

## SEMAFORI

1

### Semaforo

Una variabile di *tipo semaforico* viene definita come una variabile *intera non negativa*, cui è possibile accedere *solo* tramite le due operazioni **wait** e **signal** definite nel seguente modo:

```
wait(s):    while (!s);
            s--;

signal(s):   s++;
```

- L'operazione **wait** ritarda il processo fino a che il valore del semaforo diventa maggiore di 0 e quindi decrementa tale valore di 1.
- L'operazione **signal** incrementa di 1 il valore del semaforo.
- Le due operazioni sono **atomiche**. Il valore del semaforo viene modificato da un *solo processo* alla volta.

2

Il valore di un semaforo  $s$  è legato al numero delle operazioni *wait* e *signal* eseguite su di esso dalla relazione:

$$val(s) = s_0 + ns(s) - nw(s)$$

dove:

- **val(s)** valore del semaforo  $s$
- $s_0$  valore iniziale di  $s$ ;
- **ns(s)** numero di volte che è stata eseguita la *signal(s)*;
- **nw(s)** numero di volte che è stata completata la *wait(s)*.
- Essendo, per definizione, **val(s)  $\geq 0$** , si ha:

$$nw(s) \leq ns(s) + s_0$$

3

- La relazione  **$nw(s) \leq ns(s) + s_0$**  è **invariante** rispetto all'esecuzione di *wait* e *signal* (sempre vera qualunque sia il numero di primitive eseguite).
- La proprietà può essere utilizzata per **verificare** che un'interazione tra processi, programmata mediante il meccanismo semaforico, *avvenga correttamente*.

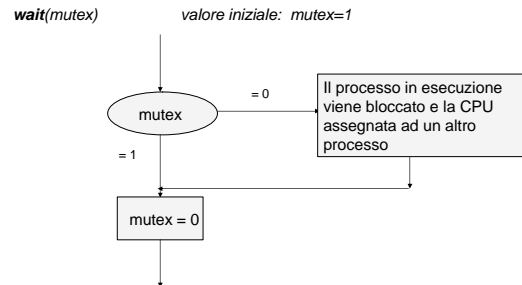
4

### Realizzazione dei semafori

- Il meccanismo di implementazione del costruito semaforo deve consentire:
  - **eliminazione** di ogni forma di **attesa attiva** dei processi (v. definizione della *wait*): sospensione del processo che non può proseguire l'esecuzione in una coda associata al semaforo.
  - **eliminazione di forme di starvation** (attesa indefinita di un processo): scelta FIFO del processo da risvegliare.

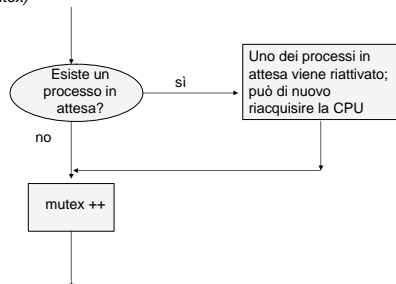
5

### Esempio di mutua esclusione



6

*signal(mutex)*



7

### Realizzazione dei semafori

Al semaforo sono associati:

- un valore intero non negativo con valore iniziale  $\geq 0$
- una coda *Qs* nella quale sono posti i descrittori dei processi che attendono l'autorizzazione a procedere.

```
typedef struct{int value;
               queue Qs;} semaphore;
```

8

### Realizzazione di wait e signal

Wait e signal possono essere realizzate come segue:

```
void wait(semaphore *s) {
    if (s->value==0)
        <il processo viene sospeso ed
        il suo descrittore
        inserito in s->Qs>
    s->value--;
}

void signal (semaphore *s) {
    if (<s->Qs non e` vuota>)
        <il descrittore del primo processo
        viene rimosso dalla coda ed il suo
        stato modificato in pronto>
    s->value++;
}
```

– L'esecuzione della signal non comporta concettualmente nessuna modifica nello stato del processo che l'ha eseguita.

– Scelta del processo da risvegliare tramite politica **FIFO**

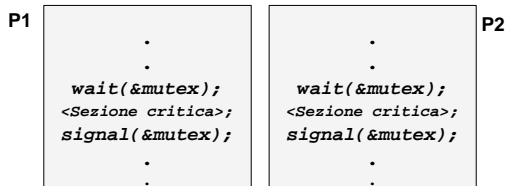
9

- *wait* e *signal* : sezioni critiche → devono essere **azioni indivisibili** (azioni atomiche).
- Analisi e modifica del valore del semaforo ed eventuale sospensione o riattivazione di un processo devono avvenire in **modo indivisibile**.
- Durante un'operazione sul semaforo nessun altro processo può accedere al semaforo fino a che l'operazione è completata o bloccata.

10

### Soluzione al problema della mutua esclusione

```
semaphore mutex;
mutex.value=1;
```



- *mutex* semaforo (binario) di mutua esclusione (0,1), con valore iniziale uguale a 1.

→ Qualunque sia la sequenza di esecuzione dei processi, la soluzione è sempre corretta.

11

### Dimostrazione

**Th:** Il numero  $n$  dei processi presenti contemporaneamente nella sezione critica  $S$  deve essere 0 o 1.

- Si ha:

$$n = nw(mutex) - ns(mutex)$$

- La relazione  $nw(s) \leq ns(s) + s0$  diventa in questo caso:

$$nw(mutex) \leq ns(mutex) + 1$$

- Dalle due relazioni si ha:

$$n = nw(mutex) - ns(mutex) \leq 1$$

12

Poiché *wait(s)* precede sempre *signal(s)* si ha:

$$nw(mutex) - ns(mutex) \geq 0$$

Quindi si ha:

$$0 \leq n \leq 1 \quad \boxed{\text{cvd}}$$

13

### Dimostrazione

**Th:** Un processo viene bloccato in ingresso solo se la sezione critica è occupata da un altro processo.

Un processo è ritardato *solo* se il valore di *mutex* è zero.

HP: *mutex.value*=0

La relazione  $nw(mutex) \leq ns(mutex) + 1$  diventa :

$$nw(mutex) = ns(mutex) + 1$$

Il numero delle operazioni *wait* eseguite con successo su *mutex* eccede il numero delle operazioni *signal* su *mutex* di 1.

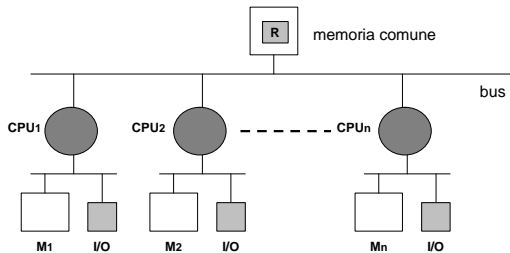
Quindi:

**un processo si trova entro la sezione critica.** cvd

14

### Mutua esclusione: alcuni problemi

1. E' sempre necessario usare *wait* e *signal* per assicurare la mutua esclusione (**overhead**)?
2. Come si ottiene la **non interrompibilità** nel caso di sistemi multiprocessori?



15

### Soluzione al primo problema

**Ipotesi:** sezioni critiche "sufficientemente brevi".

a) Sistema **monoprocessore**:



16

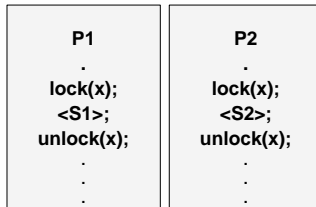
### Soluzione al primo problema

#### b) Sistema multiprocessore: uso di lock e unlock

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

/* x=0 risorsa
occupata;
x=1 risorsa libera
*/
```

int x=1;



17

- Problema dell'attesa attiva (*busy waiting*)
- Nell'ipotesi che l'hardware garantisca la mutua esclusione solo a livello di lettura o scrittura di una cella di memoria **solo unlock è indivisibile**

#### → Istruzione di **test and set lock (tsl)**

- Copia il valore di x in un registro ed inserisce in x il valore 0, in modo **indivisibile**
- La CPU che esegue *tsl* tiene occupato il bus di memoria per impedire ad altre CPU di accedere alla memoria

18

lock(x):

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

unlock(x):

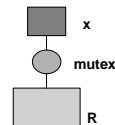
```
move x, 1          (inserisce 1 in x)
ret                (ritorna al chiamante)
```

19

### Soluzione al secondo problema

Nel caso generale in cui *wait* e *signal* siano eseguite su processori diversi si ha:

void wait(..mutex)	void signal(..mutex)
{	{
lock(x);	lock(x);
/*codice della	/*codice della
wait */	signal */
unlock(x);	unlock(x);
}	}



20

### Cooperazione tra processi concorrenti

- Scambio di **messaggi** generati da un processo e consumati da un altro
- Scambio di **segnali temporali** che indicano il verificarsi di dati eventi

La *cooperazione tra processi* prevede che l'esecuzione di alcuni di essi risulti *condizionata* dall'informazione prodotta da altri (**vincoli sull'ordinamento nel tempo delle operazioni dei processi**).

21

### ESEMPIO:

- n processi  $P_1, P_2, \dots, P_n$  attivati ad intervalli prefissati di tempo da  $P_0$ .
- l'esecuzione di  $P_i$  non può iniziare prima che sia giunto il segnale da  $P_0$
- ad ogni segnale inviato da  $P_0$  deve corrispondere una attivazione di  $P_i$

$n_1$  = numero di richieste di attivazione di  $P_i$

$n_2$  = numero di segnali di attivazione inviati da  $P_0$

$n_3$  = numero di volte in cui  $P_i$  è stato attivato

Deve essere ad ogni istante:

se  $n_2 \geq n_1$   $n_3 = n_1$

se  $n_2 < n_1$   $n_3 = n_2$

22

```
semaphore si;
si.value=0 /* valore iniziale  $s_i = 0$  */
```

```
processo  $P_i$ :
main()
{ ...
  while(...)
  { ...
    wait (&si);
    ...
  }
  ...
}
```

```
processo  $P_0$ :
main()
{ ...
  while(...)
  { ...
    signal (&si);
    ...
  }
  ...
}
```

23

### Dimostrazione:

La relazione  $nw(s) \leq ns(s) + s_0$  diventa in questo caso:

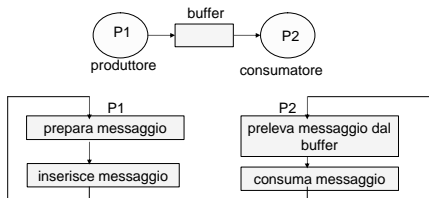
$$nw(s_i) \leq ns(s_i)$$

→ Il numero di volte che il processo  $P_i$  è stato attivato è minore o uguale al numero dei segnali inviati da  $P_0$

cvd

24

## Comunicazione



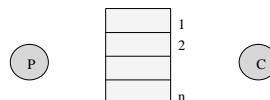
**Sequenza corretta:** inserimento-prelievo-inserimento-prelievo....

**Sequenze errate:**

- Inserimento-inserimento-prelievo....
- Prelievo-prelievo-inserimento

25

## Esempio: Produttore Consumatore (buffer di capacità n)



1. Il produttore non può inserire un messaggio nel buffer se questo è pieno.
  2. Il consumatore non può prelevare un messaggio dal buffer se questo è vuoto
- Siano:
    - d = numero dei messaggi depositati
    - e = numero dei messaggi estratti
    - n = numero dei messaggi che può contenere il buffer

- Deve valere la condizione:

$$0 \leq d - e \leq n$$

26

```
semaphore msg_disponibile;
msg_disponibile.value=0;
```

/* Processo produttore:*/	/* Processo consumatore:*/
<pre>main() {     for (;;) {         &lt;produzione messaggio&gt;;         &lt;deposito messaggio&gt;;         signal(&amp;msg_disponibile);     } }</pre>	<pre>main() {     for (;;) {         wait(&amp;msg_disponibile);         &lt;prelievo messaggio&gt;;         &lt;consumo messaggio&gt;;     } }</pre>

- questa soluzione soddisfa soltanto la condizione 2: il produttore potrebbe depositare un messaggio nel buffer pieno!

27

- Per sincronizzare correttamente gli accessi al buffer di produttore e consumatore, introduciamo **due semafori**:

- spazio\_disp (valore iniziale=n, capienza del buffer)
- msg\_disp (valore iniziale=0)

```
semaphore spazio_disp, msg_disp;
spazio_disp.value=n;
msg_disp.value=0;
```

/* Processo produttore:*/	/* Processo consumatore:*/
<pre>main() {     for (;;) {         wait(&amp;spazio_disp);         &lt;produzione messaggio&gt;;         &lt;deposito messaggio&gt;;         signal(&amp;msg_disp);     } }</pre>	<pre>main() {     for (;;) {         wait(&amp;msg_disp);         &lt;prelievo messaggio&gt;;         signal(&amp;spazio_disp);         &lt;consumo messaggio&gt;;     } }</pre>

28

### Dimostrazione

Th: la soluzione proposta soddisfa la condizione

$$0 \leq d - e \leq n$$

- La relazione  $nw(s) \leq ns(s) + s0$  scritta per i due semafori diventa:

$$1) \quad nw(spazio\_disp) \leq ns(spazio\_disp) + n$$

$$2) \quad nw(msg\_disp) \leq ns(msg\_disp)$$

- L'ordine con cui vengono eseguite le primitive comporta:

$$3) \quad ns(msg\_disp) \leq d \leq nw(spazio\_disp)$$

$$4) \quad ns(spazio\_disp) \leq e \leq nw(msg\_disp)$$

29

Dalle 3),1),4) si ha:

$$d \leq nw(spazio\_disp) \leq ns(spazio\_disp) + n \leq e + n \quad [i]$$

Dalle 4),2),3) si ha:

$$e \leq nw(msg\_disp) \leq ns(msg\_disp) \leq d \quad [ii]$$

- Combinando i due risultati [i] e [ii] si ottiene:

$$e \leq d \leq e + n$$

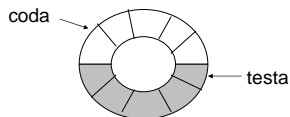
da cui:

$$0 \leq d - e \leq n$$

**cvd**

30

Affinchè la soluzione sia corretta, bisogna che produttore e consumatore *non accedano mai contemporaneamente alla stessa posizione del buffer*.



Inizialmente si ha:

$$coda = testa$$

31

### Operazioni di inserimento e prelievo:

```
typedef messaggio buffer[N];
buffer B;
int testa=0, coda=0;
messaggio M;
```

**Inserimento:**

$$B[coda] = M;$$

$$coda = (coda + 1) \% N;$$

**Prelievo:**

$$M = B[testa];$$

$$testa = (testa + 1) \% N;$$

Siano  $p1$  e  $p2$  rispettivamente il numero di volte in cui *coda* e *testa* sono stati incrementati (inizialmente  $p1 = p2$ ).

Le operazioni di deposito e prelievo agiscono sulla stessa porzione di buffer se:

$$p1 = p2 \bmod n$$

[5]

32



### Dimostrazione

**Th:** produttore e consumatore *non accedono contemporaneamente* alla stessa porzione di buffer

- Siano  $p_1$  e  $p_2$  rispettivamente il numero di volte in cui coda e testa sono stati incrementati (inizialmente  $p_1 = p_2$ ).
- Le operazioni di deposito e prelievo agiscono sulla stessa porzione di buffer se:

$$p_1 = p_2 \bmod n$$

- Durante l'operazione di **deposito** si ha:

$p_1 = ns(msg\_disp)$

$ns(msg\_disp) = nw(spazio\_disp) - 1 \wedge n + ns(spazio\_disp) - 1$  (dalla (1))

- Durante la operazione di **prelievo** si ha:

$p_2 = nw(msg\_disp) - 1$

$nw(msg\_disp) = ns(spazio\_disp) + 1 \wedge ns(msg\_disp)$  (dalla (2))

33

Si ha:

$$nw(msg\_disp) \wedge ns(msg\_disp) \wedge n + nw(msg\_disp) - 2$$

da cui:

$$0 \wedge ns(msg\_disp) - nw(msg\_disp) \wedge n - 2$$

e quindi:

$$0 \wedge p_1 - p_2 - 1 \wedge n - 2$$

da cui:

$$1 \wedge p_1 - p_2 \wedge n - 1 \quad [6]$$

La [6], che vale quando due processi stanno contemporaneamente lavorando sul buffer, è in contraddizione con la [5]. Quindi la condizione che produttore e consumatore *non accedano contemporaneamente* alla stessa porzione di buffer è soddisfatta.

cvd

34

### Nel caso di più produttori e più consumatori:

aggiungiamo i due semafori mutex1 e mutex2

```
...
semaphore mutex1, mutex2;
mutex1.value=1;
mutex2.value=1;
```

#### Processo produttore:

```
main()
{ for( ; ; )
{ <produz. messaggio>;
wait (&spazio_disp);
wait (&mutex1);
<inserimento mess.>;
signal(&mutex1);
signal(&msg_disp);
}
}
```

#### Processo consumatore:

```
main()
{ for( ; ; )
{ wait (&msg_disp);
wait (&mutex2);
<prelievo mess.>;
signal(&mutex2);
signal(&spazio_disp);
<consumo messaggio>;
}
}
```

35

### Esempi di uso dei semafori: gestione di risorse

- $R_1, R_2, \dots, R_n$   $n$  unità di uno stesso tipo di risorsa (tutte equivalenti fra loro).
- $P_1, P_2, \dots, P_m$   $m$  processi che devono operare su una qualunque risorsa in **modo esclusivo** tramite le operazioni  $A, B, \dots$

#### I Soluzione

- Si assegna un semaforo di mutua esclusione  $M_i$  (v.  $i=1$ ) ad ogni risorsa  $R_i$

36

processo  $P_s$ :

```
..
wait( $M_i$ );
 $R_i.A$ ;
signal( $M_i$ );
..
```

$R_i.A$  rappresenta l'esecuzione dell'operazione A su  $R_i$

**Inconvenienti della soluzione:**

- Come decide il generico processo su quali risorse operare (come viene scelto  $i$ )?
- Può capitare che, una volta scelta  $R_i$ , se su di essa sta operando in quel momento un secondo processo  $P_k$ , il processo  $P_s$  si blocchi su  $\text{wait}(M_i)$ , pur essendo disponibili altre risorse  $R_h$  ( $h \neq i$ ).

37

**Il Soluzione:** viene introdotta una nuova risorsa G, **gestore** di  $R_1, R_2, \dots, R_n$ . Essa può essere concepita come una **struttura dati** destinata a mantenere lo stato delle risorse gestite. Sul gestore si opera tramite due procedure:

**Richiesta e Rilascio.**

```
unsigned int Richiesta();
void Rilascio(unsigned int x);
```

(dove il parametro  $x$  rappresenta l'indice della risorsa assegnata o rilasciata)



semaforo RIS con valore iniziale = n

E' necessario un vettore di variabili *booleane* **Libero[i]** per registrare quale risorsa è in un certo istante libera ( $\text{Libero}[i] = 1$ ) e quale occupata ( $\text{Libero}[i] = 0$ ).

38

**Strutture dati del gestore:**

- le procedure **Richiesta** e **Rilascio** dovranno essere eseguite in **mutua esclusione**
- semaforo **mutex** di mutua esclusione con v.i. = 1
- Un processo che esegue **Richiesta** verifica la 'disponibilità' di una qualunque risorsa  $R_j$ .
- Un processo che esegue **Rilascio** rende nuovamente disponibile una risorsa
- semaforo **ris** con valore iniziale = n
- E' necessario un vettore di variabili *booleane* **Libero[i]** per registrare quale risorsa è in un certo istante libera ( $\text{Libero}[i] = 1$ ) e quale occupata ( $\text{Libero}[i] = 0$ ).

39

**Il Soluzione - segue**

```
semaphore mutex, ris;
int Libero[n];
void inizializza()
{ /*inizializzazione del gestore:*/
mutex.value= 1;
ris.value= n;
for(i = 0; i<n; i++)
    Libero[i] = 1; /*true*/
}
```

40

## II Soluzione - segue

```
int Richiesta ()
{
    unsigned int x, i;
    wait(&ris);
    wait(&mutex);
    i=0;
    do
        i++;
    while (! Libero[i]);
    x = i;
    Libero[i] = 0;
    signal(&mutex);
    return x;
}
```

```
void Rilascio (unsigned int x)
{
    unsigned int i;
    wait(&mutex);
    i=x;
    Libero[i]= 1;
    signal(&mutex);
    signal(&ris);
}
```

41

## Schema del processo:

```
main()
{
    unsigned int risorsa;
    ...
    risorsa=Richiesta();
    <uso della risorsa>
    Rilascio(risorsa);
    ...
}
```

42

## Realizzazione di politiche di gestione delle risorse

- Nei problemi di sincronizzazione visti precedentemente si ha che:
  - La decisione se un processo possa proseguire l'esecuzione dipende dal valore di un solo semaforo (es., "mutex", "spazio disponibile", "messaggio disponibile")
  - La scelta del processo da riattivare avviene tramite l'algoritmo implementato nella **signal** (FIFO).
- In problemi di sincronizzazione **più complessi** si ha che:
  - La decisione se un processo può proseguire l'esecuzione dipende in generale dal verificarsi di una **condizione di sincronizzazione**
  - La scelta del processo da riattivare può avvenire sulla base di **priorità tra processi**

43

## Problema dei "readers and writers"



### Condizioni di sincronizzazione:

- I processi lettori possono usare la risorsa contemporaneamente.
- I processi scrittori hanno accesso esclusivo alla risorsa.
- I processi lettori e scrittori si **escludono mutuamente** nell'uso della risorsa.

44

### Soluzione 1

Un processo lettore aspetta *solo se la risorsa è già stata assegnata ad un processo scrittore*: cioè nessun lettore aspetta se uno scrittore è già in attesa (possibilità di attesa infinita da parte dei processi scrittori).

### Soluzione 2

Un processo lettore aspetta se *un processo scrittore è in attesa* (possibilità di attesa infinita da parte dei processi lettori).

45

### Soluzione 1:

```
int    readcount=0;
semaphore mutex, w;
mutex.value=1; w.value=1;
```

READER	WRITER
<pre>main() {     wait(&amp;mutex);     readcount ++;     if (readcount == 1)         wait(&amp;w);     signal(&amp;mutex);     ..     &lt;lettura&gt;     ..     wait(&amp;mutex);     readcount --;     if (readcount==0)         signal(&amp;w);     signal(&amp;mutex); }</pre>	<pre>main() {     wait(&amp;w);     ..     &lt;scrittura&gt;     ..     signal(&amp;w); }</pre>

46

### Soluzione 2:

```
int    readcount, writecount=0;
semaphore mutex1, mutex2, mutex3, w, r;
mutex1.value=1; mutex2.value=1; mutex3.value=1; w.value=1; r.value=1;
```

READER	WRITER
<pre>main{     wait(mutex3);     wait(r);     wait(mutex1);     readcount++;     if (readcount==1)         wait(w);     signal(mutex1);     signal(r);     signal(mutex3);     ..     &lt;lettura&gt;     ..     wait(mutex1);     readcount --;     if (readcount == 0)         signal(w);     signal(mutex1);}</pre>	<pre>main(){     wait(mutex2);     writecount ++;     if (writecount==1)         wait(r);     signal(mutex2);     wait(w);     ..     &lt;scrittura&gt;     ..     signal(w)     wait(mutex2)     writecount--;     if (writecount==0)         signal(r);     signal(mutex2);}</pre>

47