

# Sincronizzazione dei processi

Capitolo 5 -- Silberschatz

# Processi cooperanti

Nei moderni SO, i processi vengono eseguiti concorrentemente

- Possono essere interrotti in qualunque momento nel corso della loro esecuzione
- I processi cooperanti possono...
  - ...condividere uno spazio logico di indirizzi, cioè codice e dati (thread, memoria condivisa)
  - ...oppure solo dati, attraverso i file
- L'accesso concorrente a dati condivisi può causare inconsistenza nei dati

# Processi cooperanti

- Un processo cooperante può influenzare gli altri processi in esecuzione nel sistema o subirne l'influenza
- Per **garantire la coerenza** dei dati occorrono meccanismi che assicurano l'esecuzione ordinata dei processi cooperanti

# esempio: produttore - consumatore

- processo **produttore** produce informazioni che sono consumate dal processo **consumatore**
  - server-client: un server web (server) fornisce (produce) pagine html e immagini, lette (consumate) dal browser web (client) che le richiede
- i due processi hanno un **buffer** in comune in cui possono scrivere e dal quale possono leggere

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} elemento;
elemento buffer[BUFFER_SIZE];
```

# esempio: produttore - consumatore

tre variabili comuni: **in**, **out** e **contatore**  
consentono l'accesso e l'utilizzo di **buffer**



Dati condivisi

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} elemento;
elemento buffer[BUFFER_SIZE];

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

# esempio: produttore - consumatore

## Processo produttore

**in** = indice della successiva posizione libera nel buffer

```
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

**out** = indice della prima posizione piena nel buffer

# esempio: produttore - consumatore

Se si considerano separatamente, le procedure del produttore e del consumatore sono corrette, ma possono “non funzionare” se eseguite in concorrenza, sebbene siano corrette

In particolare, le istruzioni:

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

possono causare problemi se non **atomiche**

# Race Condition

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
```

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

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

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



# Race Condition

- Se il produttore ed il consumatore tentano di accedere al buffer contemporaneamente, le istruzioni in linguaggio macchina possono risultare *interfogliate*
- La sequenza effettiva di esecuzione dipende da come i processi produttore e consumatore vengono schedulati

Esempio: inizialmente `contatore=5`

**T<sub>0</sub> produttore:** `registro1=contatore` (`registro1=5`)

**T<sub>1</sub> produttore:** `registro1=registro1+1` (`registro1=6`)

**T<sub>2</sub> consumatore:** `registro2=contatore` (`registro2=5`)

**T<sub>3</sub> consumatore:** `registro2=registro2-1` (`registro2=4`)

**T<sub>4</sub> produttore:** `contatore=registro1` (`contatore=6`)

**T<sub>5</sub> consumatore:** `contatore=registro2` (`contatore=4`)

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

# Race Condition

- **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 concorrenti vengano **sincronizzati**
- Tali situazioni si verificano spesso nei SO, nei quali diversi componenti compiono operazioni su risorse condivise
- Problema particolarmente significativo nei sistemi multi-core, in cui diversi thread vengono eseguiti in parallelo su unità di calcolo distinte

# Sezione critica

$n$  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

# Nota: sezione critica e kernel

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 ogniqualvolta si aprono/chiudono file
  - Due processi che tentano di aprire file in contemporanea potrebbero ingenerare nel sistema una corsa critica

# Sezione critica

**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

```
do {  
    entry section  
    sezione critica  
    exit section  
    sezione non critica  
} while (1);
```

struttura generale  
del processo  $P_i$

# Soluzione: Sezione critica

```
do {  
    entry section  
    sezione critica  
    exit section  
    sezione non critica  
} while (1);
```

**Mutua esclusione** = Se il processo  $P_i$  è in esecuzione nella sezione critica, nessun altro processo può eseguire la propria sezione critica

**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 indefinitamente (evitare il **deadlock**)

# Soluzione: Sezione critica

```
do {  
    entry section  
    sezione critica  
    exit section  
    sezione non critica  
} while (1);
```

**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**)

# Soluzione con alternanza stretta

Si assuma di avere due processi  $P_0$  e  $P_1$  e sia `turn=0`

```
do {  
    while (turn != 0);  
    sezione critica  
  
    turn = 1;  
    sezione non critica  
} while (1);
```

algoritmo per il  
processo  $P_0$

```
do {  
    while (turn != 1);  
    sezione critica  
  
    turn = 0;  
    sezione non critica  
} while (1);
```

algoritmo per il  
processo  $P_1$

$P_1$  entra nella sezione critica (**progresso**) al massimo dopo un ingresso da parte di  $P_0$  (**attesa limitata**)



# Soluzione con alternanza stretta

Si assuma di avere due processi  $P_0$  e  $P_1$  e sia `turn=0`

```
do {  
    while (turn != 0);  
    sezione critica  
  
    turn = 1;  
    sezione non critica  
} while (1);
```

algoritmo per il  
processo  $P_0$

```
do {  
    while (turn != 1);  
    sezione critica  
  
    turn = 0;  
    sezione non critica  
} while (1);
```

algoritmo per il  
processo  $P_1$

$P_1$  entra nella sezione critica (**progresso**) al massimo dopo un ingresso da parte di  $P_0$  (**attesa limitata**) : **alternanza stretta**

# Soluzione con alternanza stretta

Si assuma di avere due processi  $P_0$  e  $P_1$  e sia **turn=0**

```
do {  
    while (turn != 0);  
    sezione critica  
  
    turn = 1;  
    sezione non critica  
} while (1);
```

algoritmo per il  
processo  $P_0$

```
do {  
    while (turn != 1);  
    sezione critica  
  
    turn = 0;  
    sezione non critica  
} while (1);
```

algoritmo per il  
processo  $P_1$

Ma, se tocca a  $P_0$  (cioè  $P_1$  ha finito con la sua sezione critica ed ha messo **turn = 0**) ed intanto  $P_0$  si attarda ancora nella sua sezione non critica =>  $P_1$  rimane bloccato => si può violare il progresso; quindi possibile **deadlock**

# Soluzione di Petersen (1981)

Soluzione per due processi  $P_0$  e  $P_1$

- Supponiamo che le istruzioni LOAD e STORE siano atomiche
- I due processi condividono due variabili:
  - `int turn`
  - `boolean flag[2]`
- La variabile `turn` 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,  $i \in \{0,1\}$

# Soluzione di Petersen (1981)

algoritmo per il  
processo  $P_i$

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

$P_i$  entra nella sezione critica (progresso) al massimo dopo un ingresso da parte di  $P_j$  (attesa limitata): **alternanza stretta**

**flag[i] = false** nella exit section evita il problema evidenziato precedentemente in quanto questa sezione non è basata sulla variabile comune **turn**

# Soluzione di Petersen (1981)

```
do {  
    flag[0] = true;  
    turn = 1;  
    while (flag[1] && turn == 1);  
    sezione critica  
    flag[0] = false;  
    sezione non critica  
} while (1);
```

P<sub>0</sub>

```
do {  
    flag[1] = true;  
    turn = 0;  
    while (flag[0] && turn == 0);  
    sezione critica  
    flag[1] = false;  
    sezione non critica  
} while (1);
```

P<sub>1</sub>

# Soluzione generale

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);
```

# Hardware di sincronizzazione

- Molti sistemi forniscono supporto hardware per risolvere efficacemente il problema della sezione critica
- In un sistema monoprocesso è sufficiente **interdire le interruzioni** mentre si modificano le variabili condivise
  - Il job attualmente in esecuzione viene eseguito senza possibilità di prelazione
- Molte architetture attuali forniscono speciali **istruzioni atomiche** 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 (**Swap**)

# Hardware di sincronizzazione

```
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**

realizzazione  
della mutua  
esclusione

**lock = TRUE**  
quando un processo è  
nella sua sezione  
critica

```
boolean lock = FALSE;
```

```
do {
    while TestAndSet(&lock);
    sezione critica

    lock = FALSE;
    sezione non critica
} while (1);
```



# Semafori

I **semafori** sono strumenti di sincronizzazione

- Il semaforo S è una variabile intera
- Si può accedere al semaforo S (dopo l'inizializzazione) solo attraverso due operazioni atomiche predefinite **wait( )** e **signal( )** (-Da non confondere con le omonime funzioni Unix-)

```
wait(S) {  
    while S<=0 ; // do not op  
    S--;  
}
```

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

# Semafori

```
wait(S) {  
    while S<=0; // do not op  
    S--;  
}
```

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

- 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

# uso semafori

**Semaforo binario:** assume soltanto i valori 0 e 1

```
Semaphore mutex=1;    // inizializzazione del semaforo mutex =1
```

$n$  processi condividono il semaforo comune mutex

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

Processo  $P_i$

Ricorda:

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

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

# uso semafori

**Semaforo binario:** assume soltanto i valori 0 e 1

- Utile anche per la **sincronizzazione** oltre che per regolare l'accesso ad una risorsa comune.
  - Siano P1 e P2 due processi che contengono, rispettivamente due frammenti di codice  $S_1$  ed  $S_2$  che devono essere eseguiti in sequenza

```
Semaphore synch=0;
```

```
P1 :  
    S1;  
    signal(synch) ;
```

```
P2 :  
    wait(synch) ;  
    S2;
```

Ricorda:

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

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

# uso semafori

- **Semaforo contatore**: è inizialmente impostato al numero di esemplari disponibili e di cui regolarne l'accesso
- I processi che desiderano utilizzare un'istanza della risorsa, invocano una **wait( )** sul semaforo, decrementandone così il valore
- I processi che ne rilasciano un'istanza, 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 fino a che esso ridiventa positivo

# Semafori: busy waiting

```
wait(S) {  
    while S<=0; // do nop  
    S--;  
}
```

*Spinlock (si gira in attesa  
del semaforo)*

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

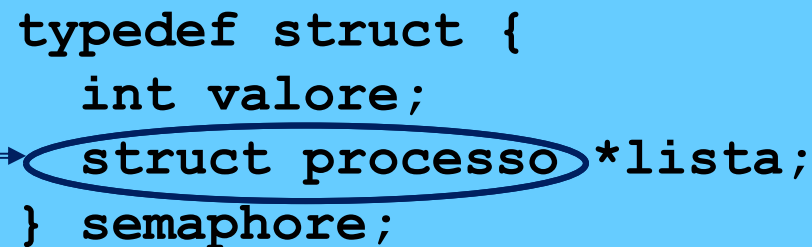
- Mentre un processo si trova nella propria sezione critica, qualsiasi altro processo che tenti di entrare nella sezione critica si trova nel ciclo del codice della entry section (**busy waiting**)
  - la condizione di attesa spreca cicli di CPU che altri processi potrebbero sfruttare produttivamente
  - è vantaggiosa solo quando è molto breve perché, in questo caso, può evitare un context switch

# Come evitare il busy waiting

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:

PCB



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

# Come evitare il busy waiting

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
- **wakeup:** rimuove un processo dalla coda di attesa e lo sposta nella ready queue



# Come evitare il busy waiting

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

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

# Come evitare il busy waiting

Mentre la definizione classica di semaforo ad attesa attiva è tale che il valore del semaforo non è mai negativo, il semaforo appena descritto può assumere valori negativi (nel campo **valore**)

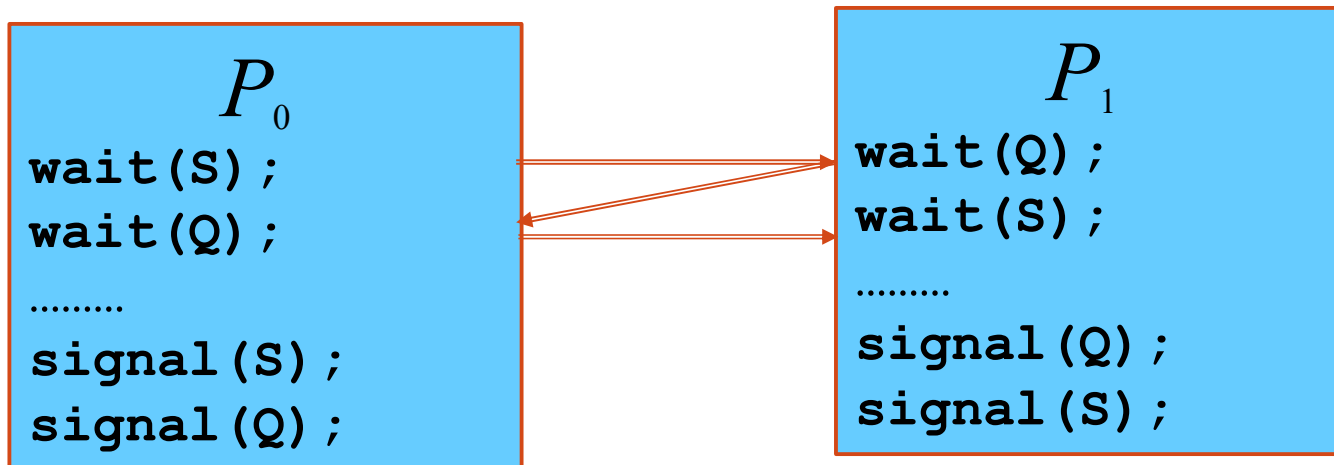
- Se **valore** è negativo  $|\mathbf{valore}|$  è il numero dei processi che attendono al semaforo
  - Ciò avviene a causa dell'inversione dell'ordine delle operazioni di decremento e verifica del valore nella **wait()**

Nota che il sistema deve garantire che **wait()** e **signal()** siano eseguite senza interruzioni

# 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 due 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 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

# produttore – consumatore

## semafori

- variabili condivise

```
semaphore full, empty, mutex;  
// inizialmente full = 0, empty = n, mutex = 1
```

- Il buffer ha **n** posizioni, ciascuna in grado di contenere un elemento
- Il semaforo **mutex** garantisce la mutua esclusione sugli accessi al buffer
- I semafori **empty** e **full** contano, rispettivamente, il numero di posizioni vuote ed il numero di posizioni piene nel buffer

# produttore – consumatore

## semafori

```
do {  
    ... /* produce un elemento in  
    next_produced */  
    ...  
    wait(empty);  
    wait(mutex);  
    ... /* inserisce next_produced nel  
    buffer */  
    ...  
    signal(mutex);  
    signal(full);  
  
} while (true);
```

Processo produttore

**empty** conta il numero  
di posizioni vuote  
**full** conta il numero di  
posizioni piene

# produttore – consumatore

## semafori

```
do {  
  
wait(full);  
wait(mutex);  
... /* sposta un elemento dal buffer  
in next_consumed */  
...  
signal(mutex);  
signal(empty);  
... /* consuma un elemento in  
next_consumed */  
  
... } while (true);
```

Processo consumatore

**empty** conta il numero  
di posizioni vuote  
**full** conta il numero di  
posizioni piene