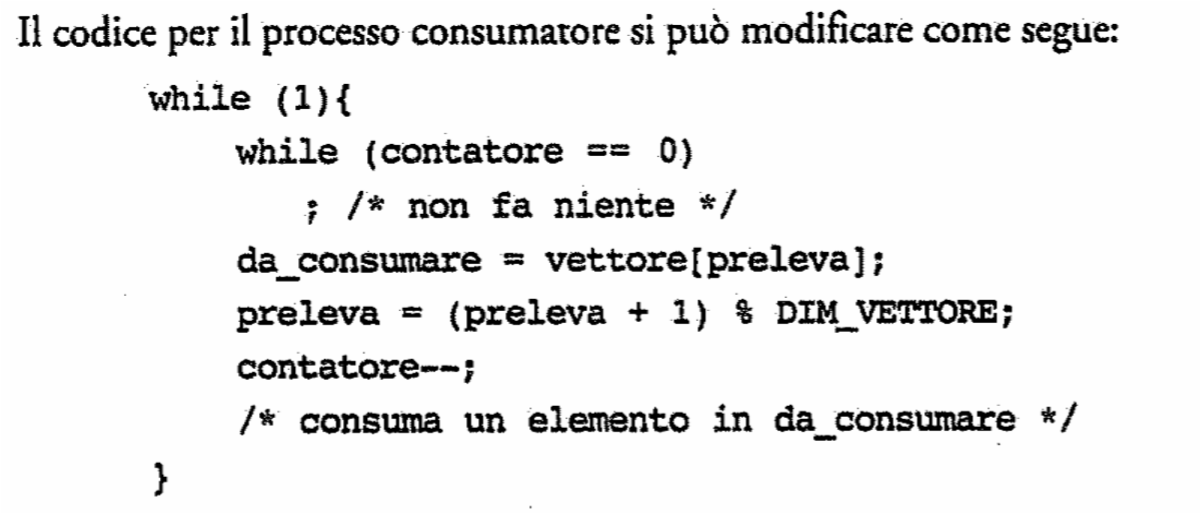
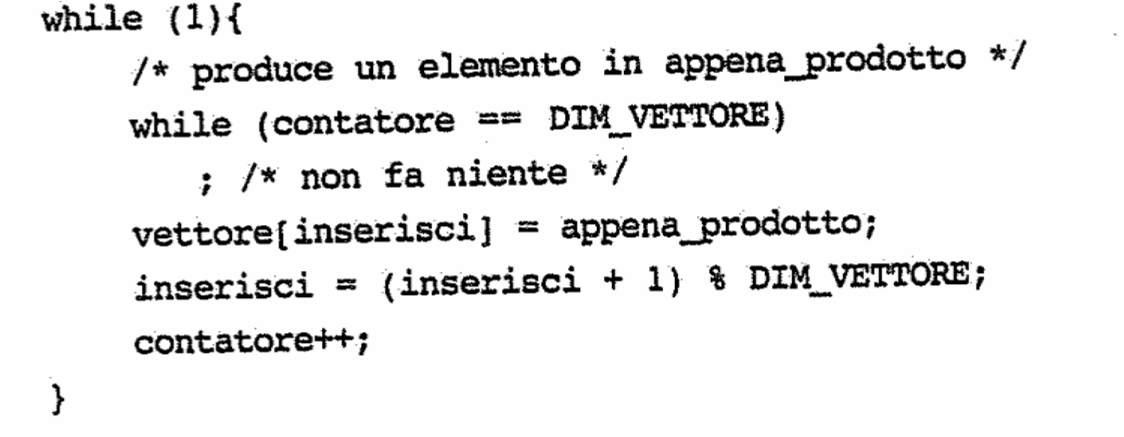
# CAPITOLO 6: SINCRONIZZAZIONE DEI PROCESSI

## 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.

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.



Le procedure del produttore e del consumatore possono non funzionare altrettanto correttamente se si eseguono in modo concorrente. 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 occorre assicurare che un solo processo alla volta possa modificare la variabile contatore. Questa condizione richiede una forma di sincronizzazione dei processi.

## 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.

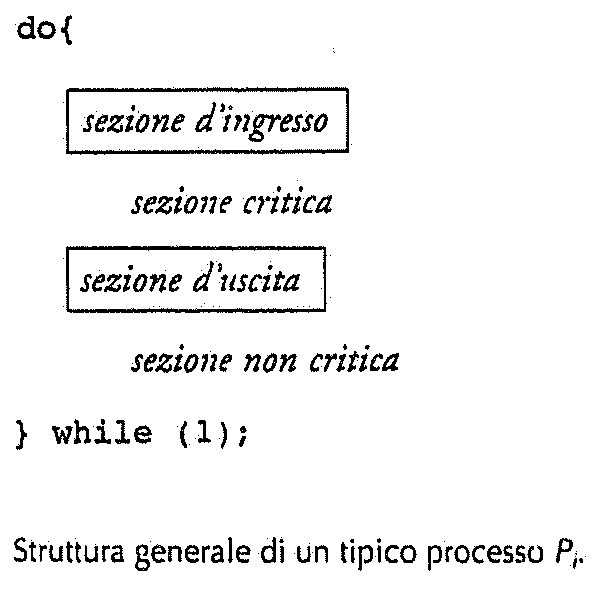
- Un kernel senza diritto di prelazione non consente di applicare la prelazione a un processo attivo

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

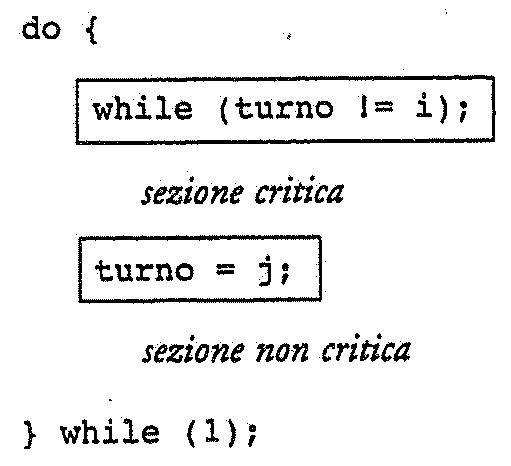
In sostanza, i kernel senza diritto di prelazione sono immuni dai problemi legati all’ordine degli accessi alle strutture dati del kernel, visto che un solo processo per volta impegna il kernel. Altrettanto non si può dire dei kernel con diritto di prelazione, motivo per cui bisogna avere cura, nella progettazione, di mantenerli al riparo dai problemi insiti nell’ordine degli accessi.

## 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.

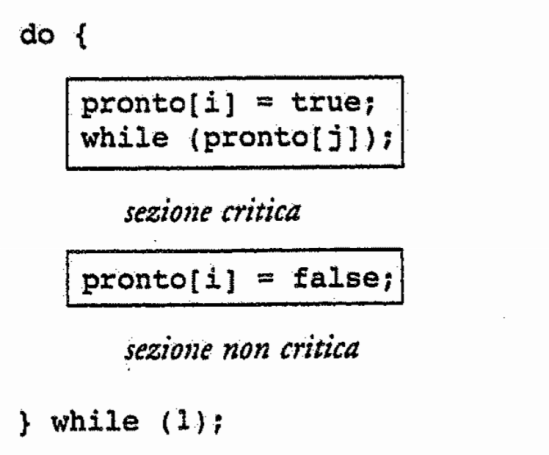


## Algoritmo 1 (non presente sull'ottava edizione)



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)



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 -> 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 ;

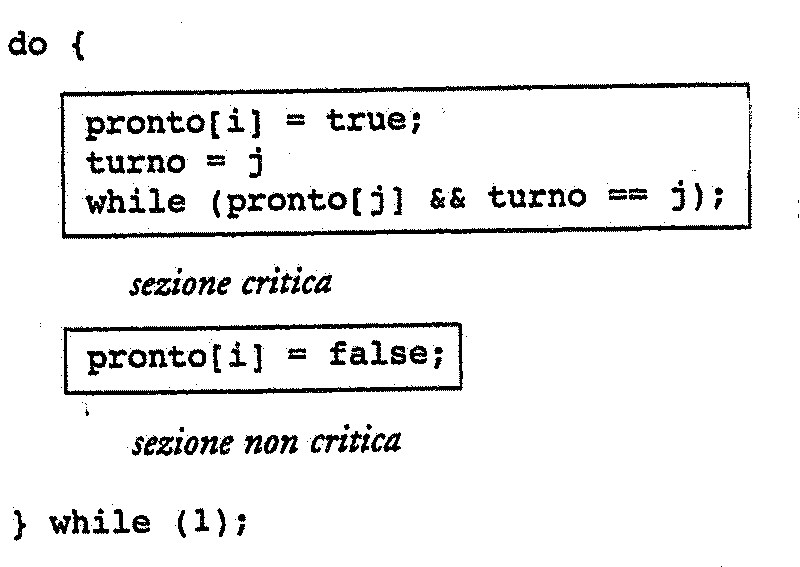
boolean flag [ 2 ] ;

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]

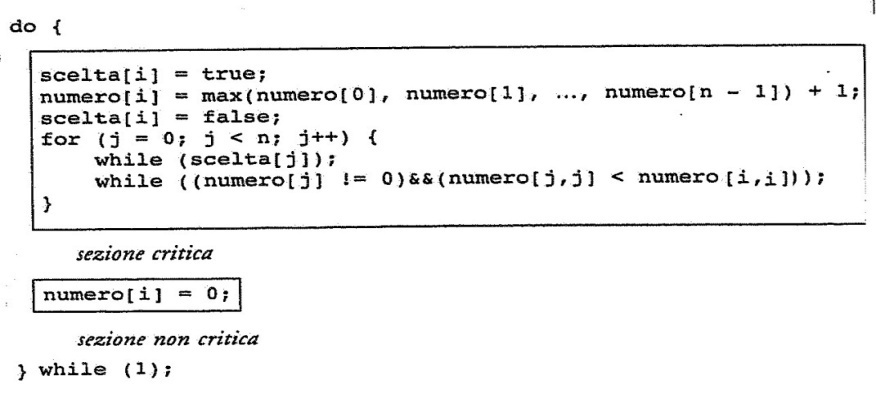


*per ulteriori sul codice e altre info : https://it.wikipedia.org/wiki/Algoritmo\_di\_Peterson*

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.

Poiché tuttavia Pi non modifica il valore della variabile turno durante l’esecuzione dell’istruzione while, Pi entrerà nella sezione 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)



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 nome minore.

Boolean scelta [n]

Int numero

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

## 3.HARDWARE PER LA SINCRONIZZAZIONE

In generale si può affermare che qualunque soluzione al problema 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.

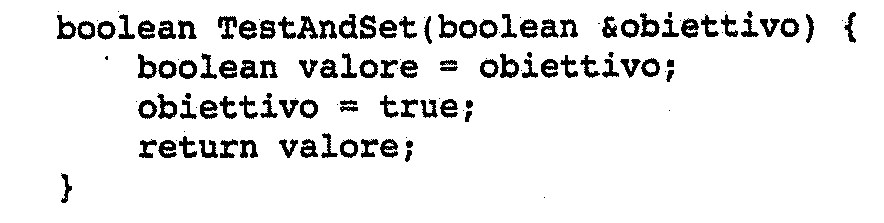
Sfortunatamente questa soluzione non è sempre praticabile; la disabilitazione delle interruzioni nei sistemi multiprocessore può comportare sprechi di tempo dovuti alla necessità di trasmettere la richiesta di disabilitazione delle interruzioni a tutte le unita d’elaborazione.

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

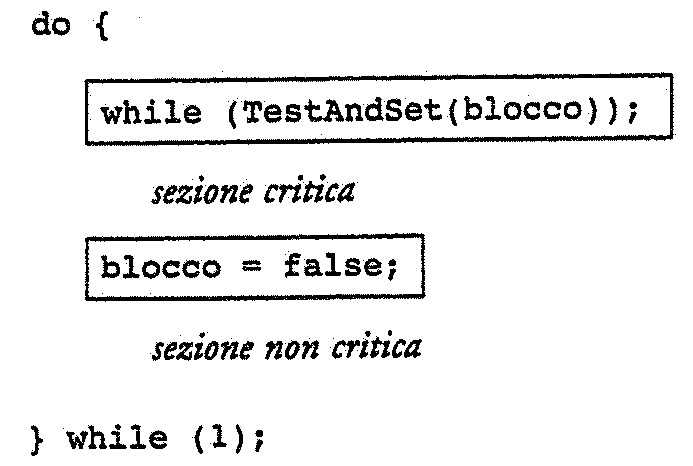
Per questo motivo molte architetture offrono particolari istruzioni che permettono di controllare e modificare il contenuto di una parola di memoria oppure di scambiare il contenuto di due parole di memoria in **modo atomico**( come un'unità non interrompibile).

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

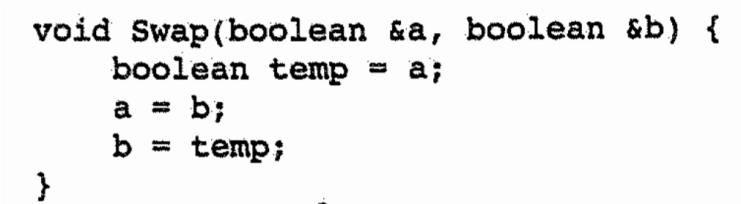
L’istruzione **TestAndSet( )** è definito nell'immagine successiva. Questa istruzione è eseguita atomicamente, cioè come un’unita non soggetta a interruzioni; quindi, se si eseguono contemporaneamente due istruzioni TestAndSet( ), ciascuna in un’unita d’elaborazione diversa, queste vengono eseguite in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



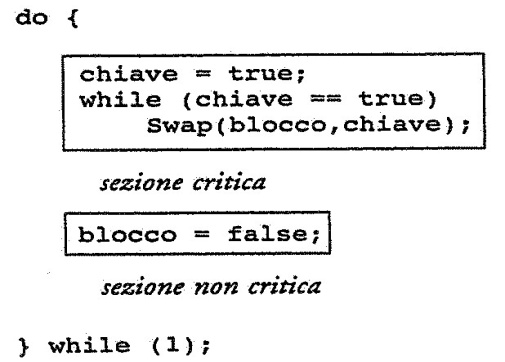
**Realizzazione MUTUA ESCLUSIONE con TestAndSet** : se si dispone dell’istruzione TestAndSet( ), si può realizzare la mutua esclusione dichiarando una variabile booleana globale lock, inizializzata a false.



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



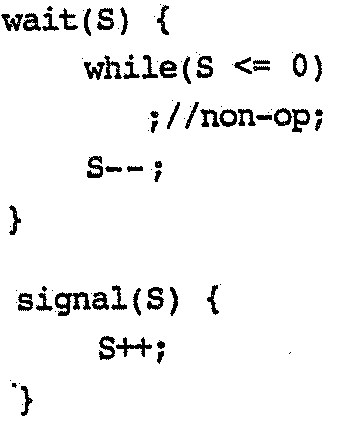
**Realizzazione MUTUA ESCLUSIONE con SWAP**



## 4.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 dei **semafori**.

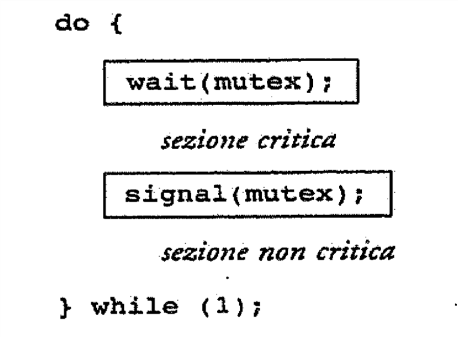
Un semaforo S è una variabile intera che si può accedere, escludendo l’inizializzazione, solo tramite due operazioni atomiche: **wait** e **signal**.



Tutte le modifiche del semaforo sono contenute nelle operazioni **wait** e **signal**.

## 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.



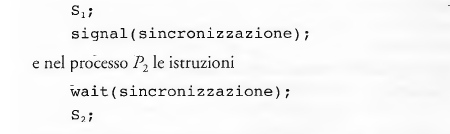
-I semafori contatore trovano applicazione nel controllo dell’accesso a una data risorsa presente in un numero finito di esemplari. Il semaforo è inizialmente impostato al numero di esemplari disponibili. I processi che desiderino utilizzare un esemplare della risorsa invocano **wait ( )** sul semaforo, 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.

## Stallo e attesa indefinita

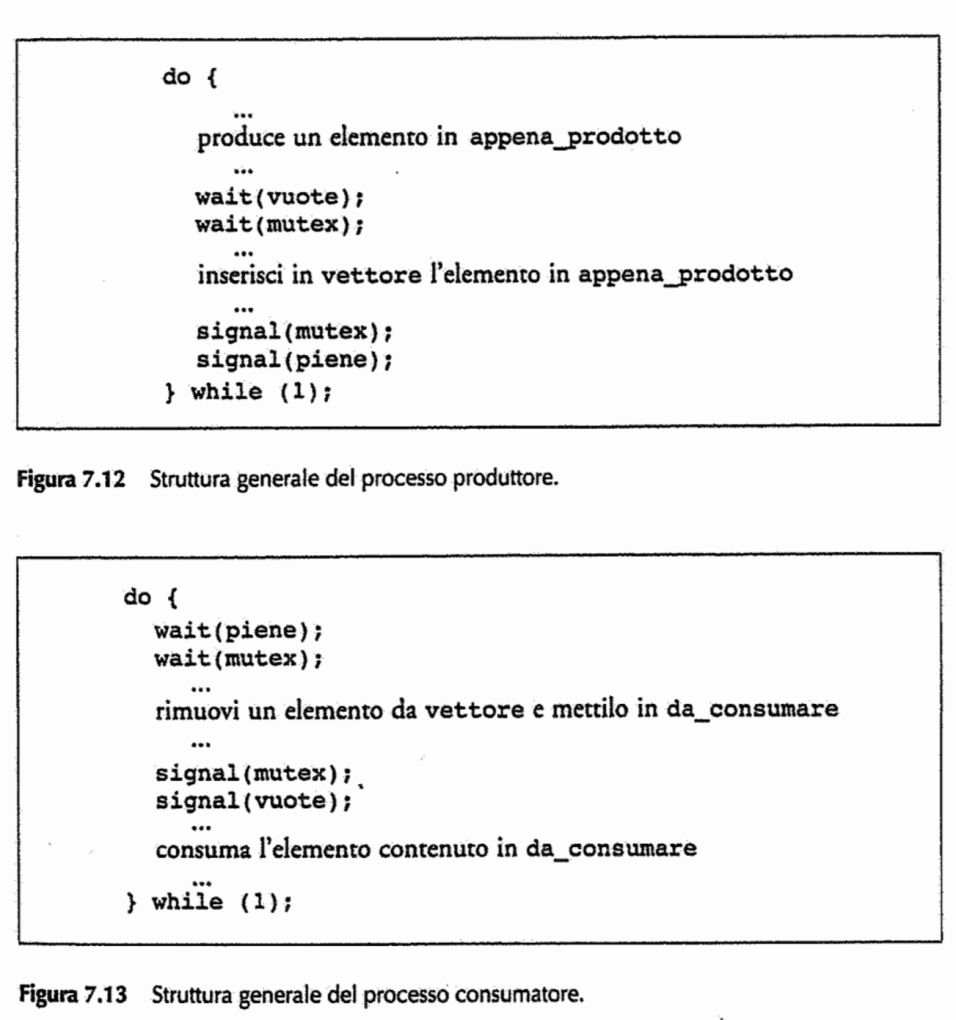
La realizzazione di un semaforo con coda d’attesa può condurre a situazioni in cui ciascun processo di un insieme di processi attende indefinitamente un evento che può essere causato solo da uno dei processi dello stesso insieme. Quando si verifica una situazione di questo tipo ci troviamo in una situazione 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**.

## 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.



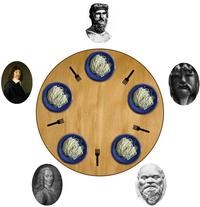
## Problema dei lettori-scrittori

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. Per impedire tale problema è necessario che gli scrittori abbiano accesso esclusivo alla base dati condivisa. Vi possono essere 2 soluzioni a tale problema: il primo che nessun lettore attenda a meno che uno scrittore abbia gia la risorsa, la seconda che lo scrittore non attenda.

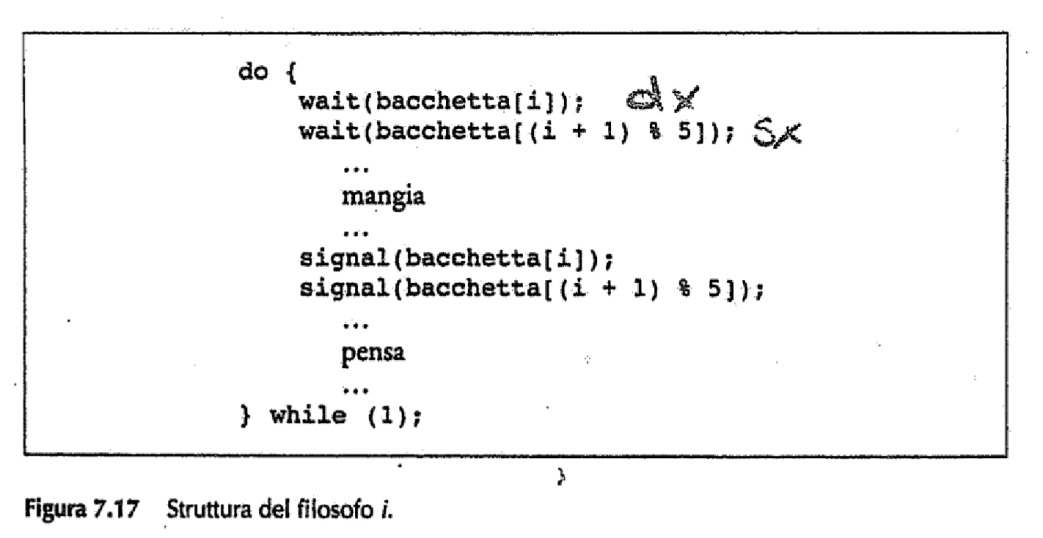
Entrambe le soluzioni presenta il problema della starvation sullo scrittore nel primo caso e nel lettore nel secondo. La soluzione prevede l’utilizzo di un semaforo.

## Problema dei 5 filosofi

Cinque filosofi siedono ad una tavola rotonda con un piatto di spaghetti davanti, una forchetta a destra e una forchetta a sinistra (bastoncini cinesi secondo un'altra versione). Ci sono dunque cinque filosofi, cinque piatti di spaghetti e cinque forchette.



Si immagini che la vita di un filosofo consista di periodi alterni di mangiare e pensare, e che ciascun filosofo abbia bisogno di due forchette per mangiare, ma che le forchette vengano prese una per volta. Dopo essere riuscito a prendere due forchette il filosofo mangia per un po', poi lascia le forchette e ricomincia a pensare. Il problema consiste nello sviluppo di un algoritmo che impedisca lo stallo (deadlock) o la morte d'inedia (starvation). Il deadlock può verificarsi se ciascuno dei filosofi tiene in mano una forchetta senza mai riuscire a prendere l'altra. Il filosofo F1 aspetta di prendere la forchetta che ha in mano il filosofo F2, che aspetta la forchetta che ha in mano il filosofo F3, e così via in un circolo vizioso. La situazione di starvation può verificarsi indipendentemente dal deadlock se uno dei filosofi non riesce mai a prendere entrambe le forchette. La presa di forchette è analoga al blocco di risorse limitate nella programmazione reale, situazione nota con il nome di *concorrenza*. Una semplice soluzione è quella di rappresentare ogni bacchetta con un semaforo. Questa soluzione garantisce che 2 vicini non mangino contemporaneamente.



*facoltativo : Monitor , ... fino alla fine*