**Sincronizzazione tra processi**

Processi indipendenti possono avanzare concorrentemente senza alcun vincolo di ordinamento reciproco.

In realtà molti processi condividono risorse e informazioni funzionali per gestire la loro condivisione servono meccanismi di sincronizzazione di accesso.

**Esempio di race condition**:

Siano A e B due processi che condividono la variabile X inizializzata al valore 10

– Il processo A deve incrementare X di 2 unità

– Il processo B deve decrementare X di 4 unità

A e B leggono concorrentemente il valore di X

– Il processo A scrive in XA il proprio risultato (12)

– Il processo B scrive in XB il proprio risultato (6)

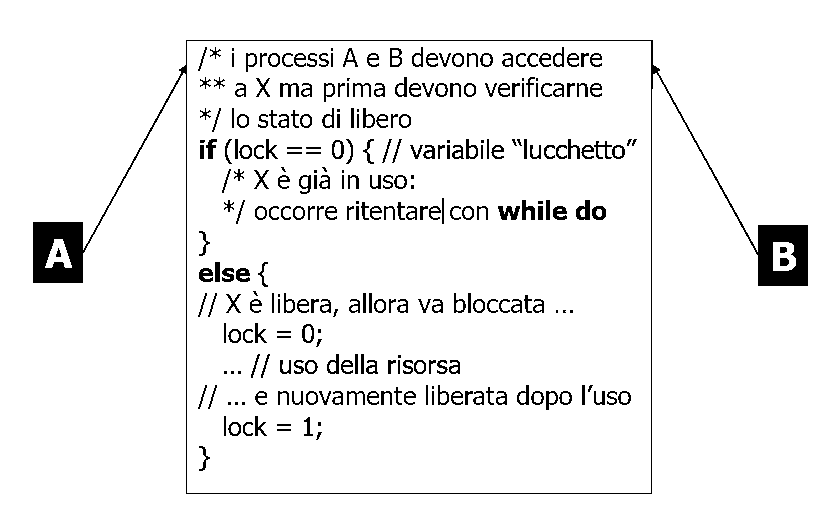
Il valore finale in X è l’ultimo tra XA e XB a essere scritto!

Il valore atteso in X invece era 8 (ottenibile solo con sequenze (A;B) o (B;A) indivise di lettura e scrittura)

La modalità di accesso indivisa a una variabile condivisa viene detta “**in mutua esclusione**”. L’accesso consentito a un processo inibisce quello simultaneo di qualunque altro processo utente fino al rilascio della risorsa.

Si utilizza una variabile logica “lucchetto” (**lock**) che indica se la variabile condivisa è al momento in uso a un altro processo (detta anche “struttura mutex” (**mutual exclusion**))

**Possibile soluzione?**

La soluzione appena vista è inadeguata:

* ciascuno dei due processi può essere prerilasciato dopo aver letto la variabile lock ma prima di esser riuscito a modificarla (questa situazione è detta **race condition** e può generare

pesanti inconsistenze)

* Inoltre l’algoritmo mostrato richiede attesa attiva che causa spreco di tempo di CPU a scapito di altre attività a maggior valore aggiunto. La tecnica di sincronizzazione tramite attesa attiva viene

detta **busy wait** (attesa con verifica continua che una variabile cambi valore).

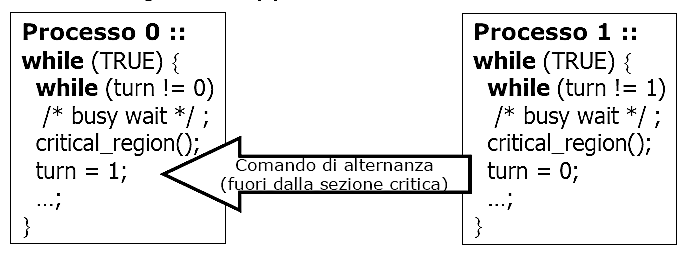
**Sincronizzazione ammissibile**

Una soluzione al problema della sincronizzazione di processi è ammissibile se soddisfa le seguenti 4 condizioni:

1. Garantire accesso esclusivo
2. Garantire attesa finita
3. Non fare assunzioni sull’ambiente di esecuzione
4. Non subire condizionamenti dai processi esterni alla sezione critica

**Soluzioni “esotiche”:**

**1)** Mutua esclusione con variabili condivise tramite alternanza stretta tra coppie di processi (non ha bisogno di supporto dalla sua macchina virtuale)

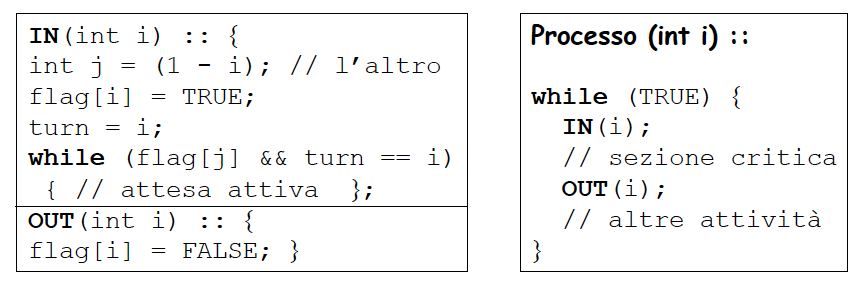


Tre gravi difetti 🡪 soluzione non ammissibile!

* Uso di attesa attiva (busy wait)
* Condizionamento esterno alla sezione critica
* Rischio di race condition sulla variabile di controllo

**2)** Proposta da G. L. Peterson

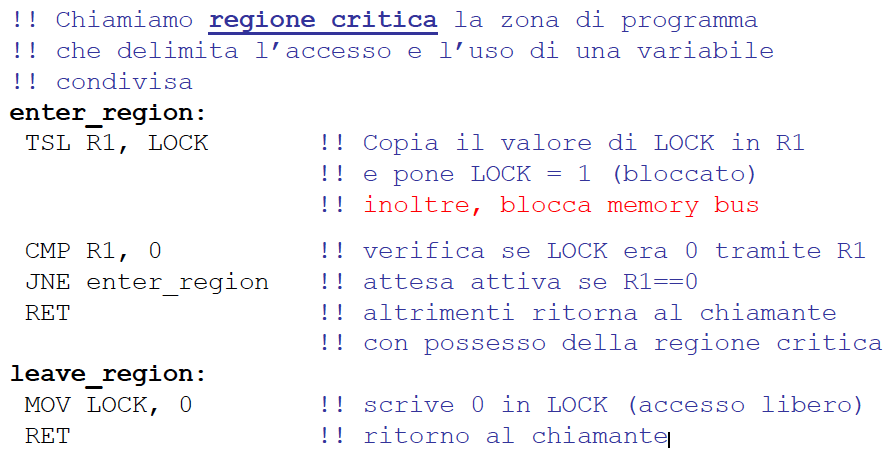
Migliore (!) della precedente ma ingenua rispetto al funzionamento dei processori di oggi. Applica solo a coppie di processi



**3)** La condizione di uscita in IN() impone che il processo i resti in attesa attiva fin quando il processo j non abbia invocato OUT(). Per cui flag[j] = FALSE. Si ha attesa attiva quando non vi sia coerenza tra la richiesta di turn e lo stato dell’altro processo.

Su flag[] non vi può essere scrittura simultanea che si ha invece su turn. La condizione di uscita è però espressa in modo da evitare il rischio di race condition. (Non viene decisa solo dal valore assunto da turn!)

**Tecniche complementari e/o alternative**

* Disabilitazione delle interruzioni: previene il prerilascio dovuto all’esaurimento del quanto di tempo e/o la promozione di processi a più elevata priorità. Può essere inaccettabile per sistemi soggetti a interruzioni frequenti. Sconsigliabile lasciare controllo interrupt a utenti (e se non li riattivano?). Inutile con due processori.
* Supporto hardware diretto: **Test-and-Set-Lock**. Cambiare atomicamente valore alla variabile di lock se questa segnala “libero”. Evita situazioni di race condition ma comporta sempre attesa attiva.

Sia la soluzione di Peterson che quella TSL sono corrette ma hanno il difetto di eseguire busy wait.

**Inversion priority**:

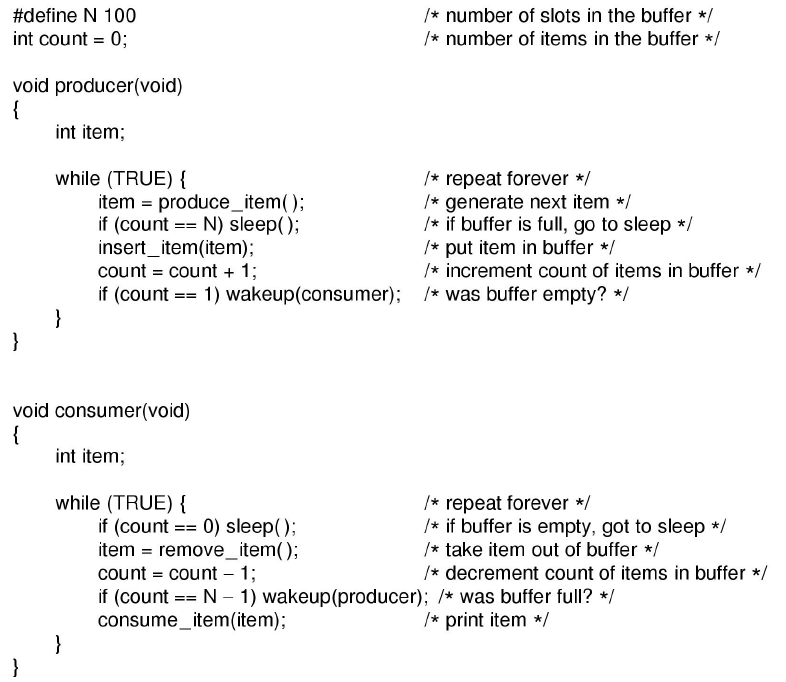
Consideriamo due processi: H (ad alta priorità) e L (a bassa priorità).

Supponiamo che H prerilasci il processore per eseguire I/O.

Supponiamo che H concluda le operazioni di I/O mentre L si trova nella sua sezione critica.

H rimarrà bloccato in busy wating perché L non avrà più modo di concludere la sezione critica.

**Problema produttore-consumatore**



Soluzione sleep & wakeup: Wake up non vengono memorizzate. Se non c’è una sleep in attesa, le wakeup vengono perse.

Rischio race condition su variabile count:

* Il buffer è vuoto e il consumer ha appena letto che count = 0
* Prima che il consumer istanzi la sleep, lo scheduler decide di fermare il consumer e di eseguire il producer
* Il producer produce un elemento e imposta count = 1
* Siccome count = 1 il producer emette wakeup (che nessuno ascolta)
* A un certo punto lo scheduler deciderà di eseguire di nuovo il consumer il quale istanzia finalmente la sleep, ma siccome count è già stato riportato a 1 e la corrispondente wakeup è già stata emessa, il consumer non verrà più risvegliato

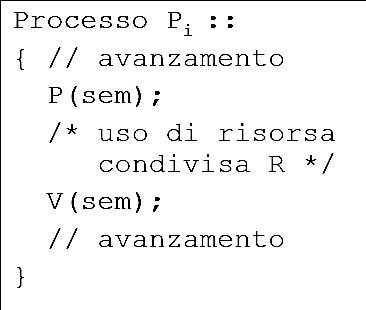
Soluzione mediante semaforo: richiede accesso indiviso (atomico) alla variabile di controllo detta semaforo. Per questo la struttura semaforo si appoggia sopra una macchina virtuale meno potente che fornisce una modalità di accesso indiviso più primitiva.

* Semaforo binario (contatore Booleano che vale 0 o 1)
* Semaforo contatore (consente tanti accessi simultanei quanto il valore iniziale del contatore)

La richiesta di accesso P (down) decrementa il contatore se questo non è già 0, altrimenti accoda il chiamante.

L’avviso di rilascio V (up) incrementa di 1 il contatore e chiede al dispatcher di porre in stato di “pronto” il primo processo in coda sul semaforo

L’uso di una risorsa condivisa R è racchiuso entro le chiamate di P e V sul semaforo associato a R

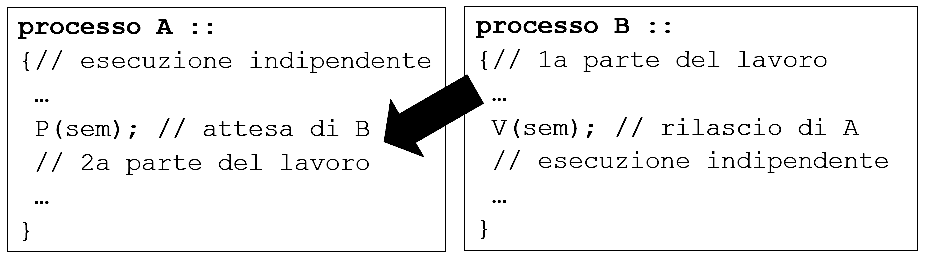


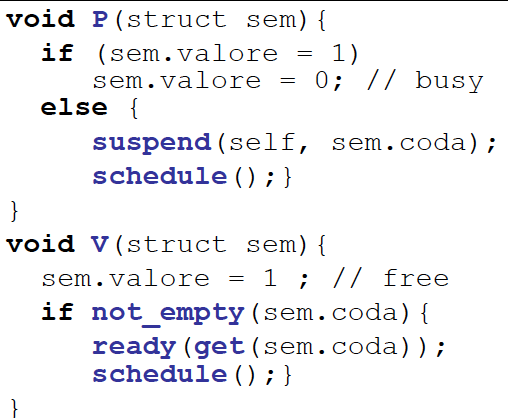
P(sem) viene invocata per richiedere accesso a una risorsa condivisa R

• Quale R tra tutte?

V(sem) viene invocata per rilasciare la risorsa

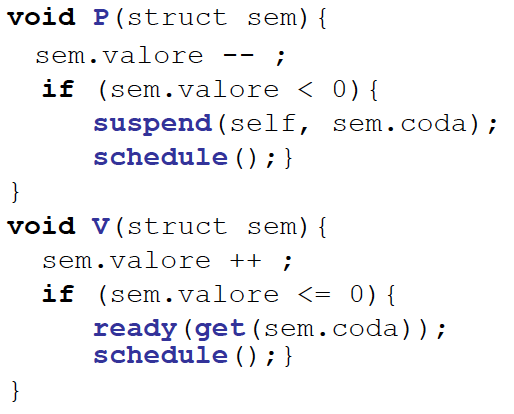
Mediante uso intelligente di semafori binari più processi possono anche coordinare l’esecuzione di attività collaborative. Esempio con semaforo inizialmente bloccato:





Il **semaforo binario** (mutex) è una struttura composta da un campo valore intero e da un campo coda che accoda tutti i PCB dei processi in attesa sul semaforo - PCB = Process Control Block

L’accesso al campo valore deve essere atomico!



Il **semaforo contatore** ha la stessa struttura del mutex ma usa una logica diversa per il campo valore:

– (Valore > 0) denota disponibilità non esaurita

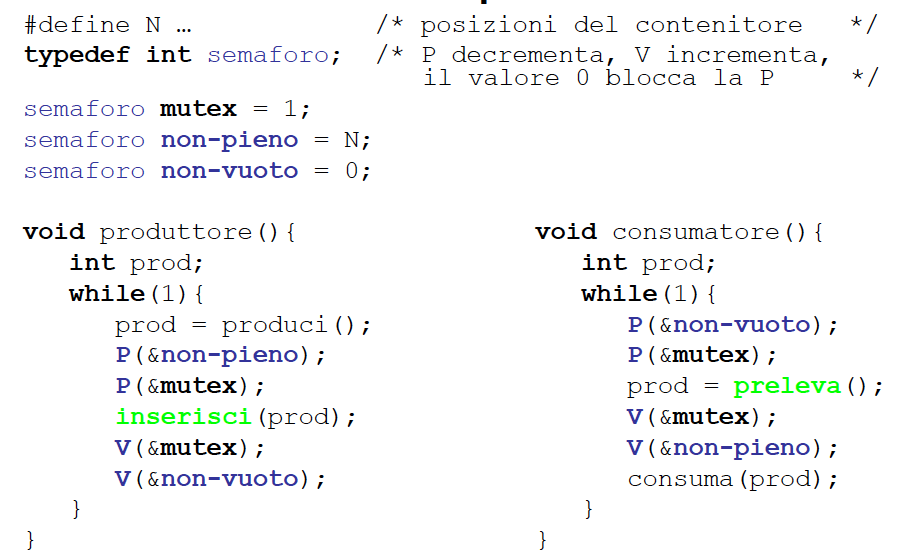
– (Valore < 0) denota richieste pendenti

Il valore iniziale denota la capacità massima della risorsa

**Monitor**

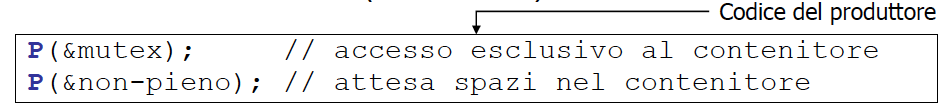
L’uso di semafori a livello di programma è ostico e rischioso. Il posizionamento improprio delle P può causare situazioni di blocco infinito (**deadlock**) o anche esecuzioni erronee di difficile verifica (race condition). È indesiderabile lasciare all’utente il pieno controllo di strutture così delicate.

**Esempio 1**



Il corretto ordinamento di P e V è critico!

Un diverso ordinamento delle P nel codice utente di esempio 1 potrebbe causare situazioni di blocco infinito (deadlock).



In questo modo il consumatore non può più accedere al contenitore per prelevarne prodotti, facendo spazio per l’inserzione di nuovi → stallo = deadlock

Linguaggi evoluti di alto livello (e.g.: Concurrent Pascal, Ada, Java) offrono strutture esplicite di controllo delle regioni critiche, originariamente dette monitor (Hoare, ’74; Brinch-Hansen, ’75).

Il monitor definisce la regione critica

Il compilatore (non il programmatore!) inserisce il codice necessario al controllo degli accessi.

Un monitor è un aggregato di sottoprogrammi, variabili e strutture dati.

Solo i sottoprogrammi del monitor possono accederne le variabili interne.

Solo un processo alla volta può essere attivo entro il monitor:

* proprietà garantita dai meccanismi del supporto a tempo di esecuzione del linguaggio di programmazione concorrente (funzionalmente molto simile al kernel del sistema operativo),
* il codice necessario è inserito dal compilatore direttamente nel programma eseguibile.

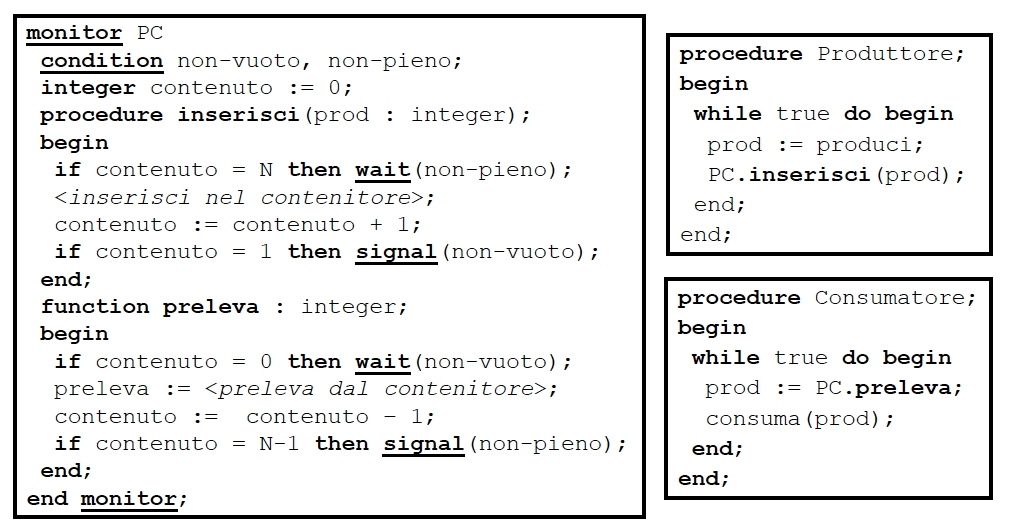
La garanzia di mutua esclusione da sola può non bastare per consentire sincronizzazione intelligente.

Due procedure operanti su variabili speciali (non contatori!) dette **condition variables**, consentono di modellare condizioni logiche specifiche del problema:

* Wait(<cond>) // forza l’attesa del chiamante
* Signal(<cond>) // risveglia il processo in attesa

Il segnale di risveglio non ha memoria, va perso se nessuno lo attende.

**Esempio 2**



La primitiva **Wait** permette di bloccare il chiamante qualora le condizioni logiche della risorsa non

consentano l’esecuzione del servizio:

* contenitore pieno per il produttore,
* contenitore vuoto per il consumatore.

La primitiva **Signal** notifica il verificarsi della condizione attesa al (primo) processo bloccato, risvegliandolo: il processo risvegliato compete con il chiamante della Signal per il possesso della CPU

Wait e Signal sono invocate in mutua esclusione, non si può verificare race condition (diverso dunque da sleep e wakeup visti in precedenza)

Java offre un costrutto simile al monitor tramite classi con metodi **synchronized** ma senza condition variable.

Le primitive wait() e notify() invocate all’interno di metodi synchronized evitano il verificarsi di race condition. In realtà il metodo wait() può venire interrotto, e l’interruzione va trattata come eccezione!

**Esempio 3**



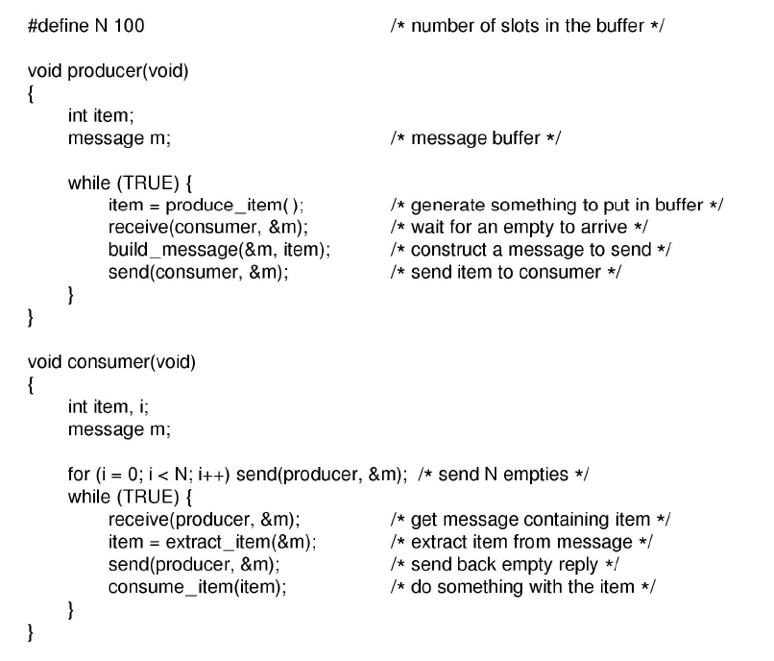
In ambiente locale si hanno 3 possibilità per supportare sincronizzazione tra processi:

1. Linguaggi concorrenti con supporto esplicito per strutture monitor (alto livello)

Linguaggi sequenziali senza supporto per monitor o semafori

1. Uso di semafori tramite strutture primitive del sistema operativo e chiamate di sistema (basso livello)
2. Realizzazione di semafori primitivi, in linguaggio assembler, senza supporto dal sistema operativo (bassissimo livello)

Monitor e semafori non sono utilizzabili per realizzare scambio di informazione tra elaboratori. Perché?



**Barriere**

Per sincronizzare gruppi di processi (attività cooperative suddivise in fasi ordinate)

La barriera blocca tutti i processi che la raggiungono fino all’arrivo dell’ultimo (si applica indistintamente ad ambiente locale e distribuito).

Non comporta scambio di messaggi esplicito. L’avvenuta sincronizzazione dice implicitamente ai processi del gruppo che tutti hanno raggiunto un dato punto della loro esecuzione.

**Problemi classici di sincronizzazione**

Metodo per valutare l’efficacia e l’eleganza di modelli e meccanismi per la sincronizzazione tra processi:

* Filosofi a cena : accesso esclusivo a risorse limitate
* Lettori e scrittori : accessi concorrenti a basi di dati
* Barbiere che dorme : prevenzione di race condition
* (Produttore e Consumatore)

Problemi pensati per rappresentare tipiche situazioni di rischio:

– Stallo con blocco (deadlock)

– Stallo senza blocco (starvation)

– Esecuzioni non predicibili (race condition)

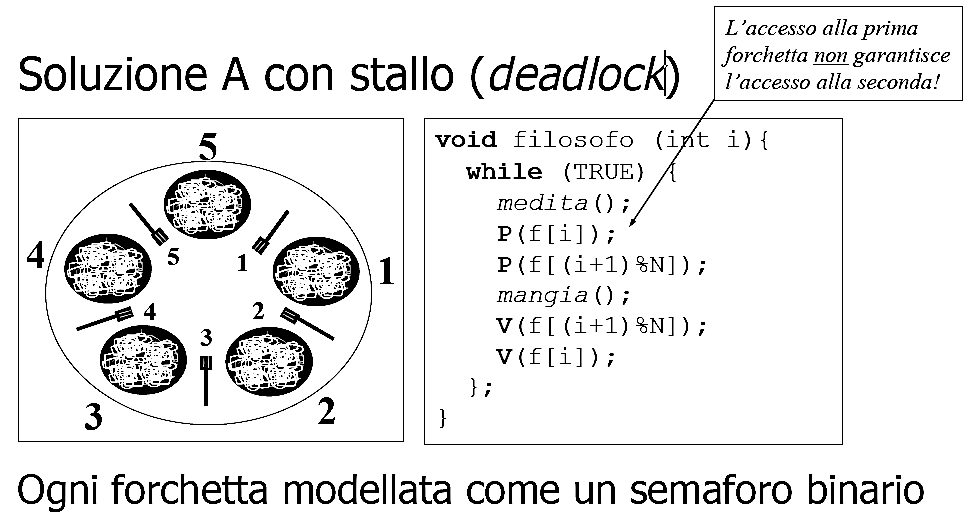
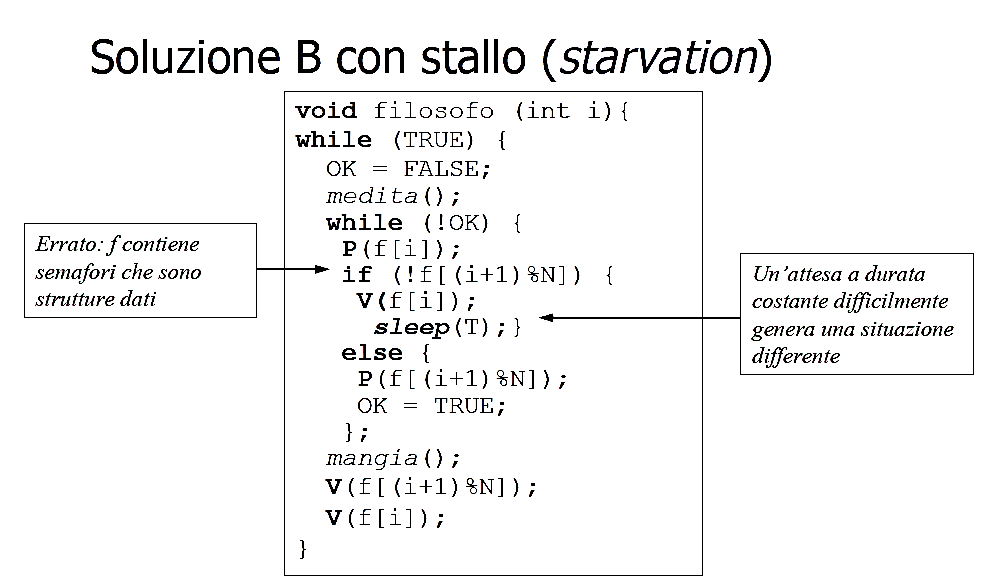
**Filosofi a cena**

N filosofi sono seduti a un tavolo circolare. Ciascuno ha davanti a se 1 piatto e 1 posata alla

propria destra. Ciascun filosofo necessita di 2 posate per mangiare. L'attività di ciascun filosofo alterna pasti a momenti di riflessione.

Il problema ammette diverse soluzioni:

1. Utilizzare in soluzione A un semaforo a mutua esclusione per incapsulare gli accessi a entrambe le forchette (Funzionamento garantito)
2. In soluzione B, ciascun processo potrebbe attendere un tempo casuale invece che fisso (funzionamento non garantito)
3. Algoritmi sofisticati, con maggiore informazione sullo stato di progresso del vicino e maggior coordinamento delle attività (funzionamento garantito)



**Stallo**

Condizioni necessarie e sufficienti

* Accesso esclusivo a risorsa condivisa
* Accumulo di risorse: i processi possono accumulare nuove risorse senza doverne rilasciare altre
* Inibizione di prerilascio: il possesso di una risorsa deve essere rilasciato volontariamente
* Condizione di attesa circolare: un processo attende una risorsa in possesso del successivo processo in catena

Stallo: prevenzione

Almeno tre strategie per affrontare lo stallo:

* Prevenzione: impedire almeno una delle condizioni precedenti
* Riconoscimento e recupero: ammettere che lo stallo si possa verificare, essere in grado di riconoscerlo, possedere una procedura di recupero (sblocco)
* Indifferenza: considerare trascurabile la probabilità di stallo e non prendere alcuna precauzione contro di esso. Che succede se esso si verifica?

Bisogna impedire il verificarsi di almeno una delle condizioni necessarie e sufficienti (si può fare staticamente (prima di eseguire) oppure a tempo d’esecuzione)

1. Accesso esclusivo alla risorsa: però alcune risorse non consentono alternative
2. Accumulo di risorse: però molti problemi richiedono l’uso simultaneo di più risorse
3. Inibizione del prerilascio: però alcune risorse non consentono di farlo
4. Attesa circolare: difficile da rilevare e complessa da evitare o sciogliere

Prevenzione sulle richieste di accesso:

* A tempo d’esecuzione: ogni richiesta di accesso si verifica se questa possa portare allo stallo. In caso affermativo non è però chiaro cosa convenga fare. La verifica a ogni richiesta è un onere molto pesante
* Prima dell’esecuzione: all’avvio di ogni processo si verifica quali risorse essi dovranno utilizzare così da ordinarne l’attività in maniera conveniente.

Stallo: riconoscimento

• **A tempo d’esecuzione**: assai oneroso. Occorre bloccare periodicamente l’avanzamento del sistema per analizzare lo stato di tutti i processi e verificare se quelli in attesa costituiscono una lista circolare chiusa. Lo sblocco di uno stallo comporta la terminazione forzata di uno dei processi in attesa. Il rilascio delle risorse liberate sblocca la catena di dipendenza circolare

• **Staticamente**: può essere un problema non risolvibile!