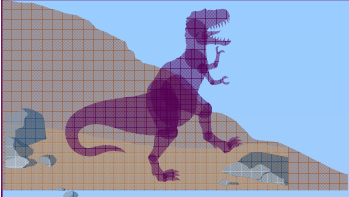




Capitolo 7: Sincronizzazione dei processi

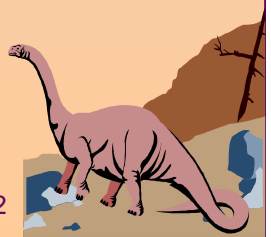
- Introduzione
- Problema della sezione critica
- Architetture di sincronizzazione
- Semafori
- Problemi tipici di sincronizzazione
- Regioni critiche
- Monitor
- Sincronizzazione nel Solaris 2 e in Windows 2000

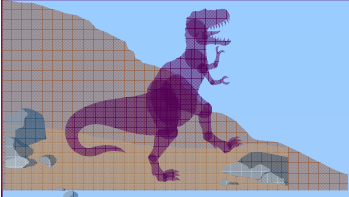




Introduzione

- L'accesso concorrente a dati condivisi può causare incoerenza degli stessi dati.
- Mantenere la coerenza dei dati richiede meccanismi atti ad assicurare un'ordinata esecuzione dei processi cooperanti.
- La soluzione del problema dei produttori e dei consumatori con memoria limitata (§ 4.4) consente la presenza contemporanea nel vettore di $n - 1$ elementi.
 - ☞ Si supponga di voler modificare il codice del produttore-consumatore aggiungendo una variabile intera **counter** (contatore) inizializzata a 0 e che si incrementa ogniqualvolta si preleva un elemento dal vettore.



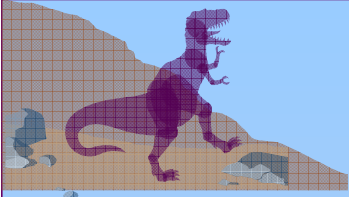


Memoria limitata

■ Dati condivisi

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE ];
int in = 0;
int out = 0;
int counter = 0;
```





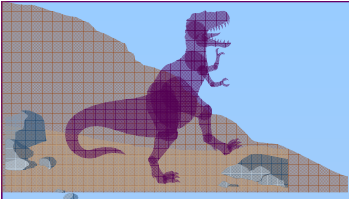
Memoria limitata

- Processo produttore

item nextProduced;

```
while (1) {  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
    buffer[in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```





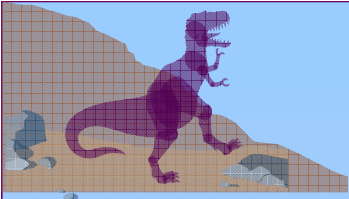
Memoria limitata

- Processo consumatore

```
item nextConsumed;
```

```
while (1) {  
    while (counter == 0)  
        ; /* do nothing */  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
}
```





Memoria limitata

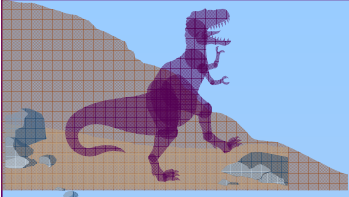
- Le istruzioni

```
counter++;  
counter--;
```

devono essere eseguite *in modo atomico*.

- Un'operazione è eseguita in modo atomico se si esegue completamente senza interruzione.





Memoria limitata

- L'istruzione “**count++**” si può codificare in linguaggio macchina come:

register1 = counter

register1 = register1 + 1

counter = register1

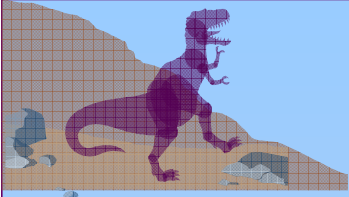
- L'istruzione “**count - -**” può essere realizzata come:

register2 = counter

register2 = register2 - 1

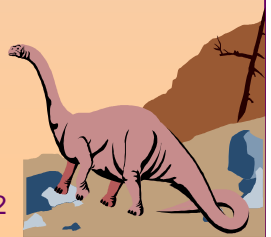
counter = register2

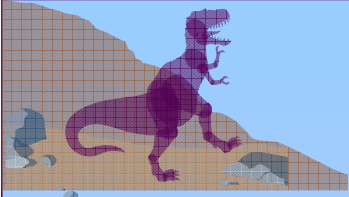




Memoria limitata

- Se sia il produttore sia il consumatore tentano di aggiornare concorrentemente il registro, le istruzioni del linguaggio macchina possono risultare intercalate (*interleaved*).





Memoria limitata

- Si supponga che il valore della variabile **counter** sia inizialmente 5. Una sequenza è:

producer: **register1 = counter** (*register1 = 5*)

producer: **register1 = register1 + 1** (*register1 = 6*)

consumer: **register2 = counter** (*register2 = 5*)

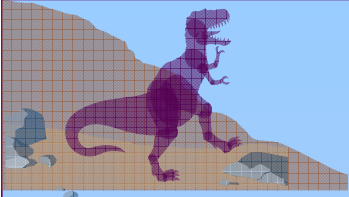
consumer: **register2 = register2 - 1** (*register2 = 4*)

producer: **counter = register1** (*counter = 6*)

consumer: **counter = register2** (*counter = 4*)

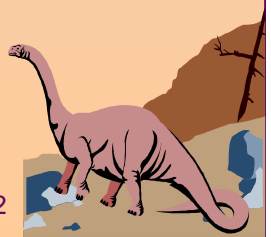
- Il valore della variabile potrebbe essere 4 o 6, dove il risultato corretto dovrebbe essere 5.

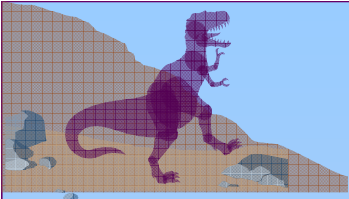




Race condition

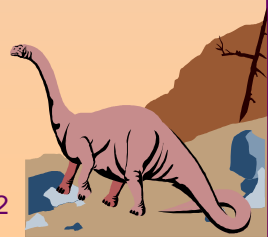
- **Race condition:** situazione in cui più processi accedono e modificano gli stessi dati in modo concorrente, e i risultati dipendono dall'ordine degli accessi.
- Per prevenire questa situazione, i processi concorrenti devono essere **sincronizzati**.

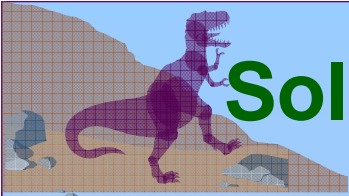




Problema della sezione critica

- Si considerino n processi concorrenti.
- Ciascun processo ha un segmento di codice chiamato **sezione critica** (*critical section*) nel quale il processo può modificare variabili comuni.
- Problema: quando un processo è in esecuzione nella propria sezione critica, non si deve consentire a nessun altro processo di essere in esecuzione nella propria sezione critica.

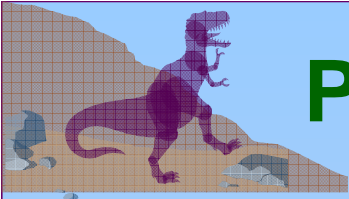




Soluzione al problema della sezione critica

1. **Mutua Esclusione.** Se il processo P_i è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
2. **Progresso.** Se nessun processo è in esecuzione nella sua sezione critica e qualche processo desidera entrare nella propria sezione critica, solo i processi che si trovano fuori delle rispettive sezioni non critiche possono partecipare alla decisione riguardante la scelta del processo che può entrare per primo nella propria sezione critica; questa scelta non si può rimandare indefinitamente.
3. **Attesa limitata.** Se un processo ha già richiesto l'ingresso nella sua sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che si accordi la richiesta del primo processo.
 - Si suppone che ogni processo sia eseguito a una velocità diversa da zero
 - Non si può fare alcuna ipotesi sulla **velocità relativa** degli n processi.



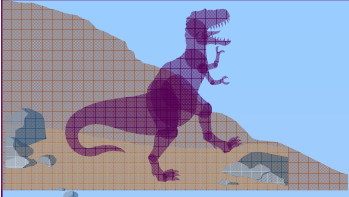


Primi tentativi di risolvere il problema

- Solo 2 processi, P_0 e P_1
- Struttura generale del processo P_i (l'altro processo è P_j)

```
do {  
    sezione d'ingresso  
    sezione critica  
    sezione d'uscita  
    sezione non critica  
} while (1);
```
- I processi possono condividere alcune variabili comuni per sincronizzare le loro azioni.



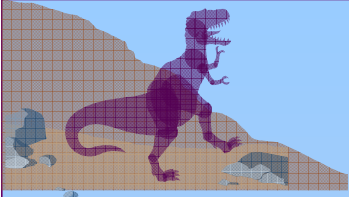


Algoritmo 1

- Variabili condivise:
 - ☞ **int turno;**
inizialmente **turno = 0**
 - ☞ **turno - i** $\Rightarrow P_i$ entra nella propria sezione critica
- Processo P_i

```
do {  
    while (turno != i) ;  
    sezione critica  
    turno = j;  
    sezione non critica  
} while (1);
```
- Soddisfa la mutua esclusione ma non il requisito di progresso





Algoritmo 2

- Variabili condivise

- ✎ **boolean pronto[2];**

- inizialmente **pronto [0] = pronto [1] = false.**

- ✎ **pronto [i] = true** $\Rightarrow P_i$ pronto a entrare nella sua sezione critica

- Processo P_i

- do {**

- pronto[i] := true;**

- while (pronto[j]) ;**

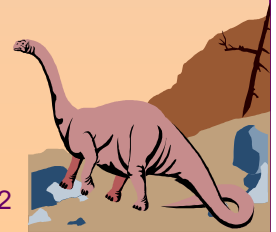
- sezione critica

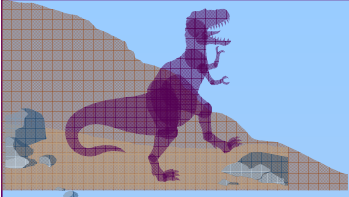
- pronto[i] = false;**

- sezione non critica

- } while (1);**

- Soddisfa la mutua esclusione ma non il requisito di progresso.

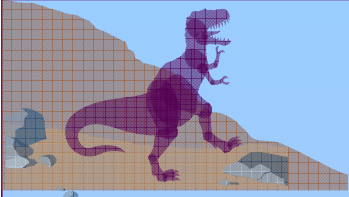




Algoritmo 3

- Combina le variabili condivise degli algoritmi 1 e 2.
- Processo P_i
 - do {
 - pronto[i] = true;
 - turno = j;
 - while (pronto[j] && turno = j) ;
 - sezione critica
 - pronto[i] = false;
 - sezione non critica
 - } while (1);
- Soddisfa tutti i tre requisiti; risolve il problema della sezione critica per i due processi.

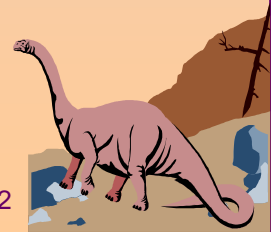


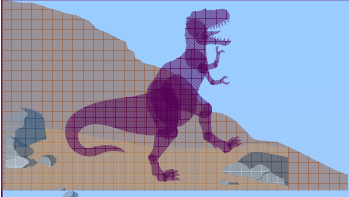


Algoritmo del fornaio

Sezione critica per n processi

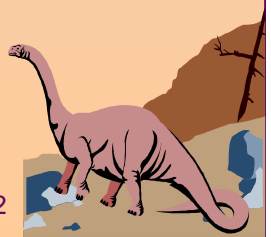
- Prima di entrare nella sua sezione critica, il processo riceve un numero. Chi detiene il numero più basso entra nella sezione critica.
- Se i processi P_i e P_j ricevono lo stesso numero, se $i < j$, allora P_i è servito per primo; altrimenti viene servito P_j .
- Lo schema di numerazione genera sempre numeri in ordine crescente; es.: 1,2,3,3,3,3,4,5...

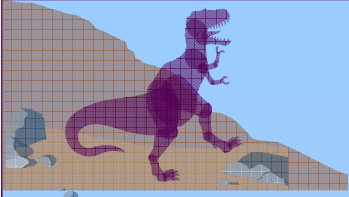




Algoritmo del fornaio

```
do {  
    scelta[i] = true;  
    numero[i] = max(numero[0], numero [1], ..., numero [n - 1])+1;  
    scelta[i] = false;  
    for (j = 0; j < n; j++) {  
        while (scelta[j]) ;  
        while ((numero [j] != 0) && (numero [j,j] < numero [i,i])) ;  
    }  
    sezione critica  
    numero [i] = 0;  
    sezione non critica  
} while (1);
```



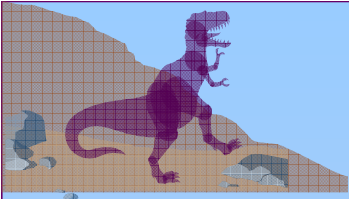


Architetture di sincronizzazione

- Controlla e modifica il contenuto di una parola di memoria in modo atomico.

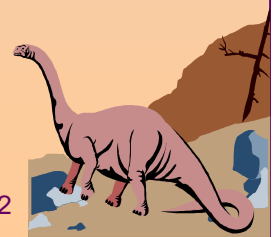
```
boolean TestAndSet(boolean &obiettivo) {  
    boolean valore = obiettivo;  
    obiettivo = true;  
  
    return valore;  
}
```

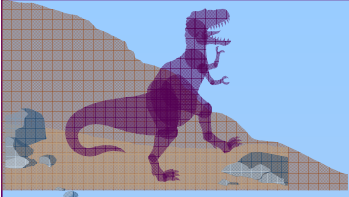




Mutua esclusione con TestAndSet

- Dati condivisi:
boolean blocco = false;
- Processo P_i
do {
 while (TestAndSet(blocco)) ;
 sezione critica
 blocco = false;
 sezione non critica
}



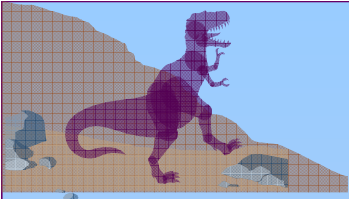


Architetture di sincronizzazione

- L'istruzione **swap** agisce sul contenuto di due parole di memoria; come per **TestAndSet** è anch'essa eseguita in modo atomico.

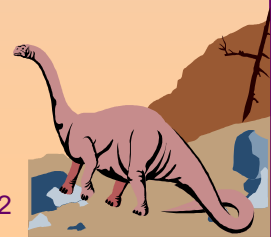
```
void Swap(boolean &a, boolean &b) {  
    boolean temp = a;  
    a = b;  
    b = temp;  
}
```

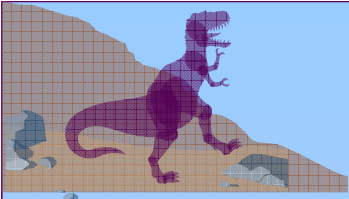




Mutua esclusione con Swap

- Dati condivisi (inizializzati a **false**):
boolean blocco;
boolean waiting[n];
- Processo P_i
do {
 chiave = true;
 while (**chiave == true**)
 Swap(blocco, chiave);
 sezione critica
 blocco = false;
 sezione non critica
}





Semafori

- Strumento di sincronizzazione che non richiede attesa attiva.
- Un semaforo S è una variabile intera.
- A questa variabile si può accedere solo tramite due operazioni atomiche predefinite

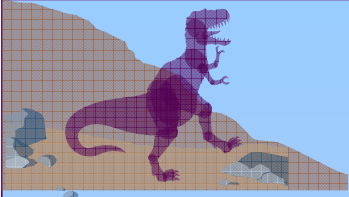
wait (S):

while $S \leq 0$ do *no-op*;
 $S--$;

signal (S):

$S++$;



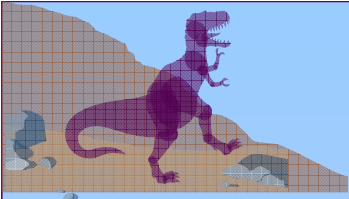


Sezione critica di n processi

- Dati condivisi:
semaphore mutex; //inizialmente $mutex = 1$
- Processo P_i :

do {
 wait(mutex);
 sezione critica
 signal(mutex);
 sezione non critica
} while (1);



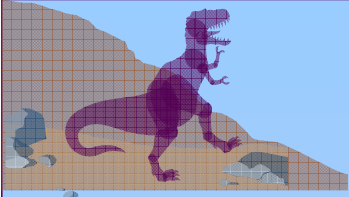


Realizzazione di semafori

- Definire il semaforo come una struttura del linguaggio C

```
typedef struct {  
    int valore;  
    struct processo *L;  
} semaforo;
```
- Due semplici operazioni:
 - ☞ **block** sospende il processo che la invoca.
 - ☞ **wakeup(*P*)** riprende l'esecuzione di un processo **P** bloccato.





Realizzazione

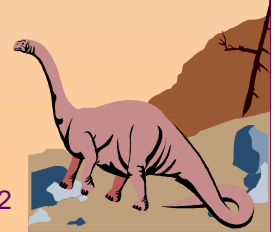
- Le operazioni dei semafori si possono definire ora come segue:

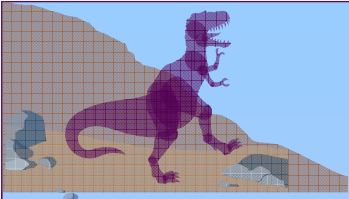
wait(S):

```
S.valore--;  
if (S.valore < 0) {  
    aggiungi questo processo a S.L;  
    block;  
}
```

signal(S):

```
S.valore++;  
if (S.valore <= 0) {  
    toglì un processo P da S.L;  
    wakeup(P);  
}
```



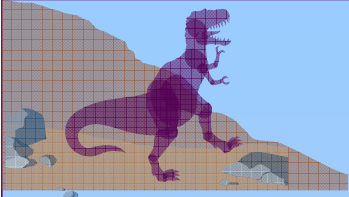


Semaforo come strumento generale di sincronizzazione

- Esegue B in P_j solo dopo che A è stato eseguito in P_i
- Usa un *flag* inizializzato a 0
- Codice:

P_i	P_j
\square	\square
A	$wait(flag)$
$signal(flag)$	B





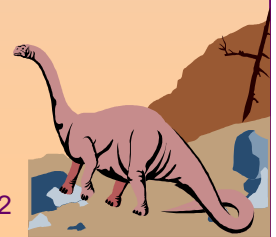
Stallo e attesa indefinita

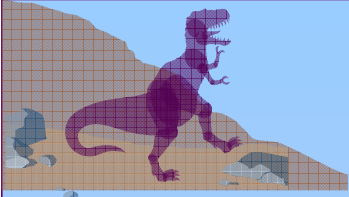
- **Stallo** (*deadlock*): situazione in cui due o più processi attendono indefinitamente un evento che può essere causato solo da uno dei processi in attesa.
- Siano S e Q due semafori inizializzati a 1

P_0
 $wait(S);$
 $wait(Q);$
 \square
 $signal(S);$
 $signal(Q)$

P_1
 $wait(Q);$
 $wait(S);$
 \square
 $signal(Q);$
 $signal(S);$

- **Attesa indefinita** (*starvation*): situazione d'attesa indefinita nella coda di un semaforo.

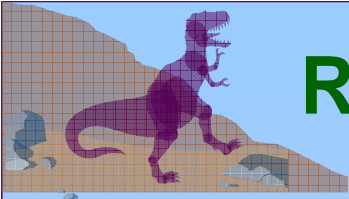




Tipi di semafori

- **Semaforo contatore** (*counting semaphore*): il suo valore intero può variare in un dominio logicamente non limitato.
- **Semaforo binario** (*binary semaphore*): il suo valore intero può essere soltanto 0 o 1; può essere più semplice da realizzare.
- È possibile implementare un semaforo contatore S in termini di un semaforo binario.





Realizzazione di S come semaforo binario

- Strutture dati:

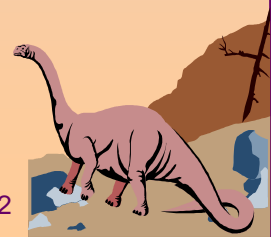
```
semaforo_binario S1, S2;  
int C;
```

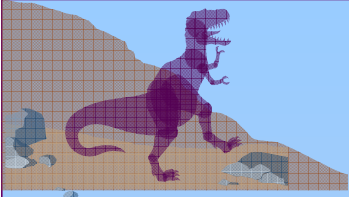
- Inizializzazione:

```
S1 = 1
```

```
S2 = 0
```

```
C = valore iniziale del semaforo contatore S
```





Realizzazione di S

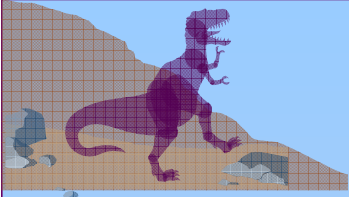
■ Operazione *wait*

```
wait(S1);  
C--;  
if (C < 0) {  
    signal(S1);  
    wait(S2);  
}  
signal(S1);
```

■ Operazione *signal*

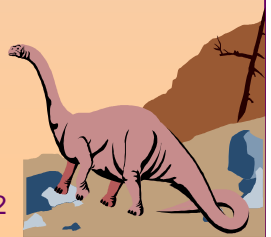
```
wait(S1);  
C ++;  
if (C <= 0)  
    signal(S2);  
else  
    signal(S1);
```

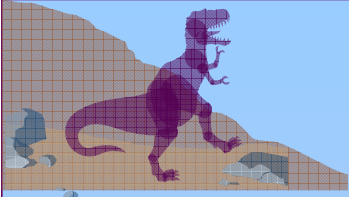




Problemi tipici di sincronizzazione

- Problema dei produttori e dei consumatori con memoria limitata
- Problema dei lettori e degli scrittori
- Problema dei cinque filosofi





Produttori e consumatori con memoria limitata

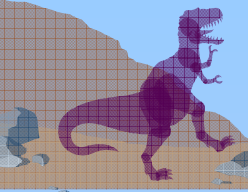
- Dati condivisi

semaforo piene, vuote, mutex;

Inizialmente:

$\text{piene} = 0$, $\text{vuote} = n$, $\text{mutex} = 1$

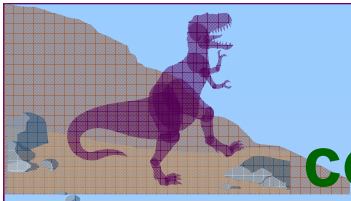




Produttori e consumatori con memoria limitata - Processo produttore

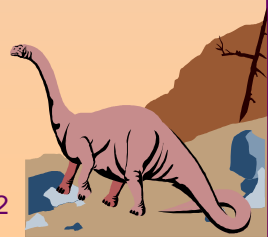
```
do {  
    ...  
    produce un elemento in appena_prodotto  
    ...  
    wait(vuote);  
    wait(mutex);  
    ...  
    inserisci appena_prodotto in vettore  
    ...  
    signal(mutex);  
    signal(piene);  
} while (1);
```

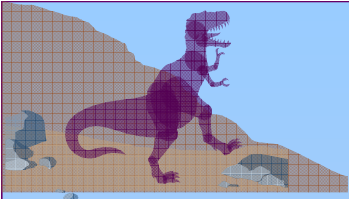




Produttori e consumatori con memoria limitata - Processo consumatore

```
do {  
    wait(piene)  
    wait(mutex);  
    ...  
    rimuovi un elemento da vettore e  
    inseriscilo in da_consumare  
    ...  
    signal(mutex);  
    signal(vuote);  
    ...  
    consuma l'elemento contenuto in da_consumare  
    ...  
} while (1);
```





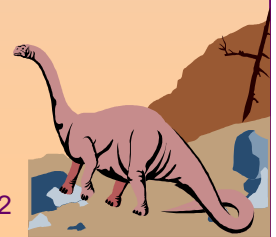
Problema dei lettori e degli scrittori

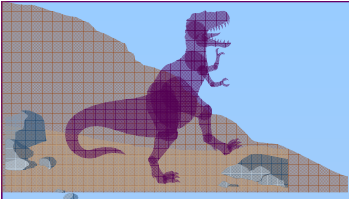
■ Dati condivisi

semaforo mutex, scrittura;

Inizialmente

mutex = 1, scrittura = 1, numlettori = 0





Problema dei lettori e degli scrittori

Processo scrittore

```
wait(scrittura);
```

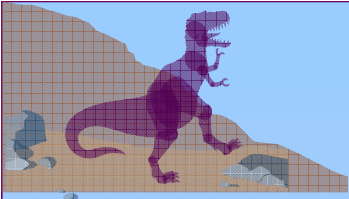
```
...
```

```
    esegui l'operazione di scrittura
```

```
...
```

```
signal(scrittura);
```

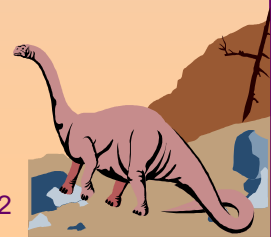


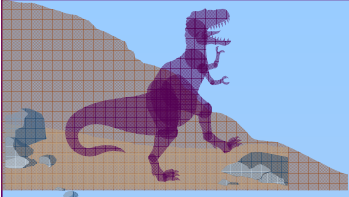


Problema dei lettori e degli scrittori

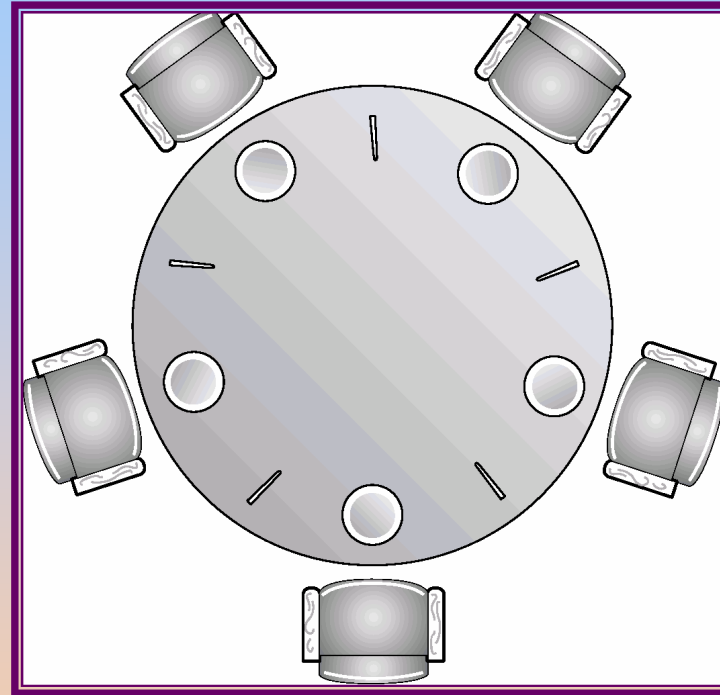
Processo lettore

```
wait(mutex);  
numlettori++;  
if (numlettori == 1)  
    wait(scrittura);  
signal(mutex);  
...  
    esegui l'operazione di lettura  
...  
wait(mutex);  
numlettori --;  
if (numlettori == 0)  
    signal(scrittura);  
signal(mutex);
```



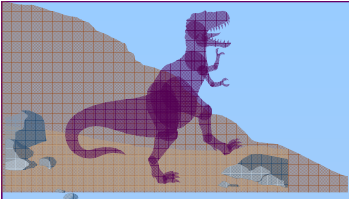


Problema dei cinque filosofi



- Dati condivisi
semaforo bacchetta[5];
Inizialmente tutti i valori sono 1





Problema dei cinque filosofi

■ Filosofo i :

```
do {  
    wait(bacchetta[i])  
    wait(bacchetta[(i + 1) % 5])  
    ...  
    mangia  
    ...  
    signal(bacchetta[i]);  
    signal(bacchetta[(i + 1) % 5]);  
    ...  
    pensa  
    ...  
} while (1);
```

