Modelli di interazione tra processi

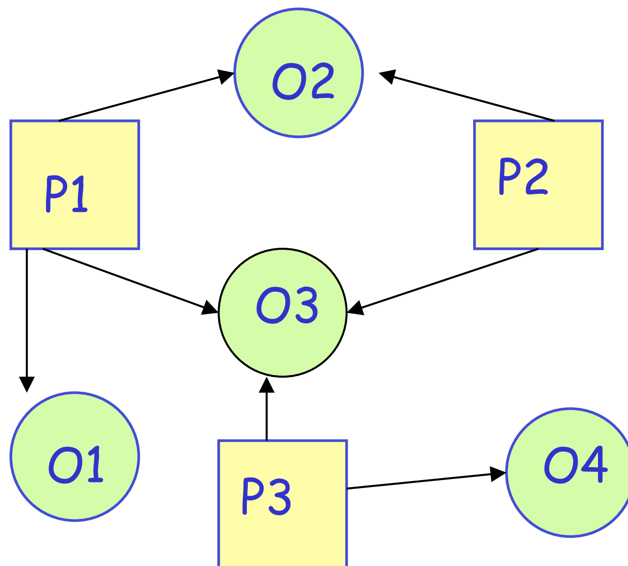
- Modello a **memoria comune** (ambiente globale, global environment)
- Modello a **scambio di messaggi** (ambiente locale, message passing)

Modello a memoria comune

Modello a memoria comune

Il sistema è visto come un insieme di

- **processi**
- **oggetti** (*risorse*)



Diritto di accesso

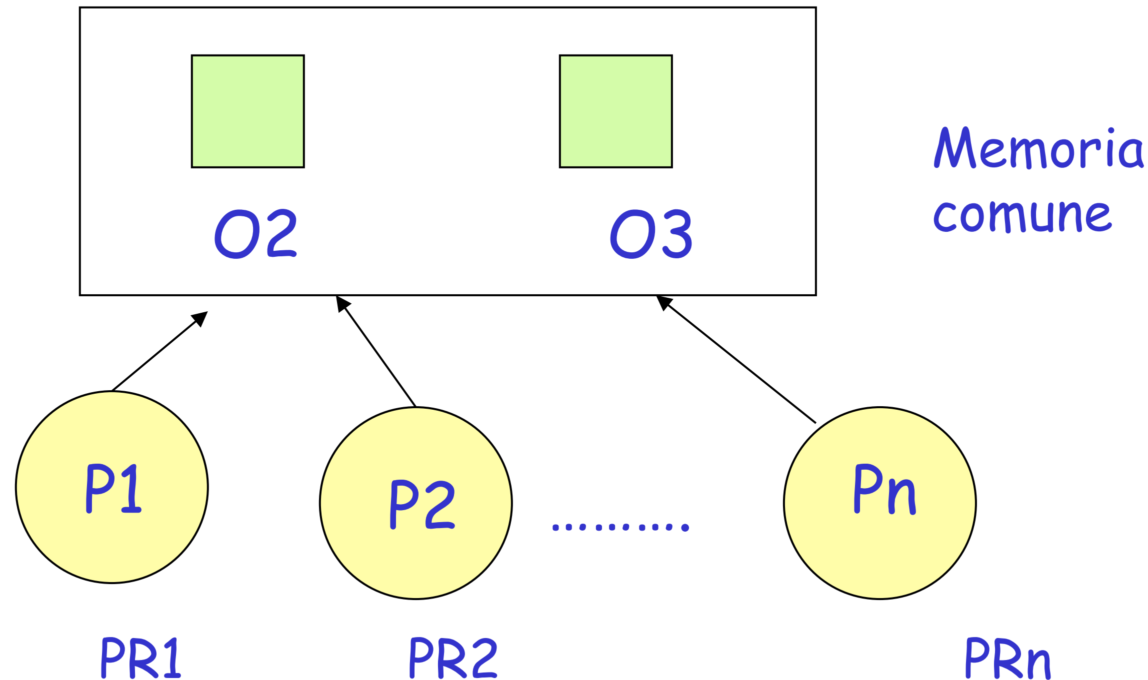
O1 , O4 risorse private

O2 , O3 risorse comuni

Tipi di interazioni tra processi:

- **competizione**
- **cooperazione**

- Il modello a memoria comune rappresenta la naturale astrazione del funzionamento di **un sistema in multiprogrammazione** costituito da uno o più processori che hanno accesso ad una memoria comune

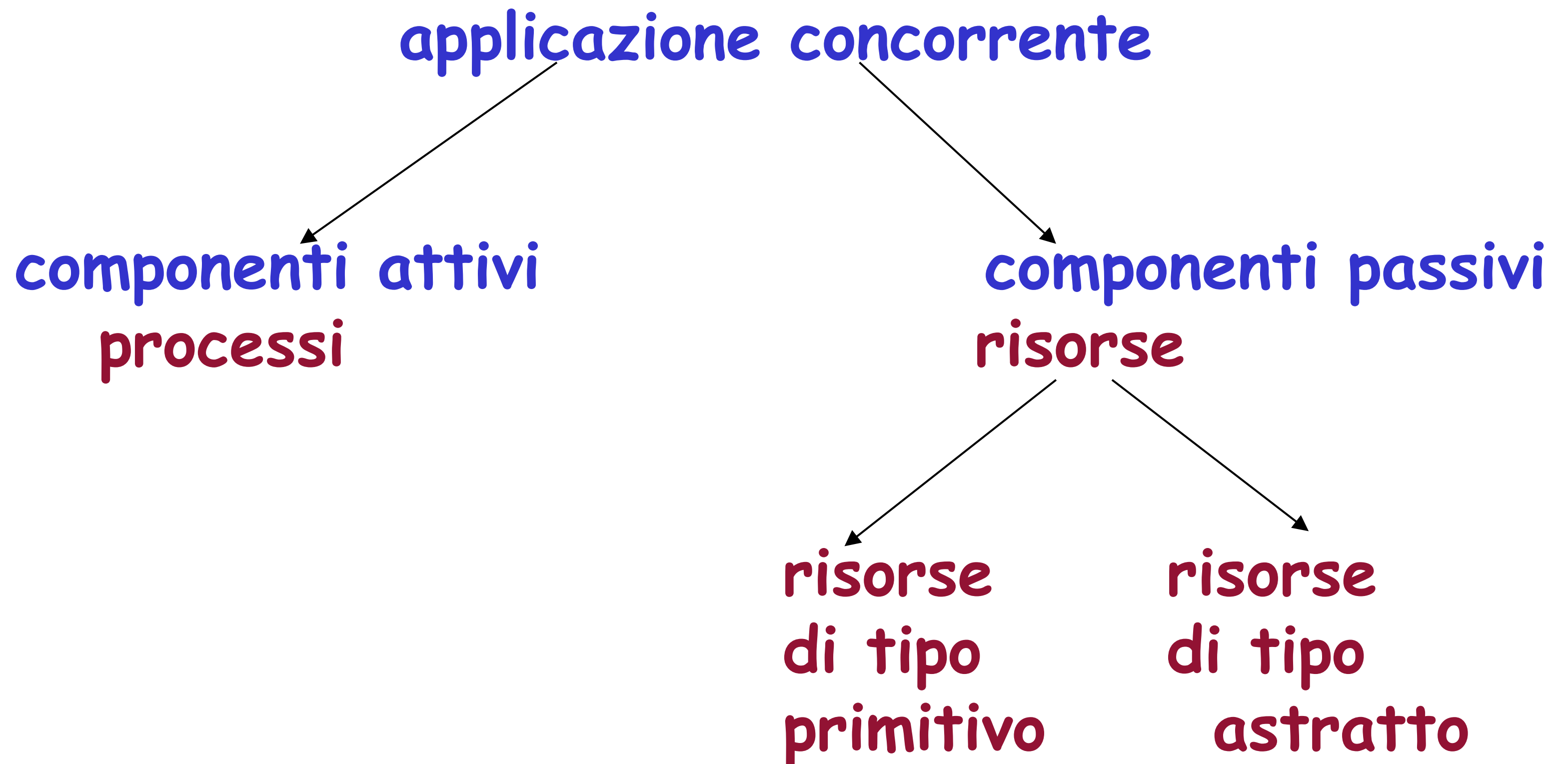


- Ad ogni processore può essere associata una memoria privata, ma ***ogni interazione avviene tramite oggetti contenuti nella memoria comune.***

Aspetti caratterizzanti

Ogni applicazione viene strutturata come un insieme di componenti, suddiviso in due sottoinsiemi disgiunti:

- processi (componenti attivi)
- risorse (componenti passivi).



Risorsa: qualunque oggetto, fisico o logico, di cui un processo necessita per portare a termine il suo compito.

- Le risorse sono raggruppate in *classi;* una classe identifica l'insieme di *tutte e sole* le operazioni che un processo può eseguire per operare su risorse di quella classe.

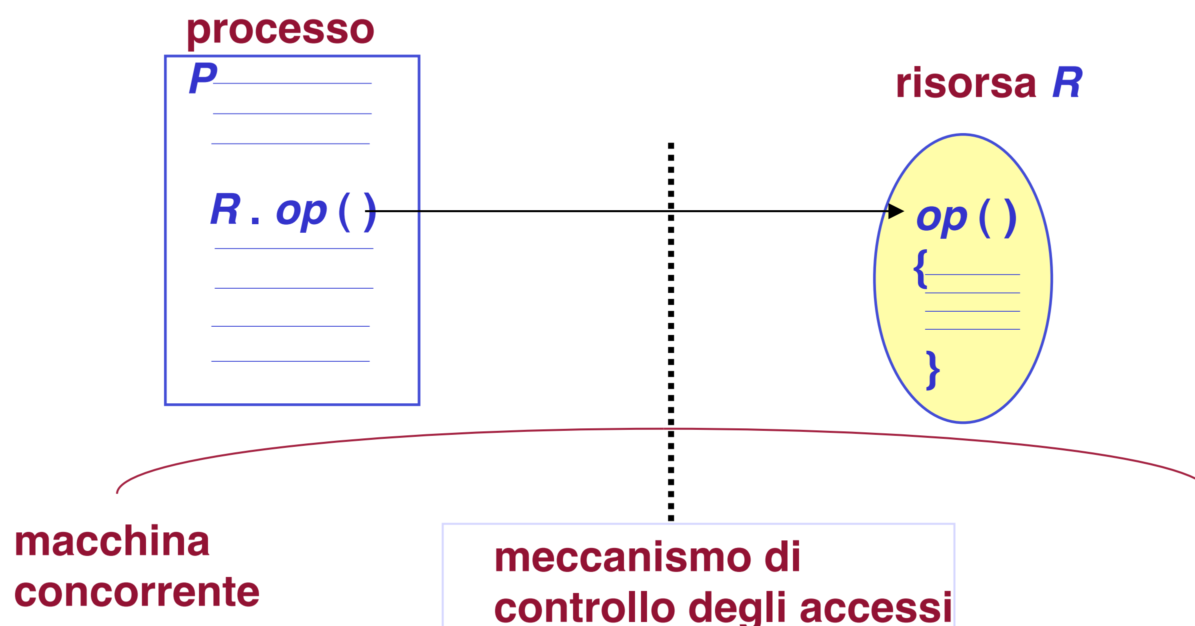
- Il termine ***risorsa*** si identifica con quello di ***struttura dati*** allocata nella memoria comune.

- Vale anche per risorse fisiche: descrittore del dispositivo

Meccanismo di controllo degli accessi

Necessita` di specificare quali processi ed in quali istanti possono accedere alla risorsa.

Il meccanismo di controllo degli accessi ha il compito di controllare che gli accessi dei processi alle risorse avvengano *correttamente.*



Gestore di una risorsa

Per ogni risorsa R, il suo gestore definisce, in ogni istante t, l'insieme *SR(t)* dei processi che, in tale istante, hanno il diritto di operare su R.

Classificazione risorse:

- Risorsa R **dedicata**: se SR (t) ha una cardinalità sempre ≤ 1
- Risorsa R **condivisa**: in caso contrario

- Risorsa R **allocata staticamente**: se SR(t) è una costante:

$$SR(t) = SR(t0), \forall t$$

- Risorsa R **allocata dinamicamente**: se SR (t) è funzione del tempo

Tipologie di allocazione delle risorse

	risorse dedicate	risorse condivise
risorse allocate staticamente	(A) risorse private	(B) risorse comuni
risorse allocate dinamicamente	(C) risorse comuni	(D) risorse comuni

Per ogni risorsa allocata staticamente, l'insieme SR(t) è definito prima che il programma inizi la propria esecuzione.

Il gestore della risorsa è in questo caso il programmatore che, in base alle regole di visibilità del linguaggio, stabilisce quale processo può "vedere" e quindi operare su R.

Per ogni risorsa R allocata dinamicamente, il relativo gestore G_R definisce l'insieme SR(t) in fase di esecuzione e quindi deve essere un componente della stessa applicazione.

Compiti del gestore di una risorsa

1. mantenere aggiornato l'insieme *SR(t)* e cioè lo stato di allocazione della risorsa;
2. fornire i meccanismi che un processo può utilizzare per acquisire il diritto di operare sulla risorsa, entrando a far parte dell'insieme *SR(t)*, e per rilasciare tale diritto quando non è più necessario;
3. implementare la strategia di allocazione della risorsa e cioè definire quando, a chi e per quanto tempo allocare la risorsa.

Gestore di una risorsa

data una risorsa R,
il suo gestore G_R è costituito da:

una risorsa condivisa
in un sistema
organizzato secondo
il modello a
memoria comune

un processo
in un sistema
organizzato secondo
il modello a
scambio di messaggi

Pool di risorse equivalenti

Sono insiemi di istanze di risorse dello stesso tipo e quindi in grado di svolgere le stesse funzioni.

Vengono gestite da un solo gestore.

Consideriamo un processo P che deve operare, ad un certo istante, su R di tipo T (op1,op2,..,opn):

- Se R è allocata **staticamente** a P (modalità *A* e B) , il processo possiede il diritto di operare in qualunque istante:

R.op$_i$(...); */*esecuzione dell'operazione op$_i$ su R*/*

- Se R è allocata **dinamicamente** a P (modalità *C* e D), è necessario prevedere il **gestore** *GR*, che implementa le funzioni di *Richiesta* e *Rilascio* della risorsa; il processo P deve eseguire il seguente protocollo:

GR.Richiesta (..); */* acquisizione della risorsa R*/*
R.opi(..); */*esecuzione dell'operazione op$_i$ su R*/*
GR.Rilascio(...); */* rilascio della risorsa R*/*

- Se R è allocata come **risorsa condivisa**, (modalità B e D) è necessario assicurare che gli accessi avvengano in modo non divisibile:
  - ➔Le funzioni di accesso alla risorsa devono essere programmate come una *classe di sezioni critiche* (utilizzando i meccanismi di sincronizzazione offerti dal linguaggio di programmazione e supportati dalla macchina concorrente).

- Se R è allocata come **risorsa dedicata**, (modalità A e C), essendo P l'unico processo che accede alla risorsa, non è necessario prevedere alcuna forma di sincronizzazione.

Tipi di interazione: competizione

- Risorse **condivise** e **allocate staticamente** (modalità B). La competizione tra processi avviene al momento dell'accesso alla risorsa. L'accesso esclusivo è garantito dal meccanismo di mutua esclusione utilizzato nel programmare le funzioni di accesso.

- Risorse **dedicate** e **allocate dinamicamente** (modalità C). La competizione tra processi avviene al momento dell'accesso alle operazioni del gestore (accesso esclusivo alle operazioni del gestore).

Tipi di interazione: cooperazione

- Risorse **condivise** e **allocate staticamente** (modalità B). Si ha cooperazione se uno dei processi memorizza in R informazioni che possono essere successivamente lette da un altro processo.

- Risorse **dedicate** e **allocate dinamicamente** (modalità C). Si ha cooperazione se un processo, acquisito dal gestore il diritto di operare su R, vi memorizza informazioni e se successivamente il diritto di accesso viene acquisito da un altro processo che legge le informazioni.

Specifica della sincronizzazione

Regione critica condizionale:

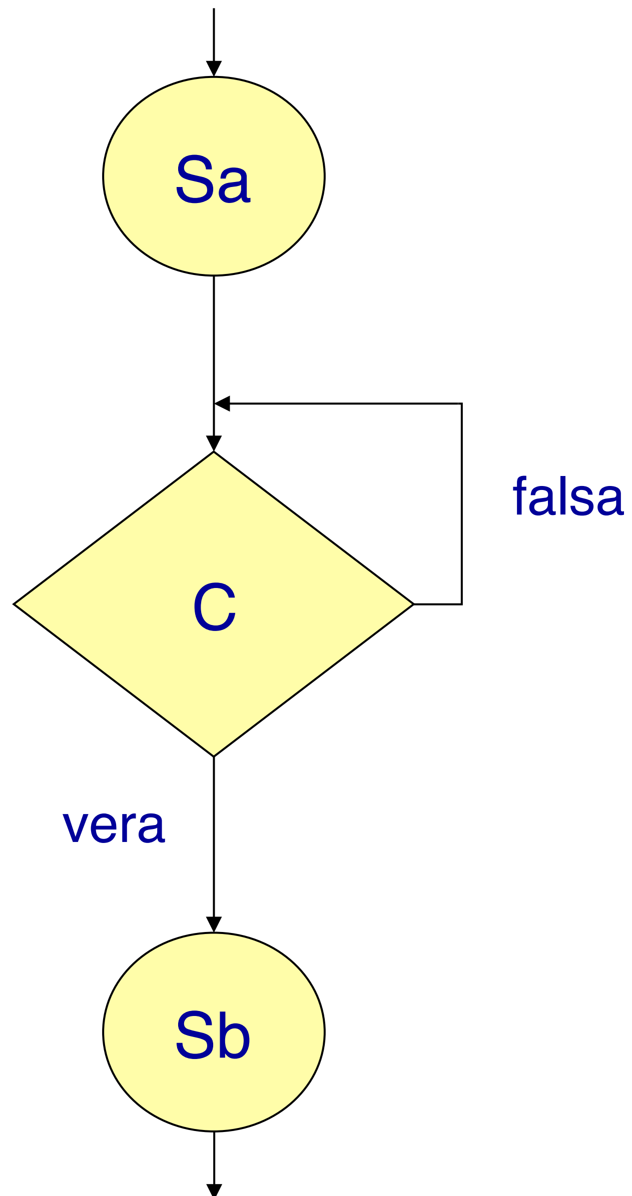
a) il corpo della ***region*** rappresenta un'operazione da eseguire sulla risorsa condivisa **R** e quindi costituisce una **sezione critica** che deve essere eseguita in **mutua esclusione** con le altre operazioni definite su R .

b) il corpo della ***region*** è costituito da due istruzioni da eseguire in sequenza: l'istruzione Sa e, successivamente, l'istruzione Sb.

In particolare,una volta terminata l'esecuzione di Sa viene valutata la condizione *C* :

- se *C* è **vera** l'esecuzione continua con Sb,
- se *C* è **falsa** il processo che ha invocato l'operazione ***attende*** che la condizione *C* diventi vera. A quel punto l'esecuzione della region può riprendere e essere completata mediante l'esecuzione di Sb.

Regione critica



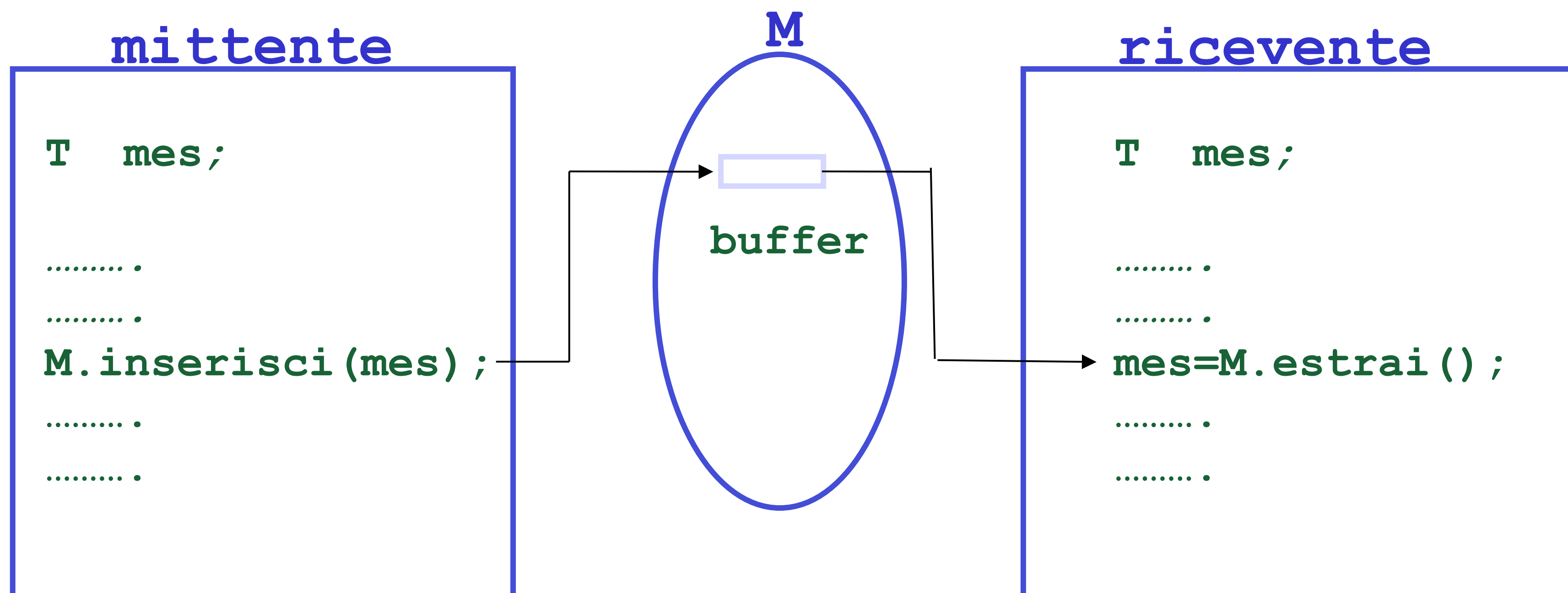
Regioni critiche: casi particolari

1) **region R << S; >>** Specifica della sola **mutua esclusione** senza che sia prevista alcuna forma di sincronizzazione diretta.

2) **region R << when(*C*) >>** Specifica di un semplice vincolo di sincronizzazione. Il processo deve attendere che *C* sia verificata prima di proseguire

3) **region R << when(*C*) S; >>** Specifica il caso in cui la condizione *C* di sincronizzazione caratterizza lo stato in cui la risorsa R deve trovarsi al fine di poter eseguire l'operazione S.

Scambio di informazioni tra processi



Scambio di informazioni tra processi: specifica della sincronizzazione

Supponendo che la risorsa M sia una struttura con i seguenti campi:

```
T  buffer;
boolean  pieno;
```

si ha:

```
void inserisci (T dato):
region M << when(pieno==false)
                buffer=dato;
                pieno=true;>>

T estrai():
region M << when(pieno==true)
                pieno=false;
                return buffer;>>
```

Meccanismi linguistici per la programmazione di interazioni

Per presentare *alcuni meccanismi linguistici* usati nella programmazione di interazioni tra processi si seguirà il seguente schema:

1. **Presentazione** del meccanismo
2. **Esempi** di uso del meccanismo
3. **Traduzione** del meccanismo linguistico in termini di un *meccanismo primitivo* di sincronizzazione fornito dal supporto a tempo di esecuzione (***nucleo***)

➔ Il meccanismo primitivo di riferimento è quello **semaforico** con le due operazioni **p** e **v** (Dijkstra).

Il problema della mutua esclusione

Mutua Esclusione

- **Il problema della mutua esclusione** nasce quando più di un processo alla volta può aver accesso a variabili comuni.
- La regola di mutua esclusione impone che le operazioni con le quali i processi accedono alle variabili comuni ***non si sovrappongano nel tempo***.
- **Nessun vincolo è imposto** ***sull'ordine*** con il quale le operazioni sulle variabili vengono eseguite.

t

Esempi di mutua esclusione

Esempio

- Due processi P1 e P2 hanno accesso ad una struttura *organizzata a* **pila** rispettivamente per inserire e prelevare dati.
- La struttura dati è rappresentata da un *vettore* ***stack*** i cui elementi costituiscono i singoli dati e da una *variabile* ***top*** che indica la posizione dell'ultimo elemento contenuto nella pila.
- I processi utilizzano le operazioni ***Inserimento (processo P1)*** *e* ***Prelievo (processo P2)*** per depositare e prelevare i dati dalla pila.

```
typedef ... item;
item stack[N];
int top=-1;

void Inserimento(item y)
{     top++;
      stack[top]=y;
}

item Prelievo()
{     item x;
      x= stack[top];
      top--;
      return x;
}
```

- L'esecuzione contemporanea di queste operazioni da parte dei processi può portare ad un uso scorretto della risorsa.
- Possibile sequenza di esecuzione delle due operazioni:

T0:	`top++;`	(P1)
T1:	`x=stack[top];`	(P2)
T2:	`top--;`	(P2)
T3:	`stack[top]=y;`	(P1)

- Viene assegnato a x un valore *non definito* e l'ultimo valore valido contenuto nella pila *viene cancellato* dal nuovo valore di y.
- Analogamente si avrebbe nel caso di esecuzione contemporanea di una qualunque delle due operazioni da parte dei due processi.

Istruzioni indivisibili

Azione atomica: esegue una trasformazione di stato *indivisibile*. Può esistere uno stato intermedio nella realizzazione dell'azione, ma non è rilevabile all'esterno.

Ipotesi:

- I valori dei tipi base (es. interi) sono memorizzati in parole di memoria che vengono lette e scritte in *modo atomico*.
- I valori sono manipolati caricandoli nei registri, operando sui *registri* e memorizzando il risultato in *memoria*
- Ciascun processo ha il proprio set di registri (v. *context switch*)
- Ogni *risultato intermedio* durante la valutazione di un'espressione viene valutato e memorizzato in registri o in memoria privata del processo in esecuzione (es. *stack privato*)

- Con questo modello di macchina se in un processo un'espressione ***e*** non fa riferimento a variabili modificate da un altro processo, la valutazione dell'espressione è *atomica* anche se risulta composta da azioni atomiche più elementari.

Infatti:

- nessuno dei valori da cui ***e*** dipende può cambiare durante la valutazione di ***e***;
- nessun altro processo può vedere valori temporanei che potrebbero essere creati durante la valutazione di ***e***.

Sezione Critica

La sequenza di istruzioni con le quali un processo accede e modifica un insieme di variabili comuni prende il nome di **sezione critica.**

Ad un insieme di variabili comuni possono essere associate *una sola sezione critica* (usata da tutti i processi) o *più sezioni critiche (classe di sezioni critiche).*

La ***regola di mutua esclusione*** stabilisce che:

Sezioni critiche appartenenti alla stessa classe devono escludersi mutuamente nel tempo.

oppure

Ad ogni istante può essere "in esecuzione" al più 1 sezione critica di ogni classe.

Realizzazione della regola di mutua esclusione

Tempificazione dell'esecuzione dei singoli processi da parte del programmatore:

Errori *time-dependent*

Inibizione delle interruzioni del processore durante l'esecuzione della sezione critica:

Soluzione parziale ed inefficiente

➔ ***Strumenti di sincronizzazione***

Schema Generale

- Ogni processo prima di entrare in una sezione critica deve chiedere l'autorizzazione eseguendo un serie di istruzioni che gli garantiscono *l'uso esclusivo della risorsa*, se questa è libera, oppure *ne impediscano* l'accesso se questa è già occupata (***PROLOGO***).

- Al completamento dell'azione il processo deve eseguire una sequenza di istruzioni per liberare la risorsa e dichiarare conclusa la sezione critica. (***EPILOGO***)

Soluzioni possibili

- **Algoritmiche** (es. Algoritmi di Dekker, Peterson, algoritmo del fornaio, ecc.): la soluzione non richiede la disponibilità di meccanismi di sincronizzazione (es. semafori, lock ecc.), ma sfrutta solo la possibilità di condivisione di variabili; l'attesa di un processo che trova la variabile condivisa già occupata viene modellata attraverso cicli di attesa attiva.

- **Hardware-based** (es. disabilitazione delle interruzioni, lock/unlock): il supporto è fornito direttamente all'architettura HW.

- **Strumenti di sincronizzazione realizzati dal nucleo della macchina concorrente** (es. semafori): prologo ed epilogo sfruttano strumenti di sincronizzazione che consentono l'effettiva sospensione dei processi in attesa di eseguire sezioni critiche.

Soluzioni Algoritmiche

Algoritmo di *Dekker (1965)*

E` stata la prima soluzione ***corretta*** al problema della mutua esclusione

Garantisce le proprietà di *mutua esclusione* e di assenza di *deadlock*.

```
int libero1 =0;
int libero2 =0;
int turno=1; /*dominio {1,2}*/
```

Il valore iniziale di *turno* è indifferente.

```
int libero1 =0;
int libero2 =0;
int turno=1; /*dominio {1,2}*/
```

```
/* processo P1: */
main()
{   ...
libero1=1;
while (libero2)
    if (turno==2)
    {   libero1=0;
        while(turno!=1);
        libero1=1;
    }
<sezione critica A>;
turno=2;
libero1=0;
...
}
```

```
/* processo P2: */
main()
{   ...
libero2=1;
while (libero1)
    if (turno==1)
    {   libero2=0;
        while(turno!=2);
        libero2=1;
    }
<sezione critica B>;
turno=1;
libero2=0;
...
}
```

Algoritmo di Peterson (1981)

- Risulta più semplice di quello di Dekker

- Le variabili utilizzate per la sincronizzazione sono:

```
int libero1 =0;
int libero2 =0;
int turno=1; /*dominio {1,2}*/
```

```
int libero1 =0;
int libero2 =0;
int turno=1; /*dominio {1,2}*/
```

```
/* processo P1: */
main()
{ ...
 libero1=1;
 turno=2;
 while(libero2 && turno==2);
 <sezione critica A>;
 libero1=0;
     ...
}
```

```
/* processo P2: */
main()
{ ...
 libero2=1;
 turno=1;
 while(libero1 && turno==1);
 <sezione critica B>;
 libero2=0;
     ...
}
```

Algoritmo del fornaio (Lamport, 1974)

- Ogni processo preleva un ticket con un numero di valore man mano crescente; entra nella sezione critica quando il numero in suo possesso è il minimo tra tutti quelli non serviti.

Nel caso di 2 processi P e Q:

- Le variabili utilizzate per la sincronizzazione sono:

```
int np=0, nq=0;
```

```c
int np=0;
int nq=0;
```

```c
/* processo P: */
main()
{ ...
 np=nq+1;
 while(np > nq && nq>0);
 <sezione critica A>;
 np=0;
     ...
}
```

```c
/* processo Q: */
main()
{ ...
 nq=np+1;
 while(nq > np && np>0);
 <sezione critica B>;
 nq=0;
     ...
}
```

Generalizzazione per n processi:

```
// var globali:
 boolean Entering [NUM_THREADS]= {false, false,… false};
 int Number[NUM_THREADS]= {0,0,0..};

// processo i-simo:
 Entering[i] = true;
 Number[i] = 1 + max(Number);
 Entering[i] = false;
 for (j = 1; j <= NUM_THREADS; j++) {
        while (Entering[j]);
        while ((Number[j] != 0) && (Number[j] < Number[i]));
}
  <sezione critica>

Number[i] = 0;
```

Algoritmo del fornaio

- Nessuna variabile è scritta da più di un processo.
- Difficile implementazione:
  - il massimo valore di number è concettualmente illimitato
- Ogni thread deve "chiedere" il numero a tutti gli altri.

Proprieta` della soluzione al problema della mutua esclusione

Proprietà necessarie

a) Sezioni critiche della stessa classe devono essere eseguite in modo ***mutuamente esclusivo***.

b) Quando un processo si trova all'esterno di una sezione critica ***non può rendere impossibile*** l'accesso alla stessa sezione (o a sezioni della stessa classe) ad altri processi.

c) Non deve essere possibile il verificarsi di situazioni in cui i processi *impediscono mutuamente* la prosecuzione della loro esecuzione (***deadlock***)

Proprietà desiderabili:

d) Se sono verificate le condizioni logiche per l'accesso ad una sezione critica da parte di un processo, questo non può essere *indefinitamente ritardato* a causa della esecuzione della stessa sezione (o di sezioni sella stessa classe) da parte di altri processi (***starvation***)

e) Devono essere eliminate *forme di attesa attiva (**busy waiting**)*. A differenza delle altre proprietà questa non riguarda la correttezza della soluzione ma *l'**efficienza** della realizzazione.*

Soluzioni hardware

Soluzioni hardware

- Nelle soluzioni precedenti si è supposto che l'hardware garantisca la mutua esclusione *solo a livello di lettura e scrittura di una singola parola di memoria.*
- L'*indivisibilità* è sempre assicurata solo riguardo all'ispezione o all'assegnamento di un valore ad una singola variabile comune.
- Molte macchine possiedono particolari istruzioni che consentono di *esaminare e modificare* il contenuto di una parola o di *scambiare* il contenuto di due parole *in un ciclo di memoria.*
- In questo caso è possibile dare una *semplice soluzione* al problema della mutua esclusione: lock e unlock

Disabilitazione delle interruzioni

1) Disabilitazione delle interruzioni durante le sezioni critiche:
   - Prologo: disabilitazione delle interruzioni
   - Epilogo: abilitazione delle interruzioni

```
/* struttura processo: */
main()
{       ...
        <disabilitazione delle interruzioni>;
        <sezione critica A>;
        <abilitazione delle interruzioni>;
        ...
}
```

Problemi:

- La soluzione è *parziale* in quanto è valida solo per sezioni critiche che operino sullo stesso processore.

- *Elimina* ogni possibilità di parallelismo.

- Rende *insensibile* il sistema ad *ogni stimolo esterno* per tutta la durata di qualunque sezione critica

Lock/unlock

- Molte macchine possiedono particolari istruzioni che consentono di *esaminare e modificare* il contenuto di una parola o di *scambiare* il contenuto di due parole *in un ciclo di memoria.*
- In questo caso è possibile dare una *semplice soluzione* al problema della mutua esclusione: lock e unlock

Lock e Unlock

```
void lock(int *x)
{      while (!*x);
       *x=0;
}
void unlock(int *x)
{      *x=1;
}
```

x *riferisce* una variabile logica associata ad una classe di sezioni critiche inizializzata al valore 1 (*true).*

(x=0 risorsa occupata, x=1 risorsa libera)

Ipotesi: lock e unlock vengono eseguite in modo atomico (o indivisibile)

Lock e Unlock

- Soluzione al problema della mutua esclusione:

```
int x=1;
```

```
/* processo P1: */
main()
{ ...
 lock(&x);
 <sezione critica A>;
 unlock(&x);
       ...
}
```

```
/* processo P2: */
main()
{ ...
 lock(&x);
 <sezione critica B>;
 unlock(&x);
       ...
}
```

Si noti che a differenza della `lock`, l'operazione `unlock` è **indivisibile.**

Ipotesi: lock(x) e unlock(x) **operazioni indivisibili.**

➔L'esecuzione contemporanea di due lock(x) (ciascuna su un diverso elaboratore) viene automaticamente sequenzializzata dall'hardware.

- I requisiti a),b),c) sono *soddisfatti*.
- Il soddisfacimento del requisito d) *non è implicito* nella soluzione. Per superare l'inconveniente della *starvation* occorre un'opportuna realizzazione del *meccanismo di arbitraggio* per l'accesso in memoria.
- Il requisito e) *non è soddisfatto*, essendo presente nella lock una forma di ***attesa attiva***

Indivisibilita` dell' operazione lock

Istruzione test and set (x):

Istruzione macchina che consente la lettura e la modifica di una parola in memoria in modo *indivisibile*, cioè in un *solo ciclo di memoria.*

```
int test-and-set(int *a)
{ int R;
        R=*a;
        *a=0;
  return R;
}
```

- Operazione lock(x):

```
void lock(int *x)
{       while (!test-and-set(x));
}
```

Implementazione di lock e unlock

Se il set di istruzioni dell'architettura prevede la test-and-set (`tsl`):

```
lock(x):
    tsl register, x      (copia x nel registro e pone x=0)
    cmp register, 1      (il contenuto del registro vale 1?)
    jne lock             ( se x=0 ricomincia il ciclo)
    ret                  (ritorna al chiamante;
                         accesso alla sezione critica)
```

```
unlock(x):
    move x,1             (inserisce 1 in x)
    ret                  (ritorna al chiamante)
```

Operazione EXCH A,X:

- Scambia i contenuti del registro A e della parola contenuta nell'indirizzo X in un *ciclo di memoria:*

```
void EXCH (int *a; int *b)
{ int temp;
  temp=*a;
  *a=*b;
  *b=temp;
}

void lock(int *x)
{   priv=0;
    do     EXCH (x,&priv)
    while (priv==0);
}
```

(*priv* è una variabile locale a ciascun processo)

Proprietà della soluzione basata su lock e unlock

- Si applica in ambiente *multiprocessore.*

- Va bene nel caso di sezioni critiche *molto brevi* (*attesa attiva*)

- Per ridurre al minimo questa attesa potrebbe essere opportuno *disabilitare il sistema di interruzioni* durante l'esecuzione della lock e unlock

Strumenti di sincronizzazione

Il semaforo

- Strumento linguistico di basso livello per risolvere problemi di sincronizzazione nel modello a memoria comune.

- Il meccanismo semaforico e` normalmente utilizzato a livello di macchina concorrente per realizzare strumenti di sincronizzazione di "più alto livello".

- Disponibile in librerie standard per la realizzazione di programmi concorrenti con linguaggi sequenziali (es. C).

Esempio: libreria LinuxThreads (standard Posix), java

Semaforo

Un semaforo e` definito come una variabile intera non negativa, cui è possibile accedere solo tramite le due operazioni *P* e *V*.

Dichiarazione di un oggetto di tipo semaphore:

semaphore s=i;

dove i ($i \geq 0$) è il valore iniziale.

Al tipo semaphore sono associati:

Insieme di valori= $\{ X \mid X \in N \}$

Insieme delle operazioni= {P,V}

Operazioni sul semaforo

- Un oggetto di tipo **semaphore** è condivisibile da due o più threads, che operano su di esso attraverso le operazioni **P** e **V**.

Semantica delle due operazioni:

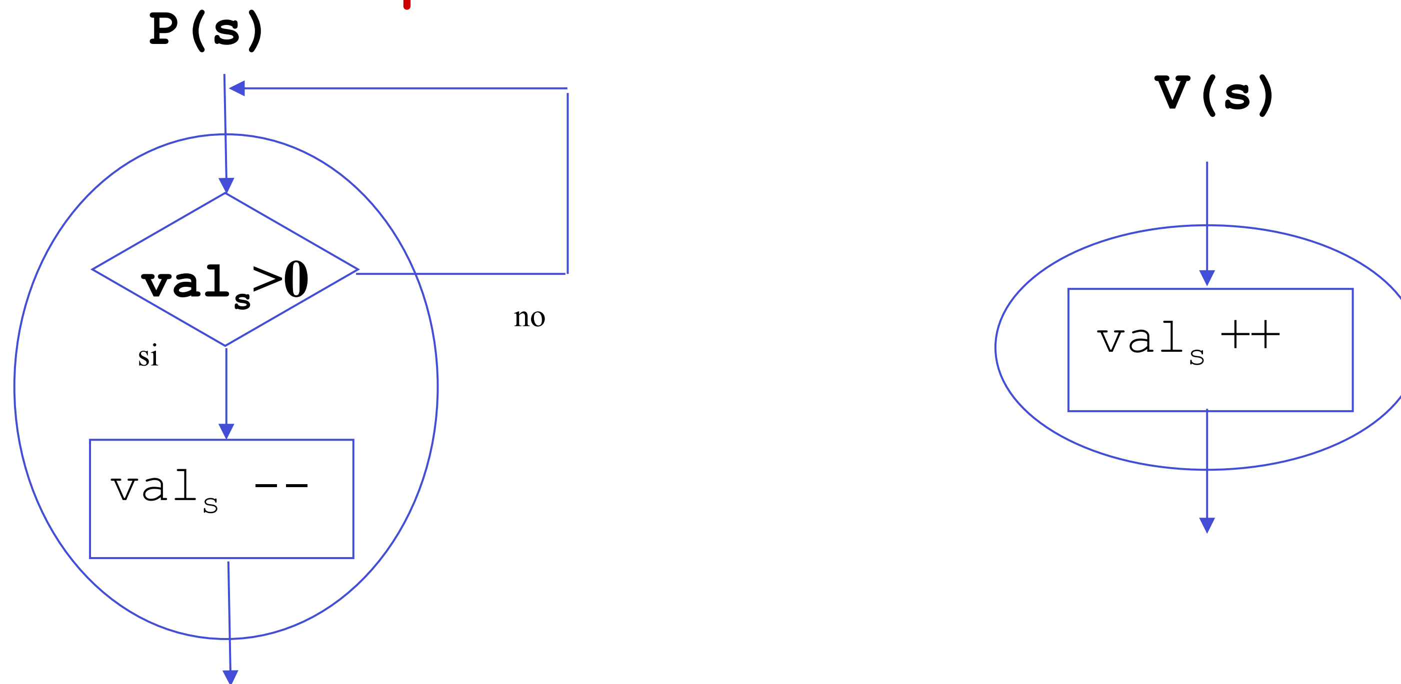
```
void P(semaphore s):
     region s << when(val_s>0) val_s--;>>

void V(semaphore s):
     region s << val_s++;>>
```

dove val_s rappresenta il valore del semaforo.

Essendo l'oggetto s condiviso, le due operazioni P e V vengono definite come **sezioni critiche** da eseguire in mutua esclusione. Le due operazioni devono essere eseguite in forma atomica.

Operazioni sul semaforo



Il semaforo viene utilizzato come strumento di sincronizzazione tra processi concorrenti:

- sospensione: P(s), val_s==0
- risveglio: V(s), se vi e` almeno un processo sospeso

Semaforo

Relazione di invarianza:

Siano:

- I_s: valore intero≥0 con cui il semaforo s viene inizializzato;
- nv_s: numero di volte che l'operazione V(s) è stata eseguita;
- np_s: numero di volte che l'operazione P(s) è stata completata;

Semaforo: relazione di Invarianza

$val_s = I_s + nv_s - np_s$

da cui ($val_s \geq 0$):

$np_s \leq I_s + nv_s$ relazione di invarianza

La relazione di invarianza è sempre soddisfatta, per ogni semaforo, qualunque sia il suo valore e comunque sia il programma concorrente che lo usa.

-> possibilità di dimostrare formalmente proprietà dei programmi concorrenti.

Uso dei semafori

- Il semaforo viene utilizzato come strumento di sincronizzazione tra processi concorrenti:
  - **sospensione**: P(s), s==0
  - **risveglio**: V(s), se vi e` almeno un processo sospeso

- Il semaforo è uno **strumento** generale, che consente la risoluzione di qualunque problema di sincronizzazione.

- Nel seguito verranno illustrati alcuni esempi notevoli di uso del meccanismo semaforico:
  - semafori di **mutua esclusione**
  - semafori **evento**
  - semafori **binari composti**
  - semafori **condizione**
  - semafori **risorsa**
  - semafori **privati**

Semafori di mutua esclusione

Semaforo binario. Può assumere solo i valori 0 e 1

```
class tipo_risorsa {
      <struttura dati di ogni istanza della classe>;
      semaphore mutex = 1;
      public void op1( )
      {     P(mutex); /*prologo*/
            <corpo della funzione op1>;
            V(mutex); /*epilogo*/
      }

      ...
      public void opn( )
      {     P(mutex);
            <corpo della funzione opn>;
            V(mutex);
      }
  }
  ...
  tipo_risorsa ris;
  ris.opi();
```

Mutua esclusione: dimostrazione di correttezza della soluzione proposta

- Dimostriamo che il protocollo (*):

```
Semaphore mutex=1;
..
p(mutex);
<sezione critica>
p(mutex);
...
```

Risolve correttamente il problema della mutua esclusione

Mutua esclusione: dimostrazione di correttezza della soluzione proposta

Hp: (*)

Th: Le condizioni necessarie sono soddisfatte:

a) Sezioni critiche della stessa classe devono essere eseguite in modo ***mutuamente esclusivo***.
b) Non deve essere possibile il verificarsi di situazioni in cui i processi *impediscono mutuamente* la prosecuzione della loro esecuzione (***deadlock***)
c) Quando un processo si trova all'esterno di una sezione critica ***non può rendere impossibile*** l'accesso alla stessa sezione (o a sezioni della stessa classe) ad altri processi.

Dimostrazione

- Dimostriamo la proprietà a):

Th: il numero Nsez dei processi nella sezione critica è sempre minore o uguale a uno:

$$0 \leq Nsez \leq 1$$

Dim:

$N_{sez} = np - nv$

Dalla relazione invariante:

$1+nv-np \geq 0 \Rightarrow np-nv \leq 1 \Rightarrow Nsez \leq 1$

Inoltre, poiché il protocollo impone che p(mutex) preceda v(mutex) in ogni processo, in ogni istante dell'esecuzione vale sempre la relazione:

$Np \geq nv \Rightarrow np-nv \geq 0 \Rightarrow Nsez \geq 0$. **c.v.d.**

Dimostrazione

- Dimostriamo la proprietà b):

Th: assenza di deadlock.

Dim: Per assurdo, se ci fosse deadlock:

1. tutti i processi sarebbero in attesa su p(mutex),
2. nessun processo sarebbe nella sezione critica:

1) => $Val_{mutex}=0$

2) $N_{sez}=np - nv = 0$

Ma sappiamo che:

$Val_{mutex}= I_{mutex} - np + nv$ => $Val_{mutex}= 1 - N_{sez}$

$\Rightarrow 0 = 1 - 0$ assurdo! **C.v.d.**

Dimostrazione

- Dimostriamo la proprietà c):

Th: un processo all'esterno della sezione critica non può impedire ad altri di entrare nella sezione critica.

Dim: Se nessun processo è nella sezione critica, deve essere possibile ad un qualunque processo di entrare nella sezione critica senza ritardi.

Se la sezione critica libera => $N_{sez}=np - nv = 0$

$Val_{mutex} = I_{mutex} - np + nv$ => $Val_{mutex} = 1$

$\Rightarrow$ p non è bloccante **C.v.d.**

Mutua esclusione tra gruppi di processi

- In alcuni casi è consentito a più processi di eseguire contemporaneamente la stessa operazione su una risorsa, ma non operazioni diverse.

- Data la risorsa condivisa **ris** e indicate con op_1, …, op_n le n operazioni previste per operare su ris, vogliamo garantire che più processi possano eseguire concorrentemente la stessa operazione op_i mentre non devono essere consentite esecuzioni contemporanee di operazioni diverse.

Anche in questo caso lo schema è:

```
public void opi() {
        <prologoi>;
        <corpo della funzione opi>;
        <epilogoi>;
         }
```

- $prologo_i$ deve sospendere il processo che ha chiamato l'operazione op_i se sulla risorsa sono in esecuzione operazioni diverse da op_i; diversamente deve consentire al processo di eseguire op_i .

- $epilogo_i$ deve liberare la mutua esclusione solo se il processo che lo esegue è l'unico processo in esecuzione sulla risorsa o è l'ultimo di un gruppo di processi che hanno eseguito la stessa op_i .

- NB: prologo ed epilogo sono sezioni critiche -> introduco un ulteriore semaforo di mutua esclusione m_i

```
semaphore mutex=1, mi,  =1;

public void opi( ) {
            P(mi);
            conti++;
            if (conti==1) P(mutex);
                          V(mi);
            <corpo della funzione opi>;
            P(mi);
            conti--;
            if (conti==0) V(mutex);
            V(mi);
      }
```

<prologo$_i$>

<epilogo$_i$>

Esempio: Problema dei lettori/scrittori

```
semaphore mutex = 1;
semaphore ml = 1
int contl = 0;
public void lettura(…) {
                    P(ml);
                    contl++;
                    if (contl==1) P(mutex);
                    V(ml);
                    <lettura del file >;
                    P(ml);
                    contl--;
                    if (contl==0) V(mutex);
                    V(ml);
}

public void scrittura(…) {
                    P(mutex);
                    <scrittura del file >;
                    V(mutex);
}
```

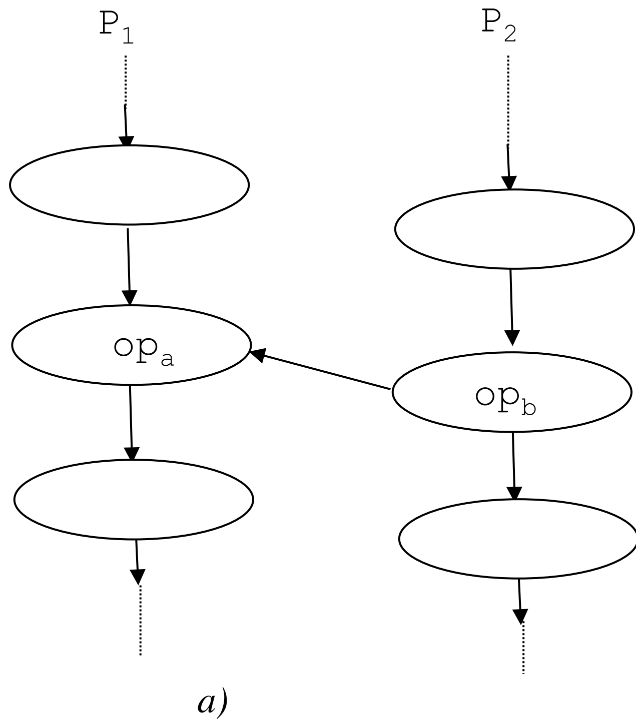
Semafori evento: scambio di segnali temporali

- semafori binari utilizzati per imporre un vincolo di precedenza tra le operazioni dei processi.

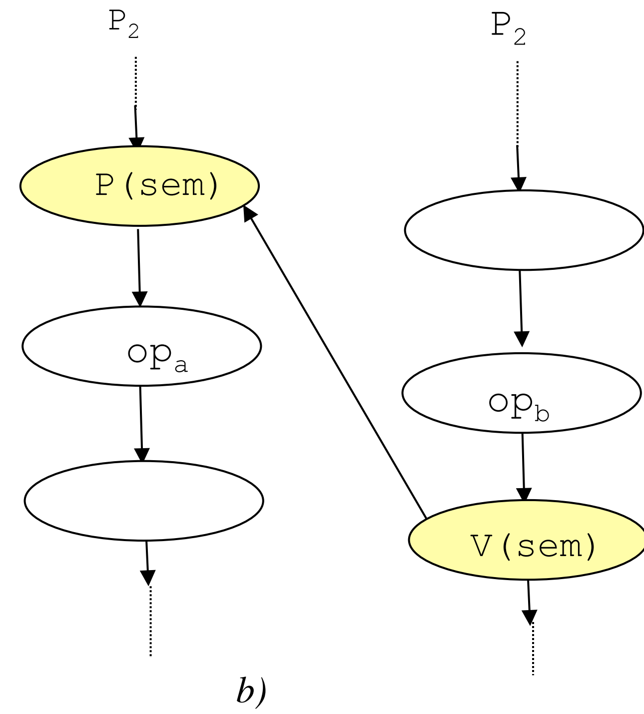
a

Esempio: op_a deve essere eseguita da P_1 solo dopo che P_2 ha eseguito op_b

➔ Introduciamo un semaforo sem inizializzato a zero:
- prima di eseguire op_a, P_1 esegue P(sem);
- dopo aver eseguito op_b, P_2 esegue V(sem).



a)



b)

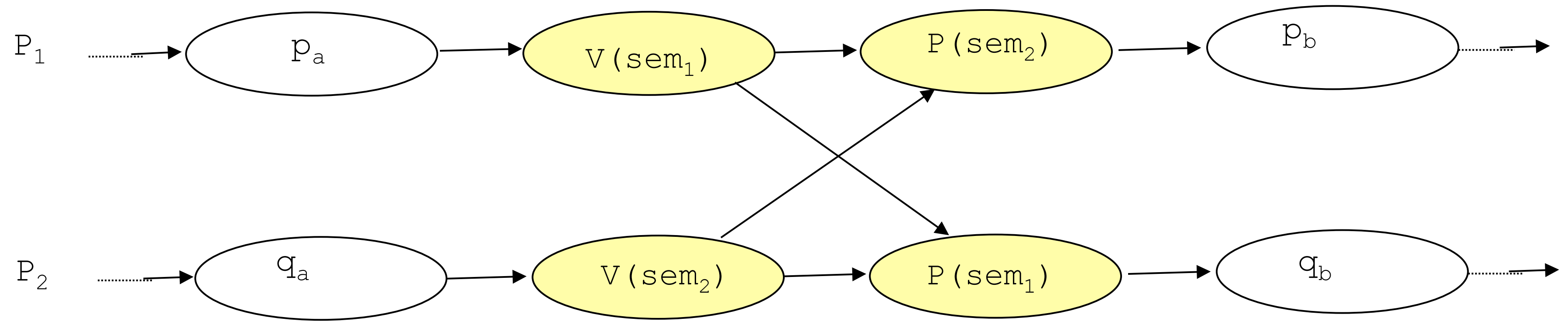
Problema del rendez-vous

Due processi P_1 e P_2 eseguono ciascuno due operazioni p_a e p_b il primo e q_a e q_b il secondo.

- vincolo di rendez-vous : l'esecuzione di p_b da parte di P_1 e q_b da parte di P_2 possono iniziare solo dopo che entrambi i processi hanno completato la loro prima operazione (p_a e q_a).

➔ Scambio di segnali temporali in modo simmetrico: ogni processo quando arriva all'appuntamento segnala di esserci arrivato e attende l'altro.

➔ Introduciamo due semafori evento sem1 e sem2


P1
pa
V(sem1)
P(sem2)
pb
P2
qa
V(sem2)
P(sem1)
qb

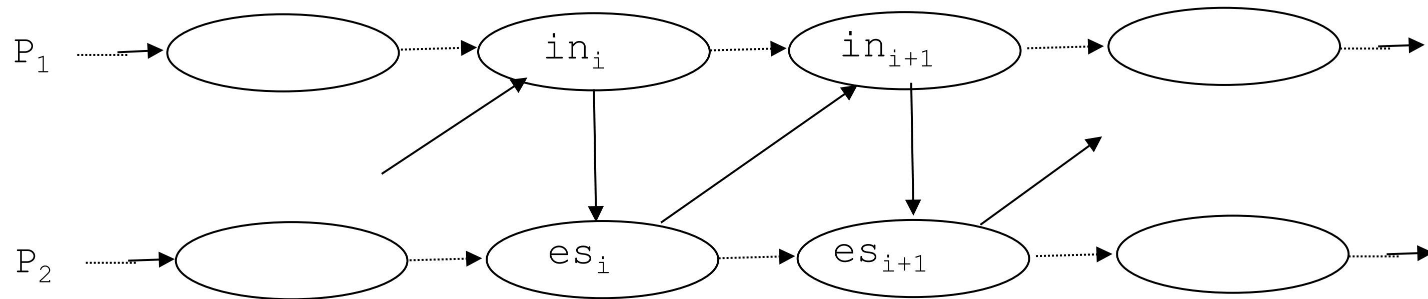
Semafori binari composti: scambio di dati

- Due processi P_1 e P_2 si scambiano dati di tipo T utilizzando una memoria (*buffer*) condivisa .

Vincoli di sincronizzazione:

- Accessi al buffer mutuamente esclusivi.
- P2 può prelevare un dato solo dopo che P1 lo ha ***inserito***.
- P1, prima di inserire un dato, deve attendere che P2 abbia ***estratto*** il precedente.

Scambio di dati: vincoli di precedenza



- Utilizziamo due semafori:
  - vu, per realizzare l'attesa di P1, in caso di buffer pieno;
  - pn, per realizzare l'attesa di P2, in caso di buffer vuoto;

```
void invio(T dato) {
      P(vu);
      inserisci(dato);
      V(pn);

}
```

```
T ricezione( ) {
T  dato;
P(pn);
dato=estrai();
V(vu);
return dato;
 }
```

buffer inizialmente vuoto
valore iniziale vu= 1
valore iniziale pn= 0

- pn e vn garantiscono da soli la mutua esclusione delle operazioni *estrai* ed *inserisci*.

- La coppia di semafori si comporta nel suo insieme come se fosse un unico semaforo binario di mutua esclusione.

Semaforo binario composto: un insieme di semafori usato in modo tale che:

  - uno solo di essi sia inizializzato a 1 e tutti gli altri a zero.
  - ogni processo che usa questi semafori esegue sempre sequenze che iniziano con la P su uno di questi e termina con la V su un altro.

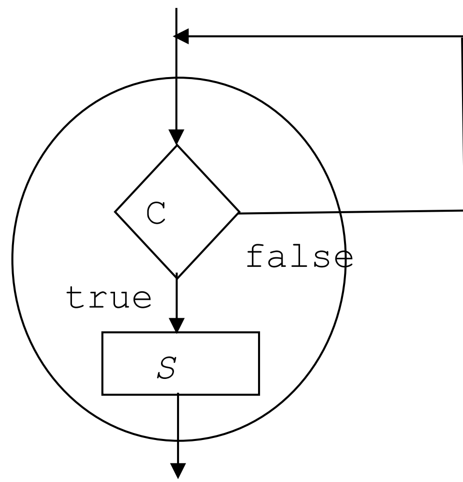
Semafori condizione

In alcuni casi, l'esecuzione di un'istruzione S1 su una risorsa R e` subordinata al verificarsi di una condizione C:

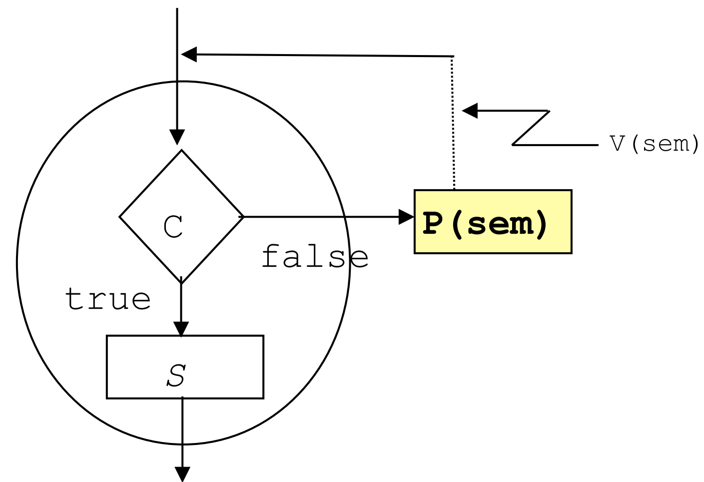
```
void op1( ): region  R << when(C) S1;>>
```

- op1( ) è una regione critica, dovendo operare su una risorsa condivisa R.
- S1 ha come precondizione la validità della condizione logica C.

➔ Il processo deve sospendersi se la condizione non è verificata e deve uscire dalla regione per consentire ad altri processi di eseguire altre operazioni su R per rendere vera la condizione C.



(a)



(b)

- Lo schema (a) presuppone una forma di attesa attiva da parte del processo che non trova soddisfatta la condizione.
- Nello schema (b) si realizza la *region* sospendendo il processo sul semaforo sem da associare alla condizione.(semaforo condizione):
  - è evidentemente necessaria un'altra operazione op2 che, chiamata da un altro processo, modifichi lo stato interno di R in modo che *C* diventi vera (*C* e` una *post condizione* di op2).
  - Nell'ambito di op2 viene eseguita la V(sem) per risvegliare il processo.

Struttura dati della risorsa R

```
semaphore mutex = 1;
semaphore sem = 0;
int csem=0
```

```
public void op1( )
{ P(mutex);
  while (!C)
  {   csem ++;
      V(mutex);
      P(sem);
      P(mutex);
  }
  S1;
  V(mutex);
}
```

```
public void op2( )
{   P(mutex);
    S2 ;
    if (csem>0)
    {   csem --;
        V(sem)
    }
    V(mutex);
}
```

Schema con *attesa circolare*

Struttura dati della risorsa R

```
semaphore mutex = 1;
semaphore sem = 0;
int csem = 0;
```

```
public void op1( )
{       P(mutex);
        if (!C) {
                csem ++;
                V(mutex);
                        P(sem);
                        csem --;

        }
        S1;
        V(mutex);
}
```

```
public void op2( )
{       P(mutex);
        S2;
        if(C && csem>0)
                V(sem);
        else V(mutex);
}
```

Schema con *passaggio di testimone*

Semafori condizione: considerazioni sulle soluzioni

- Il secondo schema è più efficiente del primo, ma consente di risvegliare un solo processo alla volta poichè ad uno solo può passare il diritto di operare in mutua esclusione.

- La condizione C verificata all'interno di op1 deve essere verificabile anche all'interno di op2 (deve essere una post condizione di S2). Ciò significa che non deve contenere variabili locali o parametri della funzione op1.

Esempio di uso di semafori condizione: Gestione di un pool di risorse equivalenti

Si consideri un insieme (*pool*) di N risorse tutte uguali:

- Ciascun processo può operare su una qualsiasi risorsa del pool purché libera.

Necessità di un gestore che mantenga aggiornato lo stato delle risorse:

1. Ciascun processo quando deve operare su una risorsa chiede al gestore l'allocazione di una di esse.
2. Il gestore assegna al processo una risorsa libera (se esiste), in modo dedicato, passandogli l'indice relativo.
3. Il processo opera sulla risorsa senza preoccuparsi della mutua esclusione.
4. Al termine il processo rilascia la risorsa al gestore.

Gestione di un pool di risorse equivalenti: operazioni del gestore

```
int richiesta(): region G <<
      when (<ci sono risorse disponibili>)
      <scelta di una risorsa disponibile>;
      int i = <indice della risorsa scelta>;
      <registra che la risorsa di indice i
      non è più disponibile>;
      return i;
      >>

void rilascio(int r): region G <<
      <registra che la risorsa r-esima
      è di nuovo disponibile>
      >>
```

Realizzazione:

```
class tipo_gestore
{       semaphore  mutex = 1; /*sem. di mutua esclusione*/
        semaphore  sem = 0; /*semaforo condizione*/
        int csem = 0; /*contatore dei proc. sospesi su sem */
        boolean  libera[N]; /*indicatori di risorsa libera*/
        int  disponibili = N; /*contatore risorse libere*/
        /*inizializzazione:*/
        {for(int i=0; i < N; i++)libera[i]=true;}
        public int richiesta()
        {       int i=0;
                P(mutex);
                if (disponibili == 0)
                {       csem ++;
                        V(mutex);
                        P(sem):
                        csem --; }
                while(!libero[i]) i++;
                libero[i]=false;
                disponibili --;
                V(mutex);
                return i;
        } /* continua..*/
```

```
    public void rilascio (int r)
    {       P(mutex);
            libero[r]=true;
            disponibili ++;
            if (csem>0) V(sem);
            else V(mutex);
    }
} /* fine classe tipo_gestore */
```

```
tipo_gestore G; /*  def. gestore*/

/*struttura del generico processo che vuole accedere a
una risorsa del pool: */
process P{
      int   ris;
      ...
      ris = G.richiesta( );
      < uso della risorsa di indice ris>
      G.rilascio (ris) ;
      ...
}
```

Semafori risorsa

- Semafori generali. Possono assumere qualunque valore ≥ 0.
- Vengono impiegati per realizzare l'allocazione di risorse equivalenti: il valore del semaforo rappresenta il numero di risorse libere.

Esempio: gestione di un pool di risorse equivalenti.

- Unico semaforo n_ris inizializzato con un valore uguale al numero di risorse da allocare.
- Esecuzione di P(n_ris) in fase di allocazione e di V (n_ris) in fase di rilascio

```
class tipo_gestore {
      semaphore  mutex = 1; /*semaforo di mutua esclusione*/
      semaphore n_ris = N; /*semaforo risorsa*/
      boolean  libero[N]; /*indicatori di risorsa libera*/

      {for(int i=0; i < N; i++)
            libera[i]=true;} /*inizializzazione*/

      public int richiesta() {
            int i=0;
            P(n_ris);
            P(mutex);
            while(libero[i]==false) i++;
            libero[i]=false;
            V(mutex);
            return i; }

      public void rilascio (int r) {
            P(mutex);
            libero[r]=true;
            V(mutex);
            V(n_ris); }
}
```

Semafori risorsa: problema dei produttori/ consumatori

Buffer di n elementi (di tipo T), strutturato come una coda:

```
coda_di_n_T  buffer;
semaphore    pn = 0; //semaforo risorsa el.pieno
semaphore    vu = n; //semaforo risorsa el.vuoto
semaphore     mutex = 1;
```

```
void invio(T dato) {
      P(vu);
      P(mutex);
      buffer.inserisci(dato);
      V(mutex);
      V(pn);
}
```

```
T ricezione() {
      T dato;
      P(pn);
      P(mutex);
     dato = buffer.estrai();
      V(mutex);
      V(vu);
      return dato;
 }
```

- il produttore richiede l'allocazione di una risorsa "*elemento vuoto*"
- il consumatore richiede l'allocazione di una risorsa "*elemento pieno*"

Realizzazione del buffer: coda circolare.

```
class coda_di_n_T {
    T  vettore[n];
    int primo = 0;
    ultimo = 0;

    public void inserisci(T dato) {
        vettore[ultimo] = dato;
        ultimo = (ultimo+1)%n;
    }

    public T estrai( ) {
        T dato= vettore[primo];
        primo = (primo+1)%n;
        return dato;
    }
}
```

Semafori privati: specifica di strategie di allocazione

Condizione di sincronizzazione:

Qualora si voglia realizzare una determinata politica di gestione delle risorse, la decisione se ad un dato processo sia consentito proseguire l'esecuzione dipende dal verificarsi di una condizione, detta condizione di sincronizzazione.

- La condizione è espressa in termini di variabili che rappresentano lo stato della risorsa e di variabili locali ai singoli processi.
- Più processi possono essere bloccati durante l'accesso ad una risorsa condivisa, ciascuno in attesa che la propria condizione di sincronizzazione sia verificata.

- In seguito alla modifica dello stato della risorsa da parte di un processo, le condizioni di sincronizzazione di alcuni processi bloccati possono essere contemporaneamente verificate.

Problema: quale processo mettere in esecuzione (accesso alla risorsa mutuamente esclusivo)?

➔ Definizione di una politica per il risveglio dei processi bloccati.

- Nei casi precedenti la condizione di sincronizzazione era particolarmente semplificata (vedi mutua esclusione) e la scelta di quale processo riattivare veniva effettuata tramite l'algoritmo implementato nella V.
- Normalmente questo algoritmo, dovendo essere sufficientemente generale ed il più possibile efficiente, coincide con quello FIFO.

Esempio 1

Su un buffer da N celle di memoria più produttori possono depositare messaggi di dimensione diversa.

Politica di gestione: tra più produttori ha priorità di accesso quello che fornisce il messaggio di dimensione maggiore.

- ➔ finché un produttore il cui messaggio ha dimensioni maggiori dello spazio disponibile nel buffer rimane sospeso, nessun altro produttore può depositare un messaggio anche se la sua dimensione potrebbe essere contenuta nello spazio libero del buffer.

Condizione di sincronizzazione: il deposito può avvenire se c'è sufficiente spazio per memorizzare il messaggio e non ci sono produttori in attesa.

Il prelievo di un messaggio da parte di un consumatore prevede la riattivazione tra i produttori sospesi, di quello il cui messaggio ha la dimensione maggiore, sempre che esista sufficiente spazio nel buffer.

Se lo spazio disponibile non è sufficiente nessun produttore viene riattivato.

Esempio 2: pool di risorse equivalenti + priorità

Un insieme di processi utilizza un insieme di risorse comuni R1,R2,..Rn. Ogni processo può utilizzare una qualunque delle risorse.

Condizione di sincronizzazione: l'accesso e` consentito se esiste una risorsa libera.

- A ciascun processo è assegnata una priorità.
- In fase di riattivazione dei processi sospesi viene scelto quello cui corrisponde la massima priorità

SEMAFORO PRIVATO

Un semaforo s si dice **privato** per un processo quando solo tale processo può eseguire la primitiva P sul semaforo s.

La primitiva V sul semaforo può essere invece eseguita anche da altri processi.

Un semaforo privato viene inizializzato con il valore zero.

Uso dei semafori privati

I semafori privati possono essere utilizzati per realizzare particolari politiche di allocazione di risorse:

- il processo che acquisisce la risorsa puo` (se la condizione di sincronizzazione non e` soddisfatta) eventualmente sospendersi sul suo semaforo privato
- chi rilascia la risorsa, risveglierà uno tra i processi sospesi (in base alla politica scelta) mediante una V sul semaforo privato del processo prescelto.

Allocazione di risorse con particolari strategie: primo schema

```
class tipo_gestore_risorsa{
      <struttura dati del gestore>;
      semaphore mutex =1;
      semaphore priv[n] = {0,0,..0}; /*semafori privati*/

      public void  acquisizione (int i)
      {     P(mutex);
            if(<condizione di sincronizzazione>){
                  <allocazione della risorsa>;
                  V(priv[i]);
            }
            else
                  <registrare la sospensione del processo>;
            V(mutex);
            P(priv[i]);
      }
```

```
    public void  rilascio( )
    {     int i;
          P(mutex);
          <rilascio della risorsa>;
          if (<esiste almeno un processo sospeso per
              il quale la condizione di sincronizz.
              è soddisfatta>)
          {     <scelta (fra i processi sospesi) del
                processo Pi da riattivare>;
                <allocazione della risorsa a Pi>;
                <registrare che Pi non è più
                sospeso>;
                V(priv[i]);
          }
          V (mutex);
    }
}
```

Allocazione di risorse

Proprieta` della soluzione:

a) La sospensione del processo, nel caso in cui la condizione di sincronizzazione non sia soddisfatta, non può avvenire entro la sezione critica in quanto ciò impedirebbe ad un processo che rilascia la risorsa di accedere a sua volta alla sezione critica e di riattivare il processo sospeso.

La sospensione avviene al di fuori della sezione critica.

b) La specifica del particolare algoritmo di assegnazione della risorsa non è opportuno che sia realizzata nella primitiva V. Nella soluzione proposta è possibile programmare esplicitamente tale algoritmo scegliendo in base ad esso il processo da attivare ed eseguendo V sul suo semaforo privato.

Lo schema presentato può, in certi casi, presentare degli inconvenienti.

1. l'operazione P sul semaforo privato viene sempre eseguita anche quando il processo richiedente non deve essere bloccato.

2. Il codice relativo all'assegnazione della risorsa viene duplicato nelle procedure acquisizione e rilascio

➔ Si può definire uno schema che non ha questi inconvenienti.

Allocazione di risorse: secondo schema

```
class tipo_gestore_risorsa{
      <struttura dati del gestore>;
      semaphore mutex = 1;
      semaphore priv[n] = {0,0,..0}; /*semafori privati */

      public void acquisizione (int i)
      {     P(mutex);
            if(! <condizione di sincronizzazione>)
            { <registrare la sospensione del processo>;
              V(mutex);
              P(priv[i]);
              <registrare che il processo
               non è più sospeso>;
            }
            <allocazione della risorsa>;
            V(mutex);
      }
```

```
public void rilascio( )
{       int i;
        P(mutex);
        <rilascio della risorsa>;
        if (<esiste almeno un processo sospeso per
            il quale la condizione di sincronizz.
            è soddisfatta>)
        {       <scelta del processo Pi da riattivare>;
                V(priv[i]);
        }
        else V(mutex);
}
}
```

➔il risveglio segue lo schema del *passaggio di testimone*

Commento

A differenza della soluzione precedente, tuttavia, in questo caso risulta più complesso realizzare la riattivazione di più processi per i quali risulti vera contemporaneamente la condizione di sincronizzazione.

Lo schema prevede infatti che il processo che rilascia la risorsa attivi al più un processo sospeso, il quale dovrà a sua volta provvedere alla riattivazione di eventuali altri processi.

Soluzione ai problemi precedenti: esempio 1

```
class buffer {
      int richiesta[num_proc]=0;/*richiesta[i]= numero di
                                   celle richieste da Pi*/
      int sospesi=0; /*numero dei processi prod.sospesi*/
      int vuote=n;   /*numero di celle vuote del buffer*/
      semaphore mutex=1;
      semaphore priv[num_proc]={0,0,..0};

      public void acquisizione(int m, int i)
      /* m dim. messaggio, i id.del processo chiamante */
      {     P(mutex);
            if (sospesi==0 && vuote>=m)
            {     vuote=vuote-m;
                  <assegnaz. di m celle al proc.i >;
                  V(priv[i]);
            }
            else
            {     sospesi++;
                  richiesta[i]=m;
            }
            V(mutex);
            P(priv[i]);
      }
```

```
public void rilascio(int m) /* m num. celle rilasciate*/
	{	int k;
		P(mutex);
		vuote+=m;
		while (sospesi!=0)
		{	<individuazione tra i processi sospesi del
			processo  Pk con la max richiesta> ;
			if (richiesta[k]<=vuote)
			{	vuote=vuote-richiesta[k];
				<ass. a Pk delle celle richieste>;
				richiesta[k]=0;
				sospesi--;
				V(priv[k]);
			}
			else break; /* fine while */
		}
		V(mutex);
	}
}
```

Soluzione esempio 2

Soluzione: *introduzione delle seguenti variabili:*

- ***PS[i]***: variabile logica che assume il valore vero se il processo Pi è sospeso; il valore falso diversamente.
- ***libera[j]:*** variabile logica che assume il valore falso se la risorsa j-esima è occupata; il valore vero diversamente.
- ***disponibili:*** esprime il numero delle risorse non occupate;
- ***sospesi:*** e` il numero dei processi sospesi;
- ***mutex:*** semaforo di mutua esclusione
- ***priv[i]:*** il semaforo privato del processo Pi.

```
class tipo_gestore {
        semaphore  mutex=1;
        semaphore priv[num_proc]={0,0,…0} /*sem. privati */
        int sospesi=0; /*numero dei processi sospesi*/
        boolean PS[num_proc]={false, false,…, false};
        int disponibili=num_ris; /*numero di risorse disp.*/
        boolean libera[num_ris]={true, true,…, true};

        public int richiesta(int proc)
        {       int i =0;
                P(mutex);
                if (disponibili==0)
                {       sospesi ++;
                        PS[proc]=true;
                        V(mutex);
                        P(priv[proc]);
                        PS[proc]=false;
                        sospesi -- ;
                }
                while (! libera[i]) i++;
                libera[i]=false;
                disponibili--;
                V(mutex );
                return i;
        }
```

```
public void rilascio (int r)/* r indice risorsa ril. */
{       P(mutex);
        libera[r]=true;
        disponibili++;
        if (sospesi>0)
        {  <seleziona il processo Pj a massima
          priorità tra quelli sospesi utilizzando PS>;
           V(priv[j]);
        }
        else  V(mutex);
}
}
```

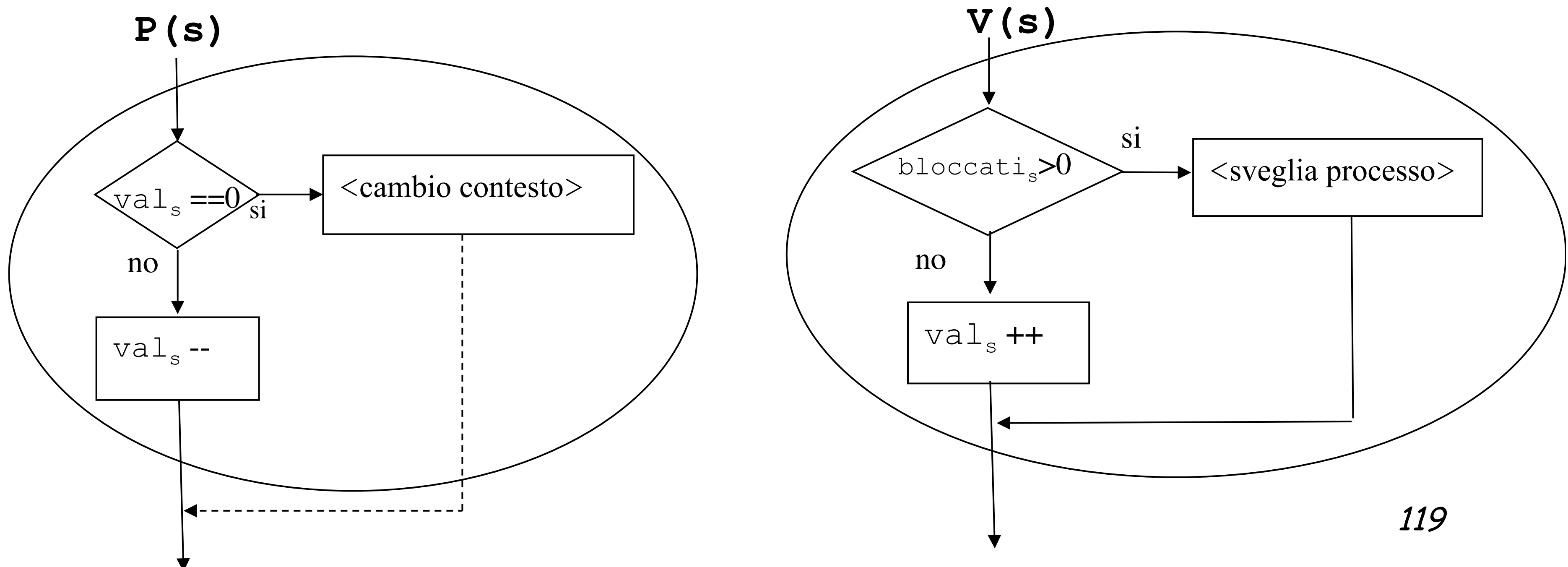
Considerazioni sulle soluzioni presentate

- Ogni processo che nella fase di acquisizione della risorsa trova la condizione di sincronizzazione non soddisfatta, deve lasciare traccia in modo esplicito della sua sospensione entro la sezione critica.
- Il processo che libera la risorsa deve infatti eseguire la primitiva V(priv(i)) solo se esistono processi sospesi. In tutte le soluzioni è stata introdotta un'apposita variabile per indicare il numero dei processi sospesi.
- La fase di assegnazione di una risorsa ad un processo è separata dalla fase di uso della risorsa stessa.
- Occorre quindi lasciare traccia in modo esplicito entro la sezione critica della assegnazione e quindi della non disponibilità della risorsa. Nel primo esempio viene decrementata la variabile vuote; nel secondo viene modificata la variabile disponibili.

Realizzazione dei semafori

In sistemi operativi multiprogrammati, il semaforo viene realizzato dal kernel, che, sfruttando i meccanismi di gestione dei processi (sospensione e riattivazione) elimina la possibilità di attesa attiva.

E` possibile definire la P e la V nel modo seguente, garantendo comunque la validita` delle proprieta` del semaforo (s.p.d).



Descrittore di un semaforo:

```
typedef struct{
        int contatore;
        coda  queue;
        }semaforo;
```

Una p su un semaforo con contatore a 0, **sospende** il processo nella coda queue, altrimenti contatore viene decrementato.

Una v su un semaforo la cui coda queue non è vuota, **estrae** un processo dalla coda, altrimenti incrementa il contatore

Architettura monoprocessore

Hp: interruzioni disabilitate (per garantire l'atomicità di p e v)

Implementazione delle operazioni p e v:

```
void P(semaphore s)
{       if(s.contatore==0)
               < sospensione del processo nella coda associata a s>;
               }
        else s. contatore--;
}


void V (semaphore s)
{if(s.queue!=NULL)
   <estrazione del primo processo da s.queue,
        che viene riportato nello stato di ready>
   else s.contatore ++;
}
```

Implementazione

L'implementazione di p e v è parte del nucleo della macchina concorrente e dipende dal tipo di architettura hardware (monoprocessore, multiprocessore, ecc.) e da come il nucleo rappresenta e gestisce i processi concorrenti.

V. Nucleo di un sistema concorrente