

02b - Sincronizzazione

Introduzione

Corsa critica

Sezione critica

Struttura generale

Spinlock

Algoritmo di Peterson

Algoritmo del fornaio di Lamport

Test and Set

Compare and Swap

Semafori e Mutex

Sezione critica -
n Processi

Mutex in Java : *ReentrantLock*

Mutex e Semafori in Java : *Semaphore*

Implementazione di *CounterSemaphore*

Problemi classici di Sincronizzazione

Barriera

Produttore e Consumatore (Semafori)

Lettore e Scrittore (Semafori)

Cinque Filosofi (Semafori)

Monitor

Equivalenza Monitor - Semaforo

Monitor in Java

Code di accesso in Java

Code di Attesa in Java

Language-based

Library-based

Eccezioni nei Thread

Cinque Filosofi (Monitor)

Modello a Scambio di messaggi

Produttore e Consumatore (Scambio di messaggi)

Designazione di mittente/destinatario

Comunicazione Diretta

Comunicazione indiretta

Buffering

Primitive di sincronizzazione

Send non bloccante

Send bloccante con buffer limitato

Send bloccante

Receive non bloccante

Receive bloccante

Meccanismi di sincronizzazione

Scambio di messaggi Asincrono

Scambio di messaggi Sincrono

Invocazione remota

Introduzione

I processi eseguiti concorrentemente nel SO possono essere indipendenti o cooperanti.

Processi **indipendenti**: l'esecuzione di un processo NON dipende dall'esecuzione degli altri processi; non c'è attesa tra processi. Un processo che non condivide dati con altri processi è indipendente.

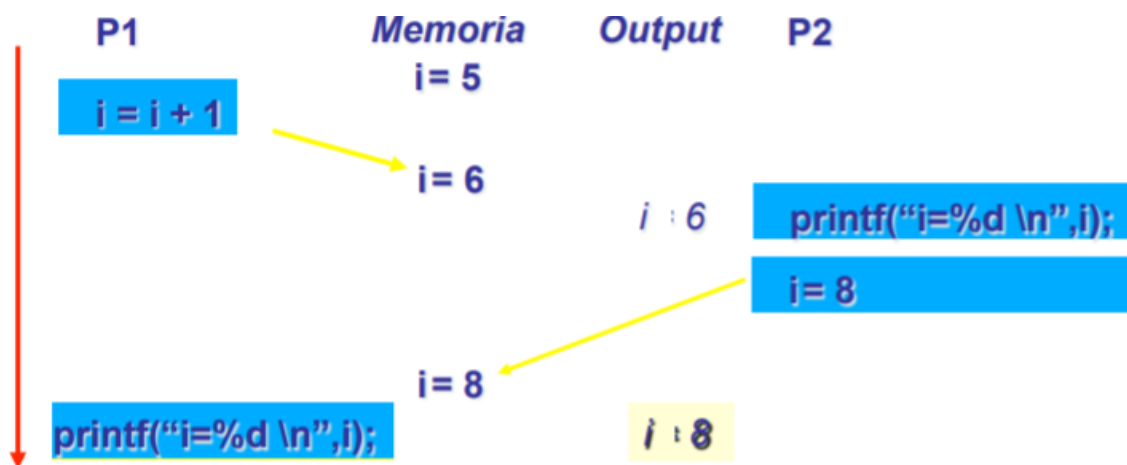
Processi **cooperanti**: l'esecuzione di un processo dipende dall'esecuzione degli altri processi.

Il loro vantaggio sta nella condivisione delle informazioni, la celerità dell'elaborazione, la modularità e la convenienza.

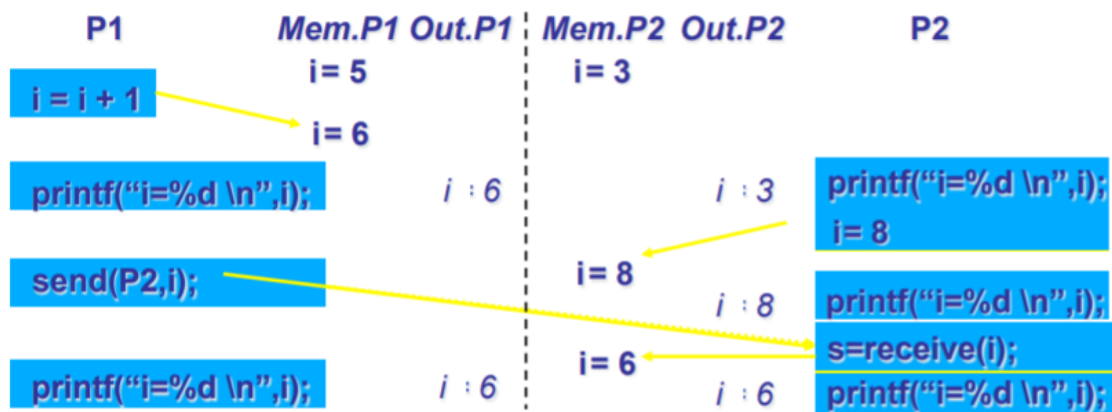
Per lo scambio di dati e informazioni i processi cooperanti necessitano di un meccanismo di comunicazione tra processi (IPC, InterProcess Communication).

I modelli fondamentali della comunicazione tra processi sono due:

- **memoria condivisa**: se un processo modifica una variabile condivisa, anche gli altri processi vedranno la modifica (i processi vanno a lavorare sulla stessa area di memoria). Può presentare problemi di corsa critica.

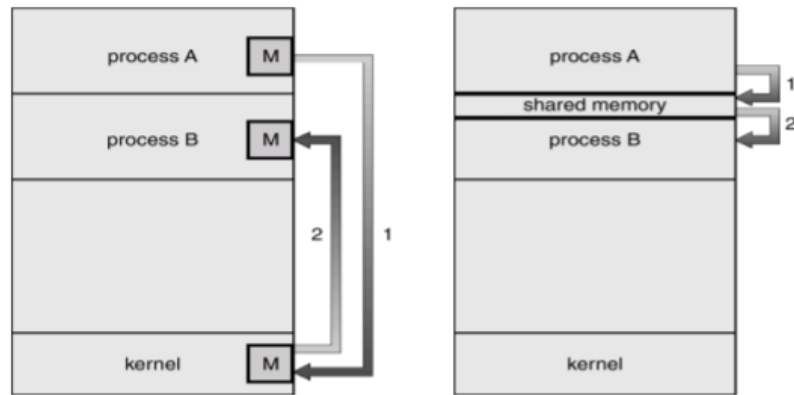


- **scambio di messaggi:** i processi non condividono variabili in memoria, ma possono interagire scambiandosi messaggi.



La memoria è locale al processo P1 e P2 ha la sua memoria.

Lo scambio dei dati avviene nel momento del invio del msg: in quel momento parte la sincronizzazione; in qualsiasi ordine vengono svolte le istruzioni non cambiano il risultato.



Scambio di Messaggi

Memoria Condivisa

Il modello a *memoria condivisa* si presta in modo più naturale ad essere utilizzato quando effettivamente ho più thread che sono in esecuzione sulla stessa macchina, quindi quando effettivamente hanno la possibilità di condividere la memoria.

Il modello a *scambio di messaggi* si presta ad essere più naturale da utilizzare quando ho processi che sono in esecuzione su macchine diverse, quando effettivamente devo utilizzare una rete per mettere in comunicazione questi messaggi.

Usiamo la memoria condivisa (anche se sembra più problematica) perchè alcuni problemi possono essere risolti più facilmente se ragioniamo in termini di memoria condivisa.

Dal punto di vista della capacità computazionale i due modelli sono equivalenti

Corsa critica

Si ha *corsa critica* quando più processi accedono e modificano gli stessi dati in modo concorrente e i risultati dipendono dall'ordine degli accessi.

Per evitare situazioni di questo tipo occorre assicurare che un solo processo alla volta possa modificare i dati condivisi. Ciò richiede una forma di sincronizzazione dei processi.

```

1. public class CCounter implements Runnable {
2.     private int counter = 0;

    // increment counter by one
3.     public void run() {
4.         int tmp = getCounter();
5.         setCounter(tmp + 1);
6.     }
...
}

```

prima lavora t e poi u

	t	u	shared
...	pc _t =4; tmp _t =1	pc _u =4; tmp _u =1	counter=0
	pc _t =5; tmp _t =0	pc _u =4; tmp _u =1	counter=0
	pc _t =6; tmp _t =0	pc _u =4; tmp _u =1	counter=1
	done	pc _u =4; tmp _u =1	counter=1
	done	pc _u =5; tmp _u =1	counter=1
	done	pc _u =6; tmp _u =1	counter=2
	done	done	counter=2

Risultato corretto

```

1. public class CCounter implements Runnable {
2.     private int counter = 0;

    // increment counter by one
3.     public void run() {
4.         int tmp = getCounter();
5.         setCounter(tmp + 1);
6.     }
...
}

```

t ed u lavorano contemporaneamente

	t	u	shared
...	pc _t =4; tmp _t =1	pc _u =4; tmp _u =1	counter=0
	pc _t =5; tmp _t =0	pc _u =4; tmp _u =1	counter=0
	pc _t =5; tmp _t =0	pc _u =5; tmp _u =0	counter=0
	pc _t =5; tmp _t =0	pc _u =6; tmp _u =0	counter=1
	pc _t =6; tmp _t =0	done	counter=1
	done	done	counter=1

Risultato errato

Sezione critica

La soluzione al problema della corsa critica consiste nell'introduzione della *sezione critica*.

Si considerino n processi che competono per accedere ad una data risorsa (esempio: area di memoria condivisa).

Ogni processo ha un segmento di codice, chiamato **sezione critica**, che contiene le istruzioni di accesso e manipolazione dei dati presenti nella memoria condivisa.

Soluzione: assicurare che quando un processo è in esecuzione nella propria sezione critica, non si deve consentire a nessun altro processo di essere in esecuzione nella propria sezione critica.

Una soluzione del problema della sezione critica deve soddisfare i 3 seguenti requisiti:

1)

Mutua Esclusione : se un processo P_i è in esecuzione nella propria sezione critica, nessun altro processo può essere in esecuzione in quella sezione critica.

2)

Progresso : se nessun processo è nella sua sezione critica e ci sono alcuni processi che desiderano entrare nella propria sezione critica, allora uno dei processi in attesa prima o poi entra in sezione critica (*assenza di deadlock*).

3)

Attesa limitata : se esiste un processo in attesa di entrare nella propria sezione critica, allora prima o poi entra in sezione critica (*assenza di starvation*).

Le proprietà viste sopra valgono se assumiamo le seguenti ipotesi:

- Si deve assumere che ogni processo ha un tempo di esecuzione non nullo.
- Nessuna assunzione deve essere fatta circa la velocità relativa dei processi.

Struttura generale



UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Sezione Critica: Struttura generale

```
// continuously  
while (true) {  
    entry section  
    critical section {  
        // access shared data  
    }  
    exit section  
    // reminder section  
}
```

ipotizziamo che ogni processo avrà bisogno di entrare in sezione critica un numero indefinito di volte

non possiamo escludere l'ipotesi che un processo, dopo che è uscito dalla sezione critica, chieda di rientrare in sezione critica, e che questa cosa possa avvenire un numero arbitrariamente grande di volte

I processi possono condividere alcune variabili per sincronizzare le loro azioni.



UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Sezione Critica: Il contatore rivisitato

```
1. public class CCounter implements Runnable {  
2.     private int counter = 0;  
  
    // increment counter by one  
3.     public void run() {  
4.         entry section;  
5.         int tmp = getCounter();  
6.         setCounter(tmp + 1);  
7.         exit section;  
8.         return;  
6.     }  
...  
}
```

Ogni thread modifica counter in modo mutuamente esclusivo

Come realizzare i protocolli di ingresso e uscita dalla sezione critica ? Esistono 2 tipi :

- **Spinlock** : basati sul concetto di mettere in attesa attiva un processo/thread fino a quando la risorsa non si libera (busy waiting).
- **Context switch** : basati sul concetto di sospendere un processo/thread, lasciando quindi la CPU disponibile agli altri, fino a quando la risorsa non si libera (blocking).

Spinlock

Sezione critica - n Processi

Viene usata una struttura dati condivisa sulla quale possono essere eseguite due operazioni: `lock()` e `unlock()` .

Processo P_i :

```
while(true) {  
    lock();  
    critical section  
    unlock();  
    reminder section  
};
```

```
interface SpinLock {  
    void lock();    // acquire  
    void unlock();  // release  
}
```

Dobbiamo trovare gli algoritmi per implementare l'interfaccia.

Sezione critica - 2 Processi/Thread

Algoritmo 1

- Variabili condivise:
 - int turn = 0; \Rightarrow valore iniziale
 - turn = i $\Rightarrow P_i$ può entrare nella sua sezione critica

- Processo P_i

```
1. while (true) {  
2.   while (turn != i) ;  
3.   critical section  
4.   turn = j;  
5.   reminder section  
6. };
```

Soddisfa la mutua esclusione, il progresso, ma non l'attesa limitata

Algoritmo 2

- Variabili condivise:
 - boolean flag[] = {false, false}; \Rightarrow valore iniziale
 - flag[i] = true $\Rightarrow P_i$ pronto ad entrare nella sua sezione critica

- Processo P_i

```
1. while (true) {  
2.   flag[i] = true;  
3.   while (flag[j]) ;  
4.   critical section  
5.   flag[i] = false;  
6.   reminder section  
7. };
```

Soddisfa la mutua esclusione, ma non il progresso

Consideriamo l'Algoritmo 2.

L'istruzione

`flag[i]` indica se P_i ha richiesto l'accesso, quindi nel mio check del `while`

controllo solo se il P_j non ha già richiesto lui la possibilità di entrare nel sezione critica.

Si usa il Busy Waiting (o attesa attiva) fino a quando

P_i non libera la sezione critica. In questo caso non è garantito il progresso, si vengono a creare situazioni di stallo; per notarlo basta cambiare l'ordine delle istruzioni (se eseguono la richiesta in contemporanea, ma "vince" P_j ((true, true)), vado in stallo in quanto entrambi i processi sono in ciclo di busy waiting, ma nessuno può uscirne).

NOTA: Quando c'è stallo, non si ha attesa limitata.

Algoritmo di Peterson

Al fine di avere una soluzione che soddisfa tutti e tre i requisiti (mutua esclusione, progresso e attesa limitata), risolvendo il problema della sezione critica, è necessario fare una fusione tra le due soluzioni proposte finora.

L'algoritmo prende il nome di *Algoritmo di Peterson*.

L'istruzione

`flag[i] = true` ha lo stesso funzionamento logico dell'istruzione `lock()`, mentre l'istruzione `flag[i] = false` ha il funzionamento logico dell'istruzione `unlock()`.

Algoritmo di Peterson

- Combina i due algoritmi precedenti:

- boolean flag[] = {false, false}; int turn = 0;

- Processo P_i

```
1. while (true) {  
2.     flag[i] = true;      lock()  
3.     turn = j;  
4.     while (flag[j] && turn==j) ;  
5.     critical section  
6.     flag[i] = false;    unlock()  
7.     reminder section  
8. };
```

Soddisfa mutua esclusione, progresso e attesa limitata

Se

P_i viene eseguito per primo, `flag[i]` viene impostato a `true` prima di

impostare `turn` a `j`. Dal momento che `flag[j]` è stato inizializzato a `false`, la condizione del `while` non è soddisfatta e P_i può accedere alla sezione critica.

Se nel frattempo viene avviato P_j , questo imposterà `flag[j]` a `true` e `turno` a `j`. `flag[i]` è già stato impostato a `true` da P_i perciò la condizione del `while` di P_j è soddisfatta, così che questo deve aspettare. Solo dopo che P_i ha abbandonato la sezione critica `flag[i]` diventa `false` e P_j può accedere alla sua sezione critica.

Se P_i viene nel frattempo riavviato, imposterà `turno` a `j`, e dovrà aspettare che P_j abbia abbandonato la sua sezione critica.

- La mutua esclusione è garantita dal fatto che il controllo di `flag` e di `turno` viene fatto in modo atomico.
- L'assenza di deadlock viene garantita dal fatto che solamente una delle due condizioni `while` può essere vera nello stesso momento.
- Il progresso è garantito dal fatto che se uno dei due processi tenta di entrare e l'altro non è in sezione critica può tranquillamente entrare.
- L'attesa di un processo è limitata in quanto, se i due processi tentano di entrare in sezione critica, entrano alternamente.

Quando un processo vuole entrare in sezione critica mette il proprio `flag` a `true` per indicare che vuole entrare.

Quindi all'inizio il processo pone `flag = true` e cede il turno all'altro processo. A questo punto, finché il flag dell'altro processo è a `true` ed è il suo turno, il processo aspetta e quando una di queste due diventa non vera, esso può entrare in sezione critica.

Se `flag = false` significa che l'altro processo non ha chiesto di entrare in sezione critica o ne è uscito; se il turno è del primo processo, significa che l'altro processo ha ceduto il turno e quindi può entrare il primo processo.

Questo algoritmo soddisfa tutte e tre le proprietà:

- la *mutua esclusione* è regolata grazie alle variabili `turn` e `flag`: o entra uno o entra l'altro.

- il

progresso è soddisfatto perché grazie la variabile `turn` è condivisa.

- l'attesa limitata pure è soddisfatta perché se entrambi hanno `flag = true` (cioè vogliono entrambi entrare in sezione critica) e supponiamo che il secondo processo entra in sezione critica, quando esso esce pone `flag = false` ; anche se fosse che per qualche ragione (per lo scheduler) ritorna a porre `flag = true` prima che il primo processo rientri in sezione critica, tale processo cede il turno all'altro, quindi non riesce ad entrare per due volte di seguito in sezione critica.

Sezione critica - n Processi/Thread

Nota: esiste anche un algoritmo di Peterson per n processi, aumentando la dimensione dell'array flag. Garantisce tutte e 3 le proprietà, ma non rispetta l'ordine cronologico dei processi dell'attesa limitata.

Algoritmo del fornaio di Lamport

Prima di entrare nella sua regione critica, ad ogni processo viene assegnato un numero.

Il processo con il numero più piccolo entra nella sua sezione critica.

Se un processo non deve entrare in sezione critica gli viene assegnato il numero 0.

Se due processi

P_i e P_j ricevono lo stesso numero, se $i < j$ allora P_i è servito per primo; altrimenti è P_j ad essere servito.

Vengono sempre generati numeri in ordine crescente; possono esserci processi con lo stesso numero.

Algoritmo del fornaio di Lamport (Lamport's Bakery algorithm)

▪ Variabili condivise

```
public static final int numberOfThreads = 5;  
private static volatile boolean[] choosing = new boolean[numberOfThreads];  
private static volatile int[] ticket = new int[numberOfThreads];
```



Accedere ad un campo (attributo) dichiarato come volatile forza la sincronizzazione e quindi impedisce qualsiasi ottimizzazione che comporti il riordino delle istruzioni

`ticket` è un array condiviso tra più thread. Ogni thread, quando ha bisogno di entrare in sezione critica, deve andare a vedere qual è il numero d'ordine più alto fino a quel momento e lo incrementa di 1, poi lui lo va a scrivere nel `ticket` corrispondente alla sua posizione.

Questo vuol dire che se arrivano più processi contemporaneamente, ci sono più processi che si mettono a cercare qual è il valore massimo dell'array; se è una operazione che eseguono contemporaneamente giungono tutti alla stesso valore, quindi entrambi trovano un determinato valore che in quel momento è il valore massimo.

Quando due thread hanno lo stesso numero d'ordine, ad entrare sarà il thread più vecchio, quello che è stato creato prima (es: id più piccolo).

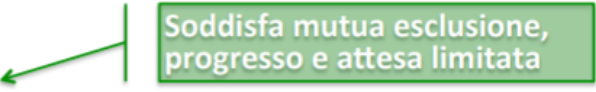
`choosing` mi serve per gestire una mini sezione critica per quanto riguarda l'aggiornamento dell'array `ticket`: quando un thread vuole aggiornare `ticket` mette `choosing = true`, quando ha finito di aggiornare `ticket` mette `choosing = false`.

L'algoritmo del fornaio è uno dei metodi di mutua esclusione che trovano applicazione pratica nella programmazione parallela per serializzare l'esecuzione delle sezioni critiche da parte di più processi o thread concorrenti. L'algoritmo deve il nome al suo inventore Lamport, che propose l'analogia con il modello reale di una frequentata panetteria, dove i clienti strappano un numero prima di mettersi in fila ad aspettare il proprio turno. I clienti del fornaio rappresentano i task in attesa del proprio turno per eseguire la sezione critica.

Algoritmo del fornaio di Lamport (Lamport's Bakery algorithm)

▪ Processo P_i

```
while (true) {  
    choosing[id] = true; lock()  
    ticket[id] = findMax() + 1;  
    choosing[id] = false;  
    for (int j = 0; j < numberOfThreads; j++) { ATTESA ATTIVA  
        while (choosing[j]) ;  
        while (ticket[j] != 0 &&  
            (ticket[id] > ticket[j] || (ticket[id] == ticket[j] && id > j))) ;  
    }  
    critical section  
    ticket[id] = 0; unlock()  
    remainder section  
};
```



Per prima cosa quindi,

P_i mette `choosing[id] = true`, quindi non dobbiamo chiedere il numero di questo processo se ancora non lo ha preso. Poi prende il numero come il massimo tra tutti i numeri presenti nel vettore `ticket` e lo incrementa di 1 (ovviamente per far sì che il suo numero diventi il più alto e quindi che il suo turno sia l'ultimo) e pone `choosing[id] = false` in quanto ha preso il numero.

Poi esegue un ciclo `for`, in cui per ogni processo si mette in attesa finché P_j sta scegliendo il numero. Poi controlla se ha la precedenza rispetto a P_j e può entrare in sezione critica. Questo controllo viene fatto tramite un altro `while`: P_i resta in attesa qualora il numero di P_j è diverso da zero (perché in caso fosse uguale a zero significa che P_i ha la precedenza in quanto questo processo non ha richiesto di entrare in sezione critica) e il numero dell'altro processo è inferiore al numero di P_i oppure è uguale ma il numero del suo process ID è inferiore (è più anziano e lo faccio passare).

Quando per tutti i processi una di queste condizioni diventa false P_i può entrare in sezione critica.

Questo algoritmo soddisfa tutte e tre le proprietà.

- La *mutua esclusione* perché, se per assurdo se non fosse rispettata, avremmo almeno due processi che sono entrambi in sezione critica: ciò significa che devono esistere due processi che hanno stesso numero e stesso process ID,

impossibile.

- II

progresso è rispettato in quanto, se per assurdo non fosse vero, vorrebbe dire che tutti i processi sono in attesa e nessuno entra, però questo significa che ogni processo ha trovato un processo che ha un numero più piccolo del suo oppure uno uguale con process ID più piccolo. Ma deve esistere per forza almeno un processo con il numero più piccolo o comunque con un process ID più piccolo che entrerà in sezione critica.

- L'

attesa limitata è rispettata in quanto ogni processo che esce in sezione critica deve riassegnare il numero che è pari al massimo +1 quindi sicuramente sta dopo di altri processi in coda.

Le operazioni però devono essere fatte in modo atomico, come abbiamo detto sopra, perché altrimenti non riusciamo a soddisfare le proprietà di questi algoritmi.

La serializzazione è assicurata dalle due iterazioni `while` consecutive. Questi cicli vengono ripetuti da ogni thread per tutti i thread, incluso quello in esecuzione e i thread non attivi. Solo quando non ci sono più altri thread con priorità più alta è possibile l'accesso alla sezione critica.

L'algoritmo del fornaio garantisce che un solo thread alla volta possa accedere alla sezione critica, indipendentemente dall'ordine delle commutazioni di contesto e dagli altri dettagli dello scheduling. Sussiste tuttavia il principale inconveniente che è proprio di tutti gli algoritmi di coordinazione decentrata, cioè non coordinata dallo scheduler: i task in attesa continuano ad utilizzare il processore in un ciclo di attesa attiva detto busy waiting, rallentando così gli altri thread nel sistema.

L'algoritmo del fornaio usa locazioni di memoria condivisa (i due array di n elementi). Si può provare che è la quantità minima di memoria richiesta per avere mutua esclusione se si usano solo operazioni di lettura e scrittura.

Per questa ragione la sincronizzazione usando solo lettura e scrittura non è praticabile. Servono primitive più potenti:

- operazioni atomiche
- operazioni di sospensione e riattivazione dei processi.

Per gestire n thread abbiamo bisogno come minimo di n elementi, con meno memoria non si riesce a fare.

Questo crea un problema: all'aumentare del numero di thread che devono

lavorare sullo stesso spazio di memoria, lo spazio di memoria cresce linearmente al crescere di n.

Test and Set

Controlla e modifica il contenuto di una variabile Booleana atomicamente.

```
boolean TestAndSet(boolean *target) {  
    boolean rv = *target;  
    *target = true;  
    return rv;  
}
```

Controlla il valore di una variabile, restituisce il valore che ha trovato e contemporaneamente assegna a quella variabile il valore `true`.

Mutua esclusione con Test-and-Set Assembly :

Mutua Esclusione con Test-and-Set Assembly

- Variabile condivisa:
boolean lock = false;

Processo Pi

```
while (true) {
```

```
    lock();
```

```
    critical section
```

```
    unlock();
```

```
    remainder section
```

```
}
```

```
enter_region:  
    TSL REGISTER,LOCK  
    CMP REGISTER,#0  
    JNE enter_region  
    RET | return to caller;
```

```
leave_region:  
    MOVE LOCK,#0  
    RET | return to caller
```

Come argomento della TestAndSet passiamo un registro REGISTER e una variabile LOCK, e la funzione prende il valore di LOCK e lo mette in REGISTER e a LOCK mette il valore true

Mutua esclusione con Test-and-Set Java :



UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Mutua Esclusione con Test-and-Set Java

- Il package `atomic` fornisce supporto ad operazioni atomiche su singole variabili

```
package java.util.concurrent.atomic;
```

```
public class AtomicBoolean {  
    AtomicBoolean(boolean initialValue);  
    boolean get();  
    void set(boolean newValue);  
    boolean getAndSet(boolean newValue);  
    ...  
}
```

Metodo costruttore, crea una nuova istanza il cui valore è `initialValue`

Metodo getter, legge il valore corrente dell'istanza

Metodo setter, assegna all'istanza il valore `newValue`

Restituisce il valore corrente ed assegna all'istanza il valore `newValue`

TestAndSet(&x) è equivalente a `x.getAndSet(true)`

30



UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Mutua Esclusione con Test-and-Set Java

Algoritmo mutua esclusione con TSL

```
public class TASLock  
    implements Lock {  
  
    AtomicBoolean held = new AtomicBoolean(false);
```

```
    public void lock() {  
        while (held.getAndSet(true))  
            {}  
    }
```

Soddisfa la mutua esclusione, il progresso, ma non l'attesa limitata

```
    public void unlock() {  
        held.set(false);  
    }
```

```
}
```

Compare and Swap

Scambia in modo atomico il contenuto di due variabili.

```
boolean CompareAndSwap(boolean *value, expected, new_value) {
    boolean temp = *value;
    if (*value == expected) *value = new_value;
    return temp;
}
```

Prende due variabili, mi restituisce il valore della prima, però scambia i valori delle variabili.

Mutua esclusione con CompareAndSwap Java :

Mutua Esclusione con CompareAnd Swap Java

- Il package `atomic` fornisce supporto ad operazioni atomiche su singole variabili

```
package java.util.concurrent.atomic;
```

```
public class AtomicBoolean {
    AtomicBoolean(boolean initialValue);
    boolean get();
    void set(boolean newValue);
    boolean getAndSet(boolean newValue);
    boolean compareAndExchange(boolean expectedValue,
                               boolean newValue)
}
CompareAndSwap(&x,a,b) è equivalente a x.getAndSet(a,b)
```

Restituisce il valore corrente ed assegna all'istanza il valore `newValue` se il valore corrente è uguale a `expectedValue`

33

Mutua Esclusione con CompareAnd Swap Java

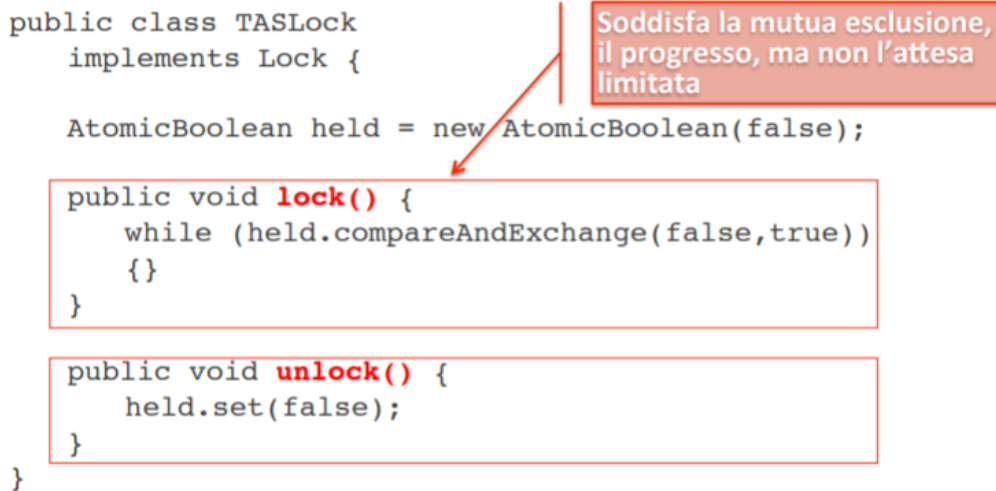
Algoritmo mutua esclusione con CompareAndSwap

```
public class TASLock
    implements Lock {

    AtomicBoolean held = new AtomicBoolean(false);

    public void lock() {
        while (held.compareAndExchange(false,true))
            {}
    }

    public void unlock() {
        held.set(false);
    }
}
```



Gli algoritmi precedenti sfruttano TSL e SWAP per la mutua esclusione, ma non garantiscono l'attesa limitata.

Dato che gli algoritmi visti finora non garantiscono l'attesa limitata, è necessario trovare un algoritmo che soddisfi tutte e 3 i requisiti TSL.

Semafori e Mutex

Tutti gli algoritmi precedenti ai semafori (come Test-and-Set oppure Swap) erano degli algoritmi basati sullo Spinlock, ovvero erano un tipo di approccio in cui c'era una variabile che mi determinava se potevo entrare in sezione critica oppure no. Quell'approccio era molto semplice e non soddisfaceva tutte e tre le proprietà dei processi, in quanto l'attesa limitata purtroppo era sempre non soddisfatta

Il semaforo è uno strumento di sincronizzazione che non richiede il busy waiting.

Esistono 2 tipi di semafori:

- **semaforo contatore** : è un tipo di dato che prende valore dall'insieme dei numeri *interi* e sul quale possiamo agire con due operazioni :
 - **wait** : finché il semaforo ha un valore negativo o nullo, il processo che esegue quella wait è sospeso; se ha un valore positivo, va avanti e il valore

del semaforo viene decrementato di 1.

-

`signal` : incrementa di 1 il valore del semaforo.

- **semaforo binario (mutex)** : prende valore dai *booleani* (0,1) e presenta due operazioni:

- `wait` : finchè il valore è 0 è sospeso; se il valore è 1 va avanti e mette il valore a 0.

-

`signal` : mette il semaforo a 1.

Sezione critica - *n* Processi

Usare i semafori significa abbandonare l'attesa attiva.

Algoritmo con Mutex :

Variabili condivise:

```
mutex m = true;
```

Processo P_i :

```
while (true) {  
    wait(mutex);  
    critical section  
    signal(mutex);  
    remainder section  
};
```

La `wait` ha due grosse differenze rispetto a `lock()` :

1) prevede la sospensione di un processo quando le condizioni non gli permettono di andare avanti (semaforo rosso);

2) sia la

`wait` che la `signal` vengono eseguite in maniera atomica.

Viene rispettato l'ordine di arrivo, quindi vengono garantite :

- mutua esclusione
- progresso

- attesa limitata

```
interface CounterSemaphore {  
    void cWait();  
    void cSignal();  
}
```

```
interface Mutex {  
    void mWait();  
    void mSignal();  
}
```

Dobbiamo trovare gli algoritmi per implementare queste interfacce.

Mutex in Java : *ReentrantLock*

Java ha una interfaccia `Lock` che di fatto corrisponde alla nostra interfaccia `Mutex` :

```
public interface Lock {  
    void lock(); // wait(  
    void unlock(); // signa  
}
```

La realizzazione di tale interfaccia più utilizzata è la classe `ReentrantLock` :

```
public class ReentrantLock implements Lock {  
    ReentrantLock(boolean fair)  
    void lock();  
    void unlock();  
}
```

Soluzione:

```
private final ReentrantLock l = new ReentrantLock(v);
```

```
public void m() {  
    l.lock();  
    try {  
        // critical session  
    } finally {  
        l.unlock();  
    }  
}
```

la clausola `finally` assicura che il lock venga rilasciato anche nel caso avvenga una eccezione nel blocco `try`

✓ Mutua esclusione: è garantita
✓ Progresso: garantito se non ci sono altri lock.

✗ Attesa limitata:
▪ se `v=false` non è garantita,
▪ se `v=true` favorisce l'accesso al thread in attesa da più tempo. Tuttavia la fairness del lock non garantisce la fairness dei thread: un thread può ottenere il lock più volte in successione mentre altri thread attivi non stanno procedendo e al momento non tengono il lock.

42

La `ReentrantLock` può essere equa o non (`true` o `false`) in base al metodo `fair` ; se non è equo viene scelto un thread casualmente, e viene risvegliato. Quindi se metto `fair = false` non ho garantito l'attesa limitata, ma tutte le altre sì. Se invece metto `fair = true` , non ho la certezza che tutti i thread vengano trattati in modo equo, ciò dipende dalla gestione della JVM (Java Virtual Machine)

Mutex e Semafori in Java : *Semaphore*

Java ha una classe `Semaphore` : quando è inizializzato a 1 abbiamo un semaforo binario (mutex), quando è inizializzato ad un valore maggiore di 1 abbiamo un semaforo contatore.

```
public class Semaphore {  
    Semaphore(int permits, boolean fair);  
    void acquire(); // wait()  
    void release(); // signal()  
}
```

`acquire()` e `release()` sono le equivalenti di `wait()` e `signal()` .

- `permits` è la capacità del semaforo
- se `fair = false` i thread vengono schedulati in modo non determinato a priori
- se `fair = true` i thread vengono schedulati secondo un ordine FIFO

Soluzione:

```
private final Semaphore s = new Semaphore (1, true);  
  
public void m() {  
    //garantisce mutua esclusione, progresso (se non ci sono altri semafori), attesa limitata  
    s.acquire();  
    try {  
        //critical session  
    }  
    finally {  
        s.release()  
    }  
}
```

```
}  
}
```

ReentrantLock VS Semaphore :

I semafori binari sono molto simili ai lock con una differenza:

- in un **lock** solo il thread che acquisisce il lock (decrementa il valore) lo può rilasciare [può fare unlock] (incrementa il valore);
- in un **semaforo binario** un thread può decrementare il valore ed un altro lo può incrementare.

Implementazione di *CounterSemaphore*

Proviamo ad implementare una nostra realizzazione di *CounterSemaphore* basata sul context switch.

Ci servono due primitive: una per bloccare un processo/thread e una per risvegliarlo.

- `wait()` : sospende il thread che lo esegue (interruzione software)

-

`notify()` : sceglie in modo non deterministico un thread in stato di *waiting* e lo porta in stato di *ready*

-

`notifyAll()` : tutti i thread in stato di *waiting* vengono portati in stato di *ready*

Devono avere effetto solo sui thread in coda sullo stesso oggetto condiviso, per questo si deve usare blocchi o metodi *synchronized*.

Implementazione Debole (weak) :

Soluzione : usiamo `wait()` e `notify()`

```
class SemaphoreWeak implements Semaphore {  
    private int count;  
    public synchronized void cSignal() {  
        count = count + 1;  
        notify();    // sveglia un thread in stato di waiting  
    }  
    public synchronized void cwait() throws InterruptedException  
        while (count == 0)
```

```

        wait();    // sospende il thread in stato di runn
        count = count - 1;    // ora count > 0
    }
}

```

La `wait()` deve essere richiamata all'interno di un loop per evitare risvegli spuri, perché Java non garantisce che non ce ne siano.

Non riusciamo a garantire l'attesa limitata, cioè che venga rispettato l'ordine di arrivo.

Implementazione Forte (strong) :

Soluzione : usiamo `wait()` e `notifyAll()`

```

class SemaphoreStrong implements Semaphore {
    public synchronized void cSignal() {
        count = count + 1;
        if (!blocked.isEmpty()) notifyAll();
    }
    public synchronized void cWait() throws InterruptedException {
        Thread me = Thread.currentThread();    // serve ad aver
                                                // thread in ese

        blocked.add(me);
        while (!(count > 0 && blocked.element() == me)) // 2°
                                                        // se
                                                        // è
            wait();
        blocked.remove();    // se posso entrare in sezione cri
                            // rimuovo dalla coda
        count = count - 1;
    }
    private final Queue<Thread> blocked = new LinkedList<>();
}

```

Viene gestita una coda di attese (blocked).

L'ultima istruzione mi permette di creare una coda di processi/thread, non è vuota in quanto ci sono almeno "me". L'Implementazione Forte ci permette di garantire tutti e 3 i requisiti. Le istruzioni vengono eseguite in modo atomico (synchronized).

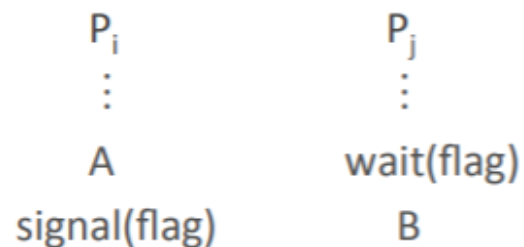
Problemi classici di Sincronizzazione

Un semplice esempio di sincronizzazione:

Problema: eseguire l'istruzione B in P_j solo dopo l'esecuzione di A in P_i .

Soluzione:

- usare un semaforo flag inizializzato a 0
- codice:



Il semaforo P_j esegue tutte le sue istruzioni, poi arrivato sul punto di eseguire B esegue prima una `wait()` sul semaforo : se è a 0 ancora non può eseguire B, se è a 1 allora P_i ha già eseguito A.

Per poter dire a

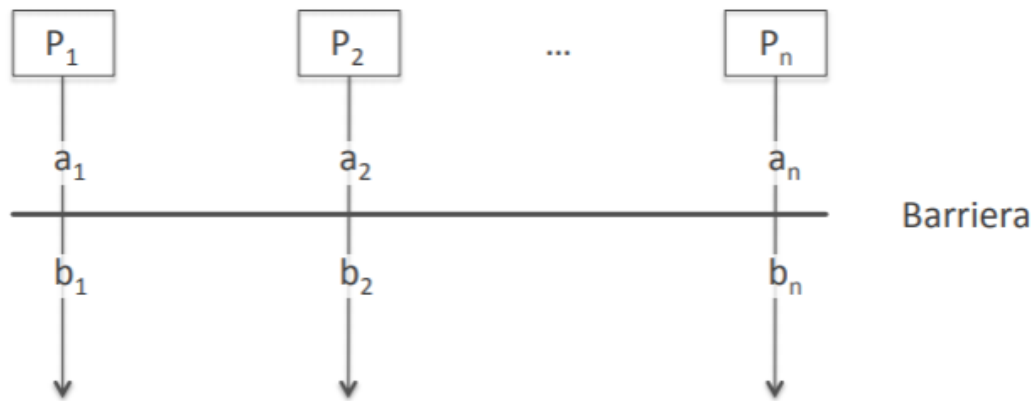
P_j che può eseguire l'istruzione B, P_i , dopo aver eseguito l'istruzione A, mette a 1 il semaforo.

Barriera

Una barriera è una forma di sincronizzazione dove esiste un punto (la *barriera*) nell'esecuzione di ogni processo di un certo gruppo che deve essere raggiunto da tutti i processi del gruppo prima che ognuno di loro possa proseguire nell'esecuzione.

Abbiamo più processi/thread in esecuzione in parallelo, arrivati alla barriera si fermano e possono proseguire solo dopo che tutti quanti sono arrivati alla barriera.

Ogni thread quando arriva sulla barriera mette ad 1 il proprio semaforo.



Soluzione per 2 thread :

Implementiamo la barriera con un array di semafori (nel caso di 2 thread l'array contiene solo 2 semafori).

```
Semaphore[] done = {new Semaphore(0), new Semaphore(0)};
```

Thread t0

```
//codice prima della barriera
done[t0].release();
done[t1].acquire();
//codice dopo la barriera
```

Thread t1

```
//codice prima della barriera
done[t1].release();
done[t0].acquire();
//codice dopo la barriera
```

Dopo la barriera non si può usare un Lock perché è acquisito da un thread e rilasciato da un altro.

`release()` segnala che il processo è arrivato alla barriera.

`acquire()` vede se l'altro processo ha segnalato che è arrivato alla barriera.

Produttore e Consumatore (Semafori)

Il problema descrive due tipi di processi, uno *produttore* e uno *consumatore*, che condividono un buffer comune, di dimensione fissata. Compito del produttore è generare dati e depositarli nel buffer di continuo.

Contemporaneamente, il consumatore utilizzerà i dati prodotti, rimuovendoli di

volta in volta dal buffer.

Il problema è assicurare che il produttore non elabori nuovi dati se il buffer è pieno, e che il consumatore non cerchi dati se il buffer è vuoto.

Buffer limitato vuol dire che lo spazio a disposizione è limitato.

I processi produttori riescono a riempire le caselle del buffer finché c'è spazio; quando sono state riempite tutte le caselle, i produttori devono aspettare che se ne liberi qualcuna; i consumatori quando le caselle sono tutte vuote devono aspettare.

Con un buffer illimitato i produttori caricano in continuazione sul buffer senza doversi trovare in attesa; con un buffer nullo un produttore produce qualcosa e aspetta che il consumatore lo prenda, nel frattempo gli altri produttori devono aspettare.

```
public class BoundedBuffer<T> {  
    Lock lock = new lock(); //semaforo binario  
    //accesso al buffer in mutua esclusione  
  
    Semaphore nItems = new Semaphore(0); //numero item  
nel buffer  
    Semaphore nFree = new Semaphore(N); //numero di pos  
izioni libere nel buffer  
    Collection storage = ...; //buffer  
}
```

- il buffer in quanto tale va gestito all'interno di una sezione critica;
- dobbiamo sincronizzare l'attività dei produttori con quella dei consumatori.

Il consumatore deve sapere se nel buffer c'è qualcosa oppure nulla, devo contare quanti elementi da consumare ci sono ancora nel buffer (semaforo contatore); inizialmente il buffer è vuoto (non ci sono elementi da consumare) e quindi inizializzo il semaforo a 0.

nItems serve ai consumatori per capire se possono prelevare qualcosa dal buffer.

nFree serve ai produttori per capire se possono caricare qualcosa sul buffer.

Produttore

```

while (true) {
// produce a new item

nFree.acquire(); //controlla lo spazio libero del buffer
lock.lock(); //"entro" nel buffer
// insirisco item
lock.unlock(); //esco
nItems.release(); //segnali il numero di items all'interno de
}

```

Consumatore

```

while (true) {
nItems.acquire(); //controllo numero elementi buffer
lock.lock();
// remove item
lock.unlock();
nFree.release(); //segnalo numero spazio libero buffer

// consume the new item
}

```

Letttore e Scrittore (Semafori)

Supponiamo di avere un'area dati condivisa e di avere un certo numero di processi che devono andare a scrivere su questa area di memoria e un certo numero di processi che invece devono leggere quello che è scritto sull'area. Rischiamo situazioni di corsa critica, quindi la modifica della nostra area deve essere fatta in modo mutualmente esclusivo :
quando un processo scrittore sta scrivendo, nessun altro deve poter accedere all'area (nessuno scrittore può scrivere e nessun lettore può leggere); quando qualcuno sta leggendo anche altri possono leggere, ma impedisce agli scrittori di operare.

```

public class SyncBoard<T> {
    int nReaders = 0; //lettori che stanno leggendo
    Lock lock = new Lock(); //accesso in mutua esclusio

```

```

ne a nReaders
    Semaphore empty = new Semaphore(1) //non si usa lock
    //acquisito da un thread e rilasciato da un altro
    /*blocca gli scrittori quando ci sono lettori
    blocca tutti quando c'è uno scrittore*/

    T message;
    --- //area condivisa
}

```

Scrittore

```

public void write(T msg) {
    empty.acquire();
    //segnalo che voglio scrivere
    //blocco gli altri processi, sia scrittori
    //che lettori

    message = msg;
    empty.release();
    //avverto che ho finito di scrivere
    //sblocco i processi
}

```

Il processo scrittore, quando deve iniziare a scrivere, fa una `acquire()` su `empty`, e una `release()` sempre su `empty` quando finisce.

Questo mi garantisce la mutua esclusione per i processi scrittore, cioè solo uno di questi processi scrittori può lavorare.

Lettore

```

public T read() {
    lock.lock(); //accesso a nReaders
    // in modo mutuamente esclusivo
    nReaders += 1;
    if (nReaders == 1) empty.acquire();
    //se sono l'unico ad aver richiesto la lettura,

```

```

// acquisisco

    lock.unlock();
    T msg = message;
    lock.lock();//accesso a nReaders in modo
//mutuamente esclusivo
    nReaders -= 1; //decremento perchè il lettore
// ha finito
    if (nReaders == 0) empty.release(); //l'ultimo
//lettore segnale la fine della lettura
    lock.unlock();
    return msg;
}

```

Per prima cosa viene fatta una `lock()` sul mutex, in modo tale che accediamo a `nReaders` in maniera mutuamente esclusiva.

Se è l'unico processo deve fare una

`acquire()` su `empty` perché è il primo processo (bloccando l'accesso agli scrittori).

Una volta finita la prima parte, si effettua

`unlock()` ; eventuali processi lettori ora possono rientrare.

Quando il processo esce rifà la

`lock()` per accedere a `nReaders` e decrementa quest'ultimo.

Poi vede che se è uguale a 0 so che tale processo è l'ultimo e quindi il semaforo contatore deve fare una

`release()` , cioè deve permettere la scrittura sull'area di memoria condivisa, e poi infine esegue una `unlock()` .

Questa soluzione garantisce tutti e 3 i requisiti.

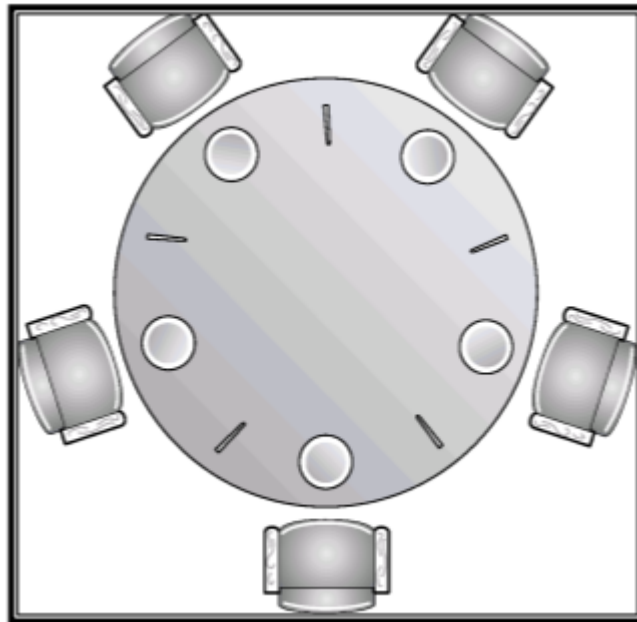
Cinque Filosofi (Semafori)

Cinque Filosofi (Monitor).

Ogni filosofo, quando intende mangiare, deve prendere le due bacchette che stano alla sua destra e sinistra; quando ha finito rilascia le due bacchette.

Le bacchette sono 5 e quando uno mangia le due bacchette devono essere libere; se lui sta mangiando, il filosofi alla sua destra e sinistra non possono

mangiare, perchè almeno una bacchetta sarà impegnata dal nostro filosofo. Quindi quando un filosofo vuole mangiare deve controllare se le bacchette sono libere : se entrambe sono libere può mangiare, se almeno una non è libera deve aspettare fino a quando entrambe le bacchette diventano disponibili. Ciascun filosofo può prendere una sola bacchetta per volta (non può prendere entrambe nello stesso momento) (non è atomica l'istruzione per prendere entrambe le bacchette).



Abbiamo un certo numero di processi che ha bisogno di utilizzare un certo numero di risorse (n° processi \neq n° risorse, non devono essere uguali) e ogni processo ha bisogno solo di un sottoinsieme di queste risorse, in cui ci sono risorse che servono anche ad altri processi.

```
interface Table {  
    // philosopher k picks up forks  
    void getChopstick(int k);  
    // philosopher k releases forks  
    void putChopstick(int k);  
}
```

Filosofo i-esimo

```
while (true) {
```

```

    think();
    table.getChopstick(k); // wait for chopsticks
    eat();
    table.putChopstick(k); // release chopsticks
}

```

Soluzione con Semafori :

Si può rappresentare ciascuna bacchetta con un semaforo.

```

public class LockTable implements Table {
    Lock[] chopstick = new Lock[N];
}

```

Ogni filosofo tenta di afferrare la k -esima bacchetta alla sua sinistra e la $(k + 1)$ -esima bacchetta alla sua destra con operazioni di `lock()` e le rilascia eseguendo le `unlock()`.

```

public void getChopstick(int k)
{
    // pick up left fork
    chopstick[k].lock();
    // pick up right fork
    chopstick[(k+1) % N].lock();
}

```

```

public void putChopstick(int k)
{
    // put down left fork
    chopstick[k].unlock();
    // put down right fork
    chopstick[(k + 1) % N].unlock();
}

```

Questa soluzione soddisfa la mutua esclusione, ma non il progresso e né l'attesa limitata.

Problema : si può cadere in una condizione nella quale tutti e 5 i filosofi hanno la sola bacchetta alla loro sinistra e, non essendocene delle altre, per poter mangiare aspetteranno il rilascio di un'altra bacchetta indefinitamente (all'infinito) → **deadlock** (progresso non soddisfatto).

Soluzione : ogni filosofo, dopo aver preso la bacchetta alla sua sinistra, verifica se è disponibile anche quella a destra; in caso contrario rilascia quella in suo possesso.

Problema : un filosofo non riesce mai a prendere entrambe le bacchette → **starvation**. (attesa limitata non soddisfatta).

Monitor

I semafori forniscono un meccanismo potente e conciso per sincronizzazione e mutua esclusione. Sfortunatamente hanno diversi limiti :

- sono intrinsecamente "globali" e non strutturati (è difficile capire il loro comportamento guardando un singolo pezzo di codice);
- sono soggetti a stallo o ad altri comportamenti scorretti (è facile dimenticare di aggiungere una singola chiamata cruciale verso l'alto o verso il basso).

In sintesi i semafori sono una primitiva di sincronizzazione di basso livello.

Se dobbiamo accedere ad una risorsa o ad un'area di memoria condivisa o a n risorse in maniera sovrapposta e complessa, facciamo gestire questo accesso ad un oggetto che riceve le varie richieste di accesso e poi sarà lui a valutare chi può accedere e chi no. Questi oggetti particolari vengono chiamati **monitor**.

I monitor forniscono un meccanismo di sincronizzazione strutturato costruito sopra i costrutti della programmazione orientata agli oggetti (classe, oggetto e incapsulamento).

Un monitor è un TDA (tipo di dato astratto) le cui operazioni possono essere richiamate in *mutua esclusione*, quindi già esso garantisce tale proprietà.

Un monitor è un oggetto che istanzia una classe di monitor.

In che modo i monitor racchiudono meccanismi di sincronizzazione ?

- gli attributi sono variabili condivise che tutti i thread in esecuzione sul monitor possono vedere e modificare
- i metodi sono sezioni critiche, con la garanzia che in qualsiasi momento al massimo un solo thread può essere attivo su un monitor.

Per problemi di sincronizzazione più complessi della mutua esclusione i monitor mettono a disposizione anche le variabili di tipo **condition**.

Le variabili di tipo

condition permettono di accodare i processi in attesa. Una variabile di tipo *condition* è una istanza di una classe con interfaccia.

Una variabile di tipo *condition* è una variabile sulla quale si possono eseguire solo 2 operazioni :

```
interface Condition {  
  
    void wait();  
        // block until signal -> prende un processo e lo sosp  
    void signal();  
        // signal to unblock  
        //prende il primo processo in coda e lo risveglia (lo  
}
```

Un processo che invoca una `wait()` è sospeso (messo in coda) fino a quando un altro processo non invoca `signal()` .

`signal()` riavvia il primo processo in coda. Se nessun processo è sospeso, allora la `signal()` non ha effetto.

Se il processo P richiama `signal()` e il processo Q è sospeso in `wait()` , cosa dovrebbe succedere dopo?

Sia Q che P non possono essere eseguiti in parallelo. Se Q viene ripreso, allora P deve attendere.

Le opzioni sono due :

1)

Segnala e attendi : P attende fino a quando Q lascia il monitor o attende un'altra condizione.

2)

Segnala e continua : Q attende che P lasci il monitor o attenda un'altra condizione.

In ogni caso, una `signal()` deve essere necessariamente l'ultima istruzione di un metodo.

1) Supponiamo che P esegua una `signal()`, Q è il primo in coda sulla variabile condition; il thread Q viene risvegliato, riprende la sua esecuzione fino alla fine del metodo che aveva invocato, e il thread P resta in attesa. Quando Q finisce ed esce dal metodo che aveva invocato e poi ripreso, riprende P.

Segnala e attendi ha implicitamente come soggetto P (il thread che esegue la `signal()`) che segnala e attende che Q si risvegli e riprenda il suo lavoro.

2) P risveglia Q (però Q non è immediatamente in esecuzione) che viene tolto dalla coda di condition e viene messo in attesa; quando P finisce il suo metodo Q può effettivamente ritornare a lavorare.

Equivalenza Monitor - Semaforo

Teorema : Monitor e Semafori sono equivalenti dal punto di vista espressivo (si possono risolvere gli stessi problemi).

Dim :

- un semaforo si può implementare usando un monitor
- un monitor si può implementare usando due semafori più un semaforo per ogni variabile condition

Monitor in Java

Java non implementa in modo completo i monitor, ma offre supporto per implementarle seguendo opportuni modelli di programmazione.

In Java ci sono due serie di primitive simili ai monitor:

- primitive basate sul linguaggio
- primitive basate sulle librerie

Le abbiamo già incontrate per implementare primitive di sincronizzazione più semplici.

Code di accesso in Java

Ogni oggetto Java ha un lock implicito a cui si può accedere usando il modificatore **synchronized**.

Method Locking (synchronized)

```
synchronized Type m() {  
    // the critical section  
    // is the whole method  
    // body  
}
```

Ogni chiamata ad `m` implicitamente :

1) fa una

`wait` sul lock

2) esegue

`m`

3) fa una `signal` sul lock

Block Locking (synchronized)

```
synchronized(this) {  
    // the critical section  
    // is the block's content  
}
```

Ogni esecuzione del blocco

implicitamente :

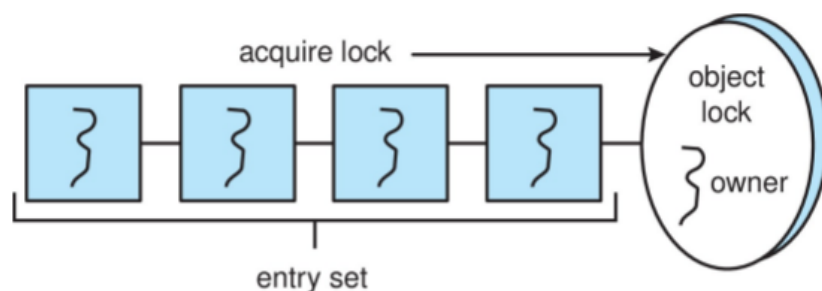
1) fa una

`wait` sul lock

2) esegue il blocco

3) fa una

`signal` sul lock



Esempio :

Method Locking (synchronized)

```
public class SyncCounter  
implements Runnable  
{
```

Block Locking (synchronized)

```
public class SyncBlockCounte  
implements Runnable  
{
```

```
private int counter = 0;
public synchronized
void run() {
    int tmp = counter;
    counter = tmp + 1;
}
}
```

```
private int counter = 0;
public void run() {
    synchronized(this) {
        int tmp = counter;
        counter = tmp + 1;
    }
}
}
```

Code di Attesa in Java

Ogni oggetto Java eredita da Object tre metodi :

- invocando

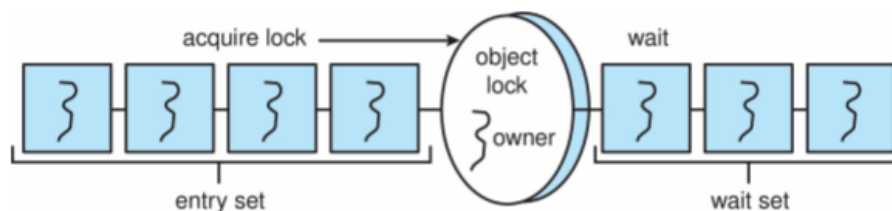
`wait()` blocca il thread in esecuzione, in attesa di un segnale

- invocando

`notify()` sblocca un thread a caso tra quelli in attesa

- invocando

`notifyAll()` sblocca tutti i thread in attesa



Un oggetto è come se avesse la possibilità di gestire due code : una coda di ingresso che si crea usando metodi o blocchi *synchronized*, più una coda di wait che si utilizza usando la `wait()` .

Language-based

Monitor :

Una classe JM può implementare un monitor M come segue :

- ogni attributo in JM è privato

- ogni metodo in JM è sincronizzato (garantendo l'esclusione in modo atomico)

```
monitor class M {
    int x, y;
```

```
class JM {
    private int x, y;
```

```

public void p()
{ /* ... */ }
public int q()
{ /* ... */ }
}

```

```

public synchronized void
{ /* ... */ }
public synchronized int q
{ /* ... */ }
}

```

Questo meccanismo non garantisce l'equità della coda di accesso associata al monitor → viene scelto dalla coda un thread in modo casuale.

Condition :

Ogni language-based monitor include implicitamente una (sola) variabile condition con semantica di tipo *segnala e continua* :

- invocando

`wait()` blocca il thread in esecuzione, in attesa di un segnale

- invocando

`notify()` sblocca un thread a caso tra quelli in attesa del monitor

- invocando

`notifyAll()` sblocca tutti i thread in attesa del monitor

```

monitor class M {
    int x;
    Condition y;
    public void p() {
        while (x < 0)
            y.wait();
    }
    public int q() {
        if (x > 0)
            y.signal();
    }
}

```

```

class JM {
    private int x;
    public synchronized void p
        while (x < 0)
            wait();
    }
    public synchronized int q(
        if (x > 0)
            notify();
    }
}

```

Non abbiamo bisogno di definire `y` perchè ogni classe ce l'ha implicitamente.

Questo meccanismo non garantisce l'equità della coda di attesa → viene scelto dalla coda un thread in modo casuale.

Library-based

Monitor :

Una classe LM può implementare un monitor M usando variabili di lock :

- aggiungere un lock come attributo privato
- ogni metodo in LM usa in modo esplicito

`lock()` e `unlock()` (garantendo l'esecuzione in modo atomico)

```
monitor class M {  
    int x, y;  
    public void p()  
    { /* ... */ }  
}
```

```
class LM {  
    private final Lock monitor  
        = new ReentrantLock(true);  
    private int x, y;  
    public void p() {  
        monitor.lock();  
        /* ... */  
        monitor.unlock();  
    }  
}
```

Questo meccanismo garantisce l'equità della coda di accesso associata al monitor.

L'accesso ai metodi pubblici del nostro monitor avviene in modo mutuamente esclusivo.

Condition :

```
monitor class M {  
    Condition y;  
    int x;  
    //...  
}
```

```
class LM {  
    private final Lock monitor  
        = new ReentrantLock(true);  
    private final Condition y  
        = monitor.newCondition();  
    private int x;  
    //...  
}
```

Le variabili condition con semantica di tipo *segnala e continua* possono essere generate da un lock :

- invocando

`await()` blocca il thread in esecuzione, in attesa di un segnale

- invocando

`notify()` sblocca un thread a caso tra quelli in attesa del monitor

- invocando

`notifyAll()` sblocca tutti i thread in attesa del monitor

```
monitor class M {  
    //...  
    public void p() {  
        while (x < 0)  
            y.wait();  
    }  
    //...  
}
```

```
class LM {  
    //...  
    public void p() {  
        monitor.lock();  
        while (x < 0)  
            y.await();  
        monitor.unlock();  
    }  
    //...  
}
```

Questo meccanismo garantisce l'equità della coda di attesa (`notifyAll()` garantisce il rispetto dell'ordine di arrivo).

Le variabili condition con semantica di tipo

segnala e continua possono essere generate da un lock :

- invocando

`await()` blocca il thread in esecuzione, in attesa di un segnale

- invocando

`signal()` sblocca il primo thread in attesa sulla condition

- invocando

`signalAll()` sblocca tutti i thread in attesa sulla condition

```
monitor class M {  
    //...  
    public int q() {  
        if (x > 0)  
            y.signal();  
    }  
    //...  
}
```

```
class LM {  
    //...  
    public int q() {  
        monitor.lock();  
        if (x > 0)  
            y.signal();  
        monitor.unlock();  
    }  
}
```

```
//...  
}
```

Questo meccanismo garantisce l'equità della coda di attesa (`signalAll()` garantisce il rispetto dell'ordine di arrivo).

Eccezioni nei Thread

Cosa succede se un thread ha una eccezione mentre possiede il controllo di un monitor? È importante che i programmi assicurino che in un caso del genere il thread lasci comunque il sistema in uno stato consistente rilasciando tutti i monitor che possiede:

- nei language-based monitor, un thread che ha una eccezione mentre si trova in un metodo `synchronized` rilascia automaticamente il monitor
- nei library-based monitor, utilizzare un blocco *finally* per rilasciare il blocco del monitor in caso di eccezione:

```
class LM {  
    private final Lock monitor = new ReentrantLock(true);  
    public void p() {  
        monitor.lock();  
        try { /* ... */ }  
        finally { monitor.unlock(); }  
    }  
}
```

Cinque Filosofi (Monitor)

Cinque Filosofi (Semafori).

Un monitor non è una semplice struttura dati, ma è un oggetto che può fare esso stesso delle operazioni, quindi ad esempio nel problema dei *cinque filosofi* può controllare se le bacchette siano libere e risponde direttamente al filosofo stesso.

Anzitutto ci serve sapere gli stati dei processi, cioè se un filosofo sta mangiando oppure sta pensando oppure ha richiesto di mangiare ma non ha le bacchette (ovvero non ha le risorse necessarie).

Poi abbiamo bisogno di gestire l'eventuale attesa di quel processo se non può mangiare: abbiamo quindi N (nel caso dei cinque filosofi sono 5) variabili di

condition, una per ogni filosofo. In questo modo il monitor lo mette in attesa finché non può mangiare.

```
public class MonitorTable implements Table {  
  
    private enum State {THINKING , HUNGRY, EATING};  
    State[] state = new State[N];  
    private final Lock mutex = new ReentrantLock(true);  
    Condition[] self = new Condition[N];  
}
```

Poi abbiamo bisogno di metodi per prendere e rilasciare le bacchette. Abbiamo bisogno di altri due metodi: uno è il costruttore per inizializzare i filosofi e un altro metodo è quello che va a controllare (`test`) se le bacchette (risorse) sono libere oppure no.

```
// costruttore -> inizializza filosofi  
public void MonitorTable() {  
    for (int i = 0; i < N; i++) {  
        state[i] = THINKING; // metti tutti i filosofi a THINKING  
        self[i] = mutex.newCondition(); //mette i processi in c  
    }  
}
```

```
//metodo che controlla le bacchette  
private void test(int i) {  
    if ( (state[(i + 4) % N] != EATING) //filosofo a destra  
        &&(state[i] == HUNGRY)           //filosofo i  
        &&(state[(i + 1) % N] != EATING))//filosofo a sinistra  
    {  
        state[i] = EATING; //mangio  
        self[i].notify();  //sblocca il processo i-esimo  
    }  
}
```

Il metodo `test` semplicemente osserva se i filosofi che stanno alla destra e alla sinistra del filosofo

i -esimo non stanno mangiando e l' i -esimo filosofo ha richiesto di mangiare → allora lui può incominciare a mangiare.

Se una di queste tre condizioni è falsa, non può mangiare.

Quindi lo stato passa a EATING e se il processo è stato sospeso lo attivo, in caso contrario non faccio niente.

Quando un filosofo chiede la bacchetta mettiamo lo stato di quel processo in richiesta delle risorse (HUNGRY, significa che vuole mangiare), poi richiamiamo la procedura di `test` per osservare se le bacchette sono libere e può lavorare (cioè mangiare e quindi il suo stato andrà in EATING) oppure no.

Quindi qualora lo stato fosse diventato EATING non faccio niente perchè può lavorare, altrimenti faccio una

`await` su quel processo.

Quando un filosofo invece ha finito di lavorare riporta lo stato del processo a non fare niente (THINKING, significa che ha smesso di mangiare), poi controlla i propri vicini se attendevano quei processi che avevo io (cioè se il filosofo di destra e di sinistra hanno bisogno delle mie bacchette).

```
public void getChopstick(int i) {
    mutex.lock();
    try {
        state[i] = HUNGRY;
        test(i);
        while (state[i] != EATING) {
            //notify potrebbe risvegliare processo non HUNGRY
            //while soluzione ai risvegli spuri -> evita l'esecuzione

            self[i].await(); //sospendo il processo fino a quando
        }
    }
    finally {
        mutex.unlock();
    }
}
```

```
public void putChopstick(int i) {
    mutex.lock();
    state[i] = THINKING; //mi metto a pensare
    test((i + 4) % 5); //verifico stato filosofo a destra e si
    test((i + 1) % 5);
    mutex.unlock();
}
```

Un effetto collaterale potrebbe essere quello di aver risvegliato i filosofi a sinistra e destra del filosofo i -esimo.

Questa soluzione soddisfa la mutua esclusione ed il progresso (in quanto ogni filosofo o prende entrambe le bacchette o non le prende affatto), ma non l'attesa limitata (in questa situazione specifica dei cinque filosofi).

Modello a Scambio di messaggi

I processi comunicano tra loro senza ricorrere alle variabili condivise (i thread non condividono delle aree di memoria) → non ci sono problemi di corsa critica.

Il modello a scambio di messaggi prevede due operazioni:

- `send(message)` ; messaggi a dimensioni fisse o variabili
- `receive(message)`

Se due processi (P e Q) desiderano comunicare, hanno bisogno:

- di stabilire un canale di comunicazione tra loro
- di scambiarsi i messaggi con le operazioni

`send` / `receive`

La realizzazione del canale di comunicazione avviene sia per via *fisica* (memoria condivisa, hardware bus, ...) sia per via *logica* (proprietà logiche).

Produttore e Consumatore (Scambio di messaggi)

I produttori devono produrre dei messaggi, però siccome il buffer è limitato a 10 non potranno essere inviati più di 10 messaggi prima che ci sia almeno un consumatore che inizi a consumarli.

```
public class Message {
    public static final int BUFFER_SIZE = 10; // dimensione del
```

```

private Type value;
public void send(Thread t) { /* ... */}
public void receive(Thread t) { /* ... */}
public void setValue(Type value) /* assegna un valore al co
{ /* ... */}
public Type getValue() // estrae il contenuto dal messaggio
{ /* ... */}
}

```

Ci serve un canale tale per cui il destinatario abbia una coda di messaggi, il buffer limitato significa che la coda dei messaggi deve avere una dimensione fissa, non può essere superiore al limite fissato.

Produttore

```

public void producer() {
    Message msg;
    Type v;

    while (true) {
        v = produce_item();
        msg.receive(consumer);
        //mette il processo in attesa fino a quando non arriva un m
        essage -> receive bloccante

        msg.setValue(v);
        msg.send(consumer);
        //send non bloccante: invia il messaggio e torna a produrre
        nuovi valori
    }
}

```

Consumatore

```

private void consumer() {
    Message msg;

```

```

    Type v;

    //quanti messaggi
    for (i=0; i<BUFFER_SIZE; i++)
        msg.send(producer);
    while(true) {
        msg.receive(producer);
        v = msg.getValue();
        msg.send(producer);
        consumer_item(v);
    }
}

```

Possiamo usare i messaggi stessi come strumento per indicare che c'è spazio per un nuovo messaggio.

Il consumatore, una volta che ha estratto un messaggio, invia un messaggio al produttore in cui riferisce che c'è uno spazio disponibile in più.

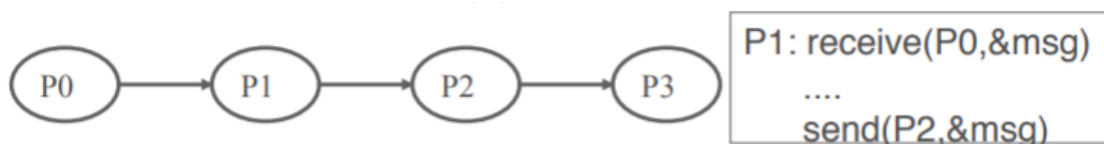
Il produttore, prima di andare a caricare nel payload quel valore che lui ha prodotto e che vuole spedire, aspetta di avere conferma dal consumatore che effettivamente c'è almeno uno spazio nel buffer.

Designazione di mittente/destinatario

Canale : collegamento logico tramite il quale due processi comunicano.

- Comunicazione simmetrica : si usano direttamente i nomi dei processi coinvolti per denotare un canale (quando il mittente indica il nome del destinatario e il destinatario, quando esegue la `receive`, indica esplicitamente da chi si aspetta di ricevere il messaggio).

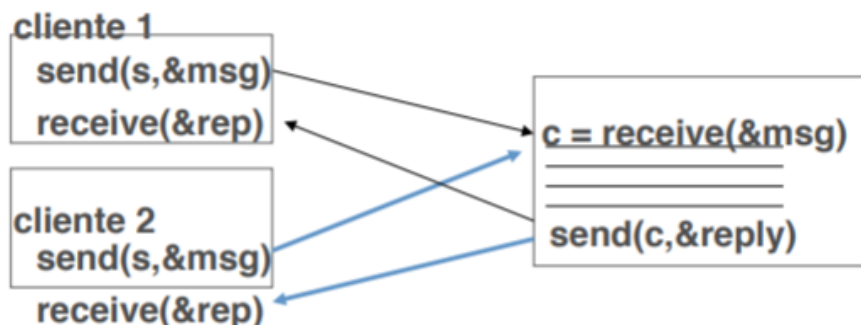
Esempio : pipeline. Un processo P_0 invia un messaggio a P_1 che sa che deve ricevere questo messaggio solo da P_0 .



- Comunicazione asimmetrica : si ha quando il processo mittente nomina esplicitamente il destinatario, ma quest'ultimo invece non esprime il nome

del processo che vuole ricevere perchè non lo sa in anticipo.

Esempio : modello client-server, schema molti (clienti) a uno (servitore).



Si può parlare di comunicazione *diretta* o *indiretta* a seconda del fatto che vengano nominati esplicitamente i processi o meno.

Comunicazione Diretta

Ogni processo deve nominare esplicitamente il processo con cui vuole comunicare.

Le operazioni sono:

- `send(P, message)` → invia un messaggio al processo P
- `receive(Q, message)` → ricevi un messaggio dal processo Q

All'interno di questo schema un canale di comunicazione ha le seguenti proprietà:

- I canali sono stabiliti automaticamente
- Ad ogni canale è associata una coppia di processi comunicanti
- Per ogni coppia esiste un solo canale
- I canali possono essere unidirezionali o bidirezionali

Questo sistema ha una *simmetria* dell'indirizzamento, cioè per poter comunicare, il trasmittente e il ricevente devono nominarsi a vicenda.

Una variante si avvale dell'*asimmetria* dell'indirizzamento, cioè soltanto il trasmittente nomina il ricevente, il ricevente non deve nominare il trasmittente

- `send (P, message)` → invia un messaggio al processo P

- receive (id, message) → in id si riporta il nome del processo con cui è avvenuta la comunicazione

Comunicazione indiretta

Si ha una comunicazione indiretta quando il destinatario del messaggio non deve essere noto, ma deve essere nota una casella postale nella quale recapitare i messaggi

I messaggi sono diretti verso / ricevuti da mailbox. Ogni mailbox ha un unico identificatore. Due processi possono comunicare solo se condividono la mailbox.

Le operazioni che si possono eseguire su una mailbox sono:

- creare una nuova mailbox
- inviare e ricevere messaggi attraverso la mailbox
- distruggere una mailbox

Le primitive di questa tipologia di comunicazione sono:

- `send(A, message)` → invia un messaggio alla mailbox A
- `receive(A, message)` → ricevi un messaggio dalla mailbox A

In questo schema si hanno le seguenti caratteristiche:

- Si stabilisce un canale solo se i due processi condividono una porta.
- Un canale può essere associato a più processi.
- Ogni coppia di processi può condividere diversi canali.
- I canali possono essere unidirezionali o bidirezionali.

Si supponga che i processi P_1 , P_2 , P_3 condividono la stessa mailbox A.

P_1 invia un messaggio ad A, e sia P_2 che P_3 ricevono il messaggio.

Si deve capire chi riceverà il messaggio, e la soluzione dipende dallo schema scelto:

- Permettere ad un canale di essere associato solo a due processi alla volta.
- Permettere ad un solo processo alla volta di effettuare una receive().
- Permettere al sistema di selezionare arbitrariamente il ricevente, per poi notificare al mittente chi ha ricevuto il messaggio.

In generale :

Se vogliamo usare un modello di comunicazione tra un numero arbitrario di processi la comunicazione diretta non è possibile.

Se abbiamo una comunicazione di tipo asimmetrico, quindi il destinatario non sa esattamente da chi riceverà il messaggio, come facciamo a dire qual è il nome del mittente?

In una comunicazione di tipo asimmetrico è preferibile usare una mailbox.

Supponiamo di avere due soli processi in cui uno deve inviare un messaggio all'altro.

In molti casi è obbligatorio usare una mailbox perchè non è possibile ipotizzare di conoscere sempre l'id dell'altro processo.

Buffering

La coda di messaggi associata ad un canale di comunicazione può essere realizzata in uno dei seguente tre modi:

1)

Capacità Zero → 0 messaggi.

Il mittente deve attendere che il ricevente legga il messaggio prima di inviare un nuovo messaggio (*rendez-vous*).

2) Capacità Limitata → lunghezza finita di n messaggi.

Il mittente deve attendere solo quando il canale è pieno.

3)

Capacità Illimitata → lunghezza infinita.

Il mittente non deve attendere mai.

Primitive di sincronizzazione

Tipi di `send` :

- **send non bloccante**
- **send bloccante con buffer limitato**
- **send bloccante**

Tipi di `receive` :

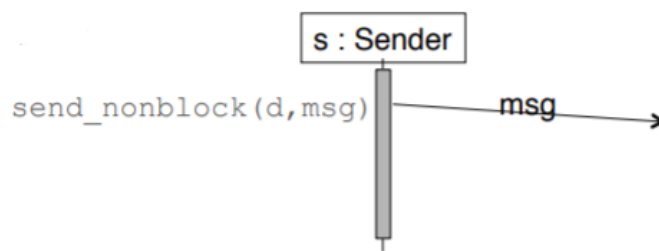
- **receive non bloccante**

- **receive bloccante**

Send non bloccante

Il mittente prosegue la sua esecuzione immediatamente dopo che il messaggio è stato inviato, senza attendere che sia stato ricevuto.

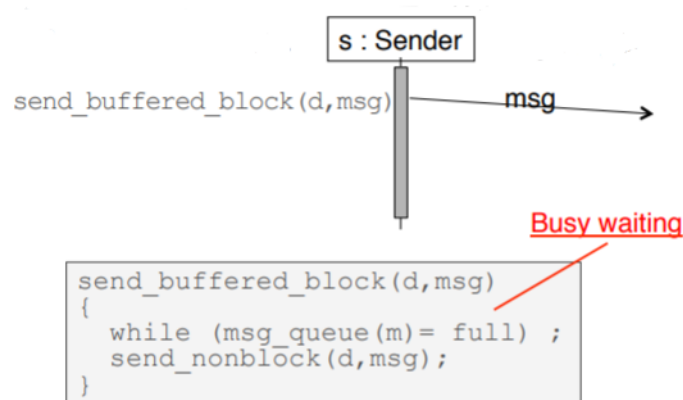
Il buffer è illimitato.



Send bloccante con buffer limitato

Il mittente prosegue la sua esecuzione immediatamente dopo che il messaggio è stato inviato, senza attendere che sia stato ricevuto, a meno che il buffer non sia pieno (fino a che non ha esaurito la coda dei messaggi del destinatario; quando la coda dei messaggi è piena, esegue una `send` e aspetta che nella coda si liberi un posto).

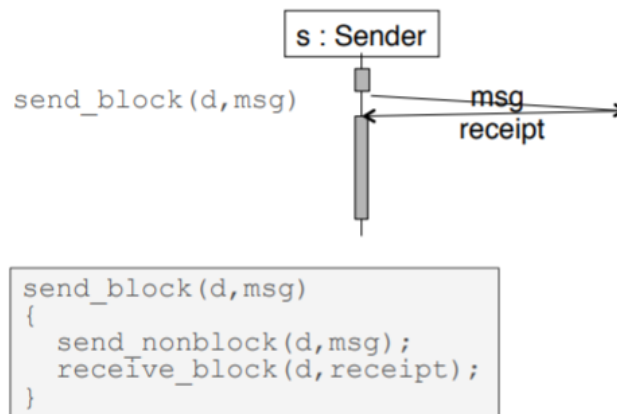
Il buffer è di lunghezza limitata.



Send bloccante

Il mittente attende che il messaggio sia stato ricevuto dal destinatario.

Il buffer è di lunghezza nulla, ovvero un processo invia un messaggio e non può andare avanti fino a quando la coda non si è liberata.

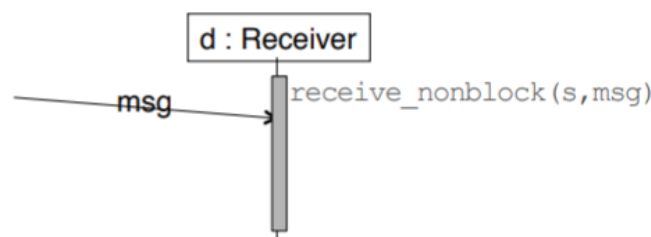


Receive non bloccante

Il destinatario che esegue una receive non rimane bloccato.

Di solito in questo caso esiste anche una funzione che permette di verificare quando il buffer indicato in una receive precedente contiene effettivamente un messaggio.

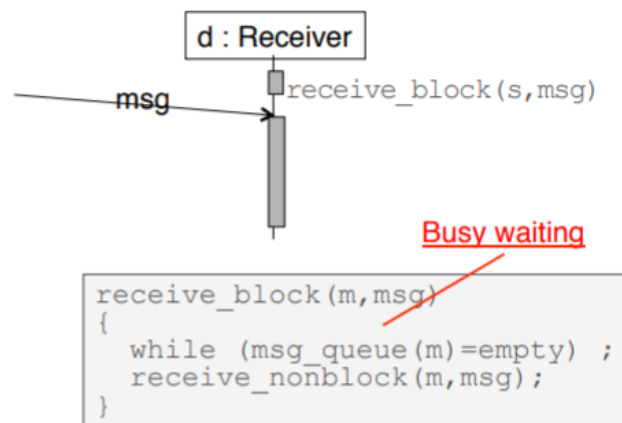
Il processo esegue le sue istruzioni e poi la `receive`: se in quel momento c'è un messaggio, legge e va avanti; se non c'è un messaggio non si ferma in attesa, ma va avanti (consapevole del fatto che non ha letto nessun messaggio).



Receive bloccante

Questa è la scelta più comune: il destinatario che esegue una receive rimane bloccato (stato *waiting*) finché non è disponibile un messaggio per lui. Quando

il messaggio arriva ad un processo bloccato su una receive, quest'ultimo viene riattivato (stato *ready*).

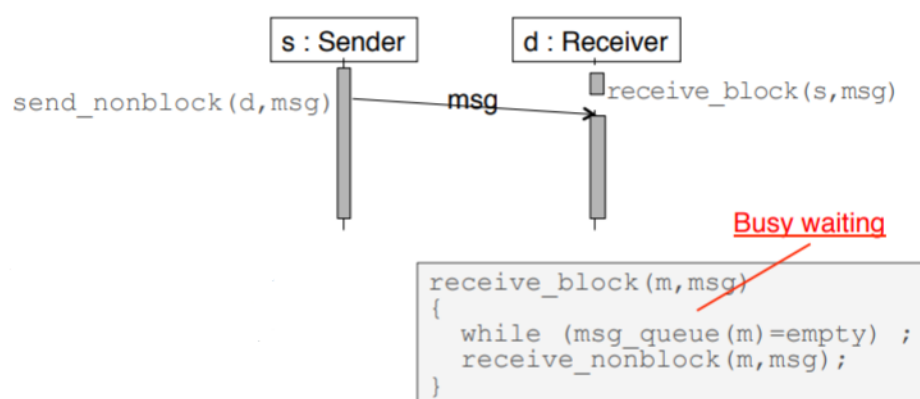


Meccanismi di sincronizzazione

Scambio di messaggi Asincrono

Da lato mittente si ha o una send non bloccante o bloccante con buffer limitato.
Da lato destinatario si ha una receive bloccante.

Si dice *asincrono* perchè il mittente non si sincronizza con il destinatario, lui continua a fare quello che deve fare senza attendere qualcosa da parte del destinatario.



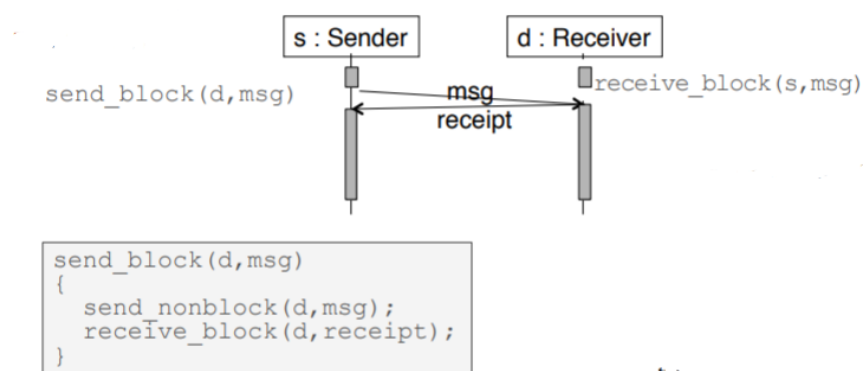
Scambio di messaggi Sincrono

Il passaggio del messaggio dal mittente al destinatario costituisce un punto di sincronizzazione (rendez-vous). Il messaggio ricevuto contiene informazioni sullo stato attuale del mittente.

Da lato mittente si ha una send bloccante.

Da lato destinatario si ha una receive bloccante.

Si dice *sincrono* perché se arriva prima il mittente invia il messaggio e si ferma fino a quando il destinatario non arriva anche lui in quel punto e riceve il messaggio, dopodiché ripartono tutti e due; viceversa, se arriva prima il destinatario si ferma e attende che anche il mittente arrivi in quel punto e gli invii il messaggio, a quel punto ripartono tutti e due.



Invocazione remota

Il processo che esegue la chiamata è un cliente, mentre la procedura viene eseguita da un processo servitore remoto. Il cliente rimane sospeso fino a quando il servitore non ha terminato il servizio (rendez-vous esteso).

Da lato mittente si ha una send bloccante.

Da lato destinatario si ha una receive bloccante.

Il mittente non resta in attesa di una conferma di lettura del messaggio, ma resta in attesa di una risposta da parte del destinatario.

