

Sistemi Operativi

Docente: Ugo Erra
ugoerr+so@dia.unisa.it



6° LEZIONE SINCRONIZZAZIONE DEI PROCESSI

*CORSO DI LAUREA TRIENNALE IN INFORMATICA
UNIVERSITA' DEGLI STUDI DELLA BASILICATA*



Sommario della lezione



- Il problema della sincronizzazione
- La sezione critica
- Hardware per la sincronizzazione
- Semafori
- Stallo e attesa indefinita
- Problemi tipici di sincronizzazione
 - Produttori/Consumatori
 - Lettori/Scrittori
 - 5 filosofi

Il problema della sincronizzazione



- Nei Sistemi Operativi multiprogrammati diversi processi o thread sono in esecuzione asincrona e possono condividere dati
 - I processi condividono normalmente file
 - I thread condividono lo stesso spazio logico di indirizzo (codice e dati)
- L'accesso concorrente a dati condivisi può comportare l'incoerenza dei dati nel momento in cui non si adottano politiche di sincronizzazione

Produttore/Consumatore



- Consideriamo il problema del produttore e del consumatore dove:
 - Un produttore genera degli oggetti
 - Un consumatore preleva gli oggetti
- Supponiamo che il buffer in cui gli oggetti vengono inseriti e prelevati permette di inserire al più `DIM_BUFFER` elementi
- Useremo una variabile `contatore` che si incrementa ogni volta aggiungiamo un elemento nel buffer e si decrementa ogni volta si preleva un elemento dal buffer
 - La variabile `contatore` è condivisa sia dal produttore che dal consumatore

Produttore/Consumatore



Produttore

```
while (true)
{
    /* produce un elemento in appena_Prodotto */
    while (contatore == DIM_BUFFER);
    /* non fa niente */
    buffer [inserisci] = appena_Prodotto;
    inserisci = (inserisci + 1) % DIM_BUFFER;
    contatore++;
}
```

Consumatore

```
while (true)
{
    while (contatore == 0);
    /* non fa niente */
    da_Consumare = buffer[preleva];
    preleva = (preleva + 1) % DIM_BUFFER;
    contatore--;
    /* consuma un elemento da da_Consumare */
}
```

Inconsistenza dei dati



- La variabile contatore è usata in modo concorrente dal produttore e dal consumatore e quindi il suo valore potrebbe essere inconsistente ad un certo punto
 - Ad esempio se `contatore` è uguale a 5 e sia il produttore che il consumatore eseguono `contatore++` e `contatore--` il valore della variabile potrebbe essere 4, 5 o 6
- Le istruzioni `contatore++` e `contatore--` potrebbero essere implementate in assembly nel seguente modo:

```
registro1 := contatore
registro1 := registro1 + 1
contatore := registro
```

```
registro2 := contatore
registro2 := registro2 - 1
contatore := registro
```

- L'esecuzione concorrente di queste due istruzioni può portare alla seguente sequenza di istruzioni intercalate:

○ T_0 : produttore	esegue	registro ₁ = contatore	{registro ₁ = 5}
T_1 : produttore	esegue	registro ₁ = registro ₁ + 1	{registro ₁ = 6}
T_2 : consumatore	esegue	registro ₂ = contatore	{registro ₂ = 5}
T_3 : consumatore	esegue	registro ₂ = registro ₂ - 1	{registro ₂ = 4}
T_4 : produttore	esegue	contatore = registro ₁	{contatore = 6}
T_5 : consumatore	esegue	contatore = registro ₂	{contatore = 4}

Race Condition



- L'ordine con cui si accede ai dati condivisi è importante e determina la consistenza dei dati (**race condition**)
- La condizione deve essere che un solo processo alla volta può modificare i dati attraverso una forma di **sincronizzazione e coordinazione dei processi**

Sezione critica



- Supponiamo di avere n processi $\{P_0, P_1, \dots, P_n\}$ ognuno con una porzione di codice, chiamata sezione critica, in cui può modificare dati condivisi
 - Ad esempio una variabile, un file, etc.
- L'esecuzione della **sezione critica** da parte dei processi deve essere *mutuamente esclusiva* nel tempo
 - Un solo processo alla volta può trovarsi nella propria sezione critica
- Un processo avrà la seguente struttura

```
do{  
  sezione d'ingresso  
  sezione critica  
  sezione d'uscita  
  sezione non critica  
}while (true)
```

Requisiti per la soluzione della sezione critica



- **Mutua esclusione**

- Se il processo P_i è in esecuzione nella sua sezione critica nessun altro processo può trovarsi nella propria sezione critica

- **Progresso**

- Se nessun processo è in esecuzione nella propria sezione critica ed alcuni processi desiderano entrarci solo i processi fuori dalla propria sezione critica possono decidere chi entrerà. Questa decisione non può essere rimandata indefinitamente. *(In altre parole, qualsiasi processo deve riuscire ad entrare nella propria sezione critica)*

- **Attesa limitata**

- Deve esistere un limite al numero di volte che si consente ai processi di entrare nella propria sezione critica dal momento in cui un processo precedentemente ha richiesto di farlo. *(In altre parole, un processo entrerà prima o poi in tempo finito nella propria sezione critica)*
 - ✦ Si assume che la velocità dei processi è diversa da zero
 - ✦ Nessuna ipotesi si può fare sulla **velocità relativa** degli n processi

Problema della sezione critica



- In conclusione, per una corretta gestione della sezione critica bisogna garantire **mutua esclusione, progresso e attesa illimitata**, indipendentemente dalla velocità relativa dei processi
 - Sotto queste condizioni non ha importanza cosa accade all'interno della sezione critica

Sezione critica nel Sistema Operativo - 1



- All'interno del Sistema Operativo la gestione della sezione critica è un aspetto particolarmente delicato
 - Ad esempio, due processi utente che aprono un file attraverso una system call effettueranno due accessi concorrenti alla stessa struttura dati del S.O. cioè la tabella dei file aperti
 - Ogni struttura dati gestita dal sistema operativo non deve essere lasciata in uno stato inconsistente a causa dei due processi concorrenti

Sezione critica nel Sistema Operativo - 2



- Nella progettazione di un Sistema Operativo bisogna decidere in che modo gestire le sezioni critiche
- Le scelte possibili si riflettono nella progettazione di uno dei seguenti kernel:
 - **Kernel senza diritto di prelazione** - un processo in kernel mode non può essere interrotto da un altro processo
 - **Kernel con diritto di prelazione** - un processo in kernel mode può essere interrotto da un altro processo (ad esempio perché è scaduto il quanto di tempo)

Kernel senza diritto di prelazione



- Progettare un kernel senza diritto di prelazione è più semplice in quanto è sufficiente disattivare gli interrupt quando un processo è in modalità kernel
 - Ad esempio, questo impedisce che l'interrupt del timer venga eseguito quando il quanto di tempo di un processo si è esaurito
- Poiché un solo processo può essere in modalità kernel non c'è più il rischio che più processi possano accedere alle strutture dati del kernel
 - La disabilitazione degli interrupt dura il tempo necessario ad una system call di accedere in modo mutuamente esclusivo ad una qualche struttura dati del Sistema Operativo
- Normalmente la disabilitazione degli interrupt sarà temporanea e di breve durata. Al termine tutto riprenderà a funzionare normalmente

Kernel con diritto di prelazione



- I kernel con diritto di prelazione sono più adatti per le applicazioni real-time, in cui la disabilitazione degli interrupt non è accettabile
 - Un processo in real time può sempre far valere il proprio diritto di esecuzione anche se un altro processo è attivo in modalità kernel
 - In generale, i kernel con diritto di prelazione hanno un tempo di risposta inferiore, per ovvie ragioni
- Windows 2000 e Windows XP sono kernel senza diritto di prelazione
- Solaris, Unix e Linux hanno introdotto recentemente kernel con diritto di prelazione

Uso della lock



- L'accesso alla sezione critica può essere risolto utilizzando uno strumento detto **lock**
 - Tutte le strutture dati condivise sono protette da uno o più lock
- Per accedere alle strutture dati è necessario acquisire il possesso del lock che verrà restituito all'uscita della sezione critica

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

Hardware per la sincronizzazione



- Per poter implementare il concetto della lock esistono istruzioni implementate in hardware nelle moderne CPU, ad esempio:
 - **TestAndSet(var)** - testa e modifica il valore di una cella di memoria
 - **Swap(var1, var2)** - scambia il valore di due celle di memoria
- L'aspetto fondamentale di queste istruzioni macchine è che sono **atomiche** ovvero non possono essere interrotte da un cambio di contesto
 - Due istruzione atomiche eseguite contemporaneamente saranno sempre eseguite completamente in modo sequenziale in un ordine arbitrario

TestAndSet



- L'istruzione TestAndSet può essere definita nel seguente modo:

```
Boolean TestAndSet(boolean *lockvar) {  
    boolean temp = *lockvar;  
    *lockvar = true;  
    return temp;  
}
```

- Il significato è quello di impostare la variabile `lock` a `true` e restituire il suo precedente valore contenuto in `temp`

Mutua esclusione mediante TestAndSet



- La mutua esclusione mediante `TestAndSet` può essere realizzata attraverso una variabile booleana globale `lock` inizializzata a `false` nel seguente modo

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

Swap



- L'istruzione Swap può essere definita nel seguente modo:

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

- Il significato è quello di invertire il contenuto delle celle di memoria puntate dalla variabile a e b

Mutua esclusione mediante Swap



- La mutua esclusione mediante Swap può essere realizzata attraverso una variabile booleana globale `lock` inizializzata a `false` nel seguente modo

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

TestAndSet nella sezione critica



- La TestAndSet per poter risolvere il problema della sezione critica deve poter soddisfare i requisiti di:
 - Mutua esclusione
 - Progresso
 - Attesa limitata
- Sono soddisfatti tutti e tre i requisiti?

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

TestAndSet – Mutua Esclusione



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

- Dati due processi P_1 e P_2 il primo dei due che esegue la TestAndSet imposta lock a true ed entra nella sezione critica
- L'altro processo troverà lock a true e si metterà in attesa nel ciclo while

TestAndSet – Progresso



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

- Un processo P dopo aver lasciato la sezione critica imposta `lock` a `false` permettendo ad un altro processo di entrare nella sezione critica

TestAndSet – Attesa limitata



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

- Un processo uscendo imposta `lock` a `false` e immediatamente dopo richiedere di entrare nuovamente nella sezione critica
- Nulla vieta che le sue `TestAndSet` vengano sempre eseguite prima delle altre. Quindi l'attesa limitata non è garantita.

Mutua Esclusione con attesa limitata



```
do {
    attesa[i] = true;
    chiave = true;
    while (attesa[i] && chiave)
        chiave = TestAndSet(&lock);
    attesa[i] = false;

    sezione critica

    j = (i + 1) % n;
    while ((j != i) && !attesa[j])
        j = (j + 1) % n;

    if (j==i)
        lock = false;
    else
        attesa[j] = false;

    sezione non critica

} while(true);
```

- Il processo che esce dalla propria sezione critica designa attraverso una coda circolare il prossimo processo che può entrare
 - Qualsiasi processo che intende entrare nella sezione critica attenderà al più $n-1$ turni

Busy waiting



- Quando un processo è nella sua sezione critica tutti gli altri restano in **attesa attiva** (*busy waiting*) all'interno di un ciclo `while` che causa un inutile spreco di cicli macchina
- Se n è il numero di processi in attesa di una risorsa un algoritmo di scheduling come il round robin potrebbe sprecare del tempo di CPU pari a $n-1$ quanti di tempo
- Una soluzione potrebbe essere quella di permettere di disabilitare gli interrupt ai processi utenti finché si trova nella sezione critica
 - Ma che succede se un processo dimentica di abilitare gli interrupt quando termina?

Semafori



- Un **semaforo** S è una variabile intera utilizzata come strumento di sincronizzazione
- Le due operazioni atomiche possibili su S sono:
 - `wait(S)`
 - `signal(S)`

```
wait(S) {  
    while(S<=0)  
        ; //no-op  
    S--;  
}
```

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

Uso dei semafori



- I semafori possono essere usati in due modi:
 - Semafori contatore – il valore del semaforo è illimitato
 - Semafori binari (o lock mutex) – il valore del semaforo è 0 o 1
- Un semaforo può essere utilizzato per risolvere il problema dell'accesso alla sezione critica tra n processi mediante un semaforo comune `mutex` inizializzato ad 1

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

Un altro uso del semaforo



- I semafori possono essere usati in diverse circostanze in cui ad esempio un risorsa presente in diversi esemplari deve essere condivisa
- Un altro uso dei semafori è nella sincronizzazione di operazioni
- Supponiamo che il processo P_1 debba compiere una operazione S_1 prima dell'operazione S_2 del processo P_2 . Quindi S_2 sarà eseguita solo dopo che S_1 termina.
 - Utilizziamo un semaforo `sincronizzazione` inizializzato a 0

P_1

```
S1 ;  
signal(sincronizzazione) ;
```

P_2

```
wait(sincronizzazione) ;  
S2 ;
```

Evitare il busy waiting con i semafori - 1



- I processi che girano inutilmente mentre attendono un semaforo per il rilascio di una risorsa sono chiamati **spin lock**
- Per evitare il busy waiting possiamo modificare la definizione di `wait` in modo che:
 - Il processo invoca una `wait` per il rilascio di una risorsa
 - Se la risorsa è in uso *blocca* se stesso inserendosi in una coda di attesa del semaforo e passando nello stato di attesa
 - Lo scheduler della CPU può quindi selezionare un altro processo pronto per l'esecuzione
 - Il processo sarà sbloccato da una `signal` da parte di qualche altro processo

Evitare il busy waiting con i semafori - 2



- Per realizzare l'operazione di bloccaggio abbiamo bisogno di una struttura dati `semaforo` implementata nel seguente modo

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

- Il sistema operativo deve inoltre fornire due operazioni:
 - `block()` – sospende il processo che la invoca
 - `wakup(P)` – pone in stato di pronto il processo *P* bloccato
- Le operazioni `wait()` e `signal()` saranno realizzate nel seguente modo:

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

```
signal(semaforo *S) {  
    S->valore++;  
    if(S->valore <= 0) {  
        toglì un processo P da S->lista;  
        wakeup(P);  
    }  
}
```

Implementazione dei semafori



- Poiché i semafori devono essere eseguiti in modo atomico si deve garantire che nessun processo possa eseguire `wait` e `signal` contemporaneamente ad un altro processo
- Il Sistema Operativo solitamente mette a disposizione delle `system call` `wait` e `signal` (magari con nomi diversi)
- `wait` e `signal` sono esse stesse sezioni critiche (circa 10 istruzioni macchina) per cui è possibile implementarle tramite:
 - Spinlock
 - Disabilitazione degli interrupt (che avviene sotto il controllo del SO)

Stallo e attesa indefinita



- I semafori implementati attraverso code di attesa sono estremamente utili all'interno di un Sistema Operativo per risolvere problemi di sincronizzazione tra processi
- Il difetto dei semafori è che sono primitive di sincronizzazione non strutturate
- E' facile usare i semafori per gestire risorse condivise ma è altrettanto facile ottenere come risultato una situazione in cui un processo attende indefinitamente il rilascio di una risorsa da parte di altri processi
- Questo genere di situazioni prendono il nome di **stallo** (*deadlock*)

Esempi di stallo



- Supponiamo di avere due processi P_0 e P_1 ciascuno dei quali utilizza due semafori S e Q impostati ad 1

- Il processo P_0 esegue `wait(S)`
- Il processo P_1 esegue `wait(Q)`
- P_0 aspetta la `wait(S)` di P_1
 - ✦ P_1 non può fare la `signal(Q)`
- P_1 aspetta la `wait(Q)` di P_0
 - ✦ P_0 non può fare la `signal(S)`

P_0	P_1
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
...	...
...	...
...	...
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>

- I processi entrano in stallo perché entrambi attendono il rilascio di una risorsa tenuta dall'altro processo

Problemi tipici di sincronizzazione



- Alcuni problemi tipici di sincronizzazione dove poter verificare schemi di sincronizzazione sono:
 - Problema dei produttori e consumatori
 - Problema dei lettori e degli scrittori
 - Problema dei cinque filosofi

Produttori e Consumatori



- Il problema del produttore e del consumatore con memoria limitata può essere risolto utilizzando i semafori e le seguenti strutture dati
 - `buffer[DIM_BUFFER]`
 - ✦ buffer circolare di dimensione limitata
 - Semafori `piene`, `vuote`, `mutex`
 - ✦ `piene` inizializzato a 0, mantiene il numero degli elementi inseriti
 - ✦ `vuote` inizializzato a n , mantiene il numero delle posizioni libere
 - ✦ `mutex` inizializzato ad 1, sincronizza gli accessi al buffer

Produttori e Consumatori



Produttore

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

Consumatore

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

- Il semaforo `mutex` serve a gestire in modo concorrente le variabili `in` ed `out`
 - Potrebbero esserci più produttori che provano ad inserire accedendo ad `in`
 - Potrebbero esserci più consumatori che consumano accedendo ad `out`

Problema dei lettori e degli scrittori



- Il problema dei lettori e degli scrittori consiste in una serie di processi che accedono ad un file in sola lettura (**lettori**) ed altri processi che possono sia leggere che scrivere il file (**scrittori**)
- I processi lettori possono accedere contemporaneamente al file
- Un processo scrittore deve accedere in mutua esclusione con tutti gli altri processi (sia lettori che scrittori)

Problema dei lettori e degli scrittori



- Strutture dati condivise:
 - numlettori
 - ✦ Inizializzato a 0, mantiene il numero di processi che leggono i dati
 - Semafori `mutex`, `scrittura`
 - ✦ `mutex` inizializzato a 1, gestisce l'accesso a `numlettori`
 - ✦ `scrittura` inizializzato a 1, gestisce la mutua esclusione nella scrittura

Letto

```
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);
```

Scrittore

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

Problema dei cinque filosofi



- Il problema dei 5 filosofi è un classico problema di sincronizzazione sviluppato da Dijkstra
- Cinque filosofi passano la loro vita pensando e mangiando
- I filosofi siedono attorno ad un tavolo rotondo con 5 posti
- Un filosofo può essere nei seguenti stati:
 - Pensa: non interagisce con niente e nessuno
 - Mangia: tenta di prendere le bacchette alla sua destra ed alla sua sinistra



Problema dei cinque filosofi



- Strutture dati condivise:
 - Semafori `bacchetta[5]`
 - ✦ Inizializzati tutti a 1, gestisce la mutua esclusione sull'accesso alle bacchette

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

Problema dei cinque filosofi



- La soluzione con i semafori non esclude la possibilità di deadlock
- Sono possibili diversi miglioramenti:
 - Solo 4 filosofi a tavola contemporaneamente
 - Prendere le due bacchette insieme ossia solo se sono entrambi disponibili
 - Prelievo asimmetrico in un filosofo. I dispari prendono la bacchetta sinistra e poi quella destra. I pari prendono la bacchetta destra e poi quella sinistra.