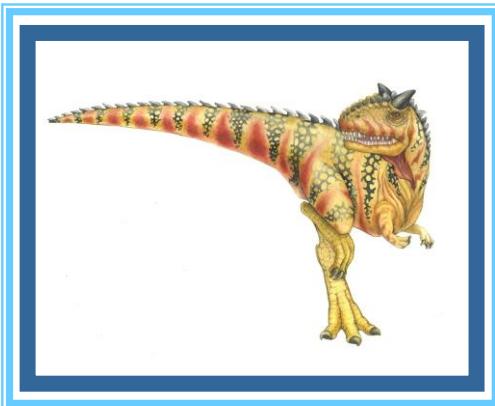
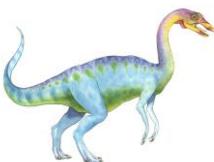


Strumenti di sincronizzazione

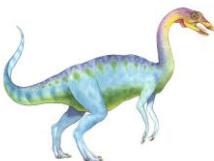




Obiettivi

- ❖ Descrivere il problema della sezione critica ed illustrare come avvengono le race condition
- ❖ Mostrare soluzioni hardware alla mutua esclusione
- ❖ Dimostrare come lock mutex, semafori, monitor e variabili condizionali possono essere utilizzati per gestire la mutua esclusione
- ❖ Valutare gli strumenti che risolvono il problema della sezione critica in scenari a bassa, moderata e alta contesa

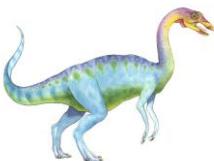




Sommario

- ❖ Introduzione
- ❖ Il problema della sezione critica
- ❖ La soluzione di Peterson
- ❖ Supporto hardware alla sincronizzazione
- ❖ Lock mutex
- ❖ Semafori
- ❖ Monitor
- ❖ Liveness, stallo e inversione di priorità
- ❖ Valutazione delle soluzioni

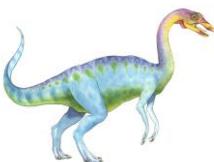




Introduzione – 1

- ❖ Nei moderni SO, i task vengono eseguiti concorrentemente, su un core, e in parallelo, su core distinti
- ❖ Un processo cooperante può influenzare gli altri processi in esecuzione nel sistema o subirne l'influenza
- ❖ I processi cooperanti possono...
 - ...condividere uno spazio logico di indirizzi, cioè codice e dati (thread, memoria condivisa)
 - ...oppure solo dati, attraverso file o messaggi
- ❖ L'accesso concorrente a dati condivisi può causare inconsistenza nei dati
- ❖ Per garantire la coerenza (l'integrità) dei dati occorrono meccanismi che assicurano l'esecuzione ordinata dei processi cooperanti

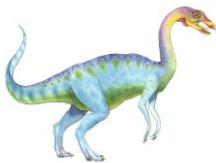




Introduzione – 2

- ❖ I processi possono essere interrotti dallo scheduler in qualsiasi punto del loro flusso di esecuzione, per assegnare la CPU (il core) ad un altro processo
- ❖ Affrontiamo di nuovo il problema del produttore–consumatore con buffer limitato
- ❖ Una soluzione in cui vengano impiegati tutti gli elementi del buffer presuppone...
 - ...la modifica del codice del produttore–consumatore con l'aggiunta di una variabile **contatore**, inizializzata a zero
 - **contatore** viene incrementato ogni volta che un nuovo elemento viene inserito nel buffer, decrementato dopo ogni estrazione





Dati condivisi



```
#define BUFFER_SIZE 10

typedef struct {

    . . .

} elemento;

elemento buffer[BUFFER_SIZE];

int in = 0;
int out = 0;
int contatore = 0;
```

in indica la successiva posizione libera nel buffer
→
out indica la prima posizione piena nel buffer
→
contatore conta il numero di elementi presenti nel buffer





Produttore–Consumatore

Processo produttore

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

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

Processo consumatore





Race condition – 1

- ❖ Sebbene siano corrette se si considerano separatamente, le procedure del produttore e del consumatore possono non “funzionare” se eseguite in concorrenza
- ❖ In particolare, le istruzioni:

```
contatore++;  
contatore--;
```

devono essere eseguite *atomicamente*





Race condition – 2

- ❖ Un'operazione è **atomica** quando viene completata senza subire interruzioni
- ❖ Le istruzioni di aggiornamento del contatore vengono realizzate in linguaggio macchina come...

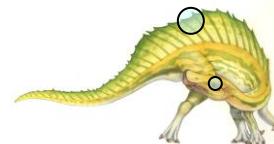
```
registro1 = contatore  
registro1 = registro1 + 1  
contatore = registro1
```

```
registro2 = contatore  
registro2 = registro2 - 1  
contatore = registro2
```

LOAD R1, CONT
ADD 1, R1
STORE R1, CONT

LOAD R2, CONT
SUB 1, R2
STORE R2, CONT

I due registri
possono
coincidere...





Race condition – 3

- ❖ Se il produttore ed il consumatore tentano di accedere al buffer contemporaneamente, poiché le istruzioni in linguaggio macchina possono risultare *interfogliate*, si ha una *race condition*
- ❖ La sequenza effettiva di esecuzione dipende da come i processi produttore e consumatore vengono schedulati
- ❖ **Esempio:** inizialmente contatore=5

T_0 produttore: **registro₁=contatore** ($\text{registro}_1=5$)

T_1 produttore: **registro₁=registro₁+1** ($\text{registro}_1=6$)

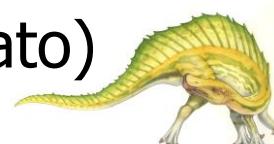
T_2 consumatore: **registro₂=contatore** ($\text{registro}_2=5$)

T_3 consumatore: **registro₂=registro₂-1** ($\text{registro}_2=4$)

T_4 produttore: **contatore=registro₁** ($\text{contatore}=6$)

T_5 consumatore: **contatore=registro₂** ($\text{contatore}=4$)

- ❖ **contatore** varrà 4, 5 o 6, mentre dovrebbe rimanere uguale a 5 (un elemento prodotto ed uno consumato)





Race condition – 4

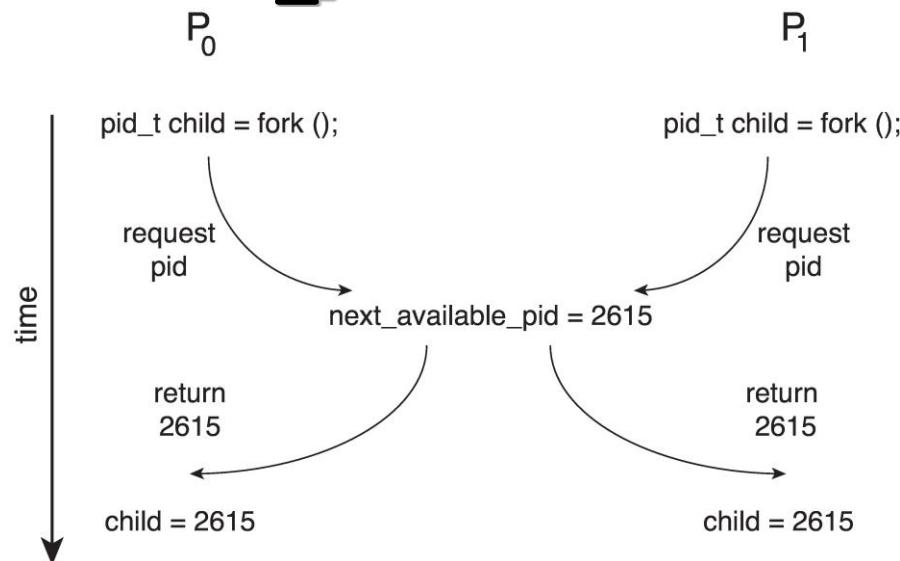
- ❖ *Race condition* — Più processi accedono in concorrenza e modificano dati condivisi; l'esito dell'esecuzione (il valore finale dei dati condivisi) dipende dall'ordine nel quale sono avvenuti gli accessi
- ❖ Per evitare le **corse critiche** occorre che i processi correnti vengano **sincronizzati**
 - Tali situazioni si verificano spesso nei SO, nei quali diversi task compiono operazioni su risorse condivise
 - Problema particolarmente significativo nei sistemi multicore, in cui i thread vengono eseguiti in parallelo su unità di calcolo distinte





Race condition – 5

- ❖ I processi P_0 e P_1 (eseguiti in parallelo) creano processi figlio utilizzando la chiamata di sistema **fork()**
- ❖ Race condition sulla variabile kernel che rappresenta il prossimo identificatore di processo disponibile (pid), **next_available_pid**



- ❖ Senza mutua esclusione, lo stesso pid potrebbe essere assegnato a due processi diversi





Il problema della sezione critica – 1

- ❖ n processi $\{P_0, P_1, \dots, P_{n-1}\}$ competono per utilizzare dati condivisi
- ❖ Ciascun processo è costituito da un segmento di codice, chiamato **sezione critica** (o *regione critica*), in cui accede a dati condivisi
- ❖ **Problema** — assicurarsi che, quando un processo accede alla propria sezione critica, a nessun altro processo sia concessa l'esecuzione di un'azione analoga
- ❖ L'esecuzione di sezioni critiche da parte di processi cooperanti è **mutuamente esclusiva** nel tempo

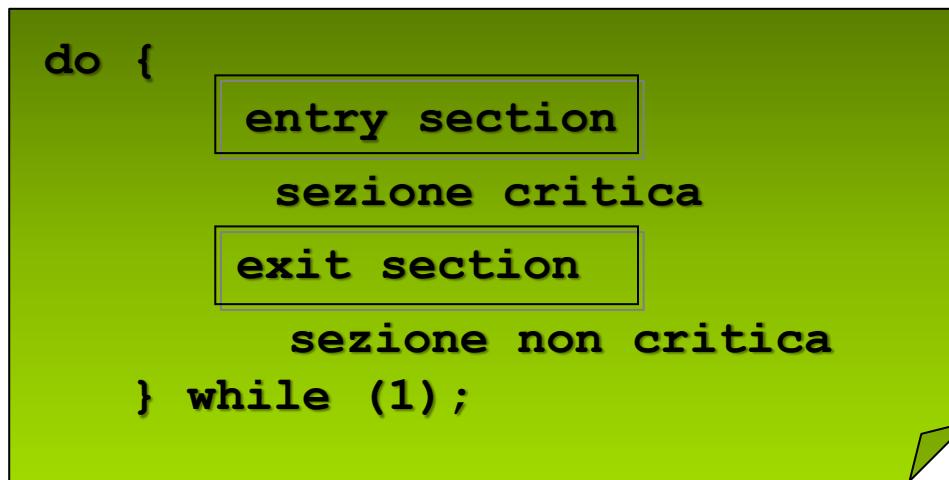




Il problema della sezione critica – 2

❖ **Soluzione** — progettare un protocollo di cooperazione fra processi:

- Ogni processo deve chiedere il permesso di accesso alla sezione critica, tramite una *entry section*
- La sezione critica è seguita da una *exit section*
- Il rimanente codice è non critico



Struttura generale del processo P_i





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 eseguire la propria sezione critica
2. **Progresso** — Se nessun processo è in esecuzione nella propria sezione critica ed esiste qualche processo che desidera accedervi, allora la selezione del processo che entrerà prossimamente nella propria sezione critica non può essere rimandata indefiniteamente
3. **Attesa limitata** — Se un processo ha effettuato la richiesta di ingresso nella sezione critica, è necessario porre un limite al numero di volte che si consente ad altri processi di entrare nelle proprie sezioni critiche, prima che la richiesta del primo processo sia stata accordata (politica *fair* per evitare la *starvation*)
 - Si assume che ciascun processo sia eseguito ad una velocità diversa da zero
 - Non si fanno assunzioni sulla velocità relativa degli n processi

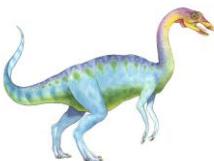




Sezione critica e kernel – 1

- ❖ In un dato istante, più processi in modalità utente possono richiedere servizi al SO
 - ➡ Il codice del kernel deve regolare gli accessi a dati condivisi
- ❖ **Esempio**
 - Una struttura dati del kernel mantiene una lista di tutti i file aperti nel sistema
 - ▶ Deve essere modificata ognqualvolta si aprono/chiudono file
 - ▶ Due processi che tentano di aprire file in contemporanea potrebbero ingenerare nel sistema una corsa critica

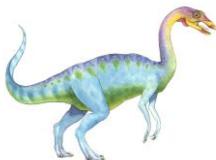




Sezione critica e kernel – 2

- ❖ **Kernel senza diritto di prelazione** — i processi vengono eseguiti fino a quando non escono dalla modalità kernel, si bloccano o restituiscono volontariamente la CPU
 - Immuni dai problemi legati all'ordine degli accessi alle strutture dati del kernel: non si consente l'interruzione forzata di processi attivi in modalità di sistema
 - Windows XP/2000
- ❖ **Kernel con diritto di prelazione** — si consente la prelazione di processi in esecuzione in modalità kernel
 - Critici per sistemi SMP, in cui due processi in modalità kernel possono essere eseguiti su processori distinti
 - Necessari in ambito real-time: permettono ai processi in tempo reale di far valere il loro diritto di precedenza
 - Utili nei sistemi time-sharing: diminuiscono il tempo medio di risposta
 - Linux, dal kernel versione 2.6





Soluzione di Peterson (1981)

- ❖ Soluzione per due processi P_0 e P_1 ($i=1-j$)
- ❖ I due processi condividono due variabili:
 - `int turno`
 - `boolean flag[2]`con entrambi gli elementi di `flag` inizializzati a `FALSE`
- ❖ La variabile `turno` indica il processo che “è di turno” per accedere alla propria sezione critica
- ❖ L’array `flag` si usa per indicare se un processo è pronto ad entrare nella propria sezione critica
 - `flag[i]=TRUE` implica che il processo P_i è pronto per accedere alla propria sezione critica





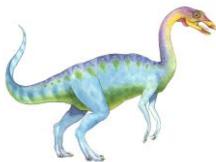
Algoritmo per il processo P_i

```
while(1) {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
        sezione critica  
        flag[i] = false;  
        sezione non critica  
}
```

P_0 : flag[0]=true
 P_1 : flag[1]=true
 P_1 : turn=0
 P_1 : while(flag[0] && turn=0);
 P_0 : turn=1
 P_0 : while(flag[1] && turn=1);
 P_1 entra in sezione critica
 P_1 : flag[1]=false
 P_0 entra in sezione critica
...
...

- ❖ **Soluzione** — Mutua esclusione garantita dal valore di **turn**; P_i entra nella sezione critica (progresso) al massimo dopo un ingresso da parte di P_j (attesa limitata): **alternanza stretta**





Soluzione di Peterson (cont.)

- ❖ La soluzione di Peterson è *basata sul software*, perché l'algoritmo garantisce la mutua esclusione senza richiedere alcun supporto speciale dal SO, né istruzioni hardware specifiche
- ❖ Sebbene rappresenti un utile approccio algoritmico, il funzionamento della soluzione di Peterson non è garantito sulle architetture attuali
- ❖ Capire perché l'approccio potrebbe non funzionare aiuta anche a comprendere le race condition
- ❖ Per migliorare le prestazioni, i processori e i compilatori possono riordinare le operazioni che non hanno interdipendenze
 - Nel caso di processi single-thread il risultato non cambia
- ❖ Tuttavia, nel multithreading, il riordino può produrre risultati inconsistenti o inaspettati





Esempio

- ❖ Due thread condividono i dati:

```
boolean flag = false;  
int x = 0;
```

- ❖ Il thread₁ esegue:

```
while (!flag);  
print x;
```

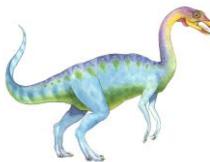
L'output dovrebbe essere 100, ma se le operazioni di thread₂, che sono indipendenti, vengono riordinate, si può stampare 0

- ❖ Mentre il thread₂ assegna:

```
x = 100;  
flag = true;
```

- ❖ Quale sarà l'output prodotto?





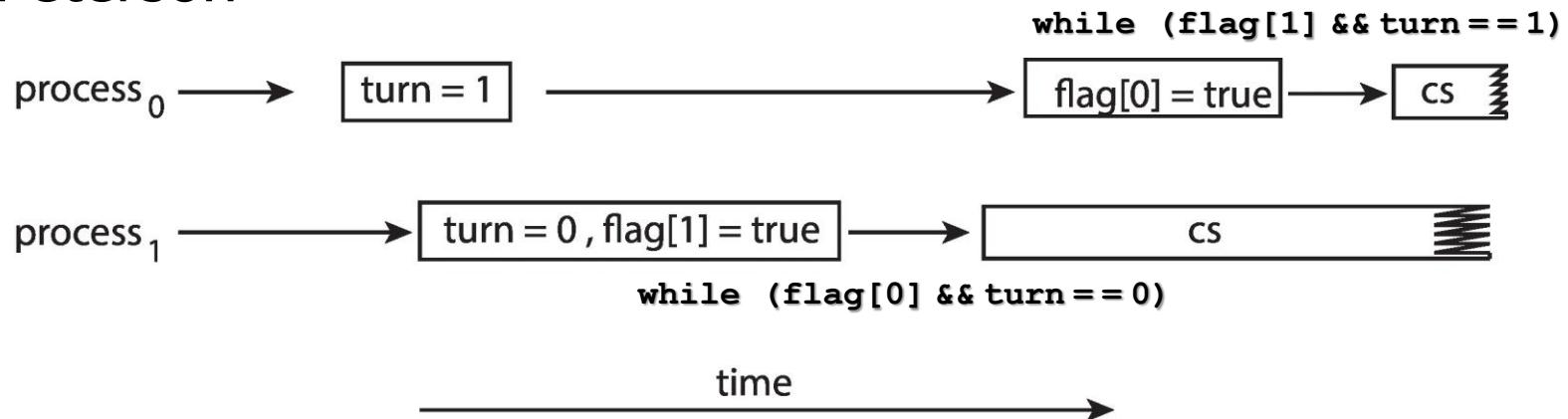
Soluzione di Peterson (cont.)

```
while(1) {  
    turn = j;  
    flag[i] = true;  
    while (flag[j] && turn == j);  
        sezione critica  
    flag[i] = false;  
        sezione non critica  
}
```

Processo P_i

P_0 : turn=1
 P_1 : turn=0
 P_1 : flag[1]=true
 P_1 : while(flag[0] && turn==0);
flag[0] è false $\rightarrow P_1$ entra in sezione critica
 P_0 : flag[0]=true
 P_0 : while(flag[1] && turn==1);
turn=0 $\rightarrow P_0$ entra in sezione critica

- ❖ Effetto dello scambio di istruzioni nella soluzione di Peterson



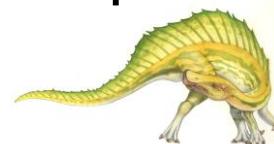
- ❖ Si consente ai processi l'accesso concorrente alla sezione critica





Hardware di sincronizzazione – 1

- ❖ Molti sistemi forniscono supporto hardware per risolvere efficacemente il problema della sezione critica
- ❖ In un sistema monoprocessoresso è sufficiente interdire le interruzioni mentre si modificano le variabili condivise
 - Il codice attualmente in esecuzione viene eseguito senza possibilità di prelazione
 - Soluzione inefficiente nei sistemi multiprocessoresso
 - SO non scalabili
- ❖ Tre forme di supporto hardware
 - Barriere di memoria
 - Istruzioni hardware
 - Variabili atomiche
- ❖ Queste primitive possono essere utilizzate direttamente come strumenti di sincronizzazione o costituire la base per costruire meccanismi di sincronizzazione più astratti





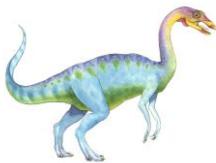
Hardware di sincronizzazione – 2

- ❖ In generale, qualsiasi soluzione al problema della sezione critica richiede l'uso di un semplice strumento, detto *lock* (lucchetto)
 - Per accedere alla propria sezione critica, il processo deve acquisire il possesso di un lock, che restituirà al momento della sua uscita

```
do {  
    acquisisce il lock  
    sezione critica  
    restituisce il lock  
    sezione non critica  
} while (1);
```

- L'implementazione del lock può essere a livello hardware o software

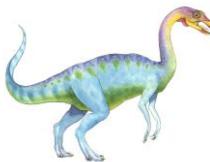




Barriere di memoria – 1

- ❖ Il modello di memoria è il modo in cui l'architettura di un computer determina quali garanzie relative alla memoria vengono fornite ai programmi applicativi
- ❖ Il modello di memoria può essere:
 - **Fortemente ordinato** — se una modifica alla memoria di un processore è immediatamente visibile a tutti gli altri
 - **Debolemente ordinato** — se una modifica alla memoria di un processore può non essere immediatamente visibile a tutti gli altri
- ❖ Una **barriera di memoria** (o recinzione di memoria) è un'istruzione che forza la propagazione a tutti i processori di ogni cambiamento alla memoria
- ❖ Sono meccanismi di basso livello utilizzati dagli sviluppatori di SO per garantire la mutua esclusione sui dati del kernel





Barriere di memoria – 2

- ❖ Relativamente all'esempio precedente, potremmo aggiungere una barriera di memoria sia in thread_1 che in thread_2 per garantire che thread_1 stampi il valore 100
- ❖ thread_1 :

```
while (!flag)
    memory_barrier();
print x;
```

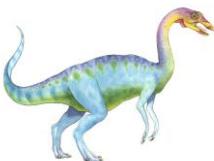
- ❖ thread_2 :

```
x = 100;
memory_barrier();
flag = true;
```



Nella soluzione di Peterson, inserire una barriera di memoria fra le due istruzioni che costituiscono la entry section garantisce l'esecuzione ordinata dei processi





Istruzioni hardware – 1

- ❖ Molte architetture attuali forniscono speciali istruzioni **atomiche** – cioè eseguibili come unità non prelazionabili e, sui multiprocessori, eseguite sequenzialmente – implementate in hardware
 - permettono di controllare e modificare il contenuto di una parola di memoria (**TestAndSet**)
 - oppure, di scambiare il contenuto di due parole di memoria (**CompareAndSwap**)





Istruzioni hardware – 2

```
boolean lock = FALSE;
```

```
boolean TestAndSet (boolean *object)
{
    boolean value = *object;
    *object = TRUE;
    return value;
}
```

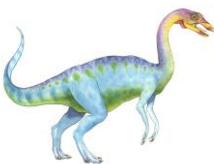
Definizione di TestAndSet

- eseguita atomicamente
- ritorna il valore originale del parametro passato
- pone il nuovo valore del parametro a TRUE

```
do {
    while TestAndSet (&lock);
    sezione critica
    lock = FALSE;
    sezione non critica
} while(1);
```

Realizzazione della mutua esclusione





Variabili atomiche – 1

- ❖ In genere, le istruzioni hardware per garantire la mutua esclusione vengono utilizzate come blocchi costruttivi per altri strumenti di sincronizzazione
- ❖ Uno di questi sono le **variabili atomiche**, normalmente definite su tipi di dati quali interi e booleani, che vengono aggiornate senza soluzione di continuità
- ❖ I SO che supportano le variabili atomiche forniscono anche tipi speciali di dati atomici e funzioni per l'accesso e la manipolazione di variabili di tali tipi
- ❖ Le variabili atomiche sono comunemente utilizzate nei SO e nelle applicazioni concorrenti, ma il loro uso è spesso limitato ad aggiornamenti di singoli dati condivisi, come contatori e generatori random
- ❖ Linux utilizza interi atomici per la sincronizzazione nel kernel





Variabili atomiche – 2

- ❖ **Esempio:** nel problema del produttore-consumatore con buffer limitato
 - Se **contatore** è una variabile atomica non vi sarà race condition all'atto dell'incremento/decremento
 - Una race condition invece si può avere nel controllo del ciclo **while**
 - Se due consumatori sono in attesa (nel loro ciclo **while**) e un produttore produce un elemento, entrambi potrebbero venire sbloccati e procedere al consumo di quell'unico elemento!





Lock mutex – 1

- ❖ Le soluzioni hardware al problema della sezione critica sono complicate e generalmente inaccessibili ai programmatore di applicazioni
- ❖ I progettisti di SO hanno implementato strumenti software per risolvere il problema: il più semplice è il **lock mutex**
- ❖ Si proteggono le regioni critiche prima acquisendo (**acquire()**) e quindi rilasciando (**release()**) il lock
 - Si definisce una variabile booleana **available** che indica se il lock è disponibile o meno

```
acquire() {  
    while (!available)  
        ; /* busy wait */  
    available = false;  
}
```

```
release() {  
    available = true;  
}
```





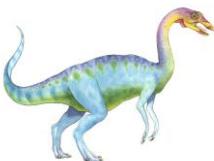
Lock mutex – 2

Processo P →

```
do {  
    acquire()  
    sezione critica  
    release()  
    sezione non critica  
} while (true);
```

- ❖ Le chiamate ad **acquire()** e **release()** devono essere atomiche
 - Normalmente vengono implementate tramite istruzioni (atomiche) hardware
- ❖ La soluzione tramite lock impone l'**attesa attiva**: mentre un processo si trova nella propria sezione critica, ogni altro processo che ne tenti l'accesso deve ciclare effettuando la chiamata ad **acquire()**
 - Per questo motivo si parla anche di **spinlock**





Semafori – 1

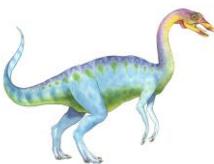
- ❖ I **semafori** rappresentano un approccio più sofisticato alla sincronizzazione fra processi
- ❖ Il semaforo **s** è una variabile intera
- ❖ Si può accedere al semaforo **s** (dopo l'inizializzazione) solo attraverso due operazioni indivisibili (operazioni atomiche, primitive) **wait()** e **signal()** (originariamente in olandese: **P** per *proberen*/verificare e **V** per *verhogen*/incrementare)

```
wait(S) {  
    while S<=0  
        ; // busy waiting  
    S--;  
}
```

Spinlock

```
signal(S) {  
    S++;  
}
```





Semafori – 2

- ❖ Tutte le modifiche al valore del semaforo contenute nelle operazioni **wait()** e **signal()** devono essere eseguite in modo indivisibile
 - Mentre un processo cambia il valore del semaforo, nessun altro processo può contemporaneamente modificare quello stesso valore
 - Nel caso di **wait()** devono essere effettuate atomicamente sia la verifica del valore del semaforo che il suo decremento





Sezione critica con n processi

- ❖ Variabili condivise:

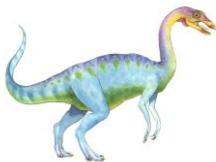
```
semaphore mutex = 1; // inizialmente mutex = 1
```

- ❖ Processo P_i :

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

Gli n processi condividono un semaforo, **mutex** per **mutual exclusion**





Definizione e uso dei semafori – 1

- ❖ **Semaforo contatore:** intero che può assumere valori in un dominio non limitato
- ❖ **Semaforo binario:** assume soltanto i valori 0 e 1
- ❖ Permettono l'approccio a diversi problemi di sincronizzazione
- ❖ Siano P_1 e P_2 due processi che contengono, rispettivamente due frammenti di codice C_1 e C_2 che devono essere eseguiti in sequenza
 - ⇒ Si crea il semaforo **synch** inizializzato a 0

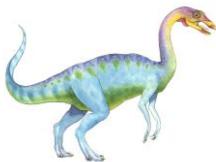
P_1 :

```
C1;  
signal(synch);
```

P_2 :

```
wait(synch);  
C2;
```





Definizione e uso dei semafori – 2

❖ Più in dettaglio...

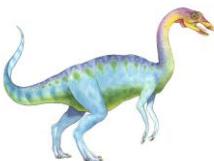
▪ Semaforo contatore

- ▶ Trova applicazione nel controllo dell'accesso a risorse presenti in un numero finito di esemplari
- ▶ Il semaforo è inizialmente impostato al numero di risorse disponibili
- ▶ I processi che desiderano utilizzare un'istanza della risorsa, invocano una **wait()** sul semaforo, decrementandone così il valore
- ▶ I processi che rilasciano una risorsa, invocano **signal()**, incrementando il valore del semaforo
- ▶ Quando il semaforo vale 0, tutte le istanze della risorsa sono allocate e i processi che le richiedono devono sospendersi sul semaforo

▪ Semaforo binario — simile al lock mutex

- ▶ Utilizzabili in maniera pressoché interscambiabile





Esempio

- ❖ Si consideri il seguente programma concorrente:

Variabili condivise

```
semaphore s1=0, s2=0;
```

processo P1

```
{  
    signal(s1);  
    wait(s2);  
    print "P"  
    wait(s2);  
    print "R"  
    signal(s1);  
}
```

processo P2

```
{  
    signal(s2);  
    wait(s1);  
    print "A";  
    signal(s2);  
    wait(s1);  
    print "I";  
}
```

- ❖ Descrivere il comportamento ed i possibili output dell'esecuzione parallela di P1 e P2

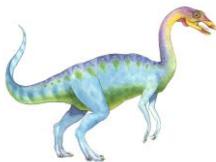




Esempio (cont.)

- ❖ Inizialmente uno qualsiasi tra P1 e P2 può essere selezionato per l'esecuzione della prima istruzione
- ❖ Il processo P1 deve attendere l'esecuzione di **signal(s2)** da parte di P2 e viceversa P2 attende l'esecuzione di **signal(s1)** da parte di P1
- ❖ L'esecuzione delle prime due stampe "P" in P1 e "A" in P2 può avvenire in qualsiasi ordine; dopo aver stampato "P", P1 attende che P2 esegua **signal(s2)** e quindi stampa "R" e "segnala" sul semaforo **s1** sul quale attende P2
- ❖ Solo dopo la **signal(s1)** eseguita da P1, P2 può stampare "I"
- ❖ I possibili output sono quindi due: la stringa "PARI" e la stringa "APRI"

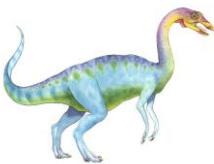




Implementazione dei semafori

- ❖ Dato che occorre garantire che due processi non eseguano mai **wait()** e **signal()** sullo stesso semaforo allo stesso istante (ovvero occorre garantire l'atomicità delle due operazioni)
 - ⇒ L'implementazione dei semafori impone che **wait()** e **signal()** vengano eseguite esse stesse all'interno di una sezione critica ⇒ possibile il busy waiting
 - Codice di implementazione molto ridotto (<10 istruzioni)
 - Brevi attese se la sezione critica è raramente occupata
- ❖ Per rendere le primitive semaforiche atomiche occorre:
 - Sui monoproessori, disabilitare le interruzioni
 - Impossibile sui multiprocessori → alternative all'uso dei semafori, costruite sulla base dello specifico hardware di sincronizzazione

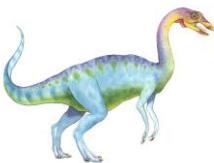




Busy waiting

- ❖ Viceversa, con i semafori spinlock, si crea una situazione di *busy waiting* (attesa attiva) dei processi in attesa ad un semaforo
 - ⇒ Poiché i processi possono passare anche molto tempo all'interno delle loro sezioni critiche, il semaforo spinlock può provocare notevole spreco di CPU da parte dei processi in attesa sullo stesso semaforo
- ❖ In altre parole... mentre un processo si trova nella propria sezione critica, qualsiasi altro processo che tenti di entrare nella sezione critica si trova nel ciclo del codice della entry section
- ❖ Il *busy waiting* o **attesa attiva** costituisce un problema per i sistemi multiprogrammati, perché la condizione di attesa spreca cicli di CPU che altri processi potrebbero sfruttare produttivamente
- ❖ L'attesa attiva è vantaggiosa solo quando è molto breve perché, in questo caso, può evitare i context switch





Da notare...

- ❖ Un lock è **conteso** se un thread si blocca mentre tenta di acquisirlo; i lock contesi possono essere soggetti ad una contesa elevata (un numero grande di thread che tentano di acquisire il lock contemporaneamente) o moderata (un numero piccolo di thread si contendono il lock) ⇒ i lock molto contesi tendono a ridurre le prestazioni complessive delle applicazioni concorrenti
- ❖ Gli spinlock sono il meccanismo di locking classico sui sistemi multiprocessore, quando il lock deve essere mantenuto per un breve periodo di tempo
 - Quanto breve?
 - Poiché l'attesa su un lock richiede due cambi di contesto (uno per spostare il thread in esecuzione nello stato di attesa ed uno per ripristinare il thread in attesa quando il lock diventa disponibile) ⇒ usare uno spinlock quando il lock deve essere trattenuto per una durata inferiore a due cambi di contesto





Come evitare il busy waiting – 1

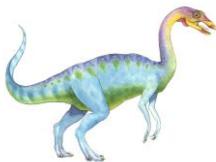
- ❖ Per evitare di lasciare un processo in attesa nel ciclo **while** si può ricorrere ad una definizione alternativa di semaforo
 - Associata ad ogni semaforo, vi è una coda di processi in attesa
 - La struttura semaforo contiene:
 - ▶ un valore intero (numero di processi in attesa)
 - ▶ un puntatore alla testa della lista dei processi

⇒ Si definisce un semaforo come un record:

```
typedef struct {  
    int valore;  
    struct processo *lista;  
} semaphore;
```

PCB

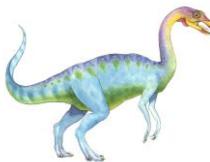




Come evitare il busy waiting – 2

- ❖ Si assume che siano disponibili due operazioni (fornite dal SO come system call):
 - **block** – posiziona il processo che richiede di essere bloccato nell'opportuna coda di attesa, ovvero sospende il processo che la invoca (stato waiting)
 - **wakeup (P)** – rimuove un processo P dalla coda di attesa e lo sposta nella ready queue (stato ready)





Come evitare il busy waiting – 3

- ❖ Le operazioni sui semafori possono essere definite come...

```
wait(semaphore *S) {
    S->valore--;
    if (S->valore < 0) {
        aggiungi il processo P a S->lista;
        block();
    }
}
```

```
signal(semaphore *S) {
    S->valore++;
    if (S->valore <= 0) {
        rimuovi il processo P da S->lista;
        wakeup(P);
    }
}
```





Come evitare il busy waiting – 4

- ❖ Mentre la definizione classica di semaforo ad attesa attiva è tale che il valore del semaforo non è mai negativo, il semaforo appena descritto (ed inizializzato a 1) può assumere valori negativi (nel campo **valore**)
- ❖ Se **valore** è negativo, $|\text{valore}|$ è il numero dei processi che attendono al semaforo
 - Ciò avviene a causa dell'inversione dell'ordine delle operazioni di decremento e verifica del campo valore (del semaforo) nella **wait()**





Errori comuni

- ❖ Errori comuni nell'uso dei semafori:

- **signal (mutex)** ... **wait (mutex)**

inversione delle chiamate di sistema: più processi possono eseguire le proprie sezioni critiche in contemporanea

- **wait (mutex)** ... **wait (mutex)**

doppia chiamata di una stessa system call: si genera un deadlock

- Omissione di **wait (mutex)**, **signal (mutex)** o di entrambe → violazione della mutua esclusione o stallo





Esempio 1

- ❖ Si considerino due processi, un produttore P ed un consumatore C, ed un buffer con capacità pari ad 1
- ❖ Sia

```
semaphore empty=1;
```

l'unica variabile condivisa (oltre al buffer)

- ❖ Si considerino le seguenti porzioni di codice includenti la sezione critica per l'accesso al buffer da parte di P e C

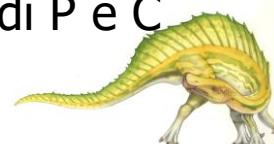
Processo P

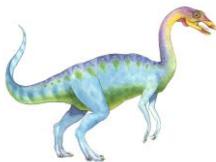
```
while(true) {  
    p1    wait(empty);  
    p2    enter_item(item);  
}
```

Processo C

```
while(true) {  
    c1    remove_item(item);  
    c2    signal(empty);  
}
```

- ❖ Supponendo che le istruzioni p1 e p2 vengano eseguite per prime, il sistema si comporta correttamente come un produttore-consumatore? Descrivere, se esiste, una sequenza di esecuzione delle istruzioni di P e C non corretta

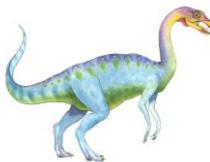




Esempio 1 (cont.)

- ❖ Il comportamento non è corretto dato che il protocollo per l'accesso ai dati condivisi (entry–exit section) non viene rispettato
- ❖ Sequenza scorretta
 - p1, p2, c1, c2, c1: attenzione, il consumatore consuma da un buffer vuoto!



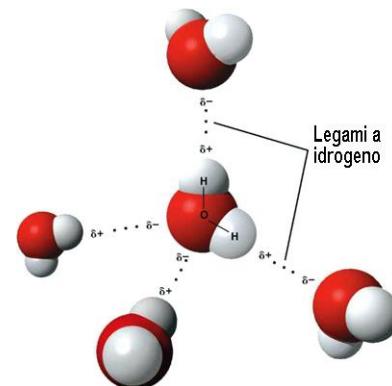


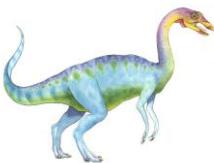
Esempio 2

- ❖ Il problema delle molecole di H_2O – Un sistema è costituito da due processi concorrenti: un processo costruttore di atomi di idrogeno e un processo costruttore di atomi di ossigeno; ogni processo ciclicamente produce un atomo e poi invoca una procedura comune (nell'esempio è una procedura di stampa)
- ❖ Si vuole che l'attività dei due processi sia sincronizzata in modo tale che le chiamate di questa procedura siano eseguite con gli input nell'ordine HHOHHOOHHO...
- ❖ Siano

semaphore Hy=0 , Oxy=2 ;

le variabili condivise





Esempio 2 (cont.)

- ❖ I processi Idrogeno ed Ossigeno possono essere descritti come segue:

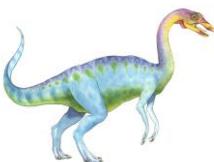
Processo Idrogeno

```
{  
while(1)  
{  
    <make a hydrogen atom>  
    wait(Oxy) ;  
    write(H) ;  
    signal(Hy) ;  
}  
}
```

Processo Ossigeno

```
{  
while(1)  
{  
    <make an oxygen atom>  
    wait(Hy) ;  
    wait(Hy) ;  
    write(O) ;  
    signal(Oxy) ;  
    signal(Oxy) ;  
}  
}
```

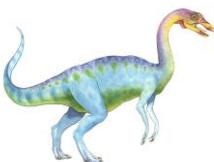




Semafori e monitor

- ❖ Benché i semafori costituiscano un meccanismo pratico ed efficace per la sincronizzazione dei processi, il loro uso scorretto può generare errori difficili da individuare, in quanto si manifestano solo in presenza di particolari sequenze di esecuzione
- ❖ I monitor semplificano molto la gestione della mutua esclusione (meno possibilità di errore)
- ❖ Sono veri costrutti, non funzioni di libreria ⇒ modifica dei compilatori
- ❖ Problema comune sia con i semafori che con i monitor: è necessario avere memoria condivisa (costrutti non applicabili in sistemi distribuiti)





I monitor – 1

- ❖ Il **monitor** è un costrutto di alto livello (un **ADT**, per *Abstract Data Type*) che fornisce un meccanismo efficiente per la sincronizzazione dei processi
- ❖ Incapsula dati mettendo a disposizione metodi per operare su di essi
- ❖ La rappresentazione di un tipo monitor è costituita da:
 - dichiarazioni di variabili i cui valori definiscono lo stato di un'istanza del tipo
 - procedure o funzioni che realizzano le operazioni sulle variabili definite nel monitor
- ❖ Supportati (implementati in certa misura) da C#, Java, Concurrent Pascal



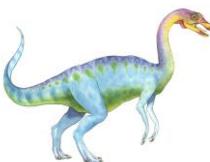


I monitor – 2

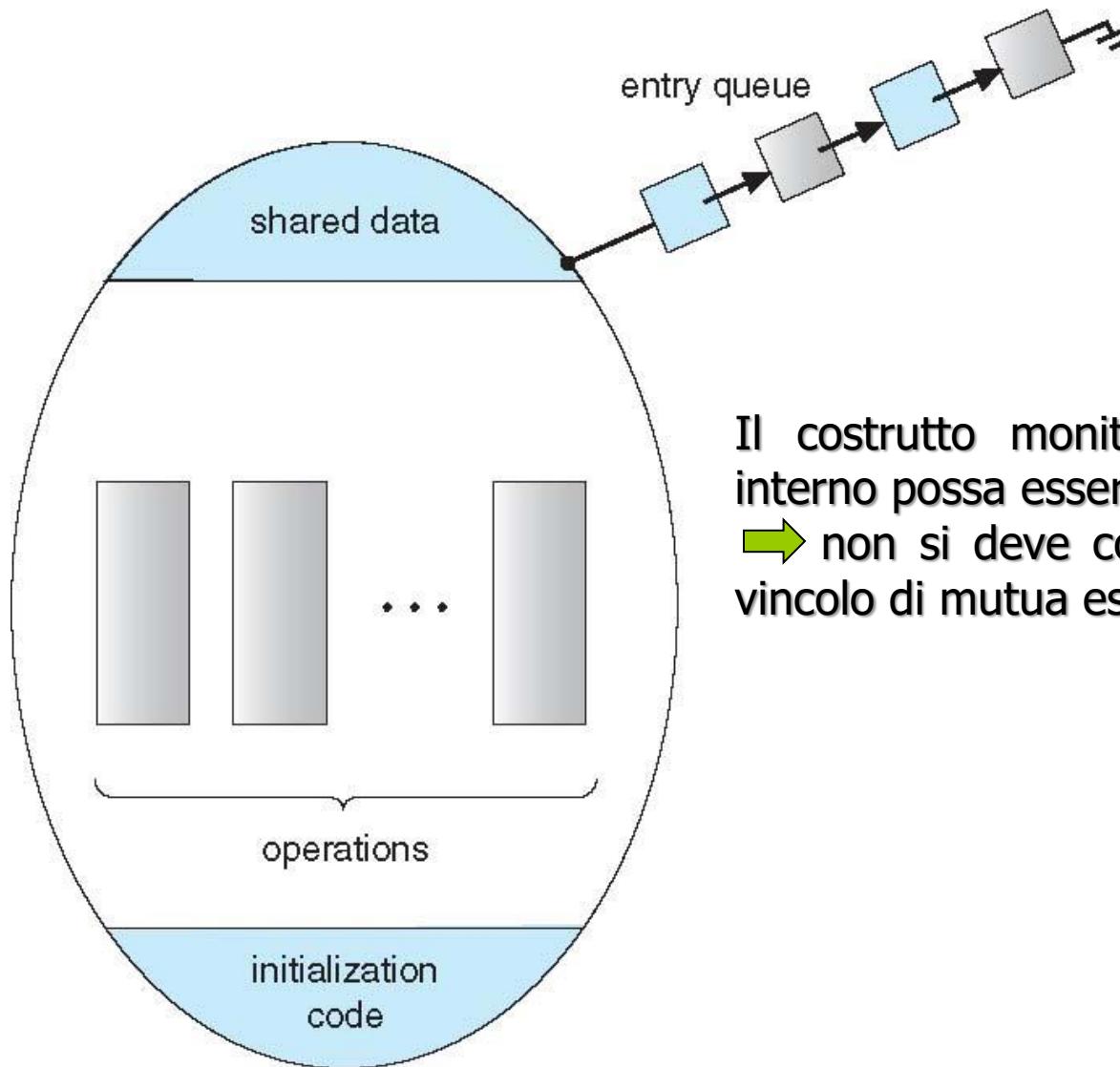
```
monitor monitor-name
{
    // dichiarazione delle variabili condivise

    function P1 (...) {
        . . .
    }
    function P2 (...) {
        . . .
    }
    function Pn (...) {
        . . .
    }
    {
        // codice di inizializzazione
    }
}
```





I monitor – 3



Il costrutto monitor assicura che al suo interno possa essere attivo un solo processo
→ non si deve codificare esplicitamente il vincolo di mutua esclusione



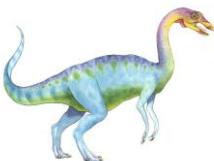


I monitor – 4

- ❖ Per effettuare l'allocazione di risorse tramite monitor, un processo deve richiamare una routine interna al monitor
 - ❖ La mutua esclusione è rigidamente forzata ai confini del monitor stesso, dove può accedere un solo processo alla volta
 - ❖ Tuttavia, tale definizione di monitor non è sufficientemente potente per modellare alcuni schemi di sincronizzazione
- ➡ Per permettere ad un processo di attendere dopo l'ingresso al monitor, causa occupazione della risorsa richiesta, si dichiarano apposite variabili *condition*

```
condition x,y;
```





I monitor – 5

- ❖ Le variabili condition possono essere usate solo in relazione a specifiche operazioni **wait()** e **signal()**
 - L'operazione

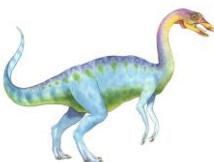
x.wait();

fa sì che il processo chiamante rimanga sospeso fino a che un diverso processo non effettui la chiamata

x.signal();

- L'operazione **x.signal()** attiva esattamente un processo sospeso; se non esistono processi sospesi l'operazione **signal()** non ha alcun effetto





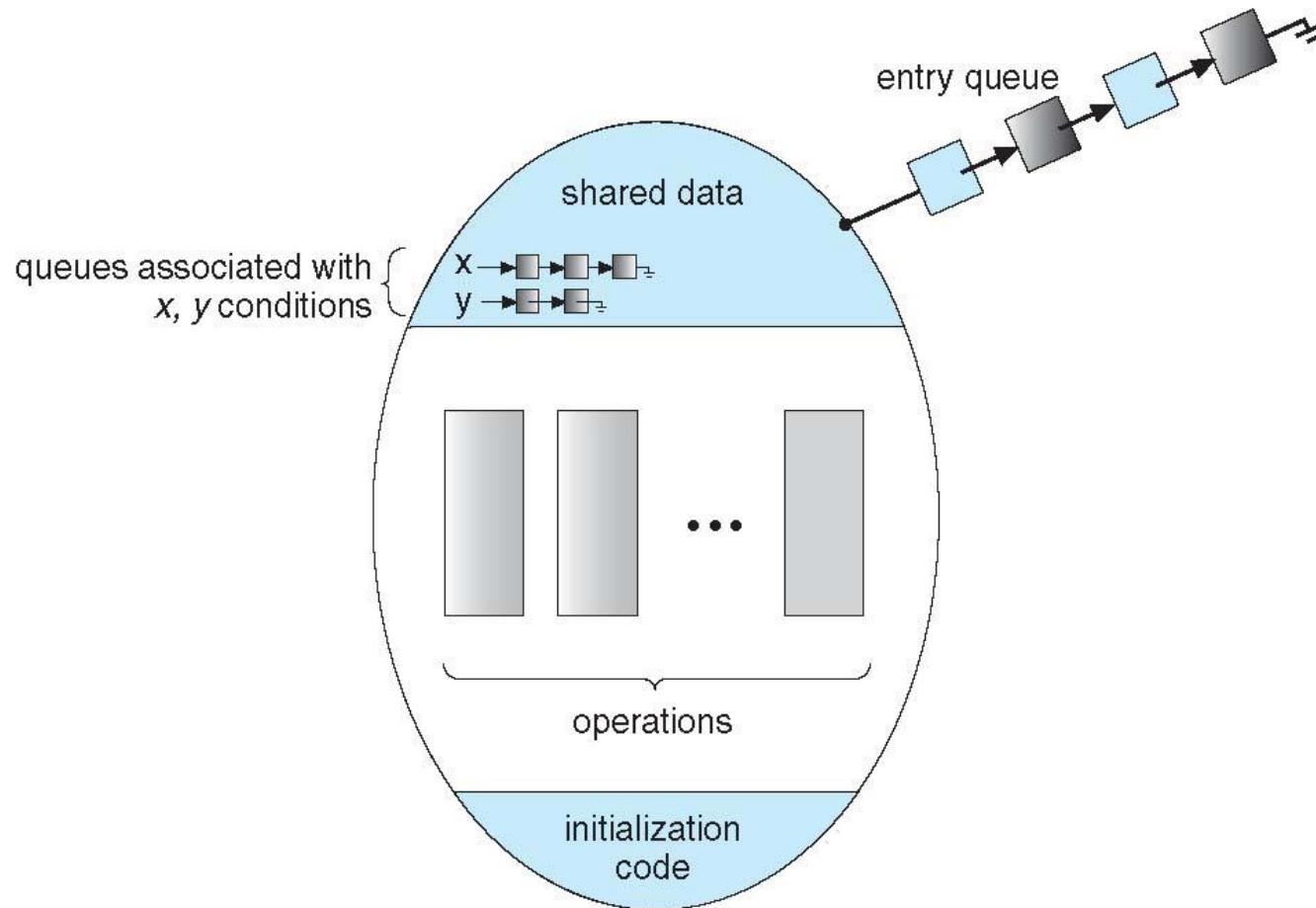
I monitor – 6

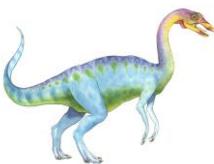
- ❖ Un processo utilizza una variabile condition (interna al monitor) per attendere il verificarsi di una certa condizione, uscendo dal monitor nell'attesa
- ❖ Un monitor associa una diversa variabile condition ad ogni situazione che potrebbe determinare l'attesa di un processo
- ❖ Un processo può uscire dal monitor mettendosi in attesa su una variabile condition o completando l'esecuzione del codice protetto dal monitor





Monitor con variabili condition





I monitor – 7

- ❖ Le variabili condition si differenziano dalle variabili convenzionali poiché ognuna di esse ha una coda associata
 - Un processo che richiama una **wait()** su una variabile condition particolare è posizionato nella sua coda e, finché vi rimane, è al di fuori del monitor, in modo che un altro processo possa entrarvi e richiamare una **signal()**
 - Un processo che richiama una **signal()** su una variabile condition particolare fa sì che un processo in attesa su quella variabile sia rimosso dalla coda associata e possa accedere al monitor
- ❖ Gestione FIFO o a priorità delle code dei processi
 - **conditional-wait** si ottiene con il costrutto **x.wait(c)**, dove **c** è un numero che rappresenta la priorità
 - Processi a più alta priorità vengono schedulati per primi

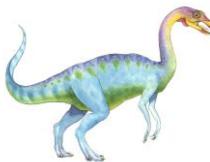




Semantica delle operazioni su variabili condition

- ❖ Se il processo P invoca **signal()** su una variabile condition ed esiste un processo Q in coda su quella variabile
 - **Segnalare ed attendere:** nel monitor entra Q e P attende (fuori dal monitor) che Q abbia terminato o si metta in attesa su una condition
 - **Segnalare e proseguire:** Q attende che P lasci il monitor o si metta in attesa su una condition
- ❖ Pros & cons
 - Dato che P era già in esecuzione nel monitor **segnalare e proseguire** sembra più ragionevole
 - Però... quando P termina, la condizione attesa da Q potrebbe non essere più valida
- ❖ Soluzione adottata da Concurrent Pascal: P deve eseguire la **signal()** come ultima istruzione prima di uscire dal monitor
- ❖ Variabili condizionali implementate in C# e Java





Esempio 1

- ❖ Allocazione di una risorsa a singola istanza tra **N** processi in competizione, che ne specificano il tempo massimo di utilizzo

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
    initialization code() {
        busy = FALSE;
    }
}
```



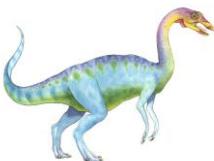


Esempio 1 (cont.)

- ❖ Se **R** è un'istanza del tipo **ResourceAllocator**, il protocollo di accesso al monitor per l'uso della risorsa per un tempo **t** è:

```
R.acquire(t);  
...  
access the resource;  
...  
R.release();
```

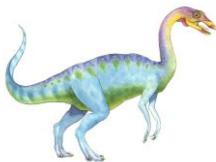




Esempio 1 (cont.)

- ❖ Sfortunatamente, il concetto di monitor non può garantire che la sequenza di acquisizione della risorsa sia rispettata
 - ⇒ Possibile accesso senza permesso o non rilascio della risorsa contesa
- ❖ Soluzione: inclusione della “risorsa” all’interno del monitor
 - Ma, in questo caso... lo scheduling sulla coda dei processi che si contendono la risorsa non è più gestito dal SO, ma coincide con lo scheduling (la politica di gestione) di accesso al monitor

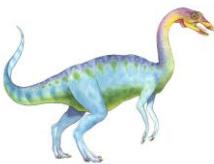




Liveness

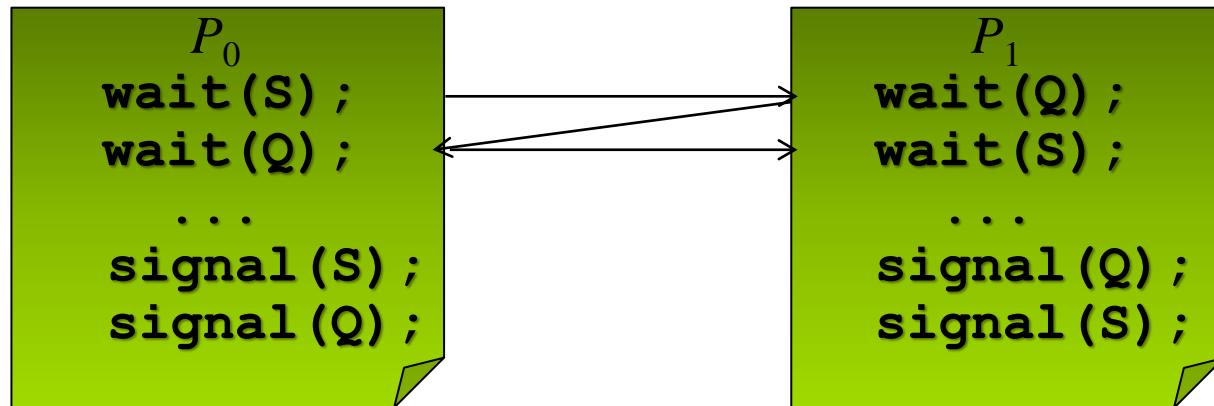
- ❖ I processi potrebbero dover attendere indefinitamente durante il tentativo di acquisire uno strumento di sincronizzazione come un lock mutex o un semaforo
- ❖ L'attesa infinita viola i requisiti di progresso e attesa limitata
- ❖ Il concetto di **liveness** fa riferimento a un insieme di proprietà che un sistema deve soddisfare per garantire che i processi progrediscano
- ❖ L'attesa infinita è un esempio di fallimento della liveness

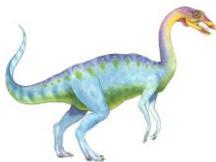




Deadlock

- ❖ La realizzazione di un semaforo con coda di attesa può condurre a situazioni in cui ciascun processo attende l'esecuzione di un'operazione **signal()**, che solo uno degli altri processi in coda può causare
- ❖ Più in generale, si verifica una situazione di **deadlock**, o **stallo**, quando due o più processi attendono indefinitamente un evento che può essere causato soltanto da uno dei processi in attesa
- ❖ Siano **S** e **Q** due semafori inizializzati entrambi ad 1

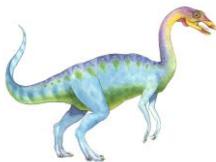




Starvation

- ❖ **Starvation** – Un processo può attendere per un tempo indefinito nella coda di un semaforo senza venir mai rimosso
- ❖ L'attesa indefinita (senza blocco) si può verificare qualora i processi vengano rimossi dalla lista associata al semaforo in modalità LIFO (*Last-In-First-Out*)
- ❖ Viceversa, per aggiungere e togliere processi dalla “coda al semaforo”, assicurando un’attesa limitata, la lista può essere gestita in modalità FIFO





Inversione di priorità – 1

- ❖ **Quando:** un processo a priorità più alta ha bisogno di leggere o modificare dati a livello kernel utilizzati da un processo, o da una catena di processi, a priorità più bassa
- ❖ **Perché:** poiché i dati a livello kernel sono tipicamente protetti da un lock, il processo a priorità maggiore dovrà attendere finché il processo a priorità minore non avrà finito di utilizzare le risorse



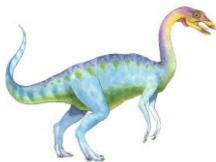


Inversione di priorità – 2

❖ Esempio

- Siano dati 3 task, T_1, T_2, T_3 , con rispettive priorità $p_1 > p_2 > p_3$
- Supponiamo che T_1 e T_3 debbano accedere alla risorsa a cui è associato il mutex **s**
- Se T_3 inizia la sua fase di elaborazione ed esegue una **wait(s)** prima che T_1 inizi la propria, quando T_1 tenterà di eseguire **wait(s)** verrà bloccato per un tempo non definito, cioè fino a quando T_3 non eseguirà **signal(s)**
 - ▶ T_1 viene penalizzato a favore di T_3 a dispetto delle priorità $p_1 > p_3$ (**blocco diretto**)
- Se, prima che T_3 esegua **signal(s)**, inizia l'esecuzione di T_2 , T_3 verrà sospeso per permettere l'elaborazione di $T_2 \Rightarrow T_1$ dovrà attendere che anche T_2 finisca (**blocco indiretto**)





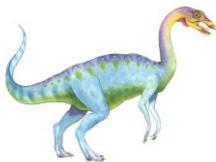
Inversione di priorità – 3

❖ Soluzioni

- Limitare a due il numero di possibili valori di priorità (poco realistico)
- Implementare un **protocollo di ereditarietà delle priorità**
 - ▶ Tutti i processi che stanno accedendo a risorse di cui hanno bisogno processi con priorità maggiore ereditano la priorità più alta finché non rilasciano l'uso delle risorse contese
 - ▶ Quando hanno terminato, la loro priorità ritorna al valore originale

❖ Esempio famoso: Mars Pathfinder/Sojourner (1997)





Valutazione delle soluzioni – 1

- ❖ La scelta di un meccanismo per affrontare le race condition può influire sulle prestazioni del sistema
 - Le soluzioni basate sull'hardware (CAS – ottimistiche e difficili da sviluppare e da testare) permettono di costruire algoritmi privi di lock che forniscono protezione dalle race condition
 - Gli interi atomici sono più leggeri e più appropriati dei lock mutex o dei semafori per aggiornamenti di singole variabili condivise (come i contatori)
 - I lock mutex sono più semplici e comportano meno overhead dei semafori; sono preferibili ai semafori binari per proteggere l'accesso alla sezione critica
 - Tuttavia, nel controllo dell'accesso ad un numero limitato di risorse, un semaforo contatore è generalmente più adatto di un lock mutex





Valutazione delle soluzioni – 2

- ❖ L'attrattiva di strumenti di alto livello (semafori, monitor, variabili condizionali) si basa sulla loro semplicità d'uso
 - Possibile overhead significativo, in dipendenza dall'implementazione
 - Poco scalabili a situazioni a contesa elevata
- ❖ In futuro:
 - Progettazione di compilatori che generano codice più efficiente
 - Sviluppo di nuovi linguaggi che forniscono supporto alla programmazione concorrente
 - Miglioramento delle prestazioni delle librerie e delle API esistenti



Fine del Capitolo 6

