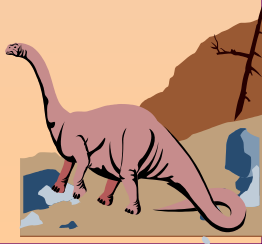
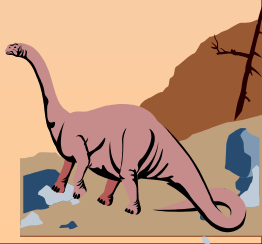
Capitolo 6: Sincronizzazione dei processi

- Introduzione
- Problema della sezione critica
- Soluzione di Peterson
- Hardware per la sincronizzazione
- Semafori
- Problemi tipici di sincronizzazione
- Monitor
- Esempi di sincronizzazione
- Transazioni atomiche



Tipi di processi

- Processi indipendenti:
  - ogni processo non può influire su altri processi nel sistema o subirne l'influsso, chiaramente, essi non condividono dati.
- Processi cooperanti:
  - ogni processo può influenzare o essere influenzato da altri processi in esecuzione nel sistema.
  - I processi cooperanti possono condividere dati.
- Un'esecuzione concorrente di processi cooperanti richiede meccanismi che consentono ai processi di comunicare tra loro e di sincronizzare le proprie azioni.



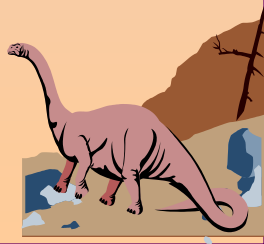
Introduzione

- L'accesso concorrente a dati condivisi può creare problemi di inconsistenza dei dati.
- Per mantenere la consistenza dei dati sono necessari meccanismi che assicurino l'esecuzione ordinata e coordinata dei processi cooperanti.
- La precedente soluzione del problema del produttore e del consumatore con memoria limitata consente la presenza contemporanea nel vettore di non più di *n–1* elementi.
  - Facendo uso di memoria condivisa.
- Una soluzione per utilizzare tutti gli *n* elementi del vettore non è semplice:
  - Potremmo aggiungere una variabile intera,*contatore*, inizializzata a 0, che si incrementa ogni volta che si inserisce un nuovo elemento nel vettore,
  - e si decrementa ogni volta che si preleva un elemento dal vettore.

Buffer di dimensioni limitate: dati condivisi

- Dati condivisi

```
#define DIM_VETTORE 10
typedef struct {
   . . .
} item;
item vettore[DIM_VETTORE];
int inserisci = 0;
int preleva = 0;
int contatore = 0;
```

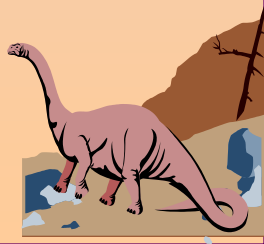


Buffer di dimensioni limitate: processo produttore

- Processo produttore

```
item appena_prodotto;

while (1) {
        while (contatore == DIM_VETTORE);
                                /* non fare niente */
        vettore[inserisci] = appena_prodotto;
        inserisci = (inserisci + 1) % DIM_VETTORE;
        contatore++;
}
```

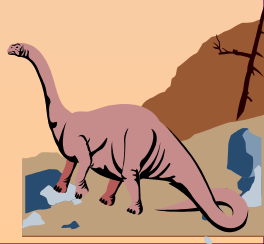


Buffer di dimensioni limitate: processo consumatore

- Processo consumatore

```
item da_consumare;

while (1) {
        while (contatore == 0); /* non fare niente */
        da_consumare = vettore [preleva];
        preleva = (preleva + 1) % DIM_VETTORE;
        contatore--;
}
```



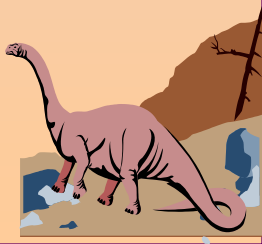
Buffer di dimensioni limitate: transazioni atomiche

- Gli statement

  contatore++;
  contatore--;

  devono essere eseguiti *atomicamente*.

- Una operazione è atomica se non può essere interrotta fino al suo completamento.



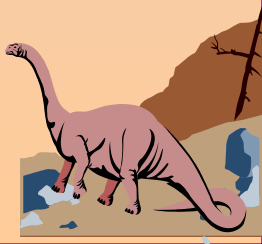
Buffer di dimensioni limitate: transazioni atomiche (II)

- Lo statement “**contatore++**” può essere implementato in linguaggio macchina come:

  registro1 = contatore
  registro1 = registro1 + 1
  contatore = registro1

- Lo statement “**count—**” può essere implementato in linguaggio macchina come:

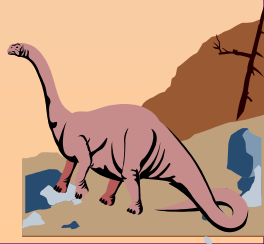
  registro2 = contatore
  registro2 = registro2 – 1
  contatore = registro2



Buffer di dimensioni limitate: transazioni atomiche (III)

- Se il produttore ed il consumatore tentano di accedere concorrentemente al buffer, le esecuzioni in linguaggio macchina dell'incremento e del decremento del contatore potrebbero interfogliarsi (**interleaving**).

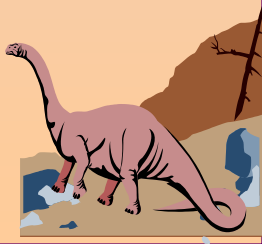
- Il risultato dell'interfogliamento dipende da come produttore e consumatore sono schedulati.



Buffer di dimensioni limitate: transazioni atomiche (IV)

- Supponiamo che **contatore** sia inizializzato a 5 e che ci siano un elemento prodotto ed uno consumato.
- Un possibile interfogliamento delle esecuzioni di produttore e consumatore è:
  produttore: **registro1 = contatore** (*registro1 = 5*)
  produttore: **registro1 = registro1 + 1** (*registro1 = 6*)
  consumatore: **registro2 = contatore** (*registro2 = 5*)
  consumatore: **registro2 = registro2 – 1** (*registro2 = 4*)
  produttore: **contatore = registro1** (*contatore = 6*)
  consumatore: **contatore = registro2** (*contatore = 4*)

- Il valore di **contatore** è quindi 4, ma il risultato corretto è 5.

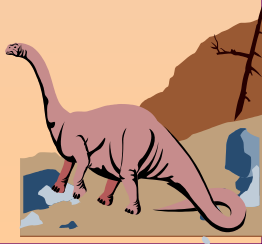


Race condition

- **Race condition**:
  - è la situazione in cui più processi accedono e modificano gli stessi dati concorrentemente e i risultati dipendono dall'ordine degli accessi.

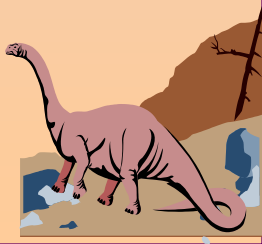
- Per prevenire le race condition, i processi concorrenti devono essere **sincronizzati**.

- Per evitare situazioni di questo tipo, in cui più processi accedono e modificano gli stessi dati in modo concorrente e i risultati dipendono dagli ordini degli accessi, occorre garantire che un solo processo alla volta possa modificare dati condivisi.



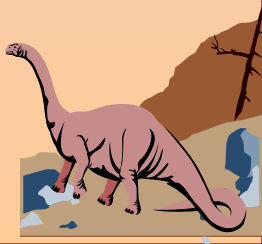
Problema della sezione critica

- Per evitare il problema delle race condition si introduce l'idea di ***sezione critica***:
  - un segmento di codice nel quale il processo può modificare variabili comuni, scrivere un file.
- Quando un processo è in esecuzione nella propria sezione critica non si deve consentire a nessun altro processo di entrare in esecuzione nella propria sezione critica.
- L'esecuzione delle sezioni critiche da parte dei processi è *mutuamente esclusiva* nel tempo.



Problema della sezione critica (II)

- *n* processi in competizione per l'uso di dati condivisi.
- Ciascun processo ha un segmento di codice chiamato *sezione critica*, in cui avviene l'accesso ai dati condivisi
- ***Problema***:
  - ☞ assicurare che quando un processo *P* esegue la sua sezione critica nessun altro processo *Q* possa eseguire la propria sezione critica
- Il problema della sezione critica si affronta progettando un protocollo che i processi possono usare per cooperare.
- Ogni processo deve chiedere il permesso di entrare nella propria sezione critica.
- La sezione di codice che realizza questa richiesta si chiama *sezione di ingresso*.
- La sezione critica può essere seguita da una *sezione di uscita*, e la restante parte del codice è detta *sezione non critica*.



Soluzione al problema della sezione critica

- Per risolvere il problema della sezione critica, occorre soddisfare i tre seguenti requisiti:

1. **Mutua esclusione**. Se il processo P_i è nella sua sezione critica, allora nessun altro processo può essere nella sua sezione critica.
2. **Progresso**. Se nessun processo è in esecuzione nella sua sezione critica ed esiste qualche processo che desidera entrare nella propria sezione critica, allora la decisione riguardante la scelta del processo che potrà entrare per primo nella propria sezione critica non può essere rimandata indefinitamente. Solo i processi che si trovano fuori dalle rispettive sezioni critiche possono partecipare alla scelta di questo processo.
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.

- Assumiamo che ogni processo sia eseguito a una velocità diversa da 0
- Nessuna ipotesi sulla *velocità relativa* degli *n* processi

Soluzione per due processi

- Solo 2 processi, P_0 e P_1
- Struttura generale di un tipico processo *P*:

do {

sezione di ingresso

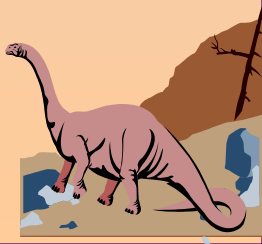
sezione critica

sezione di uscita

sezione non critica

} **while (true)**;

- I processi possono utilizzare delle variabili condivise per sincronizzarsi.

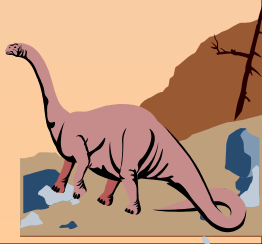


Algoritmo 1

- Variabili condivise:
  - **int turno**;
    inizialmente **turno = 0**, potra assumere valore 0 o 1.
  - **turno == i** $\Rightarrow$ *Pi* può entrare nella sua sezione critica
- Processo *Pi*

```
do {
    while (turno != i);
        sezione critica
    turno = j;
        sezione non critica
} while (true);
```

- Soddisfa la mutua esclusione, ma non il progresso.
- Se turno==0, P_i non può entrare nella sua regione critica anche se P_j è nella propria regione non critica.

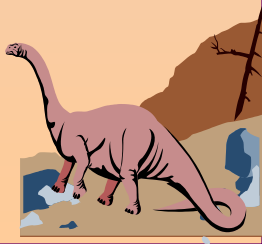


Algoritmo 2

- Variabili condivise:
  - **boolean pronto [2]**;
    inizialmente **pronto [0] = pronto [1] = false.**
  - **pronto [i] == true** $\Rightarrow P_i$ è pronto ad entrare nella sua sezione critica
- Processo P_i

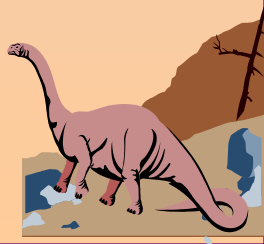
```
do {
    pronto [i] := true;
    while (pronto [j]);
        sezione critica
    pronto [i] = false;
        sezione non critica
} while (true);
```

- Soddisfa la mutua esclusione, ma non il progresso.



Algoritmo 2 (II)

- Se infatti consideriamo la seguente sequenza di esecuzione:
  - **T_0:** P_0 assegna pronto [0] = true;
  - **T_1:** P_1 assegna pronto [1] = true
- P_0 e P_1 entrano in un ciclo infinito nelle rispettive istruzioni while.
- Questo algoritmo dipende in modo decisivo dalla esatta temporizzazione dei due processi.

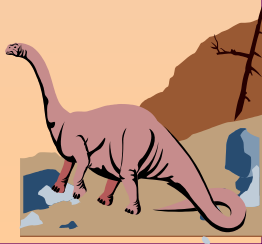


Algoritmo 3: soluzione di Peterson

- Utilizziamo le variabili condivise dei due algoritmi precedenti.
- Processo P_i

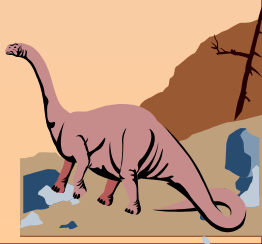
```
do {
    pronto [i]:= true;
    turno = j;
    while (pronto [j] and turno == j) ;
        sezione critica
    pronto [i] = false;
        sezione non critica
} while (true);
```

- Soddisfa tutti e tre i requisiti, risolve il problema della sezione critica per due processi.



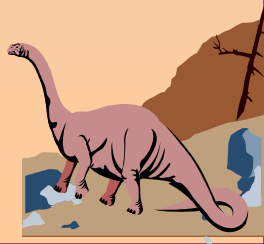
Algoritmo 3: mutua esclusione

- **Mutua Esclusione**:
  - P_i entra nella propria sezione critica solo se pronto[j]==false o turno==i.
  - Se entrambi i processi fossero contemporaneamente in esecuzione nelle rispettive sezioni critiche, si avrebbe pronto[i]==pronto[j]==true.
  - L'istruzione while non può essere eseguita da P_i e P_j contemporaneamente perché turno può assumere valore i o j, ma non entrambi.
  - Supponendo che P_j ha eseguito con successo l'istruzione while avremo che pronto[j]==true e turno=j, condizione che persiste fino a che P_j si trova nella propria sezione critica.



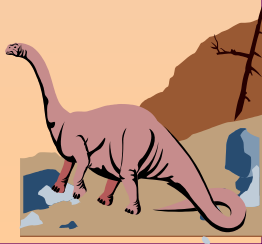
Algoritmo 3: Attesa limitata e progresso

- **Attesa Limitata e Progresso:**
  - Si può impedire a un processo P_i di entrare nella propria sezione critica solo se questo è bloccato nel ciclo while dalla condizione pronto[j]==true e turno==j.
  - Se P_j non è pronto per entrare, allora pronto[j]==false, e P_i può entrare nella propria sezione critica.
  - Se P_j è pronto per entrare, allora pronto[j]==true, se turno==i entra P_i, mentre se turno==j, entra P_j.
  - Se entra P_j, all'uscita imposta pronto[j]==false e P_i può entrare.
  - P_i entra nella sezione critica (progresso) al massimo dopo un ingresso da parte di P_j (attesa limitata).



Soluzione per più processi: algoritmo del fornaio

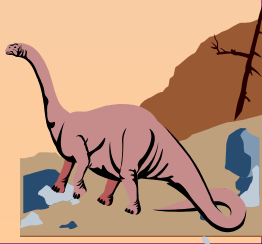
- Sezione critica per n processi.
- Questo algoritmo si basa su uno schema di servizio usato “nelle panetterie”.
- Al suo ingresso nel negozio,ogni cliente riceve un numero.
- Viene servito di volta in volta il cliente con il numero piu’ basso.
- L’algoritmo non assicura che due clienti (processi) non ricevano lo stesso numero.
- Nel caso in cui P_i e P_j hanno lo stesso numero e i < j viene servito prima P_i.
- Lo schema di numerazione genera sequenze crescenti di numeri: ad es., 1,2,3,3,3,3,4,5…
- Notazione <≡ ordine lessicografico (ticket #, process id #)
  - (*a,b*) < (*c,d*) if *a* < *c* oppure se *a* ==*c* and *b* < *d*
  - max (a_0,…, a_{n-1}) è un numero, *k*, tale che $k \geq a_i$ for $i = 0, \ldots, n-1$



Algoritmo del fornaio (II)

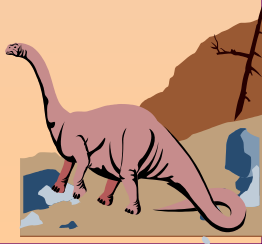
- Variabili condivise: **boolean scelta[n];**
  int numero[n];
  - inizializzate a **false** e **0** rispettivamente
- Algoritmo:

```
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 (true);
```



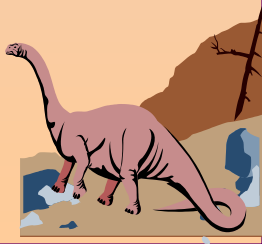
Algoritmo del fornaio: correttezza

- **Mutua esclusione**:
  - Se P_i si trova nella propria sezione critica e P_k (k!=i) ha già scelto il proprio numero[k] != 0, allora: (numero[i],i)<(numero[k],k).
  - Se P_i è nella propria sezione critica e P_k tenta di entrare nella propria, il processo P_k esegue la seconda istruzione while per j==i, trova che:
    - numero[i]!=0
    - (numero[i],i)<(numero[k],k)
  - Quindi continua il ciclo nell'istruzione while fino a che P_i lascia la propria sezione critica.
- **Progresso e Attesa Limitata:**
  - Questi requisiti sono garantiti poiché i processi entrano nelle rispettive sezioni critiche secondo il criterio FCFS .



Architetture di sincronizzazione

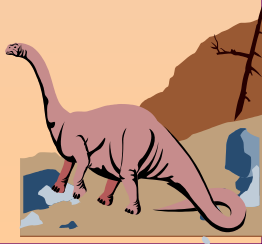
- In un sistema dotato di una singola CPU, si dovrebbero interdire le interruzioni mentre si modificano variabili condivise.
- In sistemi con più unità di elaborazione questo non è possibile.
- Esistono istruzioni, rese disponibili da alcune architetture, che possono essere impiegate efficacemente per risolvere il problema della sezione critica.
- Queste particolari istruzioni atomiche permettono di
  - controllare e modificare il contenuto di una parola di memoria,
  - di scambiare il contenuto di due parole di memoria.



Architetture di sincronizzazione (II)

- Istruzione TestAndSet: modifica il contenuto di una varabile in maniera atomica.
- Se si eseguono contemporaneamente due istruzioni TestAndSet, ciascuna in una unità di elaborazione diversa, queste vengono eseguite in maniera sequenziale in un ordine arbitrario.

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



Mutua esclusione con Test-and-Set

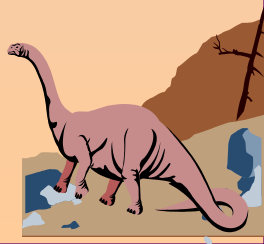
- Variabili condivise:

  boolean blocco = false;

- Processo P_i

```
do {
    while (TestAndSet(&blocco)) ;
        sezione critica
    blocco = false;
        sezione non critica
} while (true);
```

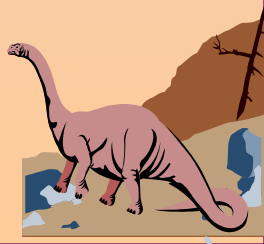
- Non soddisfa il requisito di attesa limitata.



Swap

- Scambia il contenuto di due variabili. Istruzione atomica.

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



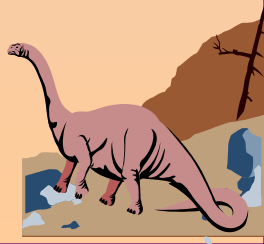
Mutua esclusione con Swap

- Variabile condivisa (inizializzata a **false**):
  boolean blocco;

- Processo P_i

```
do {
    chiave = true;
    while (chiave == true)
            Swap(&blocco, &chiave);
        sezione critica
     blocco = false;
        sezione non critica
} while (true);
```

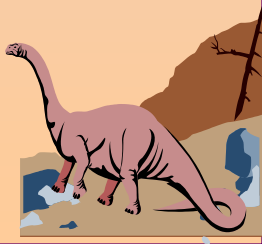
- Non soddisfa il requisito di attesa limitata.



Mutua esclusione con attesa limitata con Test-and-Set

- Variabili condivise (inizializzate a **false**): **boolean blocco;**
  boolean attesa[n];
- Processo P_i

```
do {
    attesa[ i ] = true;
    chiave = true;
    while (attesa[ i ] && chiave)
            chiave = TestAndSet(&blocco);
    attesa[ i ] = false;
        sezione critica
    j = (i + 1) % n;
    while ((j != i) && !attesa[ j ])   j = (j + 1) % n;
    if (j == i) blocco = false;
        else attesa[ j ] = false;
        sezione non critica
} while (true);
```



Requisiti

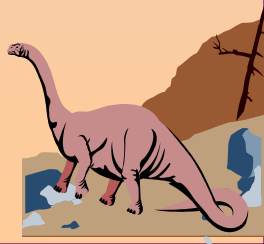
- **Mutua esclusione:**
  - Un processo può entrare in sezione critica solo se *attesa[i] == false* oppure *chiave == false*.
  - *chiave* è impostata a *false* solo se si esegue*TestAndSet*.
  - Il primo processo che esegue *TestAndSet* trova *chiave == false* , tutti gli altri devono attendere.
  - *attesa[i]* può diventare *false* solo se un altro processo esce dalla propria sezione critica, e solo una variabile *attesa[i]* vale *false*.
- **Progresso:**
  - Un processo che esce dalla sezione critica o imposta *blocco* a *false* oppure *attesa[j]* per un *j != i* a *false*, consentendo quindi ad un processo che attende di entrare.
- **Attesa limitata:**
  - Un processo, quando esce dalla sezione critica, scandisce il vettore *attesa* nell'ordinamento ciclico (i+1, i+2, …, n-1, 0, …, i-1) e designa come prossimo processo il primo presente nella sezione di ingresso (*attesa[j] == true*).
  - Ogni processo che lo chiede, entrerà in sezione critica entro **n-1** turni.

Semafori

- Un semaforo S è una variabile intera
- Ad un semaforo si può accedere solo tramite due operazioni atomiche:

wait (*S*): **{**
while (*S* <= 0) ; *ll non-op*;
***S*--;**
}

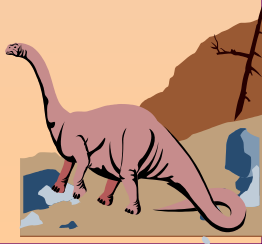
signal (*S*): **{**
***S*++;**
}



Uso dei semafori

- Eseguire *B* in P_j solo dopo l'esecuzione di *A* in P_i
- Useremo il semaforo ***pronto*** initializzato a 0
- Code:

P_i	P_j
⋮	⋮
A	*wait*(***pronto***)
signal(***pronto***)	*B*



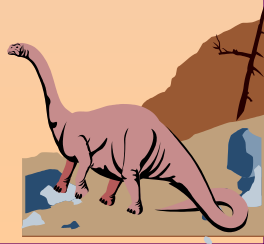
Realizzazione di mutua esclusione con semafori

- Variabili condivise:

  semaforo mutex; ***//*** inizialmente *mutex* = 1

- Process *Pi:*

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



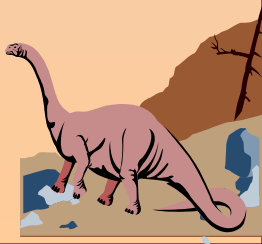
Realizzazione di un semaforo

- Sincronizzazione che non necessita di *attesa attiva* (busy waiting).

- Si può definire un semaforo come una struttura:

```
typedef struct {
   int valore;
   struct processo *L;
} semaforo;
```

- Assumiamo l'esistenza di due funzioni:
  - **block** sospende il processo che la invoca e pone il processo in una coda d'attesa associata al semaforo
  - **wakeup(*P*)** fa riprendere l'esecuzione di un processo **P** precedentemente bloccato

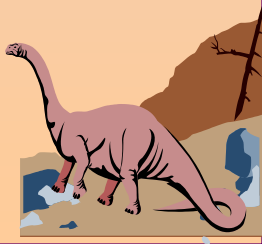


Realizzazione di un semaforo (II)

- Le operazioni sui semafori sono ora definite da:

```
void wait (semaforo *S)          {
        S.valore --;
        if (S.value < 0) {
                aggiungi questo processo a S.L;
                block();
        }

void signal (semaforo *S) {
        S.valore++;
        if (S. valore <= 0) {
                togli un processo P da S.L;
                wakeup(P);
        }
```

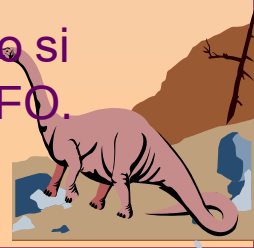


Stallo e attesa indefinita (deadlock and starvation)

- **Stallo (deadlock)**:
  - due o più processi attendono indefinitivamente un evento (ad es. un *signal*) che può essere causato solo da uno dei processi dello stesso insieme.
  - Ad es. siano *S* e *Q* due semafori inizializzati a 1

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

- **Attesa indefinita (starvation)**:
  - situazione d'attesa indefinita nella coda di un semaforo, ad esempio si può presentare se la coda del semaforo è gestita tramite criterio LIFO.



Due tipi di semafori

- Semaforo *Contatore* – a valore intero, che può variare in un dominio non limitato.
- Semaforo *Binario* – a valore intero, che può variare solo tra 0 e 1.
  - ☞ Più semplice da implementare.
- Si può implementare un semaforo contatore *S* tramite semafori binari.
- Strutture dati:

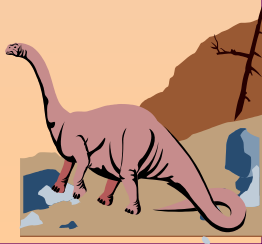
  semaforo-binario S1, S2;

  int C:
- Inizializzazione:

  S1 = 1

  S2 = 0

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



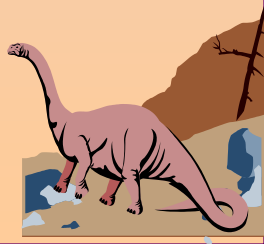
Implementare un semaforo contatore *S* tramite semafori binari

- *wait*

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

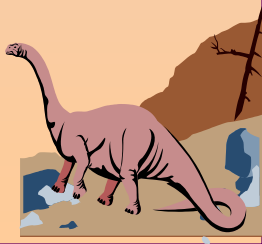
- *signal*

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



Problemi tipici di sincronizzazione

- Esamineremo alcuni problemi di sincronizzazione, come esempio di classi di problemi connessi al controllo della concorrenza.
- Presenteremo per questi problemi soluzioni basate su semafori)
- I problemi sono:
  - Problema dei produttori e consumatori con memoria limitata
  - Problema dei lettori e degli scrittori
  - Problema dei cinque filosofi



Produttori e consumatori con memoria limitata

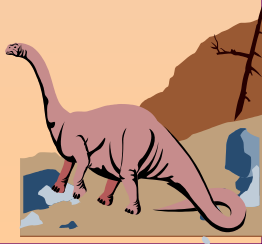
- Variabili condivise:

  item vettore [DIM_VETTORE];

  semaforo piene; /* numero di posizioni piene */
  semaforo vuote; /* numero di posizioni vuote */
  semaforo mutex; /* mutua esclusione accessi al vettore */

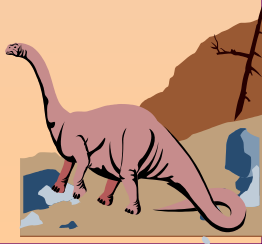
- Inizialmente:

  piene = 0, vuote = n, mutex = 1



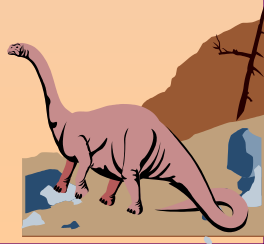
Produttori e consumatori con memoria limitata: processo produttore

```
do {
        …
    produce un elemento in appena_prodotto
        …
    wait(vuote);
    wait(mutex);
        …
    inserisci in vettore l'elemento in appena_prodotto
        …
    signal(mutex);
    signal(piene);
} while (true);
```



Produttori e consumatori con memoria limitata: processo consumatore

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



Problema dei lettori e degli scrittori

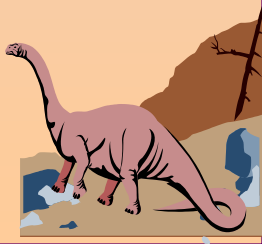
- Si consideri un insieme di dati (ad es. un file) che si deve condividere tra processi concorrenti.
- Alcuni processi leggeranno (**lettori**) altri aggiorneranno (lettura + scrittura) (**scrittori**).
- Se più lettori accedono concorrentemente all'insieme di dati condiviso non c'è nessun problema.
- Se uno scrittore accede, gli altri processi non possono accedere.
- Gli scrittori devono avere accesso esclusivo.

- Variabili condivise:

  int numlettori; /* numero di processi che stanno attualmente leggendo */

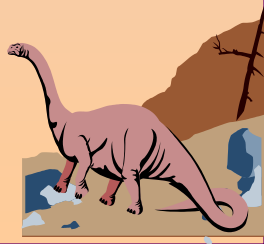
  semaforo mutex; /* mutua esclusione per aggiornamento numlettori */

  semaforo scrittura; /* mutua esclusione scrittori */
- Inizialmente
  mutex = 1, scrittura = 1, numlettori = 0



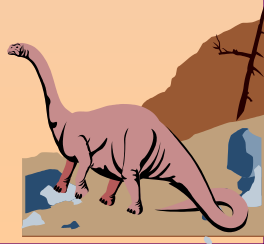
Problema dei lettori e degli scrittori: processo scrittore

```
do {
        wait(scrittura);
                ...
                esegui l'operazione di scrittura
                ...
        signal(scrittura);
} while (true);
```

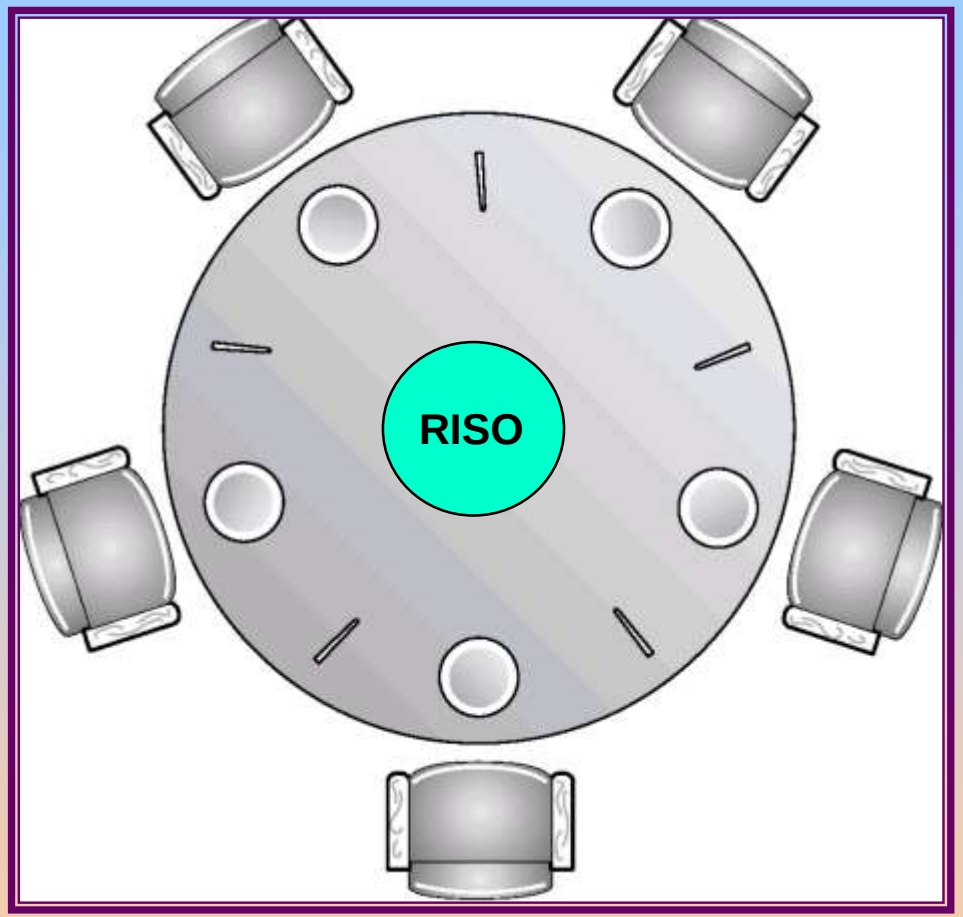


Problema dei lettori e degli scrittori: processo lettore

```
do {
        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);
        } while (true);
```



Problema dei cinque filosofi



- Variabili condivise:

  semaforo bacchetta[5];

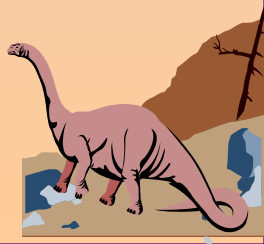
Inizialmente tutti i valori sono 1.

Struttura del filosofo *i*

- Filosofo *i* : **do {**

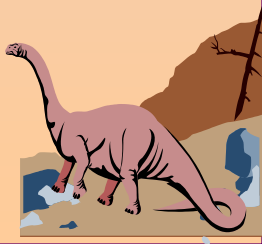
```
wait(bacchetta[i])
wait(bacchetta[(i+1) % 5])
    ...
   mangia
    ...
signal(bacchetta[i]);
signal(bacchetta[(i+1) % 5]);
    ...
   pensa
    ...
} while (true);
```

- Possibilità di stallo (deadlock).



Problema dei cinque filosofi (II)

- Alcune possibili soluzioni per evitare lo stallo:
  - Solo quattro filosofi possono stare contemporaneamente a tavola
  - Un filosofo può prendere le sue bacchette solo se sono entrambe disponibili (operazione di controllo da eseguire in sezione critica)
  - Un filosofo dispari prende prima la bacchetta di sinistra e poi quella di destra, un filosofo pari prende prima la bacchetta di destra e poi quella di sinistra
- Inoltre una soluzione soddisfacente per il problema dei 5 filosofi deve escludere le possibilità di attesa indefinita,
  - cioè che uno dei filosofi muoia di fame, da cui il termine *starvation*.

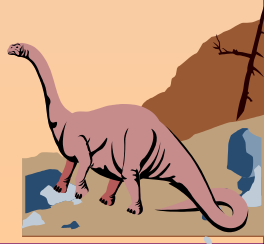


Monitor

- Un altra primitiva di sincronizzazione ad alto livello è il tipo monitor.
- Proposti da Hoara e Brinch Hansen negli anni ‘70, possono essere implementati tramite i semafori.
- Sintassi:

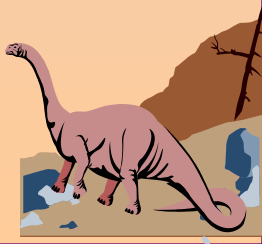
```
monitor nome-monitor
{
        dichiarazione di varabili condivise

        procedure body P1 (…) {
            . . .
        }
        procedure body P2 (…) {
            . . .
        }
        procedure body Pn (…) {
             . . .
        }
        {
            codice di inizializzazione
        }
}
```

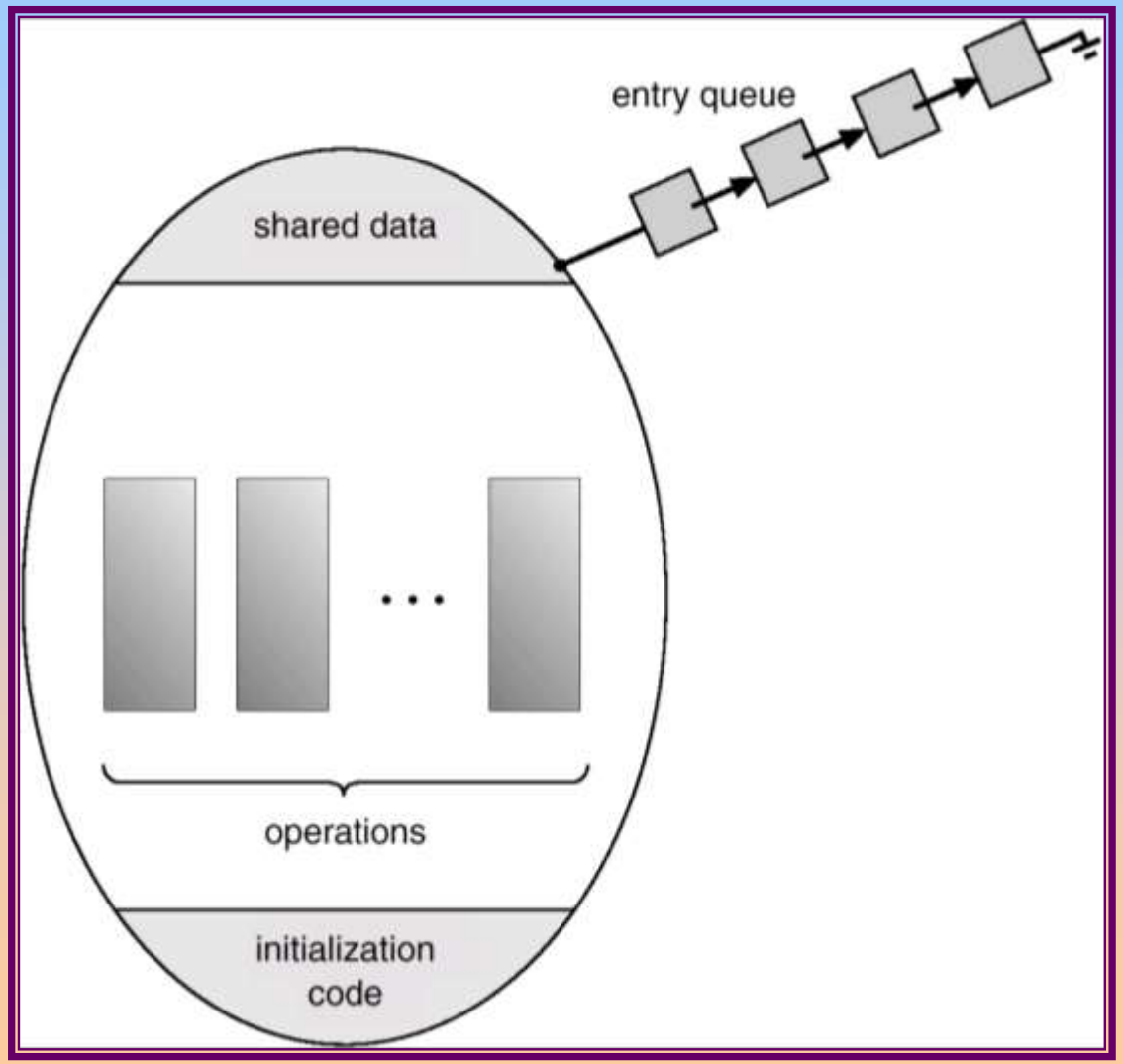


Monitor (II)

- Un monitor è un modulo software che contiene una o più procedure, una sequenza di inizializzazione e dati locali.
- Caratteristiche:
  - Le variabili locali sono accessibili solo dalle procedure del monitor e non dalle procedure esterne.
  - Un processo entra nel monitor chiamando una delle sue procedure.
  - Solo un processo alla volta può essere in esecuzione all'interno del monitor.
  - Ogni altro processo che ha chiamato il monitor è sospeso nell'attesa che questo diventi disponibile.
- Garantendo l'esecuzione di un solo processo alla volta un monitor può quindi essere utilizzato per la mutua esclusione.

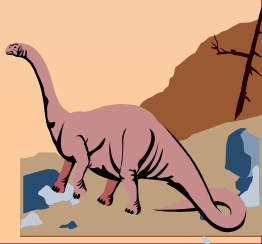


Schema di un monitor



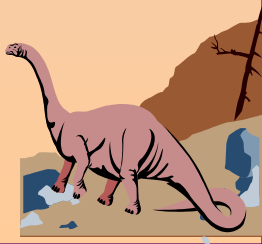
Variabili condition

- Un monitor deve contenere degli strumenti di sincronizzazione.
  - Ad esempio si supponga che un processo chiami il monitor e che mentre è al suo interno sia sospeso finché non si verifica una certa condizione.
  - E' necessario fare in modo che il processo sospeso rilasci il monitor in modo che altri processi possano entrare.
  - Quando in seguito la condizione viene soddisfatta e il monitor è nuovamente disponibile il processo deve essere riattivato e deve poter rientrare nel monitor nelllo stesso punto in cui era stato sospeso.
- Un monitor fornisce la sincronizzazione mediante l'uso di *variabili di condizione* accessibili solo dall'interno del monitor.

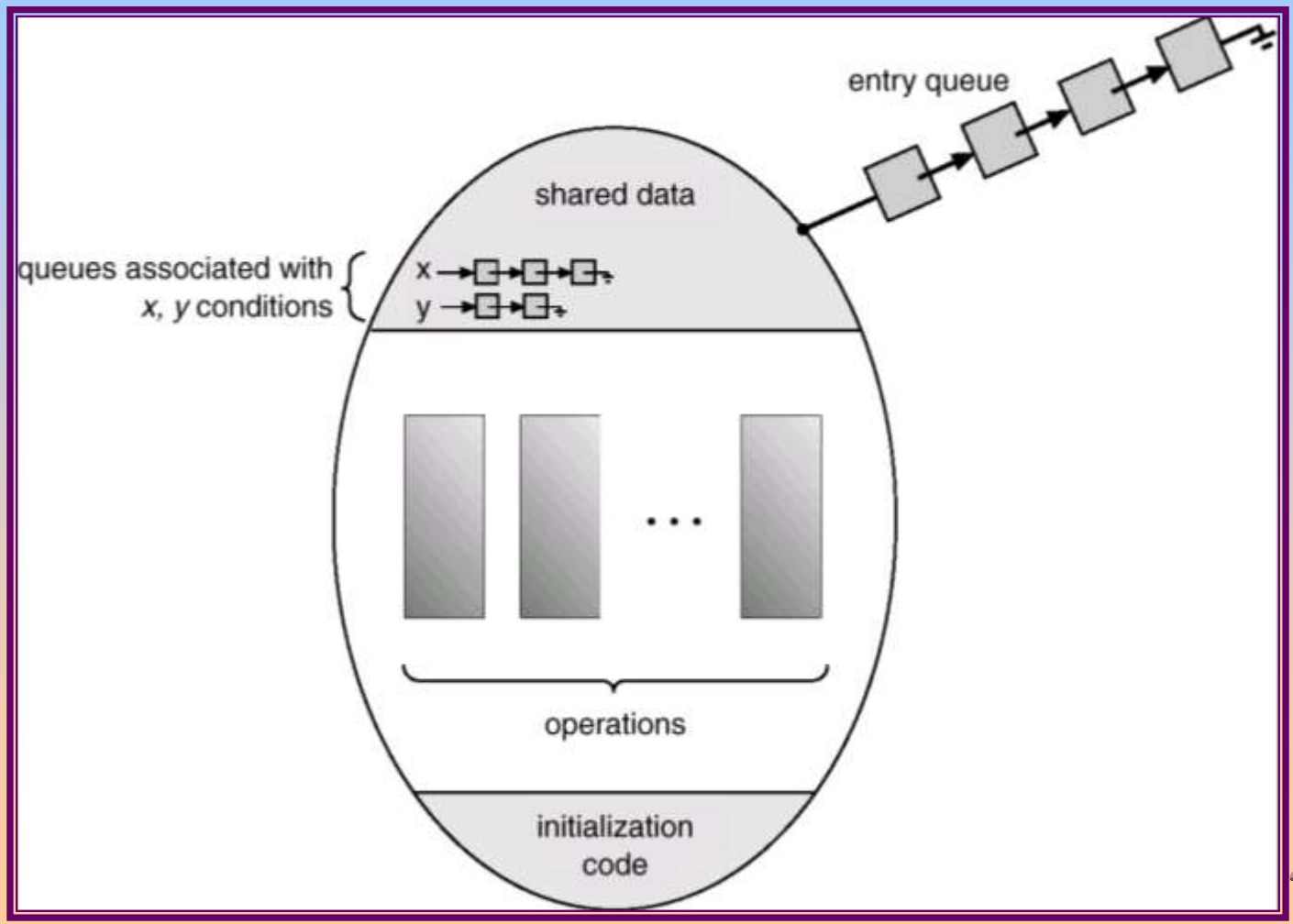


Variabili condition (II)

- Un programmatore che deve scrivere un proprio schema di sincronizzazione può definire una o più variabili condizionali tramite il costrutto **condition**.
- Ad es. **condition x, y;**
- Le uniche operazioni eseguibili su una variabile condition sono **wait** and **signal**.
  - L'operazione **x.wait ();** implica che il processo che la invoca rimanga sospeso fino a che un altro processo non invoca l'operazione **x.signal ();**
  - L'operazione **x.signal** risveglia esattamente un processo sospeso.
- Le operazioni di wait e signal sono quindi diverse da quelle sui semafori.
- Se non ci sono processi sospesi **signal** non ha alcun effetto (contrariamente alla signal sui semafori).



Schema di un monitor con variabili condition

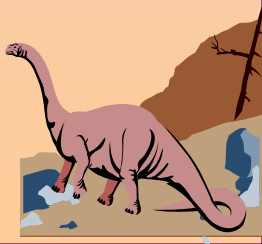


Una soluzione con monitor al problema dei cinque filosofi

```
monitor fc
   {
      enum {pensa, affamato, mangia} stato [5];
            // I tre possibili stati in cui può trovarsi un filosofo
      condition auto [5];

      void prende(int i)          // slide seguente
      void posa(int i)            // slide seguente
      void verifica(int i)        // slide seguente

      void inizializzazione() {
           for (int i = 0; i < 5; i++) stato[i] = pensa;
                                       }
   }
```



Una soluzione con monitor al problema dei cinque filosofi (II)

```
void prende (int i) {
        stato[i] = affamato;
        verifica (i);
        if (stato[i] != mangia) auto[i].wait();  }

void posa (int i) {
        stato[i] = pensa;
        // verifica lo stato dei vicini sinistro e destro
        verifica((i+4) % 5);
        verifica((i+1) % 5); }

void verifica (int i) {
        if ( (stato [(I + 4) % 5] != mangia) &&
          (stato[i] == affamato) &&
          (stato [(i + 1) % 5] != mangia)) {
                                stato[i] = mangia;
                                auto[i].signal();} }
```

- Risolto il problema dello stallo ma possibilità di attesa indefinita