💙

Lezione 7.3

Semafori

Un semaforo S è una variabile intera cui si può accedere escludendo l'inizializzazione, solo tramite due operazioni atomiche predefinite: **wait()** e **signal()**

Definizione di wait() in pseudocodice:

```
wait(S) {
    while (S <= 0); // attesa attiva
    S--; // decremento il semaforo perché mi approprio di un'unità
          // mi approprio di un'unità libera
}
```

Se:

- Vale > 0 → E' disponibile un'unità
- Vale == 0 → Non è disponibile l'unità

Il ciclo while (S ≤ 0) è detto **busy waiting**, cioè *attesa attivai:* si consumano cicli di CPU. Cicla se non sono presenti unità

brendyz13 e giosacarota

Per la wait anche le istruzioni (S ≤ 0) e S— sono istruzioni atomiche

Definizione di signal() in pseudocodice:

```
signal(S) {
    S++; // restituisco l'unità di cui mi sono appropriata e
          // quindi incremento il contatore delle unità
}
```

Se fosse solo uguale a 0 o a 1 è presente una singola unità, quindi rimarrei bloccato se usassi questo codice

Le operazioni atomiche garantiscono la mutua esclusione

Uso dei semafori

Distinguiamo quindi due tipi di semafori:

- **Semafori contatori**: Il valore è un numero intero. Controllano l'accesso ad una data risorsa, presente in un numero finito di esemplari. Il semaforo, inizialmente, è impostato sul numero di risorse disponibili. Se vale 0, tutte le risorse sono occuopate
- **Semafori binari**: Il valore è limitato a 0 e 1 (li vediamo dopo)

Realizzazione di mutua esclusione con semafori

Variabili condivise:

`semaforo mutex;` → inizialemnte mutex = 1

Processo P_i:

```
do {
    wait(mutex);
        // sezione critica
    signal(mutex);
        // sezione non critica
} while (1);
```

Realizzazione di un semaforo

Nell'implementazione che abbiamo appena visto dei semafori si necessità dell'*attesa attiva.* Vediamo adesso un'implementazione che non ne necessiti

Si può definire un semaforo come una struttura:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Andiamo quindi a modificare anche la definizione delle operazioni wait() e signal()

Operazione wait():

@redyz13 e @rosacarota

```
wait(semaphore *s) {
    s->value--;
    if(s->valuse < 0) {
        aggiungi questo processo a s->list;
        block();
    }
}
```

Si decrementa il valore della risorsa e poi si controlla se positivo o meno; se risulta essere negativo vuol dire che non sono presenti risorse e che quindi quel processo deve essere aggiunto nella coda dei processi.

L'operazione di bloccaggio pone il processo nella coda di processi associata al semaforo (e si pone lo stato del processo in waiting)

Si trasferisce il controllo allo scheduler di CPU che sceglie un processo pronto per l'esecuzione

```
signal(semaphore *s) {
    s->value++;
    if (s->value <= 0) {
        togli un processo dalla coda dei processi s->list
        wakeup(P); // P è il processo
    }
}
```

Un processo bloccato dovrà essere riavviato a seguito dell’operazione signal() su s.

All’inizio si riassegna l’unità per capire se c’è qualcuno che stava aspettando

Se s < 0 vuol dire che c’era qualcuno che stava aspettando che qualche unità si liberasse (all’arrivo del processo P che ha dovuto aspettare nella coda dei processi, il valore di value era già 0, quindi all’arrivo ha diminuito il valore del semaforo è < 0)

Il processo si riavvia tramite un’operazione wakeup(), che modifica lo stato del processo da waiting a ready. Il processo entra nella coda dei processi pronti

In sostanza:

block(): sospende il processo che la invoca e pone il processo in una coda d’attesa associata al semaforo

wakeup(P): pone in stato di pronto per l’esecuzione un processo é bloccato

Sono due chiamate di sistema

Questa definizione dei semafori porta anche a valori negativi del contatore: i valori negativi rappresentano, in valore assoluto, la quantità di processi che sta aspettando nella lista dei processi del semaforo

La lista dei processi si realizza tramite un campo puntatore nel PCB del processo

@redyz13 e @rosacarota

Un modo per aggiungere e togliere processi dalla lista assicurantdo l’attes limitata è utilizzando una coda FIFO: il semaforo conterrà sia il puntatore al primo che all’ultimo elemento.

E’ garantito che le due operazioni di **wait()** e di **signal()** vengano eseguite in modo atomico tramite l’inibizione degli interrupt in un sistema monoprocessore

Questa definizione dei semafori non consente di rimuovere del tutto la *busy waiting*: essa, però, si limiterà ad esserci solamente alle sezioni critiche delle due istruzioni wait() e signal(), che sono abbastanza brevi

Implementazione di un semaforo contatore tramite semafori binari

Necessitiamo di queste strutture dati:

- **semaforo-binario S1 e S2**
- **int C**

Inizializzazione:

- **S1 = 1** → S1 garantisce la mutua esclusione su C
- **S2 = 0** → Serve per utilizzare la coda dei processi associato al semaforo
- **C** = valore iniziale del semaforo contatore S

wait:

```
wait(S1); // controllo se posso decrementare C
C--;
if (C < 0) { // se C < 0 prima di bloccarmi in wait, devo
             // "rimettere S1 a verde" tramite la signal
    signal(S1);
    wait(S2);
}
signal(S1);
```

signal:

```
wait(S1);
C++;
if (C <= 0()
    signal(S2);
else
    signal(S1);
```

```
    Queue<process> q;

} P(semaphore s)
{
    if (s.value == 1) {
        s.value = 0;
    }
    else {
        // add the process to the waiting queue
        q.push(P) sleep();
    }
}
V(Semaphore s)
{
    if (s.q is empty) {
        s.value = 1;
    }
    else {

        // select a process from waiting queue
        Process p = q.front();
        // remove the process from wating as it has been
        // sent for CS
        q.pop();
        wakeup(p);
    }
}
```

@redyellow @rosacarota

Perché signal(s1) è presente solo una volta nell'else della signal e due volte nella wait?

Nel caso ci bloccassimo in wait(S2) poichè il contore C è diventato < 0, appena uno dei processi che stava utilizzando l'unità di elaborazione restituirà la risorsa, entrerà nell'if (C ≤ 0) e ci sbloccherà tramite signal(S2); ripartiremo da dove abbiamo lasciato il codice (cioè da wait(S2)) e a questo punto dovremmo noi "rimettere apposto S1" attivato (tramite

wait(S1) di signal) dal processo che ha restituito la risorsa (dovremmo eseguire quell’ultimo signal(S1) del nostro codice).

Problemi tipici di sincronizzazione

Esamineremo alcuni problemi di sincronizzazione, come esempio di classi di problemi connessi al controllo della concorrenza

Produttore/consumatore con memoria limitata

Produttore e consumatore condividono le seguanti strutture dati:

- int n;
- semaphore mutex = 1; → Serve per accedere in mutua esclusione
- semaphore empty = n;
- semaphore full = 0;

Supponiamo quindi di avere di una certa quantità di memoria rappresentata a un buffer con n posizioni

I semafori empty e full conteggiano rispettivamente il numero di posizioni vuote e piene nel buffer

Produttore:

```
do {
    /* produce un elemento in next_produced */

    wait(empty); // conta il numero di posizioni vuote
    wait(mutex); // se c'è una posizione vuota accedo in mutua
                 // esclusione

    /* inserisce next_produced in buffer */

    signal(mutex);
    signal(full);
} while (true);
```

@redyz13 e @rosacarota

Consumatore:

```
do {
    wait(full); // controlla che ci sia qualcosa da consumare
    wait(mutex);

    /*rimuovi un elemento dabuffer e mettilo in next_consumed*/

    signal(mutex);
    signal(empty); // segnala che c'è una posizione vuota

    /* consuma l'elemento contenuto in next_consumed */
} while (true);
```

Problema di lettori-scrittori

Si consideri un insieme di dati (ad es. un file) che si deve condividere tra processi concorrenti

Alcuni processi leggeranno (**lettori**) altri aggiorneranno (**scrittori** = lettura + scrittura)

Se più lettori accedono concorrentemente all'insieme di dati condiviso non c'è nessun problema

Se uno scrittore accede, gli altri processi non possono accedere

Gli scrittori devono avere accesso esclusivo

Quindi il problema (primo di due problemi su lettore-scrittore), è che un lettore non attenda a meno che uno scrittore abbia già ottenuto il permesso di usare l'insieme di dati condiviso

Variabili condivise:

- **int read_count = 0** → contiene il nuemri di processi che stanno leggendo
- **semaphore mutex = 1** → Serve a garantire la mutua esclusione al momento dell'aggiornamento di read_count
- **semaphor rw_mutex = 1** → E' comune ad entrambi i processi: serve per la mutua esclusione degli scrittori e anche al primo e all'ultimo lettore che entra o che esce dalla sezione critica

Scrittore:

```
do {
    wait(rw_mutex); // vede il valore del semaforo di scrittura
                    // e se nessuno sta scrivendo mette il semaforo
                    // a rosso

    /* esegui l'operazione di scrittura */

    signal(rw_mutex); // sblocca il semaforo
} while (true);
```

@redyz13 e @rosacarota

Nel caso uno scrittore stia scrivendo ed n lettori volessero entrare, si accoda un lettore a rw_mutex e n - 1 lettori a mutex

Inoltre, se uno scrittore esegue signal(rw_mutex) la scelta di riprendere l'esecuzione dai lettori di mutex o di rw_mutexe è dello scheduler

Lettore:

Bisogna capire se il processo è il primo a leggere o meno. In caso lo fosse, probabilmente starebbe scrivendo qualcuno

```
do {
    wait(mutex);  // mutua esclusione per aumentare
                  // il count dei lettori
    read_count++;
    if (read_count == 1) // verifico se sono il primo
        wait(rw_mutex);  // controllo se sta scrivendo qualcuno:
                         // se si mi blocco
    signal(mutex); // sblocco il semaforo per gli altri lettori

    /* esegui l'operazione di lettura */

    wait(mutex); // serve per decrementare il count dei lettori
    read_count--;
    if (read_count == 0) // se vale 0, nessun altro oltre me stava
```

```
        signal(rw_mutex); //leggendo quindi avverto che si può scrivere volendo
    signal(mutex); // riattivo mutex
} while (true);
```

Problema dei cinque filosofi (dining philosophers)

Si considerino 5 filosofi che trascorrono la loro esistenza epnsando e mangiando,
Al centro del tavolo c'è una zuppiera con il riso e la tavola è apparecchiata con cinque bacchette

Quando un filosofo pensa, non interagisce con gli altri; 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 sola bacchetta alla volta e non può prendere una bacchetta che si trova già nelle mani di un suo vicino

Quando il filosofo affamato tiene in mano due bacchette contemporaneamente, mangia senza lasciare le bacchette e, terminato il pasto, le posa e riprende a pensare

Una soluzione a questo problema è rappresentare la bacchetta come se fosse un semaforo: ogni filosofo cerca di afferrare la bacchetta tramite l'operazione di wait() e di posarla tramite l'operazione di signal()

Variabili condivise:

semaphore chopstick[5]; → tutti i valori sono 1

@redyz13 e @rosacarota

```
do {
    wait(chopstick[i]); // controlla se può prendere la bacchetta a Sx
    wait(chopstick[(i + 1) % 5]); // controlla se può prendere la bacchetta a DX

    /* mangia */

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

    /* pensa */
} while (true);
```

Questa soluzione garantisce che due vicini non mangino contemporaneamente ma è insufficiente: immaginiamo che tutti i filosofi abbiano fame contemporaneamente: ci ritroveremmo che tutti i filosofi prenderebbero la bacchetta alla loro sinistra in contemporanea. Tutti gli elementi di chopstick diventerebbero 0, perciò tutti i filosofi che proverebbero a prendere la loro bacchetta di destra andrebbero in stallo.

Possibilità di stallo = **deadlock:** più processi rimangono bloccati poiché aspettano una risorsa da un altro processo che aspetta a sua volta

Questa soluzione di stallo potrebbe essere evitata tramite queste 4 possibilità:

- solo quattro filosofi possono stare contemporaneamente a tavola;
- un filosofo può prendere le sue bacchette solo se sono entrambe disponibili (cioè quest'operazione deve essere eseguita in sezione critica)

- Un filosofo dispari prende prima la bacchetta di sinistra e poi quella di destra, un filosofo pari prende prima la bacchetta di destra e poi quella di sinistra

(La seconda possibilità sarà la nostra soluzione)

Inoltre una soluzione soddisfacente deve escludere l’attesa indefinita:

- cioè che uno dei filosofi muoia di fame, da cui il termine **starvation**

@redyz13 e @rosacarota