# Programmazione Concorrente

Luca Tagliavini

19 luglio 2022

## Indice

# 1 Introduzione

Fino ad ora abbiamo scritto programmi sequenziali imperativi, nella quale si ha una sola istanza del nostro codice che viene eseguita istruzione dopo istruzione, in modo ordinato. I programmi interagiscono con il sistema operativo tramite *system call*. Molte chiamate sono incluse nello standard `POSIX` e ne consentono la portabilità tra sistemi operativi che implementano tale specifica.

La programmazione concorrente viene trattata nel corso di sistemi operativi poichè la gestione concorrente dei processi è uno dei compiti che un SO deve svolgere, ed è dunque integrale sapere quali siano le sfide derivanti da questa programmazione. Un programma concorrente è un software che svolge più cose *contemporaneamente*. Al fine di comprendere questi programmi occorre che essi siano modellati, per poterli comprendere e implementare. È inoltre necessario introdurre il concetto astratto di processo, che non combacia mai con il programma stesso.

## 1.1 Processo

Un processo è semplicemente un programma in esecuzione. Esso svolge una serie di passi di avanzamento, che lo portano da uno *stato* ad uno successivo. Nella programmazione imperativa è chiaro che il nostro software esegue una istruzione alla volta avanzando passo a passo.

Un processo può essere visto come la memoria e le risorse ad esso allocate, come i dati contenuti da esso nei registri del processore ma anche come lo stato di esecuzione in cui si trova. A noi interesserà una visione più astratta, ignorando queste rappresentazione fisiche e parlando di un processo come una semplice istanza di esecuzione.

## 1.2 Assioma del Finite Progress

Questo assioma ci assicura che *ogni processo* ***non bloccato*** *prima o poi avanza*. In altro modo: ogni processo viene eseguito ad una velocità finita *non nulla*, ma sconosciuta.

Senza questo assioma anche un programma semplicissimo come la stampa di `HelloWorld` potrebbe non terminare mai poichè non avanza mai fino all'ultima istruzione.

## 1.3 Parallelismo

Il parallelismo, ovvero l'esecuzione di più compiti contemporaneamente, è di due tipi: reale e apparente. In quello *reale* si possono avere più macchine, più *core* del processore che collaborano tra di loro e dove si hanno in un dato istante due istanze del programma che eseguono istruzioni diverse (**overlapping**). Nel parallelismo *apparente* si ha un maggior numero di processi in esecuzione rispetto agli effettivi chip paralleli a disposizione: in un dato istante di tempo si hanno solo $n$ processi in esecuzione, pari al numero di core paralleli a disposizione,

tuttavia molti più software possono essere aperti contemporanemante e tutti si alternano a turni per sfruttare il tempo di calcolo (**interlapping**). All'umano che osserva appare come se i processi stiano avanzando nello stesso momento, tuttavia stanno in realtà facendo a turno molto velocemente.

Le sfide del paradigma della programmazione concorrente saranno la comunicazione e la sincronizzazione tra processi.

> Nella pratica si usa la parola *processo* qunado si parla di esecuzioni che non condividono memoria, mentre *thread* per indicare istanze che la condividono tra di loro.

## 1.4 Prime problematiche

Si pensi ad una semplice funzione che somma o sottrae ad una variabile *count* un valore $x$. Se immaginiamo esistano due istanze del programma che condividono la stessa cella di memoria per il valore *count*, esistono vari stati finali dati dall'esecuzione di un programma $+10$ e $-10$. È importante ricordarsi che ogni programma è composto da tre istruzioni separate:

1. **lettura** del dato *count* dalla memoria in un registro.
2. **aggiornamento** del valore nel registro tramite incremento o decremento.
3. **scrittura** del valore del registro nella memoria in posizione *count*.

Eseguendo le due istanze in parallello le istruzioni sono redistribuite nel tempo in ordine non garantito, e dunque possono avvenire errori di calcolo e discrepanze, dovute alla modifica concorrente di una variabile condivisa in memoria. Dunque può capitare che un processo sovrascrive la computazione svolta dall'altra controparte.

In un sistema concorrente una computazione che dipende dall'ordine casuale in cui vengono eseguite più computazioni parallele vengono chiamate **race condition**. Vogliamo evitare a tutti i costi questi avvenimenti in quanto rendono il programma non-deterministico.

## 1.5 Interazioni tra processi

È possibile classificare i processi in base a quanto sono *consapevoli* tra di loro:

1. **Ignari**: processi che sono completamente ignari dell'esistenza altrui competono (concorrenza) per l'accesso alle risorse e devono dunque avere un modo di *sincronizzarsi* (offerto dal sistema operativo).
2. **Indirettamente comunicanti**: che non conoscono il PID altrui ma comunicano tramite una risorsa che posseggono in comune (un buffer ad esempio).
3. **Direttamente comunicanti**: hanno una comunicazione diretta per mezzo dei loro ID offerta dal sistema operativo. La comunicazione avviene spesso tramite lo scambio di messaggi.

## 1.6 Iniziamo a risolvere il problema

Per risolvere il problema della sincronizzazione di due processi concorrenti è essenziale capire **quali** sono operazioni *atomiche*, inscindibili, e dunque operazioni nelle quali non può avvenire una discrepanza dovuta alla modifica dei dati da un agente terzo durante la computazione.

Noi vorremmo che le operazioni critiche diventassero *atomiche*, ovvero che le operazioni vengano eseguite in *mutua esclusione*. Questa funzionalità non è offerta dal processore, ma deve essere implementata lato software.

## 1.7 Azioni atomiche (di sistema)

Alcune operazioni di sistema sono già atomiche di natura. Ad esempio, operazioni che vengono svolte con una sola istruzione sono atomiche per proprietà hardware. In un contesto di parallelismo reale questa proprietà viene garantita dall' hardware che garantisce l'indipendenza dei *core*. Nel parallelismo apparente l'avvicendamento dei due processi, *context switch*, avviene prima e dopo l'operazione atomica il che ne preserva le proprietà (non interferisce).

In seguito useremo la sintassi `<S>` per indicare che una operazione dev'essere eseguita in modo atomico.

## 1.8 Proprietà comuni

Esistono alcune proprietà comuni ai programmi concorrenti:

- **safety**: non succederà qualcosa di male.
- **liveness**: succederà qualcosa di buono (non si va in loop infinito).

È importante svolgere i pezzi *critici* del programma, quelli che possono andare in conflitto con altri processi, in una *critical section*. Vorremo dunque prefissare e suffissare queste operazioni con l'entrata e l'uscita nella *sezione critica*. In questa sezione critica saranno rispettate le seguenti regole: **mutua esclusione** (liveness), **no deadlock**, **no ritardi** (non necessari), **no starvation**.

### 1.8.1 Deadlock (safety)

Il principio dell'*deadlock* può essere espresso con il seguente esempio: abbiamo due risorse $a$ e $b$ a cui due processi vogliono accedere in maniera *esclusiva*. Se il primo vuole prendere $a$ poi $b$ e il secondo $b$ poi $a$, e si parte prendendo uno $a$ e l'altro $b$, nessuno riuscirà mai a scambiarsi il possesso, e ci sarà dunque un blocco infinito (ci si blocca a vicenda).

### 1.8.2 Starvation e ritardi (liveness)

La starvation è la situazione in cui altri processi approfittano di tutte le risorse e dunque un processo interessato viene bloccato per l'avidità altrui. Il processo in un certo senso *muore di fame* (trad. let.) per carenza di risorse.

Inserendo ritardi superflui è un modo analogo per rubare tempo ad altri processi e generare una problematica analoga a quella della starvation.

### 1.9 Sezioni critiche

Useremo la sintassi `[enter cs]` e `[exit cs]` per aprire e chiudere un "blocco" del programma denominata *sezione critica* che deve dunque essere eseguita in mutua esclusione e rispettando le proprietà elencate in precedenza.

Programmi diversi hanno bisogno di sezioni critiche diverse in base alle risorse a cui fanno accesso e come le usano. Perciò non si può semplicemente delegare questo compito al Sistema Operativo. C'è bisogno che il programmatore usi il proprio intelleto per trvoare la soluzione più adatta al problema di concorrenza sotto esame.

## 2 Approccio di Dekker e Peterson

Seguiremo la stessa introduzione all'algoritmo di Dekker (pubblicato da Dijkstra) offerta nell'articolo della pubblicazione originale.

Vedremo in seguito la versione di Peterson che è più semplice da implementare e da generalizzare per più processi.

### 2.1 Primo tentativo

Si può usare la tecnica denominata *busy wait* nella quale si tiene una variabile condivisa *turn* che può essere impostata a 1 quando esso entra nella sezione critica, a 0 quando lo fa l'altro processo. La mutua esclusione è trivialmente garantita, così come l'assenza di deadlock, tuttavia si spreca tempo ad aspettare inutilmente.

### 2.2 Secondo tentativo

Si possono allora usare due variabili *readonly* $c_i$ all'esterno che indicano se il processo $i$ è nella critical section. Tuttavia si può verificare che entrambi i processi arrivano al controllo per vedere se la controparte è libera, entrambi vedono $false$ e dunque entrano entrambi, contemporaneamente, nella critical section, violando la *mutua esclusione*.

### 2.3 Terzo tentativo

Allora possiamo riproporre il tentativo precedente ma nel quale i processi si appropriano prima del controllo del lock. In questo modo tuttavia si può avere che entrambi bloccano la risorsa e aspettano per l'altro che la liberi, ma ciò non accadrà mai poichè stanno aspettando l'uno la mossa dell'altro. Si ha dunque un *deadlock*.

Si può anche provare una variante di questo approccio che blocca la risorsa ma se vede che l'altro la sta usando la sblocca e riprova, tuttavia si arriva sempre

ad un deadlock se i processi venogno eseguiti perfettamente all'unisono (molto raro).

## 2.4 Soluzione di Dekker

La soluzione pensata da Dekker, consiste nel combinare il tentativo 3.*b* ma usare solo nel caso in cui ci si trovi in stallo si usa la tecnica della soluzione 1.

**Algoritmo 1** Dekker

```
turn ← 0
needs_i ← {false, ..., false}
procedure DekkerProcess(i)
    while true do
        needs_i ← true
        while ∃i. needs_i do
            if turn ≠ i then
                prev ← turn
                needs_i ← false
                while turn ≠ prev do
                end while
                needs_i ← true
            end if
        end while
        ...                                   ▷ Critical section
        needs_p ← false
        turn ← i + 1
    end while
end procedure
```

### 2.4.1 Prova di correttezza

Assumiamo per assurdo che due processi $P$ e $Q$ siano contemporanemanete nella critical section. Poichè gli accessi in memoria sono esclusivi e per entrare devono camiare la variabile $need_i$. Allora supponiamo sia entrato per prima $Q$ allora $needs_Q$ sarà $true$ fino a quando $Q$ non uscirà dal ciclo. Poichè $P$ entra nella critical section mentre $Q$ lo è già significa che c'è un istante di tempo in cui $needs_Q$ è false, il che è assurdo. □

Le altre proprietà possono essere mostrate trivialmente in modo analogo.

## 2.5 Algoritmo di peterson

**Algoritmo 2** Peterson

```
turn ← 0
needs_i ← {false, ..., false}
procedure PetersonProcess(i)
    while true do
        needs_i ← true
        turn ← i
        while (∃i  needs_i) ∧ turn ≠ i do
        end while
        ...                                   ▷ Critical section
        needs_p ← false
    end while
end procedure
```

### 2.5.1 Prova di correttezza

1. **Mutua esclusione**: supponiamo che 1 sia nella sezione critica e vogliamo provare che 2 non può entrare. Poichè 1 è dentro sappiamo che $needs_1 = true$ e sappiamo che 2 entra solo quando $turn = 2$.
   Valutiamo lo stato al momento in cui 1 entra nella critical section; abbiamo due possibilità:

   - $needs_2 = false$ allora 2 dovrà ancora eseguire $needs_2 = true$ e dunque lo farà dopo 1 impedendosi di entrare.
   - $turn = 1$ analogo a prima, si preclude l'entrata.

   □

2. **Assenza di deadlock**: Supponiamo per assurdo che 1 voglia entrare nella critical section ma rimanga bloccato nel while. Allora valgono le seguenti invarianti:

   $$needs_1 = ture \quad needs_2 = true \quad turn = 2$$

   Ci sono allora tre possibili casi:

   - 2 vuole entrare nella critical section: impossibile visto che $needs_2 = true$
   - 2 è bloccato nel suo while: impossibile visto che $turn = 2$
   - 2 è nella CS e ne esce: impossibile poichè allora $needs_2 = false$ e si uscirebbe dal deadlock.

   Abbiamo provato l'assurdo. □

# 3 Soluzioni hardware

Le soluzioni software di Peterson e Dekker per quanto corrette e rispettose delle proprietà necessarie, sono tutte basate sul *busy waiting*. Con questa tecnica vengono spcreate preziose risorse e tempo per calcoli di attesa. Bisognerebbe dunque implementare qualcosa a livello hardware affinchè non si sprechi tempo con soluzoni a lato software.

## 3.1 Stop degli interrupt

Per i processori single core è semplice la soluzione: lo *stop degli interrupt*. L'esecuzione lineare di un processore può essere interrotta solo da interrupt esterni. Bloccandoli quando si entra nella critical section basta dunque disabilitare gli interrupt e l'atomicità è garantita.

Questa tecnica tuttavia è limitata solo a hardware mono-processore e può essere usata in modo maligno da programmi che hanno intenzione di bloccare il sistema interrompendo la ricezione dei segnali esterni. La tecnica viene tuttavia sfruttata a basso livello dai sistemi opeartivi che offrono ai programmi *user-space* una tecnica *sicura* per creare critical section che sono gestite dal sistema operativo stesso.

## 3.2 Test and Set

Una comune soluzione hardware al problema offerta da tutti i processori moderni è l'istruzione speciale *test-and-set* che svolge in modo *atomico* le due operazioni di lettura e scrittura in una locazione di memoria. In tal modo un programma concorrente che vuole entrare nella sezione critica tenta di impostare il lock a 1 per se stesso ma fallisce se esso è già stato preso da altri processi. In questo modo si continua a fare *busy waiting* ma l'approccio è molto più pulito e si sprecano meno risorse usando una istruzione singola.

Esistono altre istruzioni analoghe come *fetch-and-set*, *compare-and-swap*, etc.

### 3.2.1 Comunque un miglioramento

L'implementazione hardware è in generale più pulita, in cui basta una sola variabile per la creazione della sezione critica e veloce, tuttavia si usa ancora il busy waiting e bisogna ancora impiegare misure per constrastare la starvation che aggiungono complessità.

La soluzione attualmente più usata è dunque astrarre queste problematiche delegando la creazione della critical section al sistema operativo e alleggerendo il carico sul programmatore userspace.

# 4 Semafori

I semafori sono il meccanismo pricinpale dei sistemi operativi per la sincronizzazione degli accessi ad una risorsa condivisa (l'area dell'incrocio nella metafora) introdotti negli anni 60 da Dijkstra.

L'idea è che due processi possano essere bloccati durante l'esecuzione a vicenda nell'attesa di un segnale dalla controparte. Formalmente un semaforo è un *dato astratto* su cui sono disponibili due operazioni:

- $v$ dall'olandese *verhogen* tramite la quale si invia un segnale per indicare il verificarsi di un evento o il rilascio di una risorsa.
- $p$ dall'olandese *proberen* tramite il quale ci si ferma ad aspettare un segnale e dunque si attende la liberazione di una risorsa o la ricezione di un evento.

Nell'implementazione canonica il semaforo contiene un valore *intero non negativo*. La funzione $v$ incrementa il valore del semaforo, $p$ attende che il semaforo sia strettamente positivo e ne decrementa il valore.

## 4.1 Invarianti

Chiamiamo

- $n_p$ il numero di operazioni $p$ completate
- $n_v$ il numero di operazioni $v$ completate
- $init$ il numero iniziale del semaforo

Allora abbiamo le seguenti invarianti:

1. $n_p \leq n_v + init$, ovvero il valore del semaforo è pari a $init + n_v - n_p$.
2. Ci sono due possibili modalità di utilizzo:
   (a) con $init = 0$ siamo in modalità **eventi**: il numero di eventi consegnati non deve superare il numero di volte che l'evento si è verificato.
   (b) con $init > 0$ siamo in modalità **risorse**: il numero di risorse soddisfatte non deve essere superiore al numero di risorse + il numero di risorse restituite.

## 4.2 Proprietà soddisfatte

Sono banalmente soddisfatte le proprietà di mutua esclusione e assenza di ritardi non necessari tramite la definizione di semaforo. L'assenza di deadlock è facilmente dimstrabile, tuttavia spetta all'implementazione garantire **l'assenza della starvation**.

## 4.3 Rimozione della starvation

Per garantire questa proprietà vengono utilizzati i semafori cosidetti *fair*, ossia equi, giusti. Questi semafori inseriscono i processi in attesa in una coda FIFO (*first in, first out*) e dunque il processo che è in attesa da più tempo sarà anche il primo ad essere liberato dall'attesa. In questo modo tutti i processi saranno prima o poi liberati dal lock.

## 4.4 Implementazione

**Algoritmo 3** Semafori

```
procedure SEMAPHORE(val)
    value ← max{val, 0}
end procedure

procedure P
    if value > 0 then
        value ← value − 1
    else
        pid ← id of the caller
        queue ← pid
        suspend(pid)
    end if
end procedure

procedure V
    if queue is empty then
        value ← value + 1
    else
        pid ← queue
        wakeup(pid)
    end if
end procedure
```

Il corpi di queste procedure devono essere inseriti all'interno di una critical section in qunato fanno accesso alla queue che è una risorsa condivisa tra più processi e processori.

## 4.5 Busy waiting

Poichè si ha comunque bisogno di una ciritcal section per implementare le funzionalità del semaforo si ha comunque il *busy waiting*, tuttavia **se ne limitano limita** l'utilizzo. Esso è ristretto alle sezioni critiche di P e V che sono molto brevi rispetto all'intero blocco della critical section di tutti i programmi (che avrebbero tenuto il resto del sistema in busy waiting altrimenti).

## 4.6 Semafori binari

Un sottoinsieme dei semafori generali è quello dei **semafori binari**. Queste strutture possono solo assumere valori binari, e l'operazione V su una struttura contentente già 1 causa un errore secondo alcuni testi. La variante aggiornata è dunque: $0 \leq init + n_v - n_p \leq 1$

### 4.6.1 Semafori binari usando semafori generali

**Algoritmo 4** Semafori binari

```
procedure SEMAPHORE(init)
    init ← max{min{init, 1}, 0}                    ▷ sanitize init ∈ {0, 1}
    s_0, s_1 ← SEMAPHORE(init), SEMAPHORE(1 − init)
end procedure

procedure P
    call P on s_o
    call V on s_1
end procedure

procedure V
    call P on s_1
    call V on s_0
end procedure
```

## 4.7 Semafori generali tramite semafori binari

Possiamo ora implementare una classe semaforo generico utilizzando i semafori binari per la critical section dei semafori visti nella sottosezione 4.4. Il resto del codice per i metodi P e V è pressochè analogo.

Useremo dunque un semaforo binario come *mutex* per gestire la critical section dell'implementazione, e creeremo poi un semaforo allocato dinamicamente per ogni processo che sfrutta il nuovo semaforo generale.

# 5 Soluzione di problemi

Siamo ora pronti a poter risolvere problemi pratici, creati ad arte, con gli strumenti che ci vengono offerti dai semafori.

## 5.1 Produttore/Cosumatore

Esiste un processo detto `produttore` che vuole trasmettere dei dati ad un altro processo detto `consumatore` che li consuma. I due comunicano tramite una sola variabile in memoria condivisa. Devono dunque coordinare le letture/scritture in modo da non generare *race condition*.

Bisogna inoltre garantire che il produttore non scriva prima della lettura del consumatore e viceversa, ovvero che il consumatore non legga due volte prima che il produttore abbia aggiornato il valore in uscita.

**Algoritmo 5** Produttore/Consumatore

```
buf ← null
empty, full ← Semaphore(1), Semaphore(0)

procedure Producer
    while true do
        val ← Produce()
        call P on empty
        buf ← val
        call V on full
    end while
end procedure

procedure Consumer
    while true do
        call P on full
        val ← buf
        call V on empty
        Consume(val)
    end while
end procedure
```

### 5.1.1 Problema del buffer limitato

Si altera il problema del *producer/consumer* imponendo che il dato scambiato sia un *buffer* di dimensione limitata, nel quale non si devono dunque sovrascrivere i dati non ancora letti ne leggere i dati più volte.

**Algoritmo 6** Buffer limitato

```
q ← Queue(size)
empty, full, mutex ← Semaphore(size), Semaphore(0), Semaphore(1)

procedure Producer
    while true do
        obj ← Produce()
        call P on empty
        call P on mutex
        q ← val
        call V on mutex
        call V on full
    end while
end procedure

procedure Consumer
    while true do
        call P on full
        call P on mutex
        val ← q
        call V on mutex
        call V on empty
        Consume(val)
    end while
end procedure
```

## 5.2 Cena dei filosofi

Il problema viene narrato tramite una storiella: cinque filosofi si trovano ad una tavola rotonda per mangiare. Nella loro vita compiono solo due azioni: mangiano e dormono. Per mangiare hanno bisogno di 2 posate che vengono poi rilasciate quando egli va a dormire. Il testo mostra dunque la soluzione per assegnare una o più risorse a uno o più consumatori.

Definendo le variabili:

- $up_i$ indica il numero di volte che la bacchetta $i$ viene presa dal tavolo.
- $down_i$ indica il numero di volte che la bacchetta $i$ viene appoggiata sul tavolo.

Sappiamo che vale per certo la seguente invariante: $down_i \leq up_i \leq down_i+1$. Sappiamo dunque che il valore del semaforo binario $chopstick_i$ assume valore $1 - up_i - down_i$.

Si possono proporre una serie di soluzioni errate, avendo cura ad esempio di controllare se i proprio vicini stanno mangiando, o provando subito a prenotarsi per mangiare ma si incorre spesso in starvation. L'unica soluzione è quella di rompere la simmetria, facendo in modo che un filosofo sia mancino.

### 5.2.1 Lettori/Scrittori

Un database è condiviso tra un certo numero di processi, che si suddividono in due tipi: `lettori` e `scrittori`. Quando uno scrittore accede al database deve agire in *mutua esclusione*, mentre quando nessuno sta scrivendo il numero di lettori può essere arbitrario.

Questo esempio ci fa notare che mutua esclusione e condivisione possono coesistere per la stessa risorsa, e ci da una casistica in cui non sono singoli processi a comptere ma bensì *calssi di processi*.

Definiamo $nr$ il numero di lettori e $nw$ il numero di scrittori in un dato istante di tempo. Vale la seguente invariante:

$$(nr \geq 0 \wedge nw = 0) \vee (nr = 0 \wedge nw = 1)$$

Si noti che viene ammessa la situazione $nr = nw = 0$ per il passaggio del controllo da lettori a scrittori o viceversa.

Nel definire la soluzione bisogna decidere se dare la priorità agli scrittori o ai lettori in caso di risorsa libera e entrambe le code piene.

**Algoritmo 7** Lettori/Scrittori

```
nr ← 0
rw, mutex ← Semaphore(1), Semaphore(1)

procedure StartRead
    call P on mutex
    if nr = 0 then
        call P on rw
    end if
    nr ← nr + 1
    call V on mutex
end procedure

procedure StartWrite
    call P on rw
end procedure

procedure EndRead
    call P on mutex
    nr ← nr − 1
    if nr = 0 then
        call V on rw
    end if
    call V on mutex
end procedure

procedure EndWrite
    call V on rw
end procedure
```

Questa soluzione da la precedenza ai lettori, i quali possono causare starvation, mentre gli scrittori dopo aver svolgo la loro azione rilasciano il controllo al prossimo attore sulla coda quindi non causeranno mai *race condition*.

## 5.3 Andrews e l'await

Andrews da alcune definizioni preliminari per la sua opera:

- $B$ è una condizione booleana
- $S$ uno statement (istruzione)
- $\langle S \rangle$ eseguire $S$ in modo dinamico
- $\langle \text{Await}(B) \rightarrow S \rangle$ attendi fino a quando la condizione $B$ è verificata e *poi* esegui $S$. Il tutto viene fatto in modo atomico e dunque quando $S$ viene eseguito $B$ è verificata.

Andrews suggerisce poi la seguente procedura per affrontare un problema di programmazione concorrente:

1. **Definire il problema con precisione**: identificare i processi, specificare i problemi di sincronizzazione, introdurre le variabili e le invarianti.
2. **Abbozzare una soluzione**: produrre uno schema di soluzione, e identificare le situazioni in cui un accesso atomico o in mutua esclusione è necessario.
3. **Garantire l'invariante**: verifica che l'invariante sia sempre verificata.
4. **Implementare le azioni atomiche**: esplicitare le azioni atomiche e le AWAIT tramite le primitive di sincronizzazione disponibili.

## 5.4 Lettori/Scrittori secondo Andrews

- **Variabili**:
  1. $nr$ il numero di lettori
  2. $nw$ il numero di scrittori
- **Invariante**:

$$(nr \geq 0 \wedge nw = 0) \vee (nr = 0 \wedge nw = 1)$$

- **Schema di soluzione**:

**procedure** READER
  $\langle$AWAIT$(nw = 0) \rightarrow nr \leftarrow nr + 1\rangle$
  ▷ Read the database
  $\langle nr \leftarrow nr - 1\rangle$
**end procedure**

**procedure** WRITER
  $\langle$AWAIT$(nr = 0 \wedge nw = 0) \rightarrow nw \leftarrow 1\rangle$
  ▷ Write the database
  $\langle nw \leftarrow 0\rangle$
**end procedure**

Useremo poi un *mutex* per la mutua esclusione, un array di semafori $sem_i$ per ogni condizione booleana $B_i$. Su questi semafori verranno messi in attesa i processi che attendono il verificarsi della condizione $B_i$. Avremo infinie un array di interi *waiting* dove nella posizione $i$-esima ci sarà il contatore del numero di processi in attesa della condizione $B_i$.

Tradurremo le istruzioni nel seguente modo:

1. $\langle S\rangle$:
   call $P$ on *mutex*

$S$
SIGNAL()

2. $\langle await(B_i) \rightarrow S_i \rangle$:
   call $P$ on $mutex$
   **if** $\neg B_i$ **then**
      $waiting_i \leftarrow waiting_i + 1$
      call $V$ on $mutex$
      call $P$ on $sem_i$
   **end if**
   $S_i$
   SIGNAL()

3. La funzione ausiliaria SIGNAL():
   **if** $B_0 \wedge waiting_o > 0$ **then**
      $waiting_0 \leftarrow waiting_0 - 1$
      call $V$ on $sem_0$
   **else if** ... **then**
   **else if** $B_{n-1} \wedge waiting_{n-1} > 0$ **then**
      $waiting_{n-1} \leftarrow waiting_{n-1} - 1$
      call $V$ on $sem_{n-1}$
   **else if** $\neg(B_0 \wedge waiting_0 > 0) \wedge \ldots \wedge \neg(B_{n-1} \wedge waiting_{n-1} > 0)$ **then**
      call $V$ on $mutex$
   **end if**

# 6 Monitor

I *monitor* sono un paradigma di programmazione concorrente inventato da Hoare e presente in moltissimi linguaggi sotto varie forme.

Un monitor è un modulo software che contiene *dati locali*, *una sequenza di inizializzazione* e *una o più procedure*. I dati locali sono accessibili **solo alle procedure** del modulo stesso ed un processo entra in un monitor invocando una delle sue procedure. Solo un processo può essere all'interno del monitor mentre gli altri attendono (mutua esclusione). Eccone una descrizione in pseudo-sintassi:

**Algoritmo 8** Monitor

```
monitor NAME
    ...                                              ▷ Variables

    procedure INIT                    ▷ An optional initialization procedure
    end procedure

    procedure entry PROC1(...)                       ▷ A pubblic procedure
    end procedure

    procedure PROC2(...)                             ▷ A private procedure
    end procedure
end monitor
```

## 6.1 Variabili di condizione

Per garantire la mutua esclusione tra le varie procedure si necessita di una qualche tecnica di sincronizzazione per poter sospendere i processi, farli uscire dalla mutua esclusione quando sono in attesa e farli tornare quando le condizioni si verificano.

Si usano a tal file le *variabili di condizione* che possono essere definite con la seguente sintassi:

**condition** c

Ed accettano le operazioni:

- WAIT: attende il verificarsi di una condizione.
  Rilascia la mutua esclusione e il processo viene sospeso in una coda di attesa della condizione $c$.
- SIGNAL: segnala agli altri processi il verificarsi di una condizione.
  Viene riattivato il processo successivo secondo una policia *fair* mentre il chiamante viene messo in attesa. Il chiamante verrà riattivato quando il processo risvegliato uscirà dalla mutua esclusione (*urgent stack*), mentre la chiamata non avrà effetto se la coda per $c$ è vuota.

## 6.2 Lettura/Scrittura

Reimplementazione di una soluzione al problema 5.2.1 con i monitor. I metodi STARTREAD, STARTWRITE, ENDREAD ed ENDWRITE vengono costruiti a partire dal seguente template:

```
procedure entry TEMPLATE()
    if can't read/write then
        call WAIT on can_{read/write}
    end if
    increase or decrease nr or nw
    if nw = 0 then
```

call SIGNAL on *can_read*
**end if**
**if** $nw = 0 \wedge nr = 0$ **then**
call SIGNAL on *can_write*
**end if**
**end procedure**

Ecco il monitor che risolve questo problema:

**Algoritmo 9** Lettura/Scrittura con monitor

```
monitor RW
    nr, nw
    condition can_read, can_write

    procedure INIT
        nr, nw ← 0
        can_read ← nw = 0
        can_write ← nr = 0 ∧ nw = 0
    end procedure
    procedure entry STARTREAD()
        if nw ≠ 0 then
            call WAIT on can_read
        end if
        nr ← nr + 1
        call SIGNAL on can_read
    end procedure
    procedure entry ENDREAD()
        nr ← nr − 1
        if nr = 0 then
            call SIGNAL on can_write
        end if
    end procedure
    procedure entry STARTWRITE()
        if ¬(nr = 0 ∧ nw = 0) then
            call WAIT on can_write
        end if
        nw ← nw + 1
    end procedure
    procedure entry ENDWRITE()
        nw ← nw − 1
        call SIGNAL on can_read
        if nw = 0 ∧ nr = 0 then
            call SIGNAL on can_write
        end if
    end procedure
end monitor
```

# 7 Message Passing

Il *message passing* consente una comunicazione tra processi che non hanno memoria condivisa, tramite la comunicazione e lo scambio di dati attraverso un canale. I processi non utilizzano alcuna struttura dati condivisa (come potevano essere i Semafori o Monitor).

La ricezione e consegna dei messaggi viene gestita dal sistema operativo, che non consentono politiche di broadcast o multicast.

Questa tecnica di sincronizzazione sfrutta due procedure:

1. SEND: il processo mittente (dove avviene la chiamata) invia un messaggio al destinatario *specificato*.
2. RECIEVE: utilizzata dal processo ricevente per leggere i messaggi in arrivo o in coda, con la possibilità di specificare (filtrare) quale sia il mittente a cui siamo interessati.

Esistono poi tre tipologie di *message passing*:

- **sincrono**: Il SEND è sincrono, mentre il RECIEVE è bloccante. La dichiarazione dei metodi è la seguente:
  1. sSEND$(m, q)$ dove $m$ è il *payload* e $q$ il processo destinatario. La chiamata si blocca fino a quando $q$ non riceve il messaggio.
  2. aRECIEVE$(p)$ dove $p$ è il destinatario (opzionale). Se il mittente non ha ancora inviato nessun messaggio il ricevente (chiamante) si blocca in attesa.
- **asincrono**: Il SEND è asincrono, mentre il RECIEVE rimane bloccante. La dichiarazione dei metodi è la seguente:
  1. aSEND$(m, q)$ dove $m$ è il *payload* e $q$ il processo destinatario senza bloccarsi nell'attesa del ricevente. Un qualche soggetto terzo (sistema operativo) si prende cura di tenere una coda dei messaggi ancora non ricevuti.
  2. aRECIEVE$(p)$ dove $p$ è il destinatario (opzionale). Il ricevente si blocca in attesa di un messaggio dal mittente.
- **completamente asincrono**: Il SEND è asincrono, mentre il RECIEVE **non bloccante**. La dichiarazione dei metodi è la seguente:
  1. aSEND$(m, q)$ è analogo al punto precedente.
  2. nbRECIEVE$(p)$ dove $p$ è il destinatario (opzionale). Il ricevente restituisce un messaggio nullo se la coda dei messaggi è vuota (nessun messaggio in entrata).

La versione asincronca di una chiamata può essere implementata tramite il suo analogo asincrono aggiungendo un messaggio extra di ackowledgment. Per fare il viceversa si deve invece utilizzare un ulteriore processo *demone* che ha il compito di ricevere immediatamente i messaggi (sbloccando subito la chiamata sincrona) e smistandoli a dovere tra i dialoganti.