# CAPITOLO 6: SINCRONIZZAZIONE DEI PROCESSI

## 6.1 INTRODUZIONE

Un processo cooperante è un processo che può influenzarne un altro in esecuzione nel sistema o anche subirne l’influenza. I processi cooperanti possono condividere direttamente uno spazio logico di indirizzi.

L’accesso concorrente a dati condivisi può tuttavia causare situazioni di incoerenza degli stessi dati. Parlando del problema del produttore/consumatore e rifacendoci al concetto di buffer. Come sappiamo la nostra memoria può condividere al massimo DIM\_BUFFER -1 elementi. Quindi per tenere traccia del numero di elementi si aggiunge una variabile, contatore, inizializzata a 0 che incrementa ad ogni produzione e decrementa ad ogni consumo.



**Contatore==DIM\_VETTORE** (busy waiting, attesa indefinita.

Le procedure del produttore e del consumatore possono non funzionare altrettanto correttamente se si eseguono in modo concorrente e si ha l’interfogliamento **(interleaving)**, cioè che il risultato dipende solo dal come sono schedulati i processi. Si supponga per esempio che il valore della variabile contatore sia attualmente 5, e che i processi produttore e consumatore eseguano le istruzioni contatore++ e contatore- - in modo concorrente. Si può avere attraverso le istruzioni del calcolatore un risultato diverso da quello che dovrebbe essere. Per evitare situazioni di questo tipo, il cui più processi accedono e modificano gli stessi dati in modo concorrente e i risultati dipendono dall’ordine degli accessi(**race condition)** occorre assicurare che un solo processo alla volta possa modificare la variabile contatore. Questa condizione richiede una forma di sincronizzazione dei processi.

## 6.2 PROBLEMA DELLA SEZIONE CRITICA

Si consideri un sistema composto di n processi (P0, P1, …,PN-1), ciascuno avente un segmento di codice chiamata sezione critica in cui il processo può modificare variabili comuni, aggiornare una tabella, scrivere in un file. 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. Quindi, l’esecuzione delle sezioni critiche da parte dei processi è ***mutuamente esclusiva*** nel tempo. Il problema della sezione critica si affronta progettando un protocollo che i processi possono utilizzare per cooperare. Ogni processo deve chiedere il permesso per entrare nella propria sezione critica. La sezione di codice che realizza questa richiesta è la **sezione d’ingresso**. La sezione critica può essere seguita da una **sezione d’uscita** e la restante parte del codice è detta **sezione non critica**.

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

* **Mutua esclusione**. Se il processo Pi è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
* **Progresso**. Se nessun processo è in esecuzione nella sua sezione critica e qualche processo desidera entrare nella propria sezione critica, solo i processi che si trovano fuori delle rispettive sezioni non critiche possono partecipare alla decisione riguardante la scelta del processo che può entrare per primo nella propria sezione critica; questa scelta non si può rimandare indefinitamente.
* **Attesa limitata**. Se un processo ha già richiesto l’ingresso nella sua sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che si accordi la richiesta del primo processo.

La due strategie principali per la gestione delle sezioni critiche nei sistemi operativi prevedono l’impiego di: (1) **kernel con diritto di prelazione** e (2) **kernel senza diritto di prelazione.**

**-** Un kernel con diritto di prelazione consente che un processo funzionante inmodalità di sistema sia sottoposto a prelazione, rinviandone in tal modo l’esecuzione.

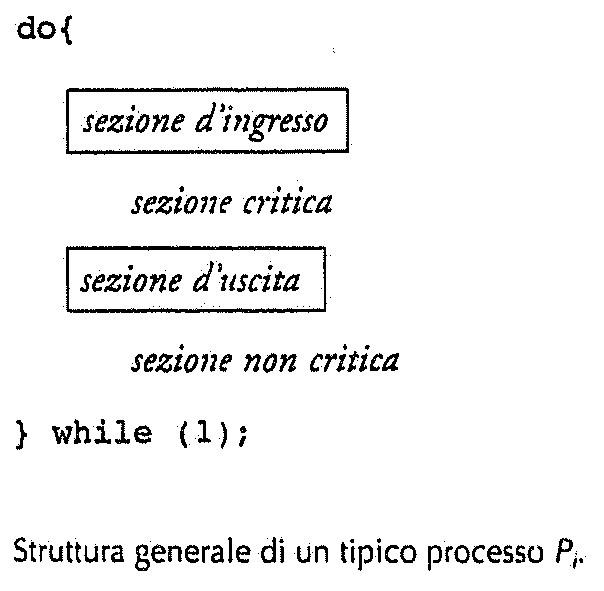
- Un kernel senza diritto di prelazione non consente di applicare la prelazione a un processo attivo

in modalità di sistema: l’esecuzione di questo processo seguiterà finché la stessa esca da tale modalità, si blocchi o ceda volontariamente il controllo della CPU.

In sostanza, i kernel senza diritto di prelazione sono immuni dai problemi legati all’ordine degli accessi alle strutture dati del kernel, visto che un solo processo per volta impegna il kernel. Ma i kernel con diritto di prelazione sono preferiti perché sono più adatti alla programmazione in real-time, dal momento che permette ai processi in tempo reale di far valere il loro diritto di precedenza nei confronti di un processo attivo nel kernel.

## Soluzione per 2 processi (non presente sull'ottava edizione)

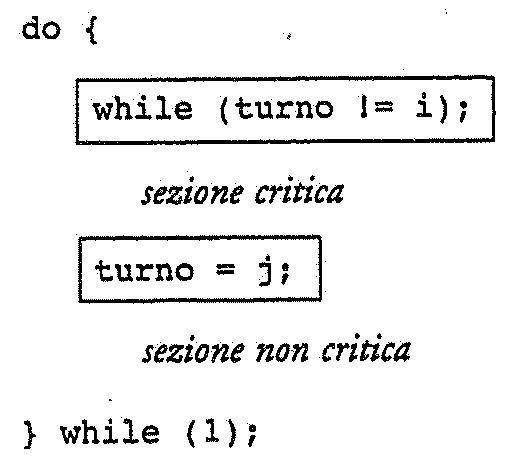
I processi sono indicati con P0 e P1. una volta introdotto un processo Pi l’altro è Pj dove j=1-i.



## Algoritmo 1 (non presente sull'ottava edizione)

**Variabili condivise:**

* int turno; //inizialmente =0
* turno -i; //Pi entra nella propria sezione critica

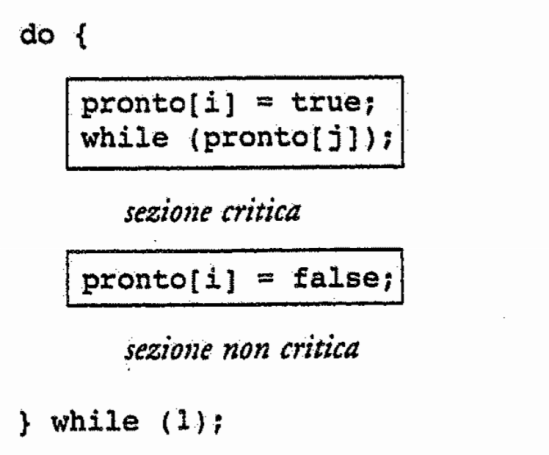


Il primo tentativo di soluzione consiste nel far condividere ai processi una variabile intera, turno, inizializzata a 0. se turno==i si permette al processo Pi di entrare nella propria sezione critica. Questa soluzione assicura che in un dato momento un solo processo può trovarsi nella propria sezione critica. Tuttavia la soluzione non soddisfa il requisito di progresso poiché richiede una stretta alternanza dei processi nell’esecuzione della sezione critica. Se ad esempio turno==0, P1 non può entrare nella propria sezione critica, anche se P0 si trova nella propria sezione non critica.

## Algoritmo 2 (non presente sull'ottava edizione)

**Variabili condivise:**

* Boolean pronto[2]; //inizialmente pronto[0]=pronto[1]= **false**
* Pronto[i]= true;//Pi pronto ad entrare nella sua sez critica.



L’algoritmo 1 non possiede informazioni sufficienti sullo stato di ogni processo; ricorda solo il processo cui si permette l’ingresso nella propria sezione critica. Per questo motivo si sostituisce la variabile turno con un vettore boolean pronto [2] che viene inizializzato a false. Se pronto[i] è true significa che Pi può entrare nella sezione critica, altrimenti entra Pj. Inizialmente viene posto pronto[i]=true permettendo al processo Pi di entrare nella sezione critica verificando che Pj non sia pronto per entrare nella propria sezione critica. Questa soluzione soddisfa il requisito di mutua esclusione ma no quello di progresso. Nel momento in cui sia pronto[i] che pronto[j] diventano true, Pi e Pj entrano nel ciclo infinito. Questo accade quando nello stesso momento vengono poste sia pronto[i] che pronto[j] a true.

## 

**Algoritmo 3 -> 6.3 soluzione di PETERSON**

La soluzione di Peterson si applica a 2 processi, ognuno dei quali esegue alternativamente la propria sezione critica e la sezione rimanente.

I processi sono indicati con P0 e P1. una volta introdotto un processo Pi l’altro è Pj dove j=1-i.

La soluzione di Peterson richiede che i processi condividano i seguenti dati:

int turno ;

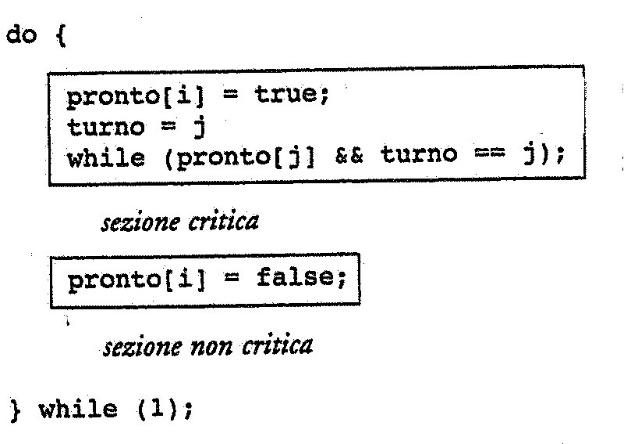
boolean flag [ 2 ] ;

La variabile turno segnala, per l’appunto, di chi sia il turno d’accesso alla sezione critica;

quindi, se turno == i , il processo *Pi* è autorizzato a eseguire la propria sezione critica.

L’array flag , invece, indica se un processo *sia pronto* a entrare nella propria sezione critica.

NB : flag[i] -> pronto[i]



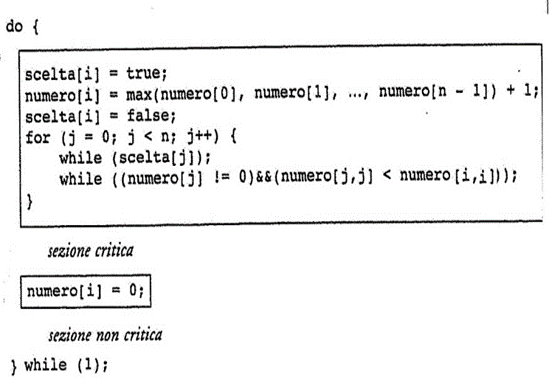
*//Pronto[i]=true “avverte all’altro processo che vuole andare in sez critica.*

//Pi aspetta l’altro processo che ha turno=j e pronto = true;

**Soddisfa tutti e 3 i requisiti e risolve tutti i problemi della sezione critica per 2 processi.** Per dimostrare che la **mutua esclusione** è preservata si osserva che Pi accede alla propria sezione critica solo se pronto [j]=false oppure turno==i, se entrambi i processi fossero contemporaneamente in esecuzione nelle rispettive sezioni critiche, si avrebbe che pronto[i]==pronto[j]==true (cosa impossibile) perché l’istruzione while non può essere eseguita da Pi e Pj contemporaneamente in quanto la variabile turno può assumere solo valore di i o j, ma non entrambi.

Per dimostrare la proprietà di **progresso** e **attesa limitata**, osserviamo come l’ingresso di un processo Pi nella propria sez. critica possa essere impedito solo se il processo è bloccato nella sua interazione while, con le condizioni flag[j] == true e turno == j. Qualora Pj non sia pronto a entrare nella sez. critica, flag[i] == false, e Pi può accedere alla propria sez. critica. Se Pj ha impostato flag[j] a true e sta eseguendo il proprio ciclo while, turno==i, oppure turno == j. Se turno == i, Pi entrerà nella propria sez. critica. Se turno ==j, Pj entra nella sez. critica. Tuttavia al momento di uscita dalla sez. critica, Pj imposta flag[j] a false, consentendo a Pi di entrarvi. Poiché tuttavia Pi non modifica il valore della variabile turno durante l’esecuzione dell’istruzione while, Pi entrerà nella sez. critica (**progresso**) dopo che Pj abbia effettuato non più di un ingresso(**attesa limitata**)**.**

## Soluzione per più processi: algoritmo del fornaio (non presente sull'ottava edizione, consulta slide6)



//commenti istruzioni sul quaderno

L’algoritmo 3 risolve il problema della sezione critica per due processi, mentre l’algoritmo del fornaio lo risolve per n processi. È basato su uno schema di servizio usato nella panetteria dove si deve evitare la confusione dei turni. Al suo ingresso nel negozio ogni cliente riceve un numero. Si serve progressivamente il cliente con il più basso. A parità di numero si serve il cliente con il numero **pid** più basso, ordine lessicografico.

*Boolean scelta [n]*

*Int numero*

*(a,b)<(c,d) se a<c oppure se a=c e b<d*

**Mutua esclusione:** Se Pi si trova nella propria sezione critica e Pk (k!= i) ha già scelto il numero [k]!= da 0 allora (numero[i], i < numero[k],k) //ordine lessicografico

Se Pi è nella propria sez. critica e Pk tenta di entrare nella propria sez. critica il processo Pk esegue la seconda istruzione while per j==i, trovando che:

* numero[i]!=0;
* (numero[i],i) < (numero[k],k)

Quindi continua il ciclo nell’istruzione while fino a che Pi lascia la propria sez. critica.

**Progresso e Attesa Limitata:** Questi requisiti sono garantiti perché i processi entrano nelle rispettive sez. critiche secondo il criterio **FCFS** (first come first served) e al più un processo deve aspettare 1 giro, o altri n-1 giri per andare in sez. critica.

## 6.4 HARDWARE PER LA SINCRONIZZAZIONE

In generale si può affermare che qualunque soluzione al problema della sez. critrica richiede l’uso di un semplice strumento detto lock. Il corretto ordine degli accessi alle strutture dati del kernel è garantito dal fatto che sezioni critiche sono protette da lock. In altri termini un processo per accedere alla propria sezione critica deve ottenere il permesso di un lock.

In un sistema dotato di una singola CPU tale problema si potrebbe risolvere semplicemente se si potessero interdire le interruzioni. Non si potrebbe eseguire nessun'altra istruzione quindi non si potrebbe apportare alcuna modifica inaspettata alle variabili condivise. È questo l’approccio seguito dal kernel senza diritto di prelazione.

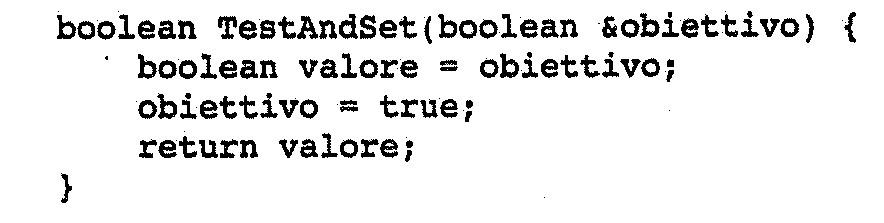
Sfortunatamente questa soluzione non è sempre praticabile; la disabilitazione delle interruzioni nei sistemi multiprocessore può comportare sprechi di tempo dovuti alla necessità di trasmettere la richiesta di disabilitazione delle interruzioni a tutte le unita d’elaborazione.

Tale trasmissione ritarda l’accesso a ogni sezione critica determinando una diminuzione dell’efficienza.

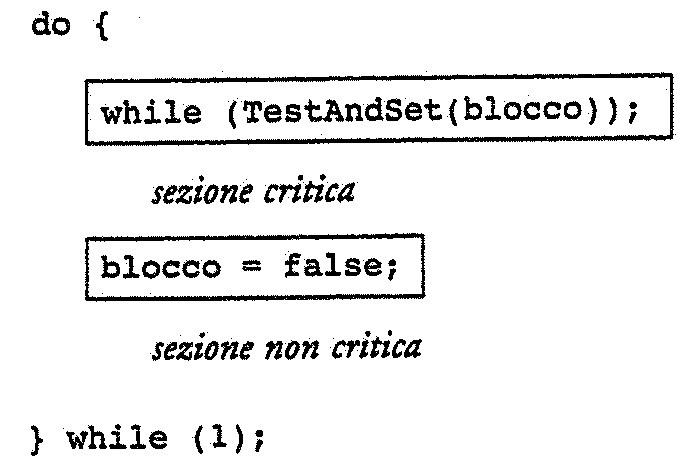
Per questo motivo molte architetture offrono particolari istruzioni che permettono di controllare e modificare il contenuto di una parola di memoria oppure di scambiare il contenuto di due parole di memoria in **modo atomico**( come un'unità non interrompibile).

Queste speciali istruzioni sono utilizzabili per risolvere il problema della sezione critica in modo relativamente semplice.

L’istruzione **TestAndSet()** è definito nell'immagine successiva. Questa istruzione è eseguita atomicamente, cioè come un’unita non soggetta a interruzioni; quindi, se si eseguono contemporaneamente due istruzioni TestAndSet(), ciascuna in un’unita d’elaborazione diversa, queste vengono eseguite in modo sequenziale in un ordine arbitrario.

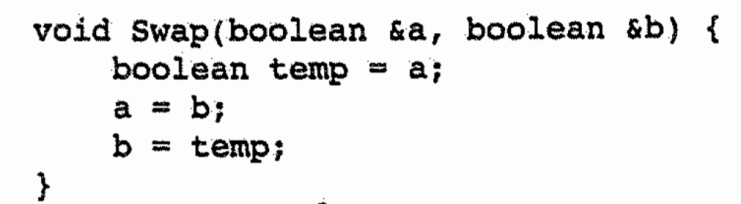


**Realizzazione MUTUA ESCLUSIONE con TestAndSet**: se si dispone dell’istruzione TestAndSet(), si può realizzare la mutua esclusione dichiarando una variabile booleana globale lock, inizializzata a false.

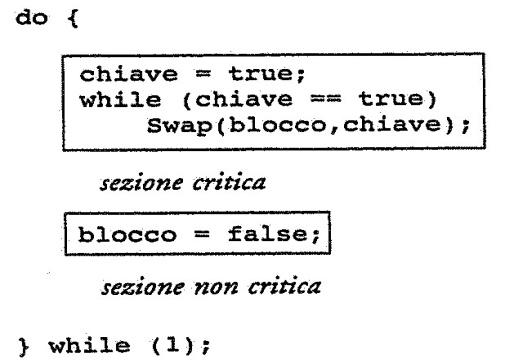


L’istruzione **Swap** agisce sul contenuto di 2 parole di memoria, anche essa è eseguita atomicamente.

Se si dispone dell’istruzione **swap()**, la mutua esclusione si garantisce dichiarando e inizializzando al valore **false** una variabile booleana globale lock, inoltre ogni processo possiede una variabile booleana **chiave** (algortimo che soddisfa la mutua esclusione ma non gli altri 2 requisiti).



**Realizzazione MUTUA ESCLUSIONE con SWAP**



**Test and set() che soddisfa tutti e 3 i requisiti**

La figura successiva mostra un algoritmo che soddisfa tutti e 3 i requisiti sfruttando l’istruzione **test and set()**

## 

**Le strutture condivise sono:**

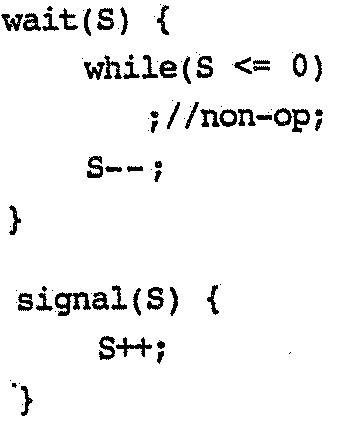
* Boolean attesa[n];
* boolean lock;

Entrambe inizializzate a false, per mostrare che l’algoritmo soddisfa il requisito di **mutua esclusione**, si considera che il processo Pi possa entrare nella propria sez. critica solo se attesa[i]==false o chiave==false. Il valore di chiave può diventare false solo se si esegue test and set(), il primo che esegue T&S trova chiave==false e tutti gli altri devono attendere, la var. attesa[i] può diventare false solo se un altro processo esce dalla propria sez. critica. Per dimostrare che l’algoritmo soddisfa il requisito di **progresso**, basta osservare che le argomentazioni fatte per la mutua esclusione valgono anche in questo caso; infatti un processo che esce dalla propria sez. critica imposta lock o attesa[j] a false, così si consentirà ad un processo in attesa l’ingresso della propria sez. critica. Per dimostrare che l’algoritmo soddisfa il requisito di **attesa limitata** occorre osservare che un processo, quando lascia la propria sez. critica, scandisce il vettore attesa nell’ordinamento ciclico(i+1, i+2,…,n-1, 0,.., i-1) e designa il primo processo in questo ordinamento nella sez. d’ingresso(attesa[j]==true) come il primo processo che deve entrare in sez. critica, qualsiasi processo che attende l’ingresso nella propria sez. critica può farlo entro n-1 turni.

## 6.5 SEMAFORI

Le varie soluzioni hardware al problema della sezione critica basate su istruzioni quali **TestAndSet()** e **Swap()** complicano l’attività del programmatore. Per ovviare questo problema si fa uso di uno strumento di sincronizzazione chiamato **semaforo**.

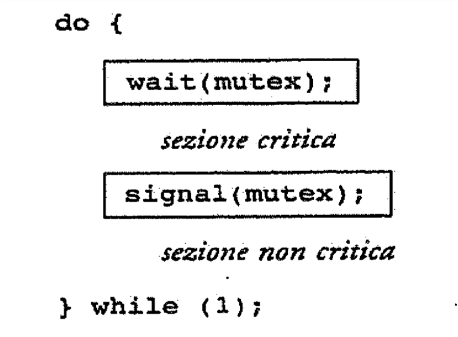
Un semaforo S è una variabile intera che si può accedere, escludendo l’inizializzazione, solo tramite due operazioni atomiche: **wait()** e **signal()**.



Tutte le modifiche del semaforo sono contenute nelle operazioni **wait()** e **signal()** e si devono eseguire in modo indivisibile: mentre un processo cambia il valore al semaforo, nessun altro processo può contemporaneamente modificare quello stesso valore. Inoltre nella **wait()** si devono eseguire senza interruzione anche la verifica del valore intero di S (S<=0) e la sua possibile modifica(S--).

## 6.5.1 Uso dei semafori

Si usa distinguere tra semafori contatore il cui valore è **illimitato** e semafori binari il cui valore è **0 o 1**. I semafori sono utilizzati per risolvere il problema della sezione critica con n processi che condividono un semaforo chiamato **mutex** (binario)inizializzato a 1.



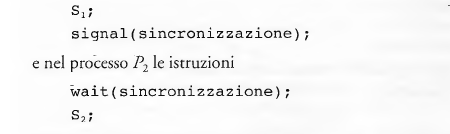
I semafori contatore trovano applicazione nel controllo dell’accesso a una data risorsa presente in un numero finito di esemplari. Il semaforo è inizialmente impostato al numero di esemplari disponibili. I processi che desiderino utilizzare un esemplare della risorsa invocano **wait ( )** sul semaforo, decrementandone così il valore; i processi che restituiscono un esemplare della risorsa, invece, invocano **signal( )** sul semaforo, incrementandone il valore.

Quando il semaforo vale 0, vengono allocati tutti gli esemplari della risorsa, e i processi che ne richiedano l’uso dovranno bloccarsi fino a che il semaforo non ritorni positivo.

I semafori sono utilizzabili anche per risolvere diversi problemi di sincronizzazione. Si considerino, per esempio, due processi in esecuzione concorrente: *P1*con un’istruzione *S1* e

*P2* con un’istruzione S2. Si supponga di voler eseguire S2 solo dopo che S1 è terminata. Questo schema si può prontamente realizzare facendo condividere a *P1 e P2* un semaforo comune,

"sincronizzazione", inizializzato a 0, e inserendo nel processo *P1* le istruzioni

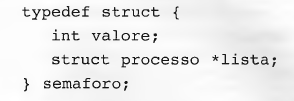


Poichè "sincronizzazione" è inizializzato a 0, *P2* esegue S2 solo dopo che *P1* ha eseguito "signal (sincronizzazione)", che si trova dopo S1.

**6.5.2 Realizzazione**

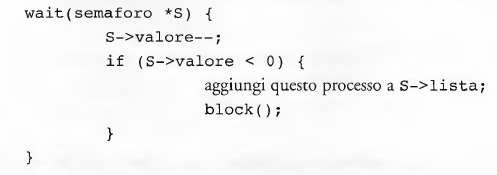
Il principale svantaggio della definizione di un semaforo e che richiede una condizione di attesa attiva (busy waiting). Mentre un processo si trova nella propria sezione critica, qualsiasi altro processo che tenti di entrarvi si trova sempre nel ciclo di codice della sezione di ingresso. Chiaramente questa soluzione costituisce un problema per un sistema con multiprogrammazione, poiché la condizione di attesa attiva spreca cicli della CPU che un altro processo potrebbe sfruttare in modo produttivo. Questo tipo di semaforo è detto anche **spinlock**, poiché i processi girano mentre attendono al semaforo. Però questi semafori hanno il vantaggio di non richiedere il cambio di contesto nel caso in cui un processo sia fermo in attesa. Per superare la necessità dell'attesa attiva, si possono modificare le definizioni delle operazioni wait () e signal (): quando un processo invoca l'operazione wait() e trova che il valore del semaforo non è positivo, deve attendere, ma anziché restare nell'attesa attiva può bloccare se stesso. L'operazione di bloccaggio pone il processo in una coda di attesa associata al semaforo e cambia lo stato del processo nello stato d'attesa. Quindi, si trasferisce il controllo allo scheduler dell'CPU che sceglie un altro processo pronto per l'esecuzione. Un processo bloccato, sarà riavviato in seguito all'esecuzione di un'operazione signal () su s da parte di qualche altro processo. Il processo si riattiva tramite un'operazione wakeup() che modifica lo stato del processo da attesa a pronto.

**Per realizzare i semafori si può definire il semaforo come una struttura in linguaggio C:**

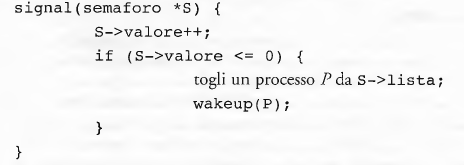
****

A ogni semaforo sono associati un valore intero e una lista di processi, contenente i processi in attesa a un semaforo: l’operazione signal() preleva un processo da tale lista e lo attiva

**L’operazione di wait() del semaforo si può definire come segue:**



//S.val<0 non ci sono unità di risorse disponibili, aggiungo questo processo alla lista associata a S.(semaforo) e blocco il processo

**L’operazione di signal() del semaforo si può definire come segue:**

//togli un processo “toglie un processo dalla lista associata al semaforo e lo sveglia con wakeup(), quando ci sarà un unita di risorsa libera

## L'operazione block() sospende il processo che la invoca; l'operazione Wakeup(P) pone in stato di pronto per l'esecuzione un processo P bloccato punto queste due operazioni sono fornite dal sistema operativo come chiamate di sistema base. Occorre notare che, mentre la definizione classica di semaforo ed attesa attiva è tale che il valore del semaforo non è mai negativo, tale definizione può condurre valori negativi. Se il valore del semaforo è negativo, la sua dimensione è data dal numero dei processi che attendono a quel semaforo. La lista dei processi che attendono ad un semaforo si può facilmente realizzare inserendo un campo puntatore in ciascun blocco di controllo del processo (PCB). Ogni semaforo contiene un valore intero è un puntatore a una lista di PCB. Per aggiungere togliere processi assicurando un'attesa limitata si può usare una coda FIFO della quale il semaforo contiene i puntatori al primo e l'ultimo elemento. I semafori devono essere eseguiti in modo atomico, si deve garantire che nessuno dei due processi possa eseguire operazioni wait() e signal() contemporaneamente sullo stesso semaforo. È importante rilevare che questa definizione delle operazioni wait() and signal() non consente di eliminare completamente l'attesa attiva, ma piuttosto di rimuoverla dalle sezioni di ingresso dei programmi applicativi.

## Stallo e attesa indefinita

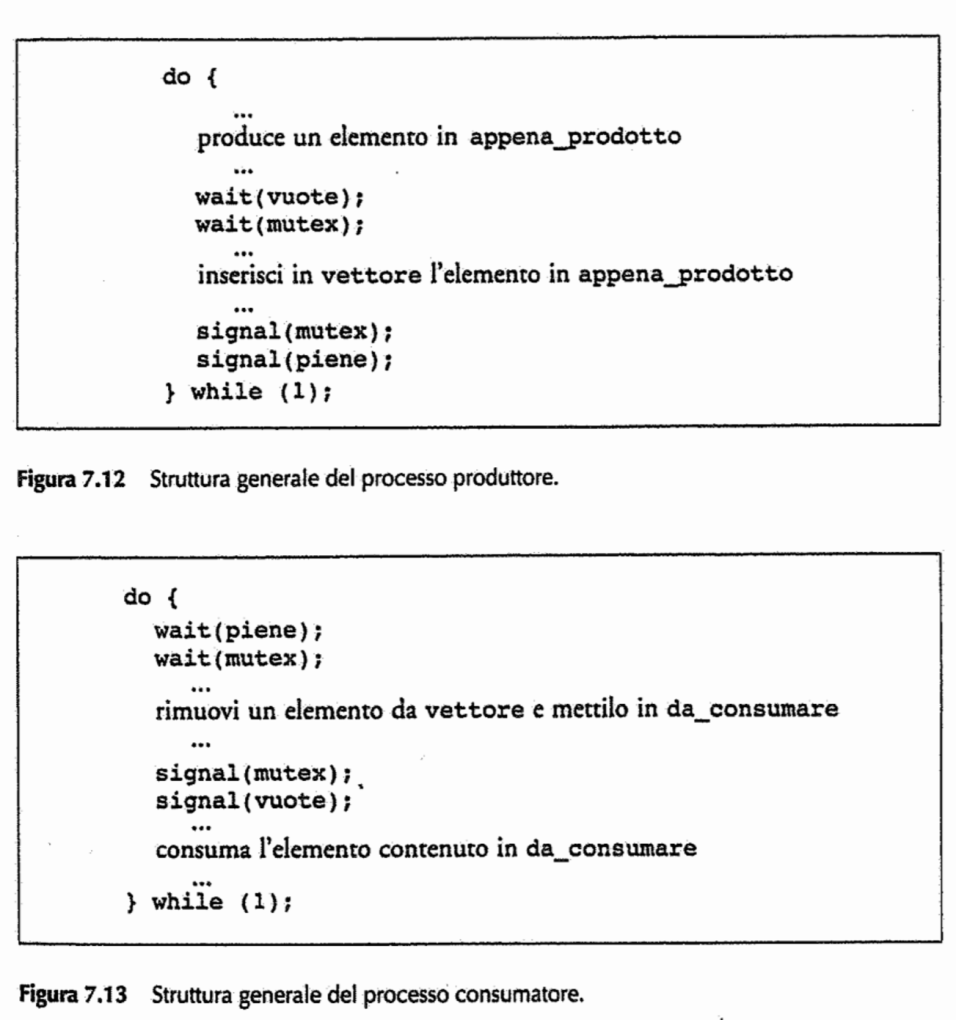
La realizzazione di un semaforo con coda d’attesa può condurre a situazioni in cui ciascun processo di un insieme di processi attende indefinitamente un evento che può essere causato solo da uno dei processi dello stesso insieme. Quando si verifica una situazione di questo tipo ci troviamo in una situazione di **stallo**. Un insieme di processi è in stallo se ciascun processo dell’insieme attende un evento che può essere causato solo da un altro processo dell’insieme. Un’altra questione connessa alle situazioni di stallo è quella dell’**attesa indefinita**(starvation), Con tale termine si definisce una situazione di attesa indefinita nella coda di un semaforo, che si può per esempio presentare sei processi si aggiungono e si rimuovono dalla lista associata a un semaforo secondo un criterio FIFO.

## 5.PROBLEMI TIPICI DI SINCRONIZZAZIONE

## Produttori consumatori con memoria limitata

Il problema dei produttori/consumatori con memoria limitata si usa generalmente per illustrare la potenza delle primitive di sincronizzazione.

Si supponga di disporre di una certa quantità di memoria rappresentata da un buffer con *n* posizioni, ciascuna capace di contenere un elemento. Il semaforo mutex garantisce la mutua esclusione degli accessi al buffer ed è inizializzato al valore 1 . I semafori ***vuote*** e ***piene*** conteggiano rispettivamente il numero di posizioni vuote e il numero di posizioni piene nel buffer. Il semaforo ***vuote*** si inizializza al valore n; il semaforo ***piene*** si inizializza al valore 0.



## Problema dei lettori-scrittori

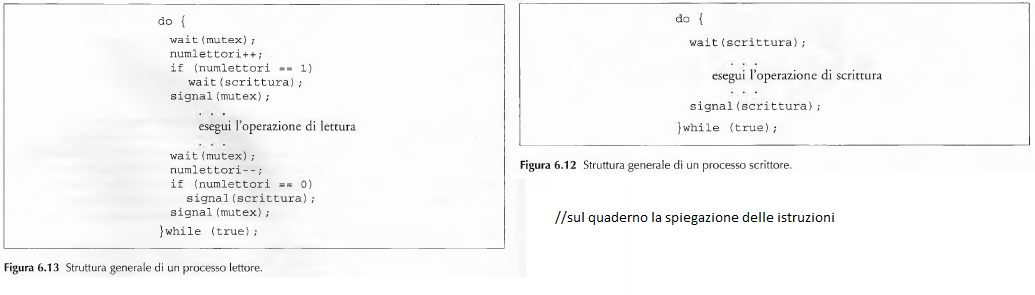
Si consideri una base di dati da condividere tra numerosi processi concorrenti. Alcuni processi possono richiedere solo la lettura del contenuto mentre altri possono scriverci. Qualsiasi può effettuare letture concorrenti ma nel momento in cui uno scrive, l’altro non può ne leggere ne scrivere altrimenti vi è una incoerenza di dati. Questo problema di sincronizzazione è conosciuto come problema dei lettori scrittori. Da quando tale problema fu enunciato, è stato usato per verificare quasi tutte le nuove preventivo di sincronizzazione. Il problema dei lettori - scrittori ha diverse varianti, che implicano tutte l'esistenza di priorità, la più semplice , cui si fa riferimento come il primo problema dei lettori- scrittori , richiede che nessun lettore attenda , a meno che uno scrittore abbia già ottenuto il permesso di usare l'insieme dei dati condivisi. Il secondo problema dei lettori- scrittori si fonda sul presupposto che uno scrittore , una volta pronto, eseguirà il proprio compito di scrittura al più presto. La soluzione del primo problema e quella del secondo può condurre a uno stato di attesa indefinita (starvation), degli scrittori nel primo caso e dei lettori nel secondo.

La soluzione del primo problema dei lettori- scrittori prevede dunque la condivisione da parte dei processi lettori delle seguenti strutture dati:

* **semaforo** mutex, scrittura;
* **int** numLettori;

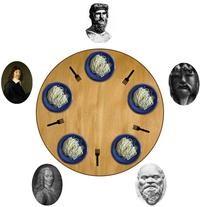
I semafori mutex e scrittura sono inizializzati a 1, numLettori è inizializzato 0. il semaforo scrittura è comune entrambi i processi(lettura e scrittura ). il semaforo mutex si usa per assicurare la mutua esclusione al momento dell' aggiornamento di numLettori. la variabile numLettori contiene il numero dei processi che stanno attualmente leggendo insieme di dati. Il semaforo scrittura funziona come semaforo di mutua esclusione per gli scrittori e serve anche al primo o all'ultimo lettore che intro esce dalla sezione critica (non serve ai lettori).

Le soluzioni al problema dei lettori- scrittori sono state generalizzate su alcuni sistemi in modo da fornire **lock di lettura-scrittura.** Per acquisire tale lock, è necessario specificare la modalità di lettura o scrittura: se il processo desidera solo leggere i dati condivisi, richiede un lock in modalità lettura; se invece desidera anche modificare i dati lo può richiedere in modalità scrittura. È permesso più processi di acquisire un lock di lettura ma un solo processo alla volta può avere il lock di scrittura.

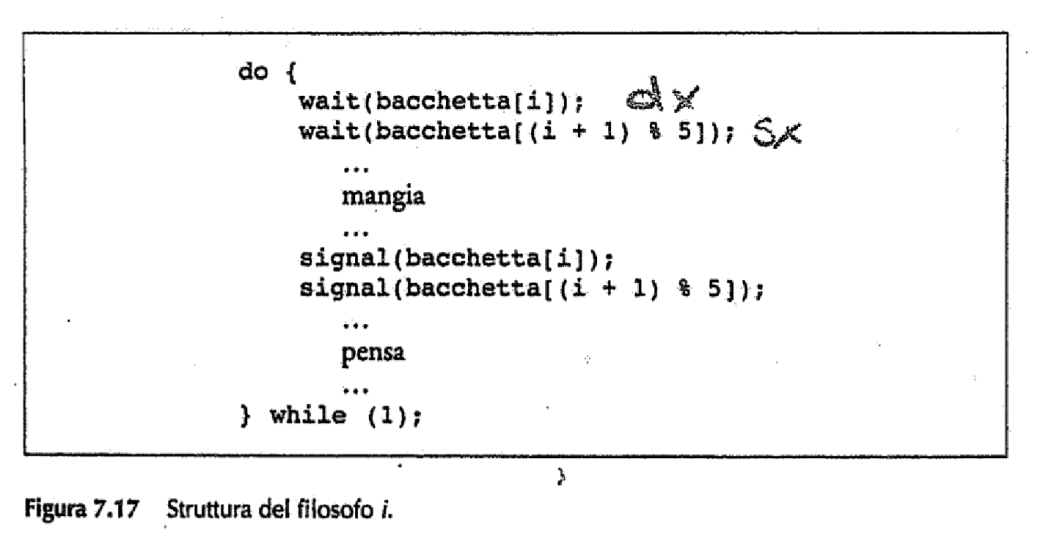


## 6.6.3 Problema dei 5 filosofi

Cinque filosofi siedono ad una tavola rotonda con un piatto di riso davanti, una bacchetta a destra e una bacchetta a sinistra. Ci sono dunque cinque filosofi, cinque piatti di riso e cinque bacchette.



Si Consideri no 5 filosofi che trascorrono la loro esistenza pensando e mangiando. Quando un filosofo pensa, non interagisce con i colleghi; quando gli viene fame, tenta di prendere le bacchette più vicine: Quelle che si trovano tra lui e commensali alla sua destra e alla sua sinistra. Un filosofo può prendere una bacchetta alla volta e non può prendere una bacchetta che si trova già nelle mani di un suo vicino. Quando un filosofo affamato tieni in mano due bacchette contemporaneamente, mangia senza lasciare le bacchette. Terminato il pasto, le posa e riprende a pensare.



Una semplice soluzione consiste nel rappresentare ogni bacchetta con un semaforo: un filosofo tenta di afferrare ciascuna bacchetta eseguendo un'operazione di wait() su quel semaforo e la si posa con un'operazione di signal() semafori appropriati. Quindi i dati da condividere sono:

**semaforo** bacchetta[5];

Questa soluzione garantisce che due vicini non mangiano contemporaneamente, ma è insufficiente poiché non esclude la possibilità che si abbia una situazione di stallo. Si supponga che tutti 5 filosofi abbiano fame contemporaneamente e che ciascuno tenti di afferrare la bacchetta di sinistra; tutti gli elementi di bacchetta diventa uguale a zero, perciò ogni filosofo che tenta di afferrare la bacchetta di destra entra in stallo. Di seguito sono elencate possibili soluzioni per tale situazione di stallo:

* solo quattro filosofi possono stare contemporaneamente a tavola.
* Un filosofo può prendere le sue bacchette solo se sono entrambe disponibili.
* si adotta una soluzione simmetrica: un filosofo dispari prende prima la bacchetta di sinistra e poi quella di destra, invece un filosofo pari prende prima la bacchetta di destra e poi quella di sinistra.

**6.7 Monitor**

Benché i semafori costituiscano un meccanismo pratico ed efficace per la sincronizzazione dei processi, il loro uso scorretto può generare errori difficili da individuare, in quanto si manifestano solo in presenza di particolari sequenze di esecuzione che non si verificano sempre. Neanche l'uso dei semafori, purtroppo, esclude la possibilità che si verifichi qualche errore di sincronizzazione. Per capire perché, analizziamo la soluzione al problema della sezione critica. Tutti i processi condividono una variabile semaforo mutex, inizializzata a 1. ogni processo deve seguire wait(mutex) prima di entrare nella sezione critica e signal(mutex) al momento di uscirne. Se questa sequenza non è rispettata, può accadere che i due processi occupino simultaneamente le rispettive zone critiche. Per rimediare a questi errori, i ricercatori hanno sviluppato costrutti con un linguaggio ad alto livello. Un costrutto fondamentale di sincronizzazione ad alto livello e chiamato **monitor**.

**6.7.1 Uso del costrutto monitor**

Il tipo monitor presenta un insieme di operazioni definite dal programmatore che, all'interno del monitor, sono contraddistinte dalla mutua esclusione. Il tipo monitor contiene anche la dichiarazione delle variabili i cui valori definiscono lo stato di un'istanza del tipo, oltre ai corpi delle procedure ho funzioni che operano su tali variabili. Lana presentazione di un tipo monitor non può essere usata direttamente dai vari processi. Pertanto, una procedura definita all'interno di un monitor ha accesso unicamente alle variabili dichiarate localmente, situate nel monitor, gli ai relativi parametri formali. In modo analogo, alle variabili locali di un monitor possono accedere solo le procedure locali. Il costrutto monitor assicura che all'interno di un monitor possa essere attivo un solo processo alla volta. Tale definizione di monitor non e abbastanza potente per esprimere alcuni schemi di sincronizzazione, sono perciò necessari ulteriori meccanismi che, in questo caso, sono forniti dal costrutto **condition.**

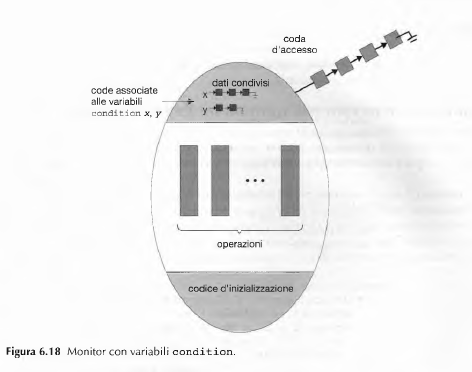
Un programmatore che deve scrivere un proprio schema di sincronizzazione può definire una o più variabili condizionali: **condition** x,y;

Le uniche operazioni eseguibili su una variabile **condition** sono **wait()** e **signal()**.

**x.wait()** Implica che il processo che l'ha invocato rimanga sospeso finché un altro processo non invochi l'operazione **x.signal()** Che risveglia esattamente un processo sospeso. Se non esistono processi sospesi l'operazione non ha alcun effetto.

Si supponga, che quando un processo P invoca l'operazione x.signal(), ed esista un processo sospeso Q associato alla variabile x di tipo condition. Occorre notare che, concettualmente, entrambi i processi possono continuare l'esecuzione. Sussistono quindi due possibilità:

* **segnalare ed attendere.** P Attende che Q lasci il monitor o attenda su un'altra variabile condition.
* **segnalare e proseguire**. Q Attende che P lasci il monitor o attenda su un'altra variabile condition.



**6.7.2 Soluzione al probl dei 5 filosofi con monitor**

Illustriamo quindi i concetti relativi al costrutto monitor presentando una soluzione sente da stallo del problema dei 5 filosofi. La soluzione però impone il vincolo che un filosofo posso prendere le sue bacchette solo quando siano entrambi disponibili. Per codificare questa soluzione si devono distinguere i tre diversi stati in cui un filosofo può trovarsi , a tale scopo si introduce la seguente struttura dati .

**Enum**{pensa, affamato, mangia} stato [5];

Il filosofo i può impostare la variabile stato[i]= mangia solo se i suoi due vicini non stanno mangiando:

((stato[(i+4)%5 != mangia) && (stato[(i+1)%5 != mangia)).



//sul quad la spiegazione

Inoltre occorre impiegare la seguente struttura dati: **condition** auto[5];

dove il filosofo i può ritardare se stesso quando ha fame, ma non riesce ad ottenere le bacchette.

La distribuzione delle bacchette e controllata dal monitor **fc**. Ciascun filosofo, prima di cominciare a mangiare, devi invocare l'operazione prende (); ciò può determinare la sospensione del processo filosofo. Completata con successo l'operazione , il filosofo può mangiare; in seguito, in bocca l'operazione posa() e comincia a pensare.

**6.7.3 Realizzazione di un monitor per mezzo di semafori**

A questo punto si considera la possibilità di realizzare il meccanismo del monitor usando i semafori. Ogni monitor si associa un semaforo mutex, inizializzato a 1; un processo deve seguire wait(mutex) prima di entrare nel monitor, e signal(mutex) dopo aver lasciato il monitor. Poiché un processo che esegue una Signal() deve attendere finché il processo risvegliato si mette in attesa o lasci il monitor, si introduce un altro semaforo, **prossimo,** inizializzato a 0, a cui i processi che eseguono una signal() possono autosospendersi. Per contare i processi sospesi al semaforo prossimo, si usa una variabile intera **prossimo\_contatore.** In questo modo si assicura la mutua esclusione all'interno del monitor. A questo punto si può descrivere la realizzazione delle variabili condition. per ogni variabile x di tipo condition si introducono un semaforo **x\_sem** e una variabile intera **x\_contatore**, entrambi inizializzati a 0.

**6.7.4 Ripresa dei processi all’interno di un monitor**

A questo punto si discute il problema dell'ordine di ripresa dei processi all'interno di un monitor. Sei più processi sono sospesi alla condizione x, e se qualche processo esegui l'operazione x.signal(), è necessario stabilire quale tra i processi sospesi si debba riattivare per primo. Una semplice soluzione consiste nell’usare un ordinamento **FCFS.**

tuttavia, in molti casi uno schema di scheduling di questo tipo non risulta adeguato; in questi casi si può usare un costrutto di **attesa condizionale** della forma: **x.wait(c).**

il valore di c, chiamato **numero di priorità,** viene memorizzato col nome del processo sospeso. Quando si esegue **x.signal(),** si riprende il processo cui è associato il numero di priorità più basso.