•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

• 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

Chiamiamo **regione critica** la zona di programma che delimita l'accesso e l'uso di una variabile condivisa.

• Problema di 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

Il problema produttore- consumatore *Soluzione sleep & wakeup*

```
#define N100
int count =0;

void producer (void)

{
int item;
while (TRUE){
        item=produce_item();                    produci l' oggetto
        if (count==N) sleep();                  se buffer pieno,allora dormi
        insert_item(item);                      metti l'oggetto nel buffer
        count=count+1;                          aumenta di +1 il contatore degli oggetti
        if (count==1) wakeup(consumer);         se c'è almeno un oggetto nel buffer sveglia il consumatore
        }
}

void consumer (void)

{
int item
while(TRUE){
        if (count==0) sleep();                  se il buffer è vuoto dormi
        item=remove_item();                     altrimenti rimuovi un oggetto dal buffer
        count=count-1;                          decrementa il contatore degi oggetti del buffer
        if (count== N-1) wakeup(producer);      se il buffer ha 99 elementi,sveglia il produttore
        consume_item(item);                     stampa l' oggetto
        }
}
```

Wake up non vengono memorizzate

– Se non c'è una sleep in attesa, le wakeup vengono perse

• Rischio race condition su variabile count ESEMPIO

– 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

Il problema produttore- consumatore *Soluzione mediante **semaforo***

Richiede accesso indiviso (atomico) alla variabile di controllo detta semaforo

1) Semaforo binario (contatore Booleano che vale 0 o 1) detto anche mutex è una struttura composta da un campo valore intero e da un campo coda che accoda tutti i PCB (ProcessControlBlock) dei processi in attesa sul semaforo.L'accesso al campo valore deve essere atomico.
2) 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.Ha la stessa struttura del mutex ma usa una logica diversa per il campo valore (>0 disp non esaurita <0 significa che ci sono richieste pendenti).*Il valore iniziale denota la capacità massima della risorsa.*

Rischio sull uso dei semafori:
– In generale *l'uso di semafori a livello di programma è molto rischioso*.Infatti il posizionamento improprio delle P (down) può causare blocco infinito (deadlock) o casi di race condition.Non è quindi conveniente lasciare pieno controllo all utente.

– Alcuni linguaggi di alto livello (cuncurrent pascal,ada,java) offrono strutture Esplicite di controllo delle regioni critiche,chiamate monitor.Il monitor definisce quindi la regione critica ed è il compilatore stesso a inserire 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

La garanzia di mutua esclusione da sola può non bastare per consentire sincronizzazione intelligente
Intervengono quindi due procedure operanti su variabili speciali dette condition variables che consentono di modellare le condizioni logiche specifiche del problema:

Wait (condizione) forza l attesa del chiamante qualora le condizioni logiche della risorsa non consentano l'esecuzione del servizio (contenitore pieno per il produttore,vuoto per il consumatore)
Signal (consizione) risveglia il primo processo in attesa (non ha memoria,e va quindi perso se nessono lo attende)

Le due procedure sono invocate in mutua esclusione (non si può verificare alcun problema di race condition)

In ambiente locale si hanno tre possibilità per supportare la sincronizzazione tra processi

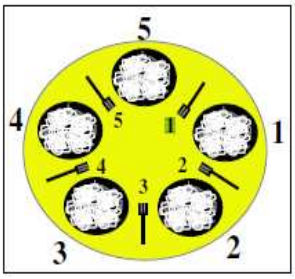
-Linguaggio con supporto esplicito per strutture di tipo monitor (alto livello)
-Linguaggi sequenzali con uso di semafori tramite strutture del SO e chiamate di sistema (basso livello)
-Linguaggi sequenziali con uso di semafori primitivi,realizzati in assembler senza supporto al SO (bassissimo livello)

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

Soluzione A con stallo (*deadlock*)

L'accesso alla prima forchetta non garantisce l'accesso alla seconda!




```
void filosofo (int i){
  while (TRUE) {
    medita();
    P(f[i]);
    P(f[(i+1)%N]);
    mangia();
    V(f[(i+1)%N]);
    V(f[i]);
  };
}
```

Ogni forchetta modellata come un semaforo binario

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

Condizioni necessarie e sufficienti per avere uno stallo
-***Accesso esclusivo*** a una 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

Strategie POSSIBILI per affrontare uno Stallo:

-***Prevenzione*****:**impedire almeno una delle condizioni precedenti

A tempo d'esecuzione
-Ad ogni richiesta di accesso si verifica se questa possa portare allo stallo
(oneroso è in caso affermativo non è ben chiaro come intervenire)

Prima dell'esecuzione
-All avvio di ogni processo si verifica quai risorse essi dovranno utilizzare
cosi da ordinare l'attività in maniera conveniente

-***Riconoscimento e recupero***,ossia ammettere che lo stallo si possa verificare,essere in grado di riconoscerlo e possedure una procedura di sblocco

A tempo d'esecuzione
oneroso perchè bisogna bloccare periodicamente il sistema e controllare lo stato di tutti i processi verificandone che noncostituiscano una lista circolare chiusa e in caso affermativo non è chiaro cosa fare).Inoltre lo sblocco di uno stallo comporta la terminazione forzata di uno dei processi in attesa.

Staticamente :può essere un problema non risolvibile
-***Indifferenza*****:** considerare trascurabile la proabilità di stallo e non prendere alcuna precauzione

Unità metriche

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^{3}	1,000	Kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	Mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000	Yotta