e **`mutex_unlock()`**
  - Spinlock e semafori (con variante lettore–scrittore)
  - Interi atomici, rappresentati mediante il tipo opaco **`atomic_t`**
    - Un tipo di dato la cui reale struttura non è esposta nell'interfaccia (non vengono forniti dettagli sui campi e, di conseguenza, l'accesso alle informazioni può avvenire solo tramite l'utilizzo di particolari funzioni)

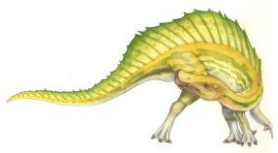
Sincronizzazione in Linux – 2

```
atomic_t counter;
int value;
atomic_set(&counter,5);   /* counter = 5 */
atomic_add(10,&counter);  /* counter = 15 */
atomic_sub(4,&counter);   /* counter = 11 */
atomic_inc(&counter);     /* counter = 12 */
value = atomic_read(&counter); /* value = 12 */
```

- Nel caso di interi atomici, tutte le operazioni matematiche vengono eseguite senza interruzioni
- Gli interi atomici sono particolarmente utili per gli aggiornamenti dei contatori
  - Non risentono dell'overhead dei meccanismi di lock
- Insufficienti se la race condition può coinvolgere diverse variabili ⇨ strumenti di lock più sofisticati

Sincronizzazione in Linux – 3

- Se il lock deve essere mantenuto per breve tempo:
  - sulle architetture SMP, il meccanismo fondamentale per la gestione della sezione critica è lo spinlock
  - Sui monoprocessori, come nel caso dei core embedded, si ricorre all'abilitazione/inibizione del diritto di prelazione (abilitazione/inibizione interruzioni)
  - Non è tuttavia possibile sottoporre a prelazione un task, attivo nel kernel, che sia possessore di un lock
- Quando sia necessario mantenere il lock più a lungo, si utilizzano lock mutex o semafori implementati senza busy waiting



Sincronizzazione in POSIX – 1

- La API Pthreads (lo standard POSIX) fornisce diversi tipi di lock e variabili condizionali per la sincronizzazione dei thread a livello utente
- API utilizzata su sistemi UNIX, Linux e MacOS
- I lock mutex rappresentano la tecnica di sincronizzazione fondamentale
  - Un thread, che sia in procinto di accedere alla propria sezione critica, si appropria del lock, salvo rimuoverlo all'uscita
- POSIX prevede due versioni di semafori (definiti nell'estensione POSIX SEM), con e senza nome
  - I semafori con nome possono essere utilizzati da processi non correlati, diversamente da quelli senza nome

Sincronizzazione in POSIX – 2

- Creazione e inizializzazione del lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

Tipo predefinito per la dichiarazione di lock

Il primo parametro è un puntatore al mutex, con il secondo parametro posto a **NULL** si inizializza il lock con gli attributi di default

- Acquisizione e rilascio del lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

Tutte le funzioni di gestione dei lock mutex restituiscono 0 in caso di corretto funzionamento; viceversa, restituiscono un codice di errore diverso da 0

Semafori con nome in POSIX

- Creazione e inizializzazione del semaforo

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

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

Tipo predefinito per la dichiarazione di semafori

- Un diverso processo può riferirsi al semaforo usandone il nome, **`SEM`** (le successive chiamate a **`sem_open`** restituiscono un descrittore al semaforo esistente)
- Acquisizione e rilascio del semaforo

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

Tutte le funzioni di gestione dei semafori restituiscono 0 in caso di corretto funzionamento; viceversa restituiscono un codice di errore diverso da 0

Semafori senza nome in POSIX

```
#include<semaphore.h>
sem_t sem;
…
/* crea il semaforo e lo inizializza a 1 */
sem_init(&sem,0,1);
…
```

```
/* acquisisci il semaforo */
sem_wait(&sem);

/* sezione critica */

/* rilascia il semaforo */
sem_post(&sem);
```

Indica il livello di condivisione: semaforo utilizzabile solo dai thread appartenenti al processo che ha creato il semaforo

Variabili condition in POSIX – 1

- Poiché POSIX è tipicamente utilizzato in C/C++, che non forniscono il tipo monitor, le variabili condition POSIX sono associate ai lock mutex per garantire la mutua esclusione
- Creazione e inizializzazione della variabile condition

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

Variabili condition in POSIX – 2

- Thread in attesa che valga **a==b**

```
pthread_mutex_lock(&mutex);
while (a != b)
      pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

Rilascia il lock mutex, consentendo ad un altro thread di accedere al dato condiviso

- Thread che "sveglia" un altro thread in attesa sulla variabile condition

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

Approcci alternativi – 1

- OpenMP
  - Include la direttiva

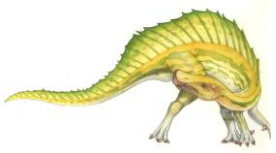
```
#pragma omp critical
```

    che specifica che la successiva regione di codice è una sezione critica in cui solo un thread alla volta può essere attivo
  - La direttiva per la sezione critica si comporta come un semaforo binario o un lock mutex (**`counter`** variabile condivisa)

```
void update (int value)
{
   #pragma omp critical
   {
        counter += value;
   }
}
```

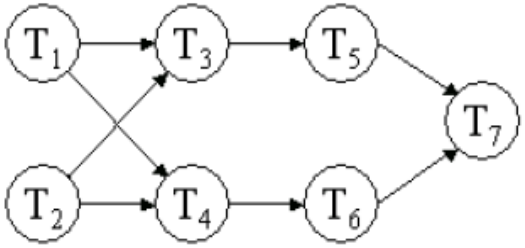
Approcci alternativi – 2

- Linguaggi di programmazione funzionali
  - Il flusso di esecuzione del programma assume la forma di una serie di valutazioni di funzioni matematiche
  - La differenza fondamentale fra linguaggi procedurali e funzionali è che i linguaggi funzionali non mantengono uno stato
  - Una volta che una variabile è stata definita e inizializzata il suo valore è immutabile ⇨ no race condition
  - Erlang, Scala, Haskell
    - Erlang: notevole supporto alla concorrenza, particolarmente adatto in ambiente parallelo
    - Scala: orientato agli oggetti (in gran parte simile a Java e C#)
    - Haskell: supporta una semantica di tipo *lazy* in cui gli argomenti delle funzioni vengono valutati solo se e quando richiesto



Esercizio 1

- Si consideri il seguente grafo di precedenza fra processi:



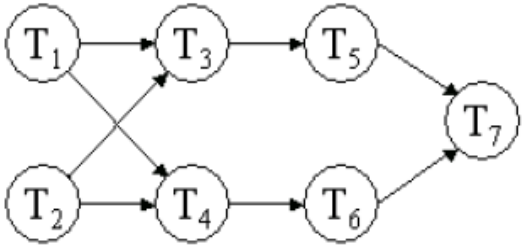
- Ciascun processo viene eseguito in base allo schema:

```
processo Ti (i=1,…,7) {
attende l'esecuzione dei predecessori, se ce ne sono;
esegue il proprio compito;
segnala la terminazione ai successori, se ce ne sono;
}
```

- Descrivere il codice relativo ai sette processi, inserendo opportunamente **`wait()`** e **`signal()`** ed utilizzando il minimo numero di semafori (definirne il valore di inizializzazione).

Esercizio 1 – Soluzione

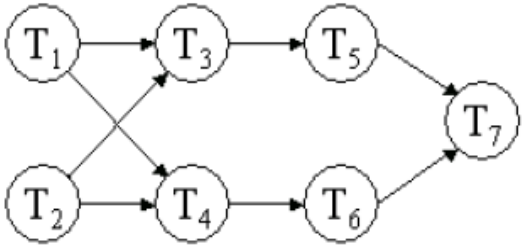
- Utilizzando 5 semafori si ottiene:




```
semaphore S[3:7] = ([5] 0); // init
T1: …signal(S[3]); signal(S[4]);
T2: …signal(S[3]); signal(S[4]);
T3: wait(S[3]); wait(S[3])… signal(S[5]);
T4: wait(S[4]); wait(S[4])… signal(S[6]);
T5: wait(S[5]); … signal(S[7]);
T6: wait(S[6]); … signal(S[7]);
T7: wait(S[7]); wait(S[7]);…
```

Esercizio 1 – Soluzione (cont.)

- Poiché **T_7** viene eseguito dopo **T_3** e **T_4**, un semaforo, per esempio **S[3]**, può essere riutilizzato. Una possibile soluzione con 4 semafori è la seguente:




```
semaphore S[3:6] = ([4] 0); // init
T1: …signal(S[3]); signal(S[4]);
T2: …signal(S[3]); signal(S[4]);
T3: wait(S[3]); wait(S[3])… signal(S[5]); signal(s[5]);
T4: wait(S[4]); wait(S[4])… signal(S[6]);
T5: wait(S[5]); … signal(S[3]);
T6: wait(S[6]); … signal(S[3]);
T7: wait(S[5]); wait(S[3]); wait(S[3]);…
```

Esercizio 2

- Monitor per la gestione di mailbox
  - Utilizziamo il costrutto monitor per risolvere il problema della comunicazione tra processi:
    - il monitor incorpora il buffer dei messaggi (gestito in modo circolare)
    - i processi produttori (o consumatori) inseriranno (o preleveranno) i messaggi mediante le funzioni **`send()`** (o **`receive()`**) definite nel monitor
  - La struttura dati che rappresenta il buffer fa parte delle variabili locali al monitor e quindi le operazioni **`send()`** e **`receive()`** possono accedere solo in modo mutuamente esclusivo a tale struttura

Esercizio 2 – Soluzione

```
monitor buffer_circolare
{
   messaggio buffer[N];
   int contatore=0; int testa=0; int coda=0;
   condition pieno;
   condition vuoto;

   void send(messaggio m)
   {
      if (contatore==N) pieno.wait;
      buffer[coda] = m;
      coda = (coda + 1)%N;
      ++contatore;
      vuoto.signal;
}
```

Esercizio 2 – Soluzione (cont.)

```
messaggio receive()
   {
      messaggio m;
      if (contatore == 0) vuoto.wait;
      m = buffer[testa];
      testa = (testa + 1)%N;
      --contatore;
      pieno.signal;
      return m;
   }
}/* fine monitor */
```

Esercizio 3

- Gestione parcheggio
  - Il gestore di un parcheggio vuole automatizzare il siste-ma di ingresso/uscita in modo da impedire l'accesso quando il parcheggio è pieno (cioè la sbarra d'ingresso deve sollevarsi solo se ci sono posti disponibili) e da consentire l'uscita solo in caso di parcheggio non vuoto (cioè la sbarra d'uscita deve rimanere abbassata in caso di parcheggio vuoto)
  - Scrivere un monitor che permetta di gestire l'accesso al parcheggio

Esercizio 3 – Soluzione

```
monitor CarPark
     condition full, empty;
     integer spaces, capacity;

     procedure enter();
          begin
            while(spaces=0) do full.wait;
            spaces := spaces-1;
            empty.signal;
          end;
     procedure exit();
          begin
            while(spaces=capacity) do empty.wait;
            spaces := spaces+1;
            full.signal;
          end;

     spaces := N;
     capacity := N;
end monitor;
```

Esercizio 4

- Variabili condivise:

```
semaphore S=1, M=1;
int x=10;
```

```
{
R1  wait(M);
    x=x+1;
    signal(M);

R2  wait(M);
    write(x);
    signal(M);

    signal(S);
}
```

Processo P_1

```
{
    wait(S);

R3  wait(M);
    x=x-1;
    signal(M);

R4  wait(M);
    write(x);
    signal(M);
}
```

Processo P_2

Esercizio 4 (cont.)

- Individuare le regioni critiche nel codice di P_1 e P_2; si possono verificare race condition per la variabile condivisa **x**?
  - Le regioni critiche (in R1, R2, R3 e R4) sono protette da semafori per cui non vi sono race condition su **x**
- Determinare tutti i possibili output del programma concorrente (P_1 in parallelo con P_2).
  - Il semaforo **s** non influisce sul flusso di esecuzione e quindi l'output dipende dall'ordine di esecuzione di R1–R2 e R3–R4:
    - Prima R1, R3 (in qualsiasi ordine) e poi R2, R4 (in qualsiasi ordine) → stampa: 10 10
    - R1 R2 R3 R4 → stampa: 11 10
    - R3 R4 R1 R2 → stampa: 9 10
- Supponiamo di inizializzare il semaforo **s** a 0 invece che a 1; determinare tutti i possibili output del programma (P_1 in parallelo con P_2) così modificato.
  - Il semaforo **s** forza l'ordine di esecuzione $P_1 \rightarrow P_2$ e quindi ci sarà un solo output: 11 10

Esercizio 5

- Variabili condivise:

```
semaphore S=2, T=0;
int x=0;
```

```
{
     wait(T);
     x=5;
     signal(S);
     x=6;
}
```

Processo P_1

```
{
     wait(S);
     x=1;
     signal(T);
     wait(S);
     x=10;
     write(x);
}
```

Processo P_2

Esercizio 5 (cont.)

- Si supponga che i processi vengano eseguiti concorrentemente sulla stessa CPU:
  - Determinarne tutti i possibili output.

    Con **`S=2`** e **`T=0`**, l'insieme di possibili output è {5,6,10}
  - Cosa succede se inizialmente i semafori hanno valore **`S=1`** e **`T=0`**?

    Con **`S=1`** e **`T=0`**, l'insieme di possibili output è {6,10}
  - Nel caso **`S=1`** e **`T=0`**, inserire una sola nuova chiamata **`wait()`** su **`S`** o **`T`** in uno dei processi P_1 e P_2 in modo da ottenere come unico possibile output il valore 10.

    Ponendo **`wait(T)`** prima di **`x=6`** in P_1 si forza l'output a 10

Esercizio

- Si considerino i processi A, B e C che si sincronizzano attraverso i semafori **`Sem1`**, **`Sem2`** e **`Sem3`** (inizializzati a 0) e che operano sulle variabili condivise **`x`**, **`y`** e **`z`**, inizializzate rispettivamente a 1, 2 e 1.

```
{
    wait(Sem1);
    z=(x-z)*y;
    x=x+z+y;
    signal(Sem3);
    wait(Sem1);
    x=x+y;
    print(x);
}
```

Processo A

```
{
    wait(Sem2)
    x=x+y;
    z=x-z;
    y=(y-z)+x;
    print(y);
    signal(Sem1);
}
```

Processo B

```
{
    signal(Sem2);
    wait(Sem3);
    x=x/y;
    y=2*z+x;
    signal(Sem1);
    z=x+z;
    print(z);
}
```

Processo C

- Con quale ordine i processi stampano i valori delle tre variabili? Quali valori vengono stampati?
- Motivare la risposta data.

Fine del Capitolo 7