ISLab Intelligent Systems Laboratory

UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO

Gestione delle risorse e problematiche connesse

Prof. Giuseppe Pirlo | Prof. Donato Impedovo | Dott. Stefano Galantucci

Architettura degli Elaboratori e Sistemi Operativi @ Corso di Laurea in Informatica

Gestione delle risorse

I processi utilizzano delle risorse (es. bus, CPU, stampanti…). Tali risorse sono gestite dal Sistema Operativo, che le governa e le amministra

Il Sistema Operativo decide in merito all'assegnazione delle risorse ai processi, disponendo l'ordine di accesso dei processi a queste e le modalità di accesso

I processi utilizzano le risorse per tramite del Sistema Operativo, che trasmette le operazioni agli strati sottostanti per conto dei processi

La gestione delle risorse può portare a diverse problematiche, le quali devono essere accuratamente attenzionate

Tali problematiche sono legate alla concorrenza e alla mutua esclusione delle risorse

Una risorsa singola può generalmente essere assegnata a un solo processo alla volta (mutua esclusione)

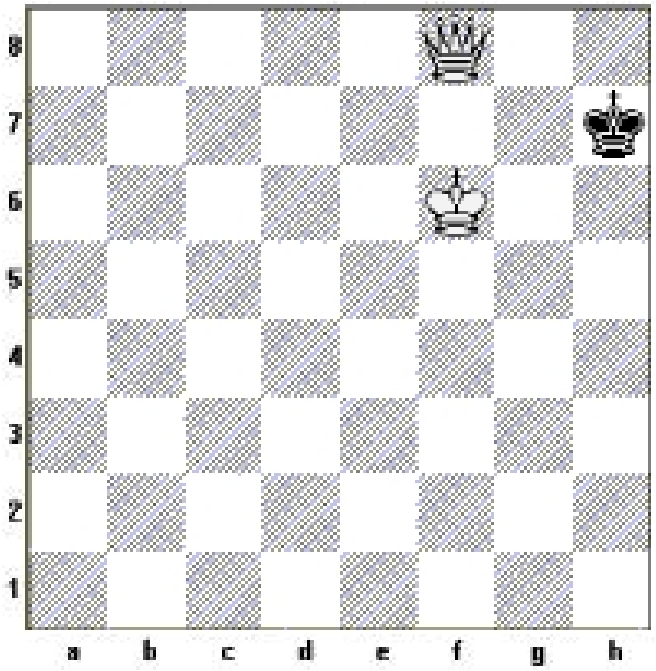
Grafo di allocazione delle risorse

Un grafo di allocazione delle risorse è un grafo che rappresenta in che modo le risorse sono assegnate ai processi e le richieste dei processi

(a) Resouce is requested

(b) Resource is held

Stallo di processi



Negli scacchi, lo stallo è la condizione in cui il Re non può effettuare mosse legittime pur non essendo sotto scacco

Lo **stallo di processi** (**deadlock**) è una delle condizioni problematiche che possono verificarsi nell'ambito della gestione delle risorse

La definizione è

Un insieme di processi è detto «in stallo» se ciascun processo nell'insieme è in attesa di un evento che solo un altro processo nello stesso insieme può generare

Generalmente l'evento è il rilascio di una risorsa detenuta da uno dei processi

Stallo di processi

Lo stallo comporta che, essendo tutti i processi in attesa di un evento, sono nello stato di blocked

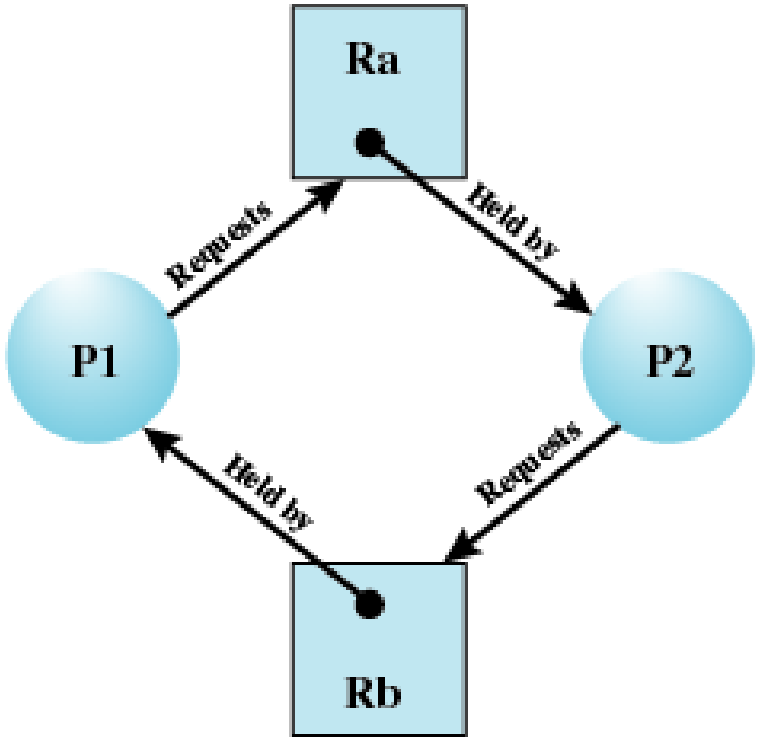
Tale situazione comporta che nessuno dei processi in stallo può

- passare in esecuzione (poiché necessita del verificarsi dell'evento)
- rilasciare risorse (in quanto non può passare in esecuzione)

Nessuno dei processi in stallo può quindi passare in running, pertanto nessuno dei processi in stallo può essere riattivato

I processi in stallo sono quindi congelati nella situazione di stallo a meno di un intervento da parte del Sistema Operativo o dell'amministratore

Stallo di processi



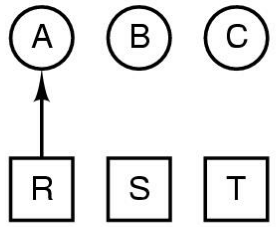
A
Request R
Request S
Release R
Release S
(a)

B
Request S
Request T
Release S
Release T
(b)

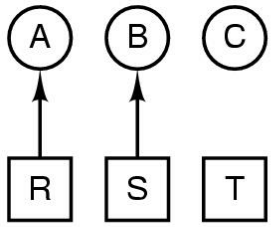
C
Request T
Request R
Release T
Release R
(c)

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

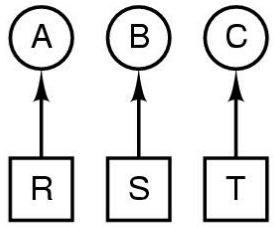
(d)



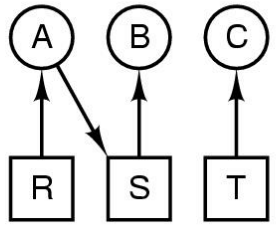
(e)



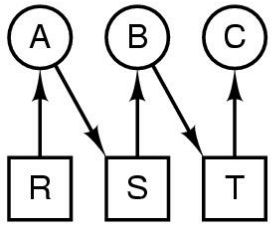
(f)



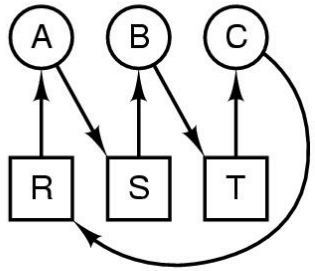
(g)



(h)



(i)



(j)

Condizioni per lo stallo

Lo stallo di processi per verificarsi necessità della congiunzione di quattro condizioni:

Mutua esclusione (oppure *Risorse seriali*)
Un solo processo alla volta può utilizzare una risorsa

Hold & Wait (oppure *Possesso e attesa*)
Un processo può mantenere il possesso delle risorse allocate mentre attende di averne altre

Assenza di prerilascio (oppure *Assenza di prelazione* o *Risorse non prerilasciabili*)
I processi non possono essere forzati a rilasciare in anticipo le risorse acquisite

Attesa circolare
Un processo aspetta una risorsa occupata da un altro processo in attesa circolare

Condizioni per lo stallo

- Mutua esclusione
- Hold & Wait
- Risorse non prerilasciabili
- Attesa circolare

Condizioni necessarie ma non sufficienti

Condizioni necessarie e sufficienti

Condizioni per lo stallo

- Mutua esclusione
- Hold & Wait
- Risorse non prerilasciabili

Condizioni derivanti direttamente dalla progettazione (in molti casi sono auspicabili)

- Attesa circolare

Evento che si può verificare e che dipende dalla particolare sequenza di richieste e rilasci

Strategie per affrontare lo stallo

Sono possibili quattro diverse strategie per affrontare il problema dello stallo di processi

1. **Ignorare il problema** (algoritmo dello struzzo)

2. **Consentire il verificarsi dello stallo, rilevarlo e risolverlo**

3. **Evitarlo con politiche di allocazione**: le tre condizioni necessarie sono permesse, un algoritmo verifica dinamicamente che una richiesta non produca una situazione di stallo

4. **Impedirlo rimuovendone le condizioni**: progettare un SO in modo che la possibilità di avere uno stallo sia esclusa a priori tramite la negazione di una delle 4 condizioni

Algoritmo dello struzzo

L'**algoritmo dello struzzo** pretende che il problema non esista

È una soluzione ragionevole se

- Lo stallo capita di rado
- Il costo per evitare lo stallo è troppo elevato

Si ha quindi un compromesso tra convenienza e correttezza

Windows e UNIX utilizzano principalmente questo approccio

Consentire lo stallo, rilevarlo e risolverlo

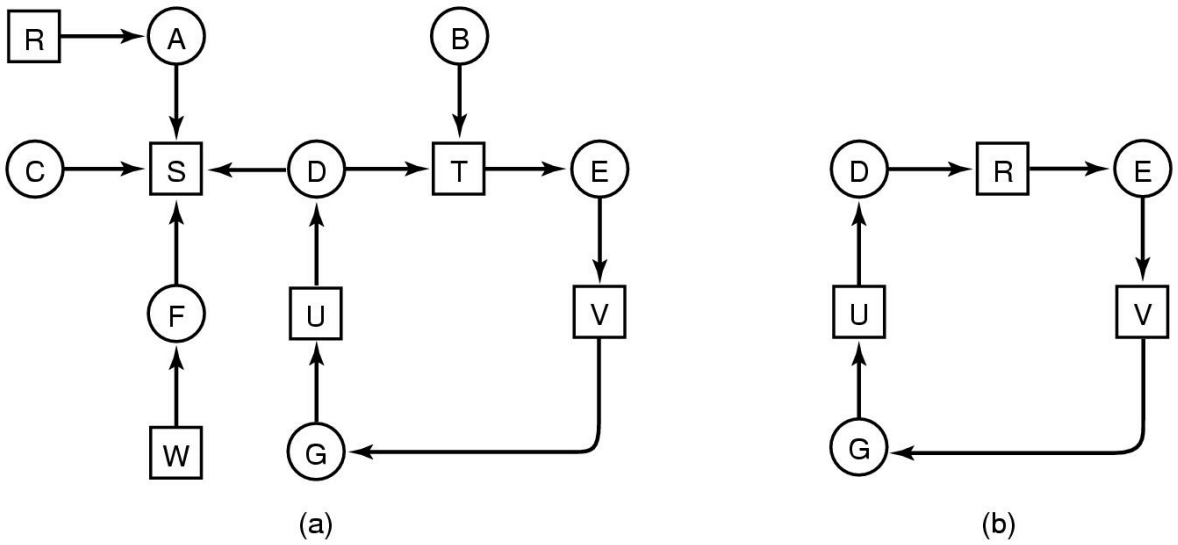
La seconda strategia possibile è quella di **consentire il verificarsi dello stallo, rilevarlo e risolverlo**

Tale strategia prevede quindi di operare normalmente e verificare periodicamente se si è verificato uno stallo

Possiamo dividerla in due fasi:

1. Rilevare lo stallo
2. Eliminare la situazione di stallo

Rilevare lo stallo



Avendo a disposizione una sola risorsa di ogni tipo si può rilevare lo stallo costruendo il grafo di assegnazione delle risorse

Un ciclo all'interno del grafo denota uno stallo

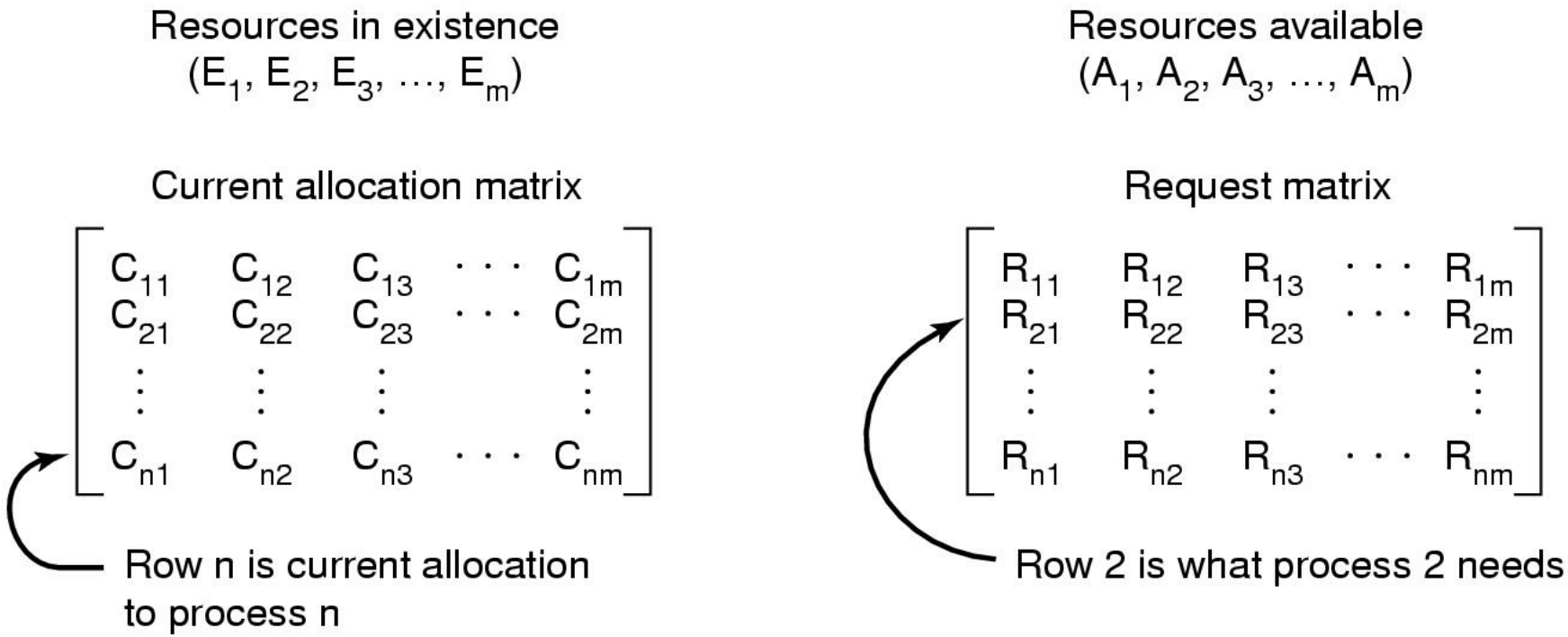
È concretamente irragionevole da implementare in quanto molto complesso

Applicabile solo nei casi di una risorsa per tipologia

Rilevare lo stallo

Serve quindi una soluzione che funzioni in generale, ovvero anche nei casi in cui si hanno più risorse per ogni tipologia

Definiamo quindi le strutture dati di cui necessitiamo:



Rilevare lo stallo

Notazioni:

- Sia R_i l'i-esima riga della matrice R
- Dati due vettori $X = [x_1 \ldots x_m]$ e $Y = [y_1 \ldots y_m]$
  $X \leq Y$ se e solo se $\forall i \in [1, m] \ x_i \leq y_i$

	Tape drives	Plotters	Scanners	CD Roms
E =	4	2	3	1

	Tape drives	Plotters	Scanners	CD Roms
A =	2	1	0	0

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

L'**algoritmo di individuazione** dello stallo opera nel seguente modo:

1. Inizialmente ogni processo P_i è non marcato
2. Per ogni processo non marcato P_i tale che $R_i \leq A$ → Individuo i processi che possono ottenere le risorse necessarie alla loro terminazione
   - $A = A + C_i$;
   - marca P_i; → Il processo può terminare la sua normale esecuzione: ne rilascio le risorse
3. Se rimangono processi non marcati:
   - I processi non marcati sono in stallo → Nessuno di loro può acquisire risorse sufficienti a terminare, tenendo occupate le risorse già prese

Rilevare lo stallo

$E = (4 \quad 2 \quad 3 \quad 1)$ Risorse esistenti

$A = (2 \quad 1 \quad 0 \quad 0)$ Risorse libere

$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$ Risorse assegnate

$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$ Risorse richieste

Nel caso di esempio verrà quindi:

1. Marcato il processo P_3
2. Marcato il processo P_2
3. Marcato il processo P_1

Pertanto i processi nell'esempio NON sono in stallo

Rilevare lo stallo

$E = (2 \quad 1 \quad 1 \quad 2 \quad 1)$ Risorse esistenti

$A = (0 \quad 0 \quad 0 \quad 0 \quad 1)$ Risorse libere

$$C = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$ Risorse assegnate

$$R = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$ Risorse richieste

Nel caso di esempio verrà quindi:

1. Marcato il processo P_3

$$A = A + C_3 = (0 \quad 0 \quad 0 \quad 0 \quad 1) + (0 \quad 0 \quad 0 \quad 1 \quad 0) = (0 \quad 0 \quad 0 \quad 1 \quad 1)$$

Non vi sono altri indici (processi) per i quali è soddisfatto che $R_i \leq A$

Pertanto i processi non marcati P_1, P_2, P_4 sono in stallo

Rilevare lo stallo

Utilizzo dell'algoritmo di rilevamento dello stallo

Quando e quanto spesso invocare l'algoritmo dipende da:

- Frequenza presunta con la quale si verifica lo stallo
- Numero di processi coinvolti nello stallo

In generale uno stallo si verifica quando un processo avanza una richiesta che non può essere soddisfatta immediatamente.

Utilizzo dell'algoritmo a ogni richiesta: Consente la determinazione dello stallo e del processo la cui richiesta ha cagionato lo stallo. Aumento notevole del carico… (overhead)

Invocare l'algoritmo quando la percentuale di utilizzo delle risorse scende al di sotto di una soglia

Invocare l'algoritmo ad istanti arbitrari: nel grafo di assegnazione delle risorse potrebbero esistere molti cicli e diventa difficile determinare quale processo ha "causato" lo stallo…

Eliminare lo stallo

Si può operare secondo diverse modalità

Tramite **terminazione dei processi**:

- Abort di tutti i processi coinvolti nello stallo. Soluzione più comunemente adottata.
- Abort di un processo alla volta fino a quando il ciclo non è eliminato. Dopo ogni abort si deve rieseguire l'algoritmo di determinazione dello stallo. Ordine con cui abortire i processi dato da funzione di minimo costo basata su
  - Priorità dei processi
  - Tempo trascorso in esecuzione e necessario al completamento
  - Risorse già utilizzate e risorse richieste (processo in fase di stampa…)
  - Tipo di processo (interattivo, ecc.)

Eliminare lo stallo

Tramite **prelazione di risorse**:

- Selezionare un processo vittima – minimize cost.
- Rollback del processo a cui è stata sottratta la risorsa – return ad uno stato sicuro per poterlo riavviare in seguito
  Salvataggio periodico dello stato dei processi. In caso di stallo:
  - Si ripristina il processo all'ultimo stato salvato
  - Si fa ripartire il processo…il non-determinismo dei processi concorrenti dovrebbe garantire il non ri-verificarsi dello stallo

  In generale conviene uccidere il processo
- Starvation – alcuni processi potrebbero essere selezionati costantemente come vittime, include number of rollback in cost factor

Evitare lo stallo con politiche di allocazione

La strategia di evitare lo stallo mediante politiche di allocazione prevede di definire un ordine di assegnazione delle risorse tale che non si verifichi mai uno stallo

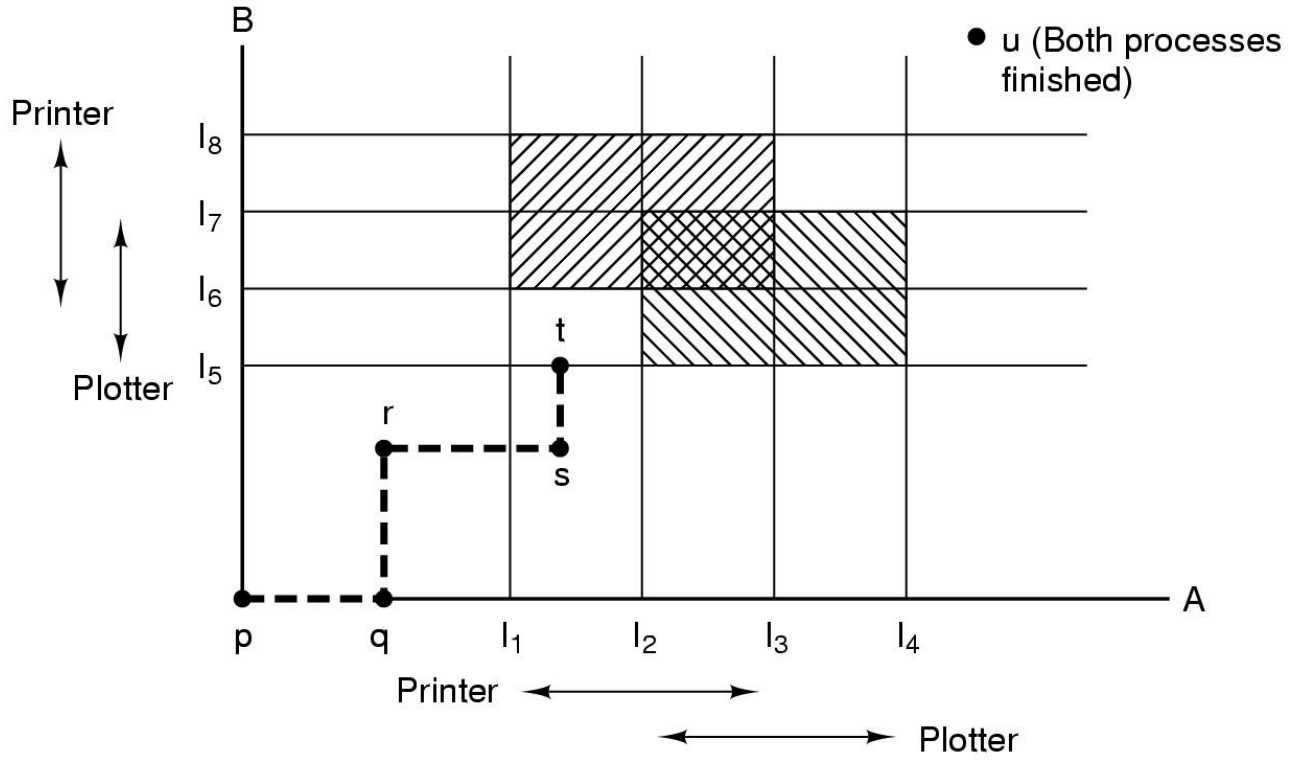
Un esempio grafico è il seguente:

Sugli assi sono riportate le istruzioni che devono essere eseguite, sull'ascissa per il processo A e sull'ordinata per il processo B

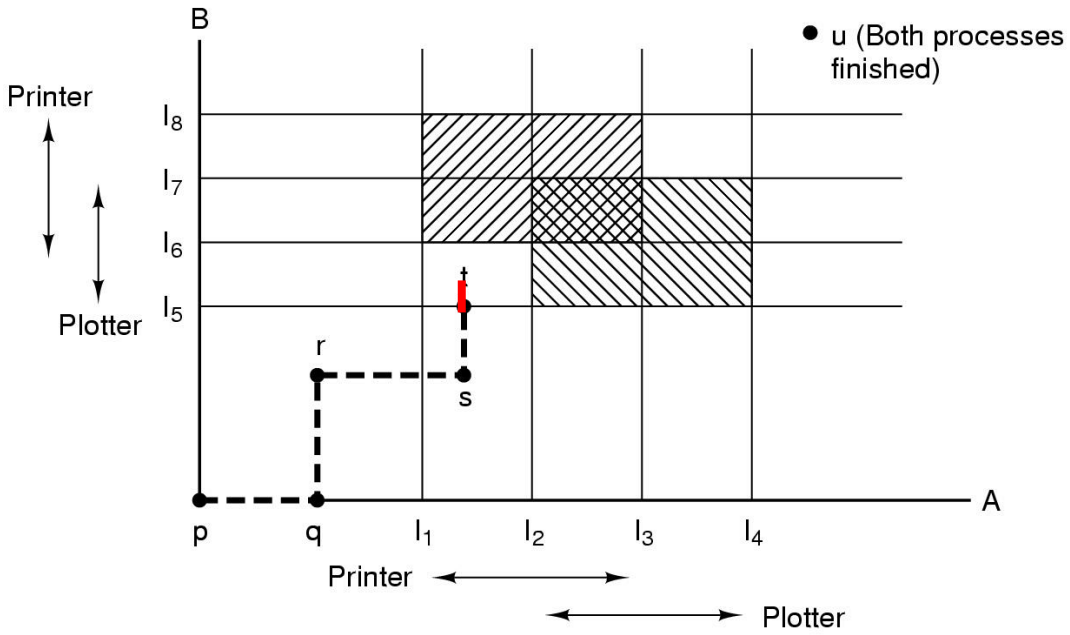
Sono presenti due risorse: printer e plotter, le quali sono usate dai processi nelle istruzioni indicate con la freccia a punta doppia

Le aree sbarrate non sono percorribili, in quanto si avrebbe la stessa risorsa assegnata a due processi

Qual è l'ordine giusto per evitare lo stallo?



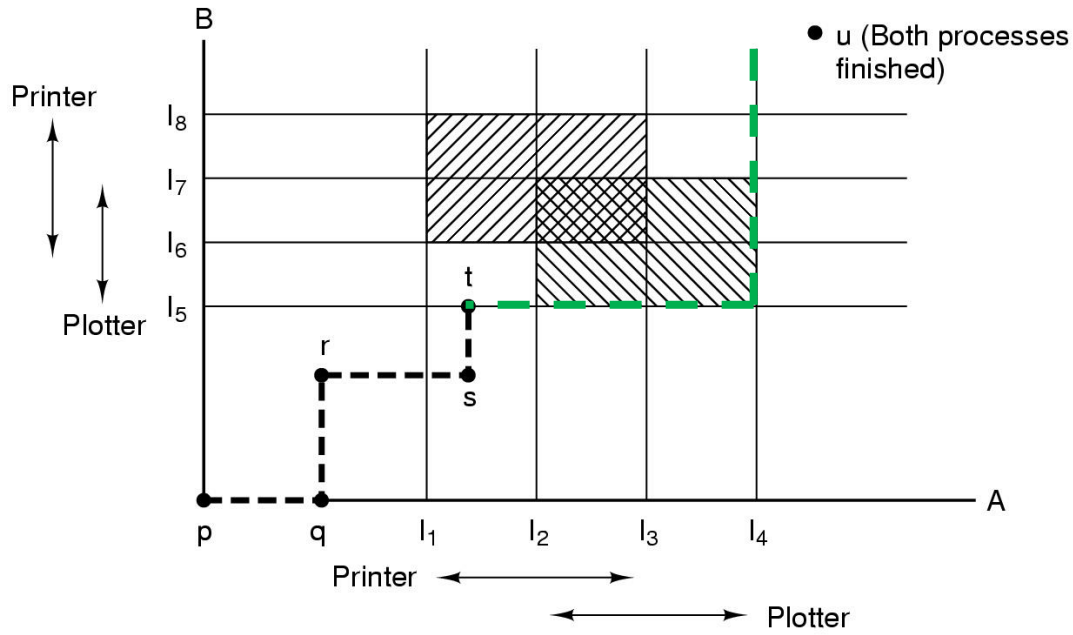
Evitare lo stallo con politiche di allocazione



Ordine di esecuzione: I_1, I_5

Si arriva allo stallo, in quanto non vi sono percorsi percorribili:

- Printer è assegnato ad A, Plotter a B
- Per proseguire, A necessita del Plotter
- Per proseguire, B necessita del Printer



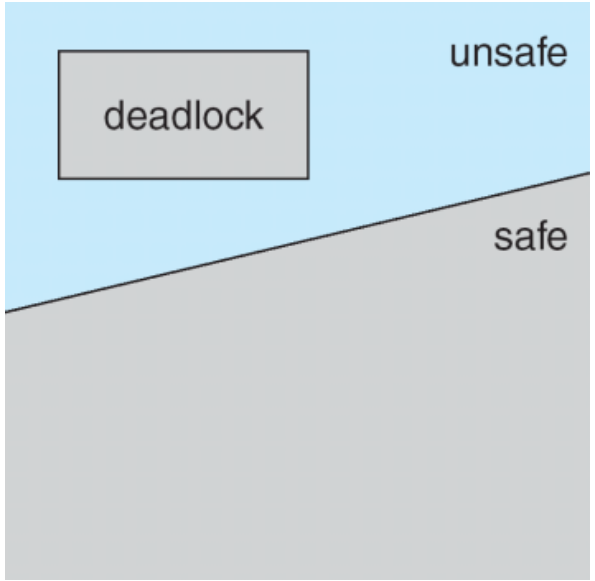
Ordine di esecuzione: $I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8$

Non si verifica stallo

Stato sicuro e insicuro

Si possono pertanto definire due stati del sistema, rispetto alle risorse assegnate:

- **Stato sicuro**: esiste almeno una sequenza di esecuzione dei processi che ne consente la terminazione senza incorrere in una situazione di stallo
- **Non sicuro**



Lo stato sicuro garantisce l'impossibilità di verificarsi dello stallo

Lo stato non sicuro indica che vi è possibilità (non certezza) che si verifichi lo stallo

La strategia di evitare lo stallo si basa sulla certezza che uno stato sicuro non possa determinare uno stallo

Deadlock avoidance significa quindi assicurarsi che il sistema non entri mai in uno stato non sicuro

Stato sicuro e insicuro

Un esempio applicato su risorse di un'unica tipologia:

Processo	Risorse assegnate	Risorse necessarie
A	3	9
B	2	4
C	2	7

Risorse disponibili: 3

Stati sicuri

Processo	Risorse assegnate	Risorse necessarie
A	3	9
B	4	4
C	2	7

Risorse disponibili: 1

Processo	Risorse assegnate	Risorse necessarie
A	3	9
B	0	-
C	2	7

Risorse disponibili: 5

Processo	Risorse assegnate	Risorse necessarie
A	3	9
B	0	-
C	7	7

Risorse disponibili: 0

Processo	Risorse assegnate	Risorse necessarie
A	3	9
B	0	-
C	0	-

Risorse disponibili: 7

Stati non sicuri

Processo	Risorse assegnate	Risorse necessarie
A	4	9
B	2	4
C	2	7

Risorse disponibili: 2

Processo	Risorse assegnate	Risorse necessarie
A	4	9
B	4	4
C	2	7

Risorse disponibili: 0

Stallo

Processo	Risorse assegnate	Risorse necessarie
A	4	9
B	0	-
C	2	7

Risorse disponibili: 4

Algoritmo del banchiere

Tale strategia prende il nome dal comportamento del banchiere: nessun banchiere presterebbe dei soldi senza avere certezze che questi tornino a sé. Il principio è attuato utilizzando le risorse come se fossero il denaro.

La strategia di evitare lo stallo mediante politiche di allocazione vede la sua applicazione pratica nell'**algoritmo del banchiere**

L'algoritmo del banchiere assegna risorse al processo solo se il sistema resta in uno stato sicuro: i processi procedono, terminano e restituiscono le risorse al sistema

Lo stato sicuro viene determinato utilizzando l'algoritmo di rilevazione dello stallo

Algoritmo del banchiere

```
struct state
{
        int resource[m];
        int available[m];
        int claim[n][m];
        int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
     < error >;                                  /* total request > claim*/
else if (request [*] > available [*])
     < suspend process >;
else                                             /* simulate alloc */
{
     < define newstate by:
     alloc [i,*] = alloc [i,*] + request [*];
     available [*] = available [*] - request [*] >;
}
if (safe (newstate))
     < carry out allocation >;
else
{
     < restore original state >;
     < suspend process >;
}
```

(b) resource alloc algorithm

Algoritmo del banchiere

```
boolean safe (state S)
{
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible)
    {
        <find a process Pk in rest such that
            claim [k,*] - alloc [k,*] <= currentavail;>
        if (found)                          /* simulate execution of Pk */
        {
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else
            possible = false;
    }
    return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)

Evitare lo stallo con politiche di allocazione

Alcune considerazioni:

Vantaggi:

- Non è necessario interrompere processi e riportarli in uno stato precedente (problema presente nelle strategie di rilevamento dello stallo)

Restrizioni:

- Il numero massimo di risorse necessitate da ogni processo deve essere noto a priori (prima dell'esecuzione)
- I processi devono essere indipendenti: l'ordine di esecuzione non deve essere vincolato a esigenze di sincronizzazione
- Deve esserci un numero fissato di risorse da allocare
- Quando un processo richiede risorse potrebbe essere posto in attesa
- Quando un processo ottiene tutte le risorse di cui necessita, deve rilasciarle in un tempo finito

Impedire lo stallo rimuovendone le condizioni

È possibile impedire il verificarsi dello stallo rimuovendo una delle quattro condizioni

Si impongono pertanto dei vincoli sulla richiesta delle risorse

Le classi di prevenzione sono:

- **Metodi indiretti:** prevengono il verificarsi di una delle tre condizioni necessarie
- **Metodi diretti:** prevengono il verificarsi dell'attesa circolare

Impedire lo stallo rimuovendone le condizioni

Negare la **Mutua Esclusione**

- Un processo non deve mai attendere una risorsa condivisibile
- Devono essere possibili accessi multipli contemporanei alle risorse condivise
- Non applicabile in generale

- Applicabile in alcuni casi specifici: alcuni dispositivi (ad esempio le stampanti) possono essere gestiti con spool
  - I processi scrivono l'output in un'area di spool
  - Solo il gestore della stampante richiede e usa la stampante
  - Quindi lo stallo per la stampante è eliminabile
- Non tutti i dispositivi possono essere gestiti con spool
- Ci puo essere stallo nell'accesso all'area di spool
- Principio:
  - Evitare di assegnare una risorsa quando non strettamente necessario
  - Far in modo che il minor numero possibile di processi possa richiedere una risorsa

Impedire lo stallo rimuovendone le condizioni

Negare l'**hold and wait**:

- Ogni processo deve richiedere all'inizio della sua esecuzione tutte le risorse
- Problemi:
  - il processo potrebbe non sapere di quali risorse avrà bisogno
  - Il processo entra in blocked fino a quando tutte le richieste non vengono soddisfatte contemporaneamente
  - Vincola risorse che altri processi potrebbero utilizzare
- Approccio inefficiente:
  - Prima di andare in run tutte le risorse devono essere disponibili, in realtà potrebbe procedere utilizzandone solo una parte
  - Le risorse assegnate al processo potrebbero rimanere inutilizzate per molto tempo, gli altri processi sono in attesa indefinita

Impedire lo stallo rimuovendone le condizioni

Negare l'**assenza di prerilascio**:

Due possibilità:

1. Un processo che non riesce ad ottenere le risorse di cui necessita, rilascia quelle che detiene per richiederle nuovamente in un istante successivo
2. OS può richiedere il pre-rilascio delle risorse al processo che le detiene per assegnarle al richiedente

- Approccio applicabile solo se lo stato della risorsa è facilmente salvabile (CPU)
- Non applicabile nel caso di stampanti…

Impedire lo stallo rimuovendone le condizioni

Negare l'**attesa circolare**:

- Si definisce un ordine lineare (di numerazione) per tutte le risorse
- Se un processo richiede una risorsa R, successivamente potrà richiedere solo una risorsa che nell'ordinamento segue R
- Es.: R_i precede R_j se $i < j$, un processo potrà chiedere le risorse solo nell'ordine R_i, R_j
- Questo metodo può essere inefficiente

Concorrenza

Multiprogrammazione: gestione di più processi su un singolo processore

Multiprocessing: gestione di più processi su più processori

Processi distribuiti: cluster

Competizione tra processi (threads) per ottenere (e condividere) le risorse:

- Cpu
- Memoria
- Canali di I/O
- Files
- ecc..

Terminologia della concorrenza

Sessione Critica: porzione di codice all'interno di un processo (thread) che richiede accesso a risorse condivise.

Deadlock: situazione nella quale due o più processi sono impossibilitati dal procedere poiché sono in attesa l'uno dell'altro

Livelock: situazione nella quale due o più processi cambiano continuamente il proprio stato a causa del cambiamento di stato degli altri (senza fare alcun lavoro utile)

Mutua esclusione: requisito per il quale, quando un processo è nella propria sezione critica, nessun altro processo può essere nella propria sezione critica se questa fa riferimento a risorse condivise con il primo processo

Race condition: situazione nella quale thread o processi leggono e scrivono un dato condiviso e il risultato dipende dalla loro velocità reciproca

Starvation: situazione nella quale un processo non riceve mai l'utilizzo di una risorsa e viene costantemente scavalcato da altri processi

Concorrenza

VANTAGGI

Benefici sulla esecuzione nonostante il sovraccarico (context switching)

Migliore utilizzo delle risorse

Concorrenza

PROBLEMI

Singolo Processore

- Condivisione pericolosa: ordine delle operazioni di lettura e scrittura su aree di memoria condivise
- Difficoltà nell'assegnare le risorse ai processi in maniera ottimale
- Difficoltà nella rilevazione degli errori nel codice e dei conflitti di interlacciamento

```
char in,out; //condivise
void echo()
{
   scanf("%c", &in);
   out = in;
   printf("%c", out);
}
```

Condivisione della procedura echo():

- Risparmio dello spazio di memoria
- Due processi concorrenti.
  - P1 viene interrotto dopo la scanf,
  - P2 esegue tutto echo,
  - P1 viene riattivato da scanf in poi e ha perso il dato che aveva letto

Soluzione: un solo processo alla volta (MUTUA ESCLUSIONE)

Concorrenza

PROBLEMI - SMP

- stessi problemi di un calcolatore a singolo processore (una interruzione può fermare l'esecuzione di un processo in un qualsiasi istante)
- interlacciamento esecuzione processi paralleli

Processo P1 – Processore1

```
……
   scanf("%c", &in);
……
   out = in;
   printf("&c", out);
……
```

Processo P2 – Processore2

```
……
……
   scanf("%c", &in);
   out = in;
   ……
   printf("&c", out);
```

Il carattere letto da P1 è perso prima di poter essere stampato (perdita di aggiornamento)

Soluzione: MUTUA ESCLUSIONE

Un solo programma per volta può entrare nella propria sezione critica (Es.: Un solo programma per volta può inviare comandi alla stampante)

Mutua esclusione

I meccanismi che provvedono alla mutua esclusione devono garantire i seguenti requisiti:

- Un solo processo alla volta deve accedere alla sezione (o risorsa) critica;
- Un processo fuori della sezione critica non deve interferire con il processo nella sezione critica
- Ogni processo deve poter accedere dopo un tempo finito di attesa in coda alla risorsa critica (no stallo o starvation)
- Se nessun processo è nella sezione critica, un processo deve poter entrare nella sezione critica senza attese
- Non ci devono essere supposizioni sulla velocità di esecuzione relativa dei processi
- Il tempo di permanenza nella sezione critica è (de)finito

Mutua esclusione

Approcci software: i processi, senza ausilio del Sistema Operativo o del linguaggio di programmazione, devono coordinarsi tra loro (Dekker)

- Aumento del tempo di esecuzione
- Errori frequenti

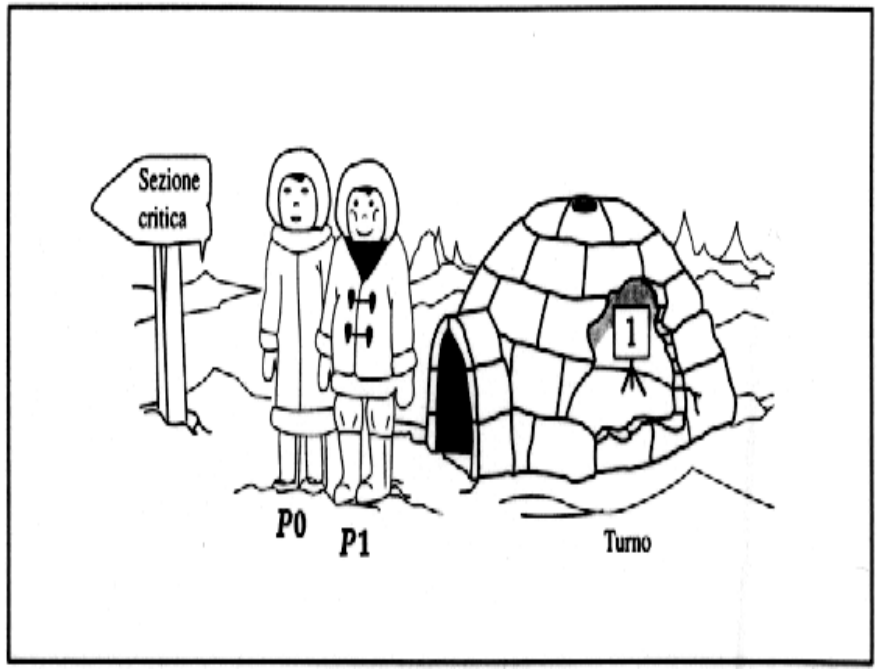
Utilizzo di particolari **istruzioni di macchina** (test-set, scambio)

- Riduzione del sovraccarico
- Non soddisfacenti

Supporto del sistema operativo o del linguaggio di programmazione

- Scambio Messaggi (Inter Process Communication)
- Semafori
- Monitor

Algoritmo di Dekker – 1° tentativo



Protocollo dell'Igloo: prima di entrare nella sezione critica, i processi controllano uno alla volta una variabile turno

Busy wait (attesa attiva): consuma tempo utile di esecuzione

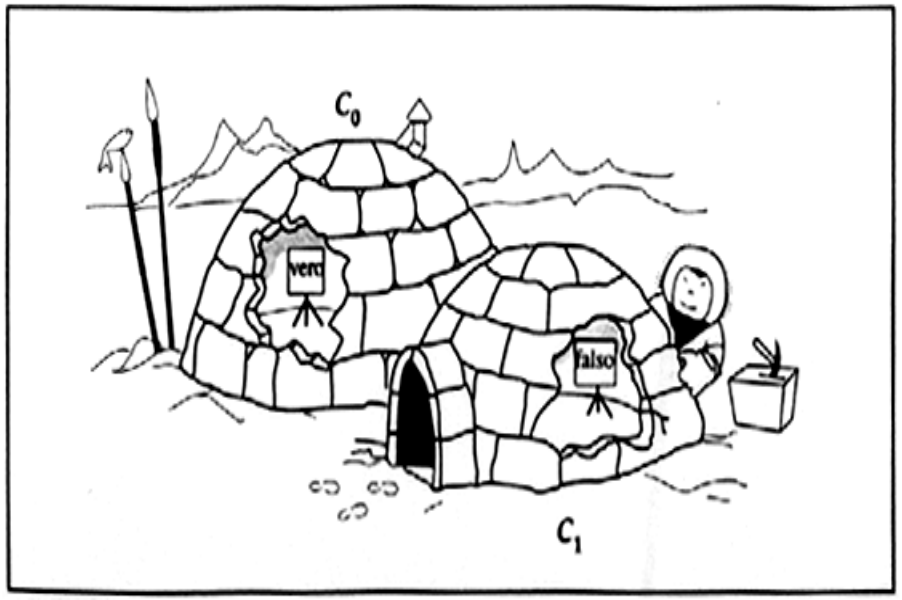
```
var turno= 0..1; //variabile globale condivisa

Processo 0
  ….
while (turno!=0)
{nulla} //busy wait
<sezione critica>;
turno=1;
  …

Processo 1
….
while (turno!=1)
{nulla} //busy wait
<sezione critica>;
turno=0;
….
```

PRO: garantisce la mutua esclusione

PUNTI DEBOLI:

1. I processi devono osservare l'alternanza, il più lento determina la velocità di avanzamento di entrambi i processi
2. Se un processo fallisce nella propria sezione critica e non, l'altro processo rimarrà bloccato per sempre

Algoritmo di Dekker – 2° tentativo



Ogni processo ha un flag relativo all'utilizzo della risorsa e può leggere il flag dell'altro senza modificarlo

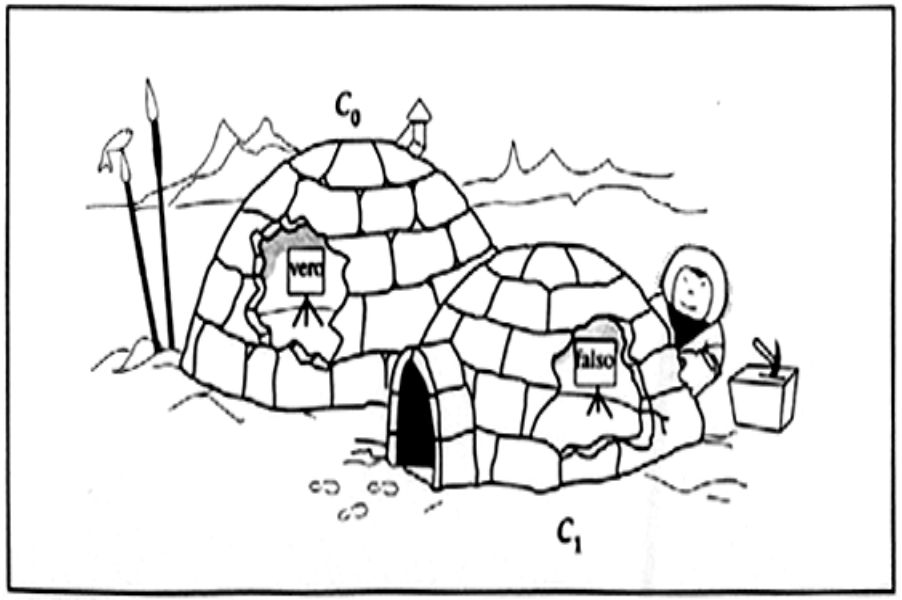
```
boolean flag[2];

//Processo 0
…
while (flag[1])
   {nulla;}
flag[0]=true;
<sezione critica>
flag[0]= false;
…
```

PRO: se un processo fallisce fuori della sua sezione critica l'altro può continuare a lavorare

CONTRO:

- se un processo fallisce entro la sezione critica o prima di mettere false nel suo flag allora l'altro è bloccato per sempre
- se entrambi vedendo il flag dell'altro false, mettono il loro flag a true, entrambi vanno nella sezione critica, senza mutua esclusione. Dipendenza della velocità dei processi

Algoritmo di Dekker – 3° tentativo




```
boolean flag[2];

// Processo 0

…

flag[0] = true; //indico prima del //test
  di voler andare in sezione //critica

while (flag[1])

{nulla}

<sezione critica>;

flag[0] =  false;

…
```

PRO: la mutua esclusione è garantita

PROBLEMI:

- Se un processo fallisce entro la sua sezione critica l'altro è bloccato
- Se entrambi settano il flag a true prima che uno dei due verifichi la condizione del while si ha stallo

Algoritmo di Dekker – 4° tentativo

Processo 0

```
…
flag[0]= true;
while (flag[1])
{
  flag[0] = false;
  <pausa>
  flag[0] = true;
}
<sezione critica>;
flag[0] = false;
…
```

Processo 1

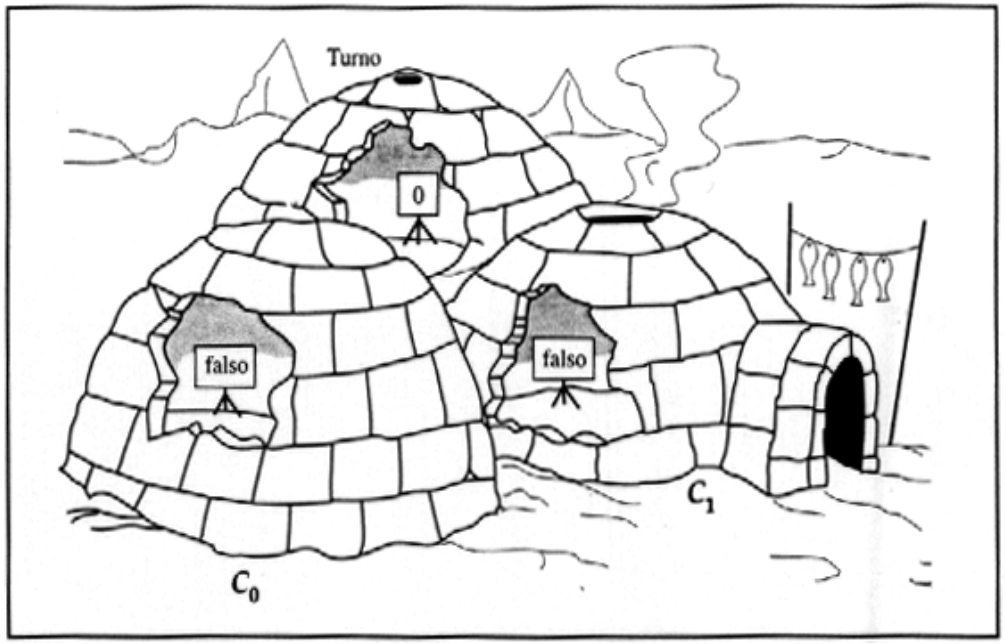
```
…
flag[1] = true;
while (flag[0])
{
  flag[1] = false;
  <pausa>;
  flag[1] = true;
}
<sezione critica>;
flag[1] = false;
…
```

PRO: mutua esclusione garantita

PROBLEMI:

- Po: “vado io”, P1: “vado io”, Po:“non vado io”, P1: “non vado io”, attesa….
  in realtà non’è uno stallo poiché chi esce prima dal ciclo di attesa riesce ad entrare nella sezione critica
- Se un processo fallisce entro la sua sezione critica l’altro è bloccato

Algoritmo di Dekker – soluzione corretta



Ogni processo ha una propria variabile **flag** che indica se vuole andare nella sezione critica o meno

Variabile **turno** specifica chi ha il diritto di insistere nel tentativo di entrare nella propria sezione critica

Es.: Po vuole entrare (pone il suo flag a true)
Po controlla il flag di P1
se falso entra nella sezione critica
se vero controlla il turno
se turno è 0 (il suo) continua a controllare periodicamente il flag di P1
se turno è 1 pone il suo flag a falso lasciando il passo a P1

Algoritmo di Dekker – soluzione corretta

```
boolean flag[2];
int turno;
void main()
{           flag[0]=false;          flag[1]=false;                  turno=1; //turno=0;
            …
            processo P0;
            processo P1;
            …
}

//processo P0
{
            …
                        flag[0]=true;
                        while(flag[1])
                        {
                                    if (turno==1)
                                    {
                                                flag[0]=false;
                                                while(turno==1)
                                                            {<nulla…ATTESA ATTIVA>}
                                                flag[0]=true;
                                    }
                         }
                        <sezione critica>
                        flag[0]=false;
            …
}
```

Algoritmo di Dekker – soluzione corretta

- Algoritmo “complesso”
- Attesa attiva: un processo controlla continuamente il flag dell’altro processo se pari a true
- Se un processo fallisce nella propria sezione critica l’altro processo rimane bloccato

Supporti hardware

Macchine monoprocessore: i processi si alternano in esecuzione (EXE)

la mutua esclusione si può ottenere evitando l'interruzione di un processo attivando e disattivando gli interrupt (supporto hardware)

<disattiva le interruzioni>

<sezione critica>

<attiva le interruzioni>

PRO: mutua esclusione garantita

PROBLEMI:

- Efficienza peggiora poiché il processore non può alternare i processi liberamente
- Non funziona su macchine SMP

Supporti hardware

Soluzioni su SMP:

Istruzioni macchina speciali per l'accesso a locazioni di memoria in modo atomico (non interrompibile)

test-and-set
scambio (swap)

l'accesso sequenziale ad una locazione di memoria è garantita dall'hardware
Se si eseguono due test-and-set (swap) contemporaneamente, esse vengono serializzate

Test & set

```
boolean TestAndSet (boolean *target)
{
  boolean val = *target;
  *target = TRUE;
  return val;
}
```

Utilizzo di test&set per garantire la mutua esclusione

Sia lock una variabile boolean condivisa inizializzata a FALSE (la risorsa è libera)

```
while (true) {
            while ( TestAndSet (&lock ))
            ;   /* do nothing
            <critical section>
            lock = FALSE; //rilascio
            …
            }
```

Swap

```
void swap (boolean *a, boolean *b)
          {
                    boolean temp = *a;
                    *a = *b;
                    *b = temp:
          }
```

Utilizzo di swap per garantire la mutua esclusione

Sia lock una variabile booleana condivisa inizializzata a FALSE (ovvero la risorsa è accessibile)

```
while (true)  {
                    key = TRUE;
                    while (key == TRUE)
                              swap (&lock, &key ); //in key torna false
                    <critical section>
                    lock = FALSE;
                    …
               }
```

Supporti hardware

Vantaggi

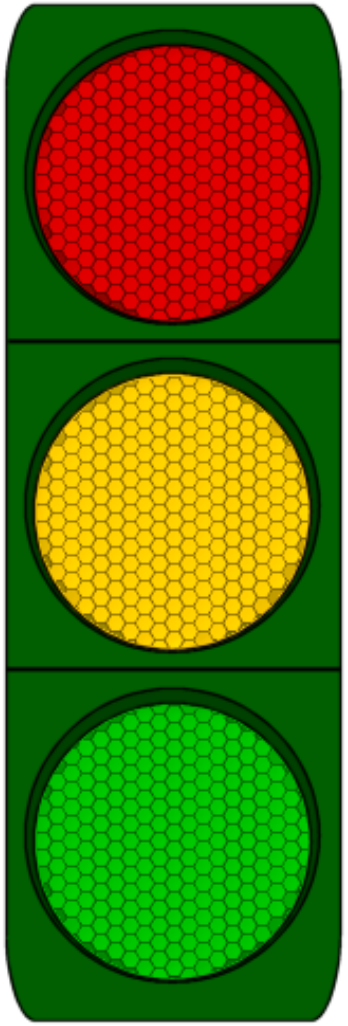
- si può applicare a un qualsiasi numero di processi anche su multiprocessori a memoria condivisa;
- si può usare per gestire più di una sezione critica, ciascuna con una propria variabile;

Svantaggi

- Attesa attiva (i processi consumano tempo di cpu)
- Starvation (la scelta di quale processo andrà nella sezione critica è arbitraria)
- Stallo

  Es (singolo processore)
  - P1 in sezione critica
  - P2 ha priorità più alta di P1
  - P2 tenterà di accedere (tramite test&set o swap) alla risorsa bloccata da P1 e nella sua attesa attiva non lascerà mai il posto a P1 che ha priorità più bassa

Supporto del SO e linguaggi di programmazione



Le soluzioni offerte dal SO e dai linguaggi di programmazione sono i **semafori** e i **monitor**

Il semaforo può essere binario (gestisce una sola risorsa) o contatore (gestisce un lotto di risorse dello stesso tipo)

SEMAFORO: variabile (booleana/intera) sulla quale sono possibili 3 operazioni:

- Inizializzazione ad un valore non negativo
- Operazione atomica* **wait()**: decrementa il valore della variabile. Se il valore della variabile diventa negativa, il processo che ha eseguito la wait viene bloccato.
- Operazione atomica* **signal()**: incrementa il valore della variabile. Se il valore della variabile era negativo, uno dei processi bloccati sull'operazione di wait viene sbloccato

*atomica = non interrompibile

Semafori

1. Si associa un semaforo ad ogni risorsa condivisa
2. Il processo che vuole utilizzare la risorsa effettua una operazione di wait
3. Il processo che rilascia la risorsa effettua il signal
4. La variabile numerica indica il numero di istanze di una specifica risorsa condivisa (semaforo contatore)
5. Se la variabile è negativa, essa rappresenta (presa in valore assoluto) il numero di processi in attesa

Wait e signal nel semaforo binario impostano il valore del semaforo rispettivamente a falso e vero

```
wait (S);
<Critical Section>
signal (S);
```

Implementazione del semaforo contatore

```
typedef struct {
                int istanze;
                struct processo *P; //lista dei processi in coda
                } semaforo;
```

```
void wait(semaforo s)
{
        s.istanze--;
        if(s.istanze<0)
                {
                        <poni processo in coda>
                        <blocca questo processo: running->blocked >
                }
}
```

```
void signal(semaforo s) :
{
        s.istanze++;
        if(s.istanze<=0)
                {
                        <rimuovi un processo in coda>
                        <sveglia il processo: blocked->ready >
                }
}
```

Implementazione del semaforo binario

Semaforo binario: semaforo il cui valore intero può essere solo 0 o 1
Gestione più complessa che non con semafori contatore

```
typedef struct {
                    boolean val;
                    struct processo *P; //lista dei processi in coda
                    } semaforo_bin;
```

```
void wait(semaforo_bin s)
{
          if (s.val==1)
                    s.val=0;
          else
                    {         <poni processo in coda a P>
                              <blocca questo processo: running->blocked >
                    }
}
```

```
void signal (semaforo_bin s)
{
          if(*P==NULL) //coda vuota
                    s.val=1;
          else
                    {         <rimuovi un processo in coda>
                              <sveglia il processo: blocked->ready >
                    }
}
```

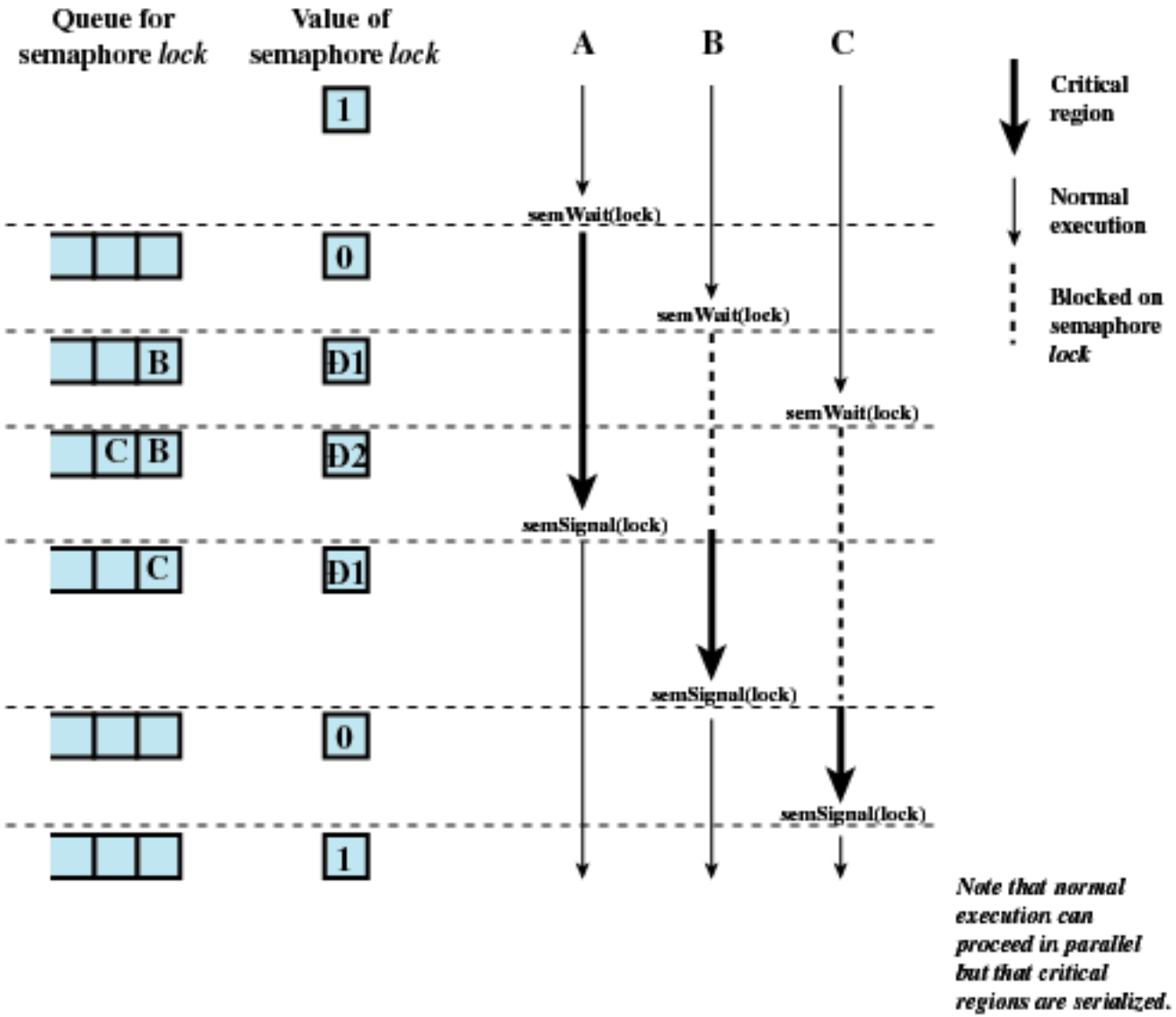
Implementazione dei semafori

Come fare affinché signal e wait siano atomiche??

1. **Implementarle in hardware o firmware**
2. Dekker o Peterson…sovraccarico di elaborazione
3. **Utilizzo dell'istruzione atomica test&set**
   Esercizio: si scriva lo pseudocodice C
4. Se il sistema è monoprocessore basta disabilitare gli interrupt durante le operazioni
   Esercizio: si scriva lo pseudocodice C

I semafori non presentano attesa attiva in quanto i processi in coda vengono bloccati. Se un processo fallisce nella sezione critica, il semaforo si occupa di dare comunque il signal della risorsa

Accesso al dato condiviso mediante semafori



Deadlock e starvation

Deadlock – due o più processi sono in attesa di un evento che può essere determinato solo da uno dei processi in attesa

Siano S e Q due semafori inizializzati a 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- P1 non rilascia Q fino a quando non ottiene S
- Po non rilascia S fino a quando non ottiene Q

Le signal non saranno mai eseguite: STALLO

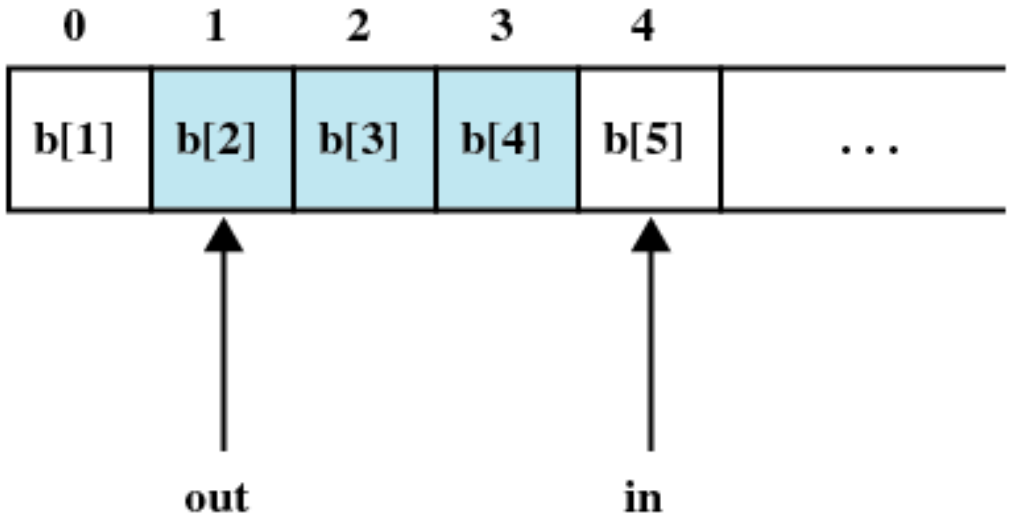
Starvation – indefinite blocking. Un processo non viene mai rimosso dalla coda al semaforo. Si immagini una gestione LIFO.

Produttore e consumatore

Tipica rappresentazione dei processi concorrenti:

- Uno o più produttori generano dati inserendoli in un buffer
- Un consumatore preleva i dati uno alla volta

Ipotesi di buffer infinito: l'accesso al buffer tra produttore e consumatore deve essere mutuamente esclusivo




```
produttore
        <produce dato>
        buffer[in]=dato;
        in++;
```

```
consumatore
    while(in >= out)
    { w = buffer[out];
      out++;
      <consuma w>
    }
```

Produttore e consumatore

```
/* program producerconsumer */
int n;  //Numero di elementi nel buffer
binary_semaphore s = 1;//gestisce l'accesso al buffer
binary_semaphore delay = 0;//gestisce il caso di nessun
                           //elemento nel buffer
void producer()
{
     while (true)
     {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     int m; /* a local variable */
     semWaitB(delay);
     while (true)
     {
          semWaitB(s);
          take();
          n--;
          m = n;
          semSignalB(s);
          consume();
          if (m==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

...con semafori binari...

Essendo il buffer infinito, il produttore può inserire tutto quello che produce

Se n=1, significa che il buffer prima era vuoto ed occorre avvisare il consumatore

Si pone in attesa che il primo elemento venga prodotto

Se il consumatore ha svuotato il buffer, si mette in attesa che venga prodotto un nuovo elemento

Produttore e consumatore

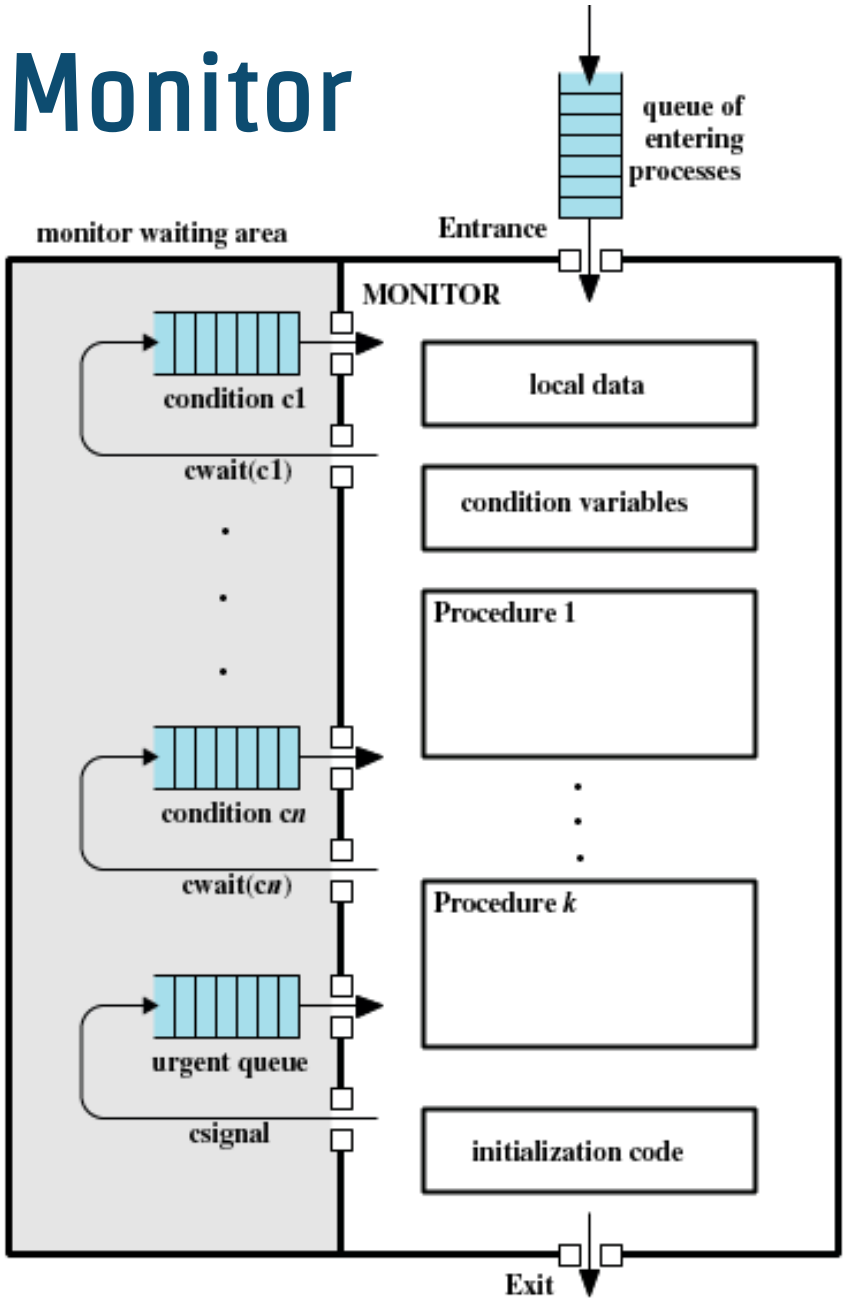
```
/* program  producerconsumer */
semaphore n = 0;
semaphore s = 1;
void producer()
{
     while (true)
     {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true)
     {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

…con semafori contatore…

È indifferente il loro ordine???
SI

È indifferente il loro ordine???
NO..
Il consumatore entrerebbe nella sezione critica quando il buffer è vuoto e nessun produttore potrebbe aggiungere elementi:
STALLO

Monitor



I semafori sono primitive potenti per gestire la mutua esclusione, ma la scrittura di un programma con l'utilizzo dei semafori può essere tutt'altro che semplice.

MONITOR:

- costrutto di sincronizzazione di alto livello
- modulo software le cui caratteristiche principali sono:
  - Variabili locali accessibili solo dalle procedure (metodi) definite al suo interno e non da procedure esterne
  - Un processo entra nel monitor chiamando le sue procedure
  - Un solo processo alla volta può essere in esecuzione all'interno del monitor. Gli altri processi sono sospesi nell'attesa che il monitor diventi disponibile

UTILIZZO:

- Mutua esclusione
- Protezione delle strutture dati condivise

Sincronizzazione tramite variabili di condizione. Su tali variabili si opera mediante:

- *cwait (c)* : sospende l'esecuzione del processo chiamante sulla condizione c
- *csignal (c)* : riattiva un processo sospeso sulla condizione c. NB: se non c'è nessun processo in attesa su tale condizione il segnale viene perso