PCD Module 2.4

Coordinamento dei processi: meccanismi base

Semafori

- Funziona come un normale semaforo: blocca e sblocca l'esecuzione di processi in base alle necessità.
- Tipo di dato primitivo fornito dalla macchina concorrente

Tipo di dato

- tipo di dato composto con due campi:
  - **S.V**, int >= 0 (inizializzato con k >= 0)
  - **S.L**, set di processi (id) (inizializzato con set vuoto)
- operazioni atomiche:
  - **wait(S)** (P(S)). Definizione, con p=processo che esegue wait:
    - se S.V > 0, S.V-- (semaforo verde)
    - altrimenti, S.L aggiunge p. p.state = blocked (semaforo rosso, processo bloccato)
  - **signal(S)** (V(S)). Definizione:
    - se S.L vuoto (nessun processo in attesa), S.V += 1
    - altrimenti, rimuove dalla lista e imposta a ready (sbloccato) un elemento arbitrario di S.L

Invarianti del semaforo

k = valore iniziale di S.V, `#signal(S)` n° di signal(S) eseguiti e `#wait(S)` n° di wait eseguite.

Un processo che è bloccato mentre esegue wait(s) non esegue con successo l'istruzione.

Teorema: un semaforo S soddisfa le seguenti invarianti: - `S.V >= 0` - `S.V = k + #signal(S) – #wait(S)`

Tipi di semaforo

- **Mutex** (binari), con S.V = {0,1}
- **General** (contatori), con S.V di valore qualsiasi >= 0
- **Event**, inizializzati a 0, con scopo di sincronizzazione

Definizioni

- **strong**: S.L non è un set ma una **coda**
  - non c'è starvation (impossibile)
- **weak**: S.L è un **set**
- **busy-wait**: non c'è S.L. Sono ancora atomici, non c'è interleaving tra le due istruzioni: `wait(S) = await(S.V > 0); S.V -= 1` e `signal(S) = S.V += 1`.
  - Può esserci starvation: non è scontato che che un processo riesca ad entrare nella sezione critica.
  - Appropriati in sistemi multiprocessore, quando il processo che attende ha il suo processore e non consuma tempo di CPU che può essere usat per altri calcoli.

Utilizzo

I semafori possono essere usati come blocchi a basso livello per risolvere quasi ogni problema riguardante l'interazione tra processi, tra cui **mutex** (critical section) e **sincronizzazione** (event semaphore/barriers).

CS con semafori

Usando un semaforo, la soluzione al problema della CS con 2 processi è triviale, usando un semaforo come **lock**.

```
CS con semafori: 2 processi
sem_bin S = (1, {})
{P, Q}
loop forever
1: NCS
2: wait(S)
3: CS
4: signal(S)
```

Correttezza: costruendo il diagramma a stati ridotto e controllando le proprietà può essere verificato che la soluzione del semaforo al problema della CS è corretto: non c'è mutex ed è libero da deadlock e starvation. Diagramma a stati pg. 15

```
CS con semafori: 2 processi (abbreviato)
sem_bin S = (1, {})
{P, Q}
loop forever
1: wait(S)
2: signal(S)
```

CS con N processi: stessa soluzione applicata a N processi, non c'è più libertà da starvation.

Utilizzo di semafori per la sincronizzazione

I semafori forniscono meccanismi base per sincronizzare i processi, risolvendo così problemi relativi all'ordine di esecuzione.

Semafori ad eventi: usati per inviare/ricevere un segnale temporale. Inizializzati a (0, {}). Esempio merge sort:

```
sem_bin S1 = (0,{})
sem_bin S2 = (0,{})
int[] A = []

sort1                    sort2                    merge
p1: sort 1st half of A   q1: sort 2nd half of A   r1: wait(S1)
p2: signal(S1)           q2: signal(S2)           r2: wait(S2)
                                                  r3: merge halves of A
```

Problemi comuni

Producer/consumer

- **producer**: esegue `produce()` per creare un elemento `data` che viene poi inviato al processo consumer
- **consumer**: al ricevimento di un elemento dal producer, esegue `consume(data)`.

Quando un elemento viene inviato da un processo a un altro, la comunicazione può essere:

- *sincrona*: la comunicazione non avviene finchè P e C non sono pronti
- *asincrona*: il canale di comunicazione ha della capacità di archiviare dei dati.
    - La separazione è molto utile per sistemi dinamici/aperte (separazione temporale tra partecipanti, set dinamico di processi)
    - È utile anche quando P e C hanno velocità diverse
    - Ha bisogno di un **buffer** per archiviare e ritrovare dati, una struttura dati con stato mutabile, letta dai consumatori e scritta dai produttori.

Buffer infinito: una sola interazione che dev'essere sincronizzata - C non deve provare a eseguire una operazione da buffer vuoto.

Invariante: nAvailItems.V = #buffer

```
buffer_illimitato buffer = {}
sem nAvailItems = (0,{}) (resource semaphore)

producer                 consumer
loop forever             loop forever
p1: item el = produce()  q1: wait(nAvailItems)
p2: buffer.append(el)    q2: item el = buffer.take()
p3: signal(nAvailItems)  q3: consume(el)
```

Buffer finito: c'è un'altra interazione che dev'essere sincronizzata: il produttore non deve provare ad aggiungere operazioni in un buffer pieno.

Invariante: nAvailItems + nAvailPlaces = N

```
buffer_limitato buffer = {}
sem nAvailItems = (0,{}) (resource semaphore) |
sem nAvailPlaces = (N,{})                     |--> semafori separati

producer                 consumer
loop forever             loop forever
p1: item el = produce()  q1: wait(nAvailItems)
p2: wait(nAvailPlaces)   q2: item el = buffer.take()
p2: buffer.append(el)    q3: signal(nAvailPlaces)
p3: signal(nAvailItems)  q4: consume(el)
```

Combinare mutex e semafori sincronizzati: come generalizzazione dei casi precedenti, consideriamo l'uso condiviso di una struttura dati nonatomica (buffer), quindi con operazioni nonatomiche. Introduciamo una mutex per garantire anche la mutex.

```
buffer_limitato buffer = {}
sem nAvailItems = (0,{}) (resource semaphore) |
sem nAvailPlaces = (N,{})                     |--> semafori separati
sem_bin mutex = (1,{})

producer                 consumer
loop forever             loop forever
p1: item el = produce()  q1: wait(nAvailItems)
p2: wait(nAvailPlaces)   q2: wait(mutex)
p3: wait(mutex)          q3: item el = buffer.take()
p4: buffer.append(el)    q4: signal(mutex)
p5: signal(mutex)        q4: signal(nAvailPlaces)
p3: signal(nAvailItems)  p4: consume(el)
```

Filosofi a cena

Problema relativo a 5 computer che competono per accedere 5 nastri condivisi. 5 filosofi eseguono due operazioni: *pensano* e *mangiano*. I piatti sono presi in comune ad un tavolo con 5 piatti e 5 forchette. Al centro del tavolo c'è una ciotola di spaghetti che viene riempita all'infinito. Gli spaghetti sono ingrovigliati, perciò ogni filosofo ha bisogno di 2 forchette per mangiare. Ogni filosofo può prendere le forchetta a destra o sinistra, ma una sola forchetta per volta.

Proprietà

```
loop forever
p1: think
p2: <pre-protocol>
p3: eat
p4: <post-protocol>
```

- un filosofo può mangiare solo se ha 2 forchette
- **mutex**: due filosofi non possono tenere la stessa forchetta contemporaneamente
- **libertà da deadlock**
- **libertà da starvation**
- comportamento efficiente in assenza di collisioni

Primo tentativo

```
sem_array[5] fork = [1,1,1,1,1]
loop forever
p1: think
p2: wait(fork[i])
p3: wait(fork[(i+1)%N])
p3: eat
p4: signal(fork[i])
p5: signal(fork[(i+1)%N])
```

- Ogni forchetta è un semaforo: con wait prende la forchetta, con signal la rilascia
- Può essere provato che nessuna forchetta è mai tenuta da due filosofi
- Crea **deadlock** (v. piu avanti): se tutti i filosofi prendono la forchetta a sx, uno prova a prendere la forchetta a dx (non disponibile)

Soluzione 1: tickets

Limitare il numero di filosofi che mangiano simultaneamente, introducendo il ticket "pasto": N-1 ticket per N filosofi.

```
sem_array[5] fork = [1,1,1,1,1]
sem ticket = (4,{})
loop forever
p1: think
p2: wait(ticket)
p3: wait(fork[i])
p4: wait(fork[(i+1)%N])
p5: eat
p6: signal(fork[i])
p7: signal(fork[(i+1)%N])
p8: signal(ticket)
```

Soluzione 2: rompere la catena wait-for

- Non puoi avere deadlock se l'ultimo filosofo prende prima la forchetta a dx e poi quella a sx, rompendo così la catena wait-for.
- Dato l'ordine totale degli identificatori delle forchette, l'ultimo filosofo prende la forchetta in ordine opposito rispetto all'ordine dei filosofi: prima quella a N-1 e poi quella a 0
- La soluzione riguarda scegliere la forchetta sempre nello stesso ordine

```
sem_array[5] fork = [1,1,1,1,1]
int first = min(i, (i+1)%N)
int second = max(i, (i+1)%N)
loop forever
p1: think
p3: wait(fork[first])
p4: wait(fork[second])
p5: eat
p6: signal(fork[first])
p7: signal(fork[second])
```

Lettori/scrittori

Astrazione del problema di accesso a un db. Divisione dei processi in due classi:

- **lettori**, escludono gli scrittori ma non altri lettori
- **scrittori**, escludono scrittori e lettori

Non c'è quindi problema ad avere processi che leggono dati concorrentemente, ma la scrittura deve avvenire in mutua esclusione.

La soluzione deve soddisfare queste invarianti (nR = n° lettori, nW = n° scrittori)

```
nR >= 0
nW = 0 || nW = 1
(nR > 0 -> nW = 0) && (nW = 1 -> nR = 0)
```

Soluzione sovravincolata tramite singolo semaforo che funziona come blocco:

```
sem_bin rw = (1,{})
database db;

reader                  writer
loop forever            loop forever
p1: wait(rw)            q1: wait(rw)
p2: item el = db.read() q2: item el = create()
p3: signal(rw)          q3: db.write(el)
                        q4: signal(rw)
```

Soluzione: i lettori non usano lo stesso blocco degli scrittori --> `mutexR` : blocco dei lettori per aggiornare strutture dati in comune (nr)

```
sem_bin mutexR = (1,{})
int nr = 0
sem_bin rw = (1,{})
database db

reader                      writer
loop forever                loop forever
p1: wait(mutexR)            q1: wait(rw)
p2: if(nr==0)               q2: item el = db.create()
p3:    wait(rw)             q3: db.write(el)
p4: nr+=1                   q4: signal(rw)
p5: signal(mutexR)
p6: item el = db.read()
p7: wait(mutexR)
p8: nr-=1
p9: if(nr==0)
p10:   signal(rw)
p11: signal(mutexR)
```

Deadlocks

Situazione in cui due o più azioni concorrenti aspettano che l'altra finisca, e nessuno lo fa. Condizioni necessarie affinchè ci sia deadlock:

- **mutex**: una risorsa che non può essere usata da più di un processo per volta
- **hold and wait**: processi che trattengono risorse potrebbero richiedere nuove risorse
- **no preemption**: nessuna risorsa può essere rimossa dal processo che lo trattiene. Le risorse possono essere rilasciate solo tramite esplicita azione del processo.
- **circular wait**: due o più processi da una catena circolare dove ogni processo attende per una risorsa che il prossimo processo in catena mantiene.

Deadlock con blocchi: accade quando più thread aspettano all'infinito a causa di dipendenze cicliche bloccanti, es. quando il thread A blocca L e prova a bloccare M, ma allo stesso tempo il thread B blocca M e prova ad acquisire L.

Ricerca e ripristino: ad esempio i database sono progettati per trovare e ripristinare dai deadlock. Il set di transazioni deadlockate si identifica analizzando il grafo di dipendenze *is-waiting*, dove si cercano cicli e, se presenti, si sceglie una vittima e si cancella la transazione. I meccanismi non sono automatici con JVM, bisogna chiudere l'applicazione.

Regola generale per evitare deadlock: N processi che condividono e acquisiscono più blocchi. Regole: assegnare un ordine totale ai blocchi e acquisire i blocchi sempre nello stesso ordine. Funziona perchè rende possibile avere dipendenze wait-for circolari tra processi.

Monitor

- Astrazione dati per la programmazione concorrente che incapsula **sincronizzazione** e **mutua esclusione** in accesso (stato + operazioni + policy di concorrenza).
- Generalizzazione del concetto del kernel o supervisore nei sistemi operativi, dove le CS come l'allocazione di memoria sono centralizzate in un programma privilegiato
  - Le app richiedono servizi che sono forniti dal kernel
  - I kernel vengono eseguiti in una modalità hw che non interferisce con le app
  - Monitor come versioni decentralizzate del kernel monolitico
- Generalizzazione della nozione di oggetto in OOP (classe che incapsula dati + operazione + sync/mutex)

Definizione astratta

```
monitor MonitorName {
   declaration of permanent variables
   initialization statements
   procedures (or entries)
}
```

Proprietà e feature

Tipi di dato astratto

- Monitor come **istanze di tipi di dato astratto**
  - Solo i nomi delle procedure sono visibili fuori dal monitor (interfacce)
  - Istruzioni all'interno del monitor non possono accedere a variabili dichiarate fuori dal monitor
  - Variabili permanenti sono inizializzate prima che ogni procedura è chiamata

Mutua esclusione

- **Mutua esclusione** implicita/intrinseca
  - Procedure eseguite di default in mutex

- Procedura del monitor chiamata da un processo esterno, attiva se un processo sta eseguendo un'istruzione nella procedura.
- Almeno un'istanza di una procedura del monitor potrebbe essere attva per volta
- Processi che trovano il monitor occupato sono sospesi.

Esempio: contatore classico, thread-safe

```
monitor Counter {
  int count;
  procedure inc(){
    count := count + 1
  }

  procedure getValue():int {
    return count;
  }
}
```

Osservazioni: mutex implicita, non richiede che i programmatori usino altri meccanismi (wait/signal/etc). Se le operazioni dello stesso monitor sono chiamate da più di un processo, l'implementazione assicura che queste siano eseguite sotto mutex (atomicamente). Se le operazioni di diversi monitor sono chiamate, la loro esecuzione può essere sovrapposta. Non c'è una coda esplicita associata con la procedura monitor (starvation).

Sincronizzazione

- Supporto esplicito alla **sincronizzazione**
  - tramite **variabili di condizione**, usate nei monitor dai programmatori per ritardare un processo che non può continuare ad essere eseguito in sicurezza finchè lo stato del monitor soddisfa qualche condizione booleana. Usato anche per risvegliare un processo ritardato quando la condizione diventa true.

Variabili di condizione: tipi di dato primitivi che possono essere usati per sospendere (wait) e riesumate (signal) processi dentro ai monitor. Rappresentano le condizioni (eventi) dello stato del monitor che aspettano di essere soddisfatte e che diventano soddisfatte. Due operazioni atomiche di base, `waitC` e `signalC`. Ogni v.c. è associata a una coda FIFO di processi bloccati.

- **`waitC`** : sospende l'esecuzione del processo e rilascia il blocco del monitor
- **`signalC`** : sblocca un processo in attesa di una condizione

```
waitC(cond) { signalC(cond) { cond.queue.push(p) if(!cond.queue.empty) { p.state = blocked q = cond.queue.pop() monitor.lock = releas
```

Esempio:

```
monitor SynchCell {
   int value
   bool available = false
   cond isAvail

   void set(int v) {
     value = v
     available = true
     signalC(isAvail)
   }

   int get() {
     if(!available)
        waitC(isAvail)
     return value
   }
}
```

Osservazioni: c'è un collegamento esplicito tra variabili di condizione e il monitor che le raccchiude. **`wait`** **rilascia il blocco del monitor**, ciò è essenziale per evitare che un processo che esegua `waitC` blocchi l'accesso al monitor.

Altre primitive:

- `emptyC(cond)` controlla se la coda è vuota
- `signalAllC(cond)` come signal, ma tutti i processi aspettano che le condizioni siano riesumate
- `waitC(cond,rank)` attende in ordine crescente il valore di rank
- `minrank(cond)` ritorna il valore di rank del processo in fronte alla coda di attesa

Esempio SynchCell, revisited:

```
monitor ImprovedSynchCell {
  int value;
  boolean available;
  cond isAvail;
  procedure set(int v){
    value := v
    available := true
    signalAllC(isAvail)
}
  procedure get():int {
    if (!available)
      waitC(isAvail)
    return value
}
}
```

```
monitor ImprovedSynchCell {
  int value;
  boolean available;
  cond isAvail;
  procedure set(int v){
    value := v
    available := true
    signalC(isAvail)
}
  procedure get():int {
    if (!available){
      waitC(isAvail)
      signalC(isAvail)
    }
    return value
  }
}
```

Implementare un semaforo

```
monitor Semaphore {
  integer s := <InitValue>
  cond notZero
  procedure wait(){
    if s = 0
      waitC(notZero)
    s := s - 1
  }
  procedure signal(){
s := s + 1
    signalC(notZero)
  }
}
```

```
monitor Semaphore {
  integer s := <InitValue>
  cond notZero
  procedure wait(){
    if s = 0
      waitC(notZero)
    else
s := s - 1 }
  procedure signal(){
    if emptyC(notZero)
s := s + 1 else
      signalC(notZero)
  }
}
```

Semafori vs Variabili di Condizione

Il semaforo:

- `wait` potrebbe/non potrebbe bloccare
- `signal` ha sempre un effetto
- `signal` slocca un processo arbitrariamente bloccato

- Un processo sbloccato da un segnale può riesumare l'esecuzione immediatamente

Il monitor: - `waitC` blocca sempre - `signalC` non ha effetto se la coda è vuota - `signalC` sblocca il processo in testa alla coda - un processo sbloccato da `signalC` deve attendere il processo segnalante per lasciare il monitor

Disciplina della segnalazione

Semantica

- S = precedenza ai processi segnalanti
- W = precedenza ai processi in attesa
- E = precedenza ai processi bloccati su una procedura

- **signal and continue**: il segnalatore continua e il processo segnalato esegue più avanti nel tempo. Non preemptive. **E < W < S**
- **signal and wait**: il processo segnalato esegue e il segnalatore aspetta, eventualmente in competizione con altri processi in attesa di entrare nel monitor. Preemptive. **E = S < W**
- **signal and urgent wait**: come signal e wait, ma il segnalatore ha priorità sui processi in attesa di blocco. Soluzione classica per i monitor. **E < S < W**

Problemi comuni usando i monitor

Produttori/consumatori

Lettori/scrittori

Allocazione e gestione di risorse

Codice sulle slide!