

Lezione T25

Sincronizzazione

Sistemi Operativi (9 CFU), CdL Informatica, A. A. 2013/2014

Dipartimento di Scienze Fisiche, Informatiche e Matematiche

Università di Modena e Reggio Emilia

<http://weblab.eng.unimo.it/people/andreolini/didattica/sistemi-operativi>

Quote of the day

(Meditate, gente, meditate...)

“Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?”

Brian Kernighan (1942-)

Accademico, informatico teorico

Coautore di UNIX e di AWK

Autore del libro

“The C Programming Language”



INTRODUZIONE

Un problema caratteristico

(Produttore-consumatore con buffer limitato condiviso)

Si usa un array circolare di N posizioni.

L'array contiene strutture di tipo **item**.

Una struttura è prodotta con il metodo **produce_item()**.

Una struttura è consumata con il metodo **consume_item()**.

Un indice **in** indica il prossimo elemento libero del buffer.

Un indice **out** indica il primo elemento pieno del buffer.

Una variabile **counter** conta il numero di elementi nel buffer.

Un problema caratteristico

(Produttore-consumatore con buffer limitato condiviso)

Gestione della variabile **counter**.

Inizializzata a 0.

Produzione di un elemento: **counter++**.

Consumo di un elemento: **counter--**.

Condizione di buffer vuoto: **counter == 0**.

Condizione di buffer pieno: **counter == N**.

Si possono inserire al più N elementi nel buffer.

Una possibile implementazione

(In pseudocodice)

Produttore

```
void producer(void) {  
    item p;  
  
    while(TRUE) {  
        p = produce_item();  
        while(counter == N)  
            /* do nothing */;  
        buffer[in] = p;  
        in = (in + 1) % N;  
        counter++;  
    }  
}
```

Consumatore

```
void consumer(void) {  
    item c;  
  
    while(TRUE) {  
        while(counter == 0)  
            /* do nothing */;  
        c = buffer[out];  
        out = (out + 1) % N;  
        counter--;  
        consume_item(c);  
    }  
}
```

Bounded buffer: concorrenza

(Sull'array `buffer[]` e relative variabili di gestione, ad esempio `counter`)

Se eseguite separatamente, le procedure ora descritte funzionano correttamente.

Se eseguite concorrentemente, le procedure ora descritte possono non funzionare correttamente.

ATTENZIONE!

L'array **`buffer[]`** è condiviso tra i due processi produttore e consumatore.

La variabile intera **`counter`** è condivisa tra i due processi produttore e consumatore.

Traduzione di `counter++` e `counter--`

(In codice macchina → Si intuisce il problema)

Si esamini, a puro titolo di esempio, la variabile **`counter`**.

Come traduce il compilatore i due statement di codice **`counter++`** e **`counter--`**?

`counter++` è tradotta in

```
register1 = counter;  
register1 = register1 + 1;  
counter = register1;
```

`counter--` è tradotta in

```
register2 = counter;  
register2 = register2 - 1;  
counter = register2;
```


Uso dello stesso registro

(Una sfortunata circostanza)

registro₁ e **registro₂** possono essere lo stesso registro.

Ad esempio, un registro accumulatore (EAX su x86).

counter++ può interrompersi “a metà” (dopo il secondo statement).

counter-- può eseguire subito dopo (esecuzione interallacciata).

Esecuzione interlacciata

(Un'altra sfortunata circostanza)

L'esecuzione delle tracce dei processi è interallacciata. Su sistemi uniprocessore:

il codice che esegue **counter++** può interrompersi "a metà" (ad es., arriva una interruzione).

lo scheduler dei processi decide di riesumare la traccia che esegue **counter--**.

Lo statement **counter--** opera su un registro il cui valore è inconsistente con il risultato dell'operazione **counter++**.

Ciò si può verificare sempre, anche quando **register₁** e **register₂** sono diversi.

Esecuzione interlacciata

(Un'altra sfortunata circostanza)

L'esecuzione delle tracce dei processi è interallacciata. Su sistemi multiprocessore:

- le tracce possono eseguire su più processori fisici concorrentemente.

- l'ordine di accesso al registro ne determina il valore finale.

- Il risultato delle operazioni su counter dipende dall'ordine di schedulazione.

In altre parole, **il risultato è non deterministico.**

Un ordine di esecuzione corretto

(Uso di due registri separati)

P₀

P₁

t₀ register₁^⑤ = counter^⑤
t₁ register₁^⑥ = register₁^⑤ + 1
t₂ counter^⑥ = register₁^⑥

t₃

t₄

t₅

Lo scheduler della CPU
decide di rischedulare un
altro processo.

register₂^⑥ = counter^⑥
register₂^⑤ = register₂^⑥ - 1
counter^⑤ = register₂^⑤

Questa sequenza di istruzioni macchina
produce il risultato corretto.

Corsa critica: approcci al problema

(Accesso in mutua esclusione, esecuzione atomica)

In questa presentazione, ci si occuperà inizialmente dell'accesso in mutua esclusione.

Successivamente si introdurrà il meccanismo di esecuzione atomica.

Infine, si cercherà di combinare i due meccanismi per fornire una soluzione più generale al problema.

Un ordine di esecuzione non corretto

(Uso di due registri separati)

P₀

t₀ $\text{register}_1^{(5)} = \text{counter}^{(5)}$

t₁ $\text{register}_1^{(6)} = \text{register}_1^{(5)} + 1$

t₂

t₃

t₄

t₅ $\text{counter}^{(6)} = \text{register}_1^{(6)}$

P₁

Lo scheduler della CPU decide di rischedulare un altro processo.

$\text{register}_2^{(5)} = \text{counter}^{(5)}$
 $\text{register}_2^{(4)} = \text{register}_2^{(5)} - 1$

$\text{counter}^{(4)} = \text{register}_2^{(4)}$

Questa sequenza di istruzioni macchina NON produce il risultato corretto.

Corsa critica

(Traccia di codice la cui esecuzione dipende dalla schedulazione)

È stato appena dimostrato che l'ordine di schedulazione impatta fortemente sul risultato delle operazioni su una variabile condivisa.

Quando il valore finale di una elaborazione su una variabile condivisa dipende dall'ordine di esecuzione delle istruzioni che la modificano, si è in presenza di una **corsa critica** (**race condition**, **race**).

Sezione critica

(La porzione di codice che accede ai dati condivisi)

Siano dati n processi P_0, P_1, \dots, P_{n-1} :

eseguiti concorrentemente.

cooperanti tramite dati condivisi.

Ciascun P_j ha una porzione di codice che accede ai dati condivisi.

Tale porzione prende il nome di **sezione critica**.

La sezione critica deve essere eseguita da al più un processo alla volta.

Corsa critica: approcci al problema

(Accesso in mutua esclusione, esecuzione atomica)

Per risolvere questo problema, è necessario che solo un processo alla volta acceda a **counter**. Esistono due approcci (radicalmente diversi).

Accesso in mutua esclusione.

Si bloccano tutti i tentativi di accesso a **counter** se quest'ultimo è già in uso.

Esecuzione atomica.

Si fa in modo che il codice macchina relativo a **counter++** o a **counter--** sia eseguito integralmente oppure non sia eseguito per niente.

Corsa critica: approcci al problema

(Accesso in mutua esclusione, esecuzione atomica)

In questa presentazione, ci si occuperà inizialmente dell'accesso in mutua esclusione.

Successivamente sarà introdotto il meccanismo di esecuzione atomica.

Infine, si cercherà di combinare i due meccanismi per fornire una soluzione più generale al problema.

Protezione delle sezioni critiche

(Il codice è suddiviso in diverse parti)

La sincronizzazione degli accessi in mutua esclusione avviene attraverso un protocollo di sincronizzazione caratterizzato dalle seguenti fasi.

Sezione di ingresso: il processo chiede il permesso di entrare nella sezione critica.

Sezione critica: accesso ai dati condivisi.

Sezione di uscita: rilascio dell'uso della sezione critica.

Sezione non critica: codice la cui esecuzione non richiede un meccanismo di mutua esclusione.

Protezione delle sezioni critiche

(Un accesso in mutua esclusione è effettuato in questo modo)

Il protocollo di sincronizzazione è implementato attraverso le seguenti istruzioni

```
do {  
    <sezione di ingresso>;  
    <sezione critica>;  
    <sezione di uscita>;  
    <sezione non critica>;  
} while (TRUE);
```

Il ciclo `do {} while(TRUE);` è puramente "accademico" (ciò che conta sono gli statement all'interno del blocco).

Alcune osservazioni

(Caveat emptor!)

Ciò che si protegge è il dato condiviso.

Il dato condiviso si protegge facendo eseguire la sezione critica che lo modifica da un solo processo.

Non si “protegge il codice”! Il codice è un'area dati “read-only” che non sarà mai modificata.

Non ha alcun senso dare accesso mutuamente esclusivo ad una porzione di codice che non accede a dati condivisi.

Requisiti di una soluzione al problema

(Mutua esclusione)

Ogni soluzione al problema della sezione critica deve soddisfare tre requisiti.

Mutua esclusione: se P_j è in sezione critica, P_i non può essere in sezione critica ($i \neq j$).

Si garantisce che la struttura non sia letta e scritta simultaneamente.

Origine del non determinismo.

Requisiti di una soluzione al problema

(Progresso)

Ogni soluzione al problema della sezione critica deve soddisfare tre requisiti.

Progresso: se nessun processo esegue la sezione critica, e se alcuni processi vogliono eseguire la sezione critica, allora solo i processi che non eseguono le sezioni non critiche possono decidere l'ammissione in sezione critica.

“Progresso”?

(Eh?)

La condizione di progresso mira ad evitare che un processo appena uscito da una sezione critica se la riprenoti immediatamente.

- Monopolizzando, di fatto, il processore.
- Gli altri processi non possono avanzare.
- Starvation degli altri processi!

L'effetto del "non progresso"

(Un esempio con pseudocodice)

P_0 do {

```
<sezione di ingresso>;  
<sezione critica>;  
<sezione di uscita>;  
<sezione non critica>;
```

} while (TRUE);

P_0 vuole entrare in sezione critica ed attende l'uscita di P_1 .

P_1 do {

```
<sezione di ingresso>;  
<sezione critica>;  
<sezione di uscita>;  
<sezione non critica>;
```

} while (TRUE);

P_1 si riassegna la risorsa subito dopo il rilascio. Con elevata probabilità, P_1 monopolizza la risorsa stessa.

“Progresso”?

(Eh?)

Ogni soluzione al problema della sezione critica deve soddisfare tre requisiti.

Attesa limitata: se un processo è in attesa di ingresso nella sezione critica, altri processi possono eseguire la sezione critica al più k volte (k finito).

Si evitano le situazioni di stallo perenne.

Si limita la “unfairness” (assegnazione non equa della risorsa) fra processi.

In caso contrario, potrebbe insorgere un fenomeno di starvation.

ALGORITMO DI PETERSON

A cosa serve

(Ad accedere in mutua esclusione ad una variabile condivisa)

Soluzione software per garantire accesso in mutua esclusione ad una sezione critica.

Permette a 2 processi di accedere ad una risorsa senza conflitti.

La risorsa in questione è una porzione di memoria condivisa.

Può essere esteso al caso di n processi (non interessa come, in questa introduzione di base).

Funzionamento

(Complicato)

Due processi P_0 e P_1 si alternano nella loro esecuzione.

I due processi condividono due informazioni.

- Una variabile intera **turn**, contenente l'indice del processo abilitato ad entrare in sezione critica.

- Un array di boolean **flag[2]** che, per ciascun processo, vale **TRUE** se tale processo è pronto per entrare in sezione critica.

Pseudocodice dell'algoritmo

(Con le varie sezioni evidenziate)

La soluzione di Peterson è rappresentata dall'algoritmo seguente (processo P_i).

```
do {  
    | flag[i] = TRUE; |  
    | turn = 1-i; | Sezione di  
    | while (flag[1-i] && turn==1-i) ; | ingresso  
    | sezione critica; |  
    | flag[i] = FALSE; | Sezione  
} while (TRUE); | di uscita
```

Il significato degli statement

(Riga per riga)

```
do {  
    flag[i] = TRUE; ←  
    turn = 1-i;  
    while (flag[1-i] && turn==1-i) ;  
    sezione critica;  
    flag[i] = FALSE;  
} while (TRUE) ;
```

P_i si dichiara pronto ad entrare in sezione critica.

Il significato degli statement

(Riga per riga)

```
do {  
    flag[i] = TRUE;  
    turn = 1-i; ←  
    while (flag[1-i] && turn==1-i) ;  
    sezione critica;  
    flag[i] = FALSE;  
} while (TRUE) ;
```

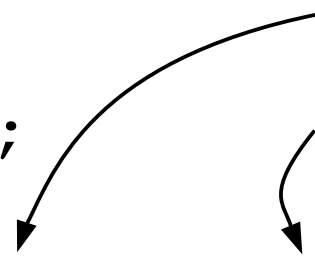
P_i lascia passare avanti
 P_{1-i} .

Il significato degli statement

(Riga per riga) P_i attende di poter entrare in sezione critica. Devono verificarsi due eventi:

1. P_{1-i} deve uscire dalla sezione critica;
2. P_{1-i} deve lasciar passare P_i , impostando $turn=i$.

```
do {  
    flag[i] = TRUE;  
    turn = 1-i;  
    while (flag[1-i] && turn==1-i) ; ←  
    sezione critica;  
    flag[i] = FALSE;  
} while (TRUE);
```



Il significato degli statement

(Riga per riga)

```
do {  
    flag[i] = TRUE;  
    turn = 1-i;  
    while (flag[1-i] && turn==1-i) ;  
    sezione critica; ←  
    flag[i] = FALSE;  
} while (TRUE);
```

P_i entra in sezione critica.

Il significato degli statement

(Riga per riga)

```
do {  
    flag[i] = TRUE;  
    turn = 1-i;  
    while (flag[1-i] && turn==1-i) ;  
    sezione critica;  
    flag[i] = FALSE; ←  
} while (TRUE);
```

P_i esce dalla sezione critica.
Si segnala tale evento a P_{1-i}
impostando $\text{flag}[i] = \text{FALSE}$.

Sono verificati i requisiti?

(Mutua esclusione)

Requisito 1: mutua esclusione.

P_i entra nella sua sezione critica solo se fallisce il while, ossia $\text{flag}[1-i] == \text{FALSE}$ o $\text{turn} == i$.

Se P_0 e P_1 fossero in sezione critica nello stesso istante, si avrebbe $\text{flag}[0] == \text{TRUE}$, $\text{flag}[1] == \text{TRUE}$.

Le due osservazioni precedenti implicano che $\text{turn} = 0$ e $\text{turn} = 1$ contemporaneamente.

Assurdo $\rightarrow P_0$ e P_1 non possono mai eseguire la sezione critica concorrentemente.

Sono verificati i requisiti?

(Progresso)

Requisito 2: progresso.

Si supponga che P_0 e P_1 non stiano usando la risorsa.
 P_0 è in procinto di usarla, P_1 l'ha appena usata.
C'è progresso se l'algoritmo sceglie P_0 .

Sono verificati i requisiti?

(Progresso)

Requisito 2: progresso.

P_1 è appena uscito dalla sezione critica ed imposta $\text{flag}[1] == \text{FALSE}$.

P_1 ricomincia il ciclo `do {} while();`, dichiara P_0 come prossimo processo avente diritto a usufruire della risorsa ($\text{turn}=0$) e ne aspetta il rilascio.

Prima o poi, P_0 raggiunge il controllo `while()` della sezione di ingresso, che sicuramente fallisce su $\text{turn}==0$; pertanto, P_0 entra in sezione critica.

P_0 e P_1 si alternano nell'uso della risorsa → Progresso.

Sono verificati i requisiti?

(Attesa limitata)

Requisito 3: attesa limitata.

La dimostrazione precedente (requisito 2) ha mostrato come P_1 passi il testimone a P_0 dopo aver utilizzato la risorsa condivisa.

Analogamente si può mostrare come P_0 passi il testimone a P_1 dopo aver utilizzato la risorsa condivisa.

In altre parole, P_0 e P_1 si alternano nell'esecuzione della sezione critica.

Se un processo è in attesa di entrare in sezione critica, l'altro processo può eseguire la sezione critica al più $k=1$ volte.

Vantaggi e svantaggi

(Indipendente dall'hw, ma soggetta alle bizze del compilatore)

Vantaggi.

Non richiede hardware di supporto per poter funzionare.

Generalizzabile a n processi.

Svantaggi.

Se la CPU effettua il reordering delle istruzioni, `flag[i]` e `turn` rischiano di essere impostati con una temporizzazione sballata.

Se il compilatore ottimizza i cicli, rischia di trasformare il `while()` in un `while(TRUE)`, perché per lui la condizione non è mai falsa.

ISTRUZIONI HARDWARE ATOMICHE

I problemi dell'algoritmo di Peterson

(Che lo rendono un semplice strumento teorico)

La soluzione di Peterson, pur essendo interamente software, è complessa.
mal sopporta ottimizzazioni sul codice da parte della CPU e del SO.

Supporto hardware

(Istruzioni atomiche)

Nei moderni SO, le sezioni di ingresso e di uscita sono implementate tramite un modello universale: il **lock** (serratura, lucchetto).

Si protegge una sezione critica che accede a dati condivisi con

- una variabile intera (lock) che indica se la risorsa è libera oppure occupata.

- un meccanismo che controlla lo stato della variabile intera e permette l'accesso esclusivo.

Modello di locking/unlocking

(Molto più semplice rispetto all'algoritmo di Peterson)

Il meccanismo generale di protezione delle sezioni critiche tramite lock si presenta nel formato seguente.

```
do {  
    <acquisizione lock>  
    sezione critica;  
    <rilascio lock>  
    sezione non critica;  
} while (TRUE);
```

Supporto hardware al locking

(Esecuzione atomica)

Molte architetture offrono particolari istruzioni che permettono di leggere e scrivere il contenuto della memoria in modalità atomica.

Esecuzione atomica: esecuzione non interrompibile (tutto o niente).

La tipica operazione atomica è un controllo di una variabile con contestuale modifica.

Le operazioni di locking sono spesso implementate tramite istruzioni atomiche.

Istruzione TestAndSet

(Imposta il valore di una variabile e ritorna il valore precedente)

L'istruzione macchina **TestAndSet** effettua le seguenti operazioni atomicamente.

Ritorna il vecchio valore di una variabile.

Imposta la variabile al valore TRUE (1).

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

Locking/Unlocking con TestAndSet

(Molto semplice)

Si usa una variabile intera blocco (inizializzata a FALSE) per contenere l'informazione "risorsa libera o occupata".

blocco=FALSE: la risorsa è libera.

blocco=TRUE: la risorsa è occupata.

Il protocollo di sincronizzazione è il seguente.

```
do {  
    while (TestAndSet(&blocco)) ;  
    <sezione critica>;  
    blocco = FALSE;  
} while (TRUE) ;
```

TestAndSet: perché funziona?

(Nelle slide seguenti viene mostrato)

In teoria, si potrebbero dimostrare i tre requisiti mutua esclusione, progresso e attesa limitata.

Viene lasciato come utile esercizio.

Nelle slide seguenti, viene mostrata una sequenza temporale di eventi che spiega il funzionamento del protocollo.

TestAndSet: perché funziona?

(Andamento temporale della variabile blocco)

Quando il primo processo esce dalla sezione critica, rilascia il lock ponendo blocco=FALSE.

Il secondo processo (che sta tentando di ottenere il lock) esegue, in uno degli istanti successivi, l'n-esima iterazione di `while(TestAndSet(blocco));`.

`blocco=TRUE` e viene ritornato il valore precedente, che ora è FALSE.

Il ciclo `while()` fallisce ed il secondo processo entra in sezione critica.

TestAndSet: la sequenza temporale

(Esplicitata)

P₀ (CPU #0)

P₁ (CPU #1)

t₀ blocco[ⓕ] = FALSE;
t₁ while (TestAndSet(blocco)[Ⓣ]);
t₂ <sezione critica>;
t₃

Imposta blocco a TRUE e
ritorna FALSE. Entra subito
in sezione critica.

t₄ blocco[ⓕ] = FALSE;
t₅

TestAndSet() imposta
blocco = TRUE e ritorna
FALSE.

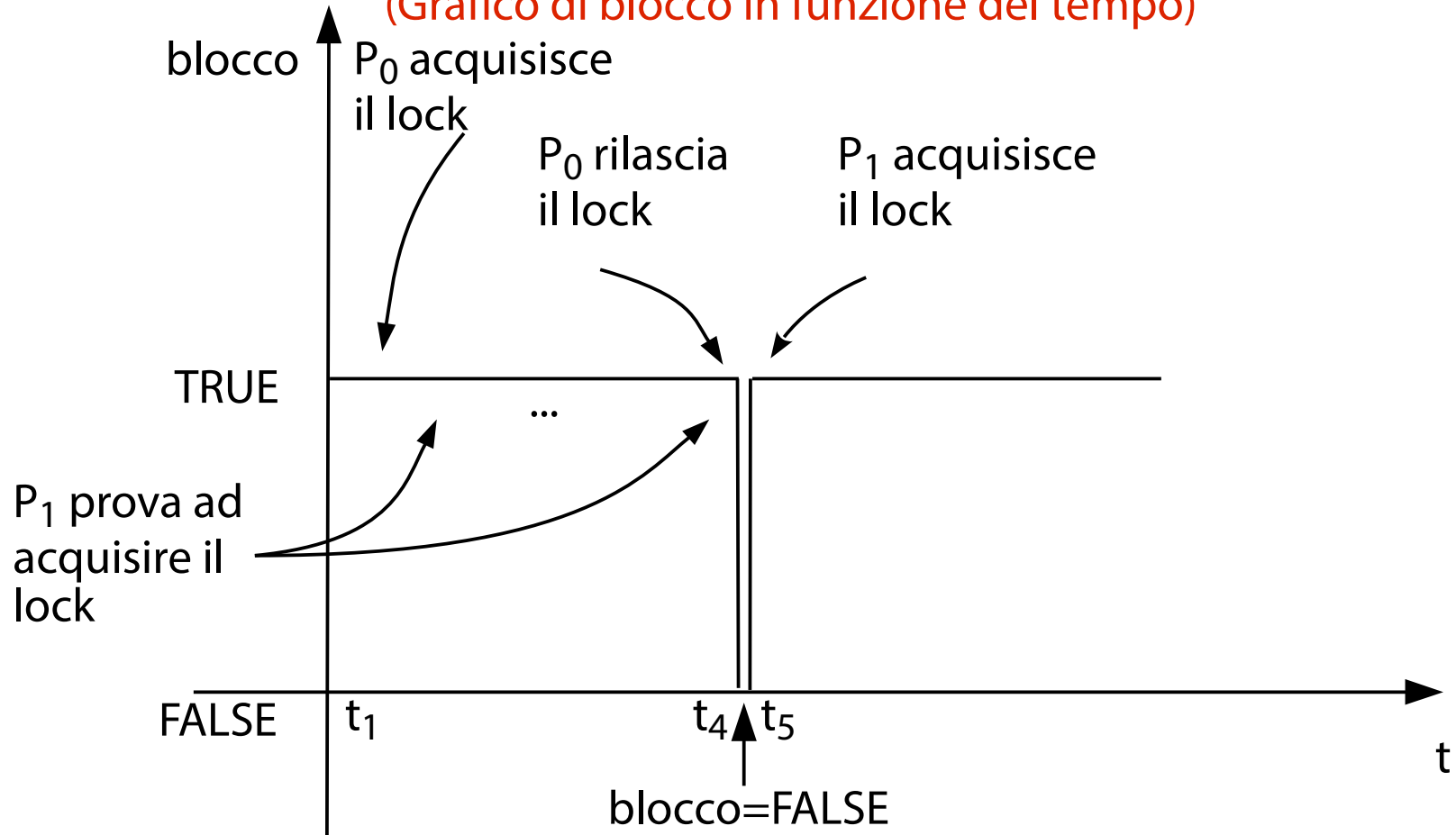
Imposta blocco = TRUE
e ritorna TRUE.

while (TestAndSet(blocco)[Ⓣ]);
Cicla sul while() fino a quando P₀
non imposta blocco=FALSE.

<sezione critica>;

TestAndSet: la sequenza temporale

(Grafico di blocco in funzione del tempo)



Istruzioni hw e mutua esclusione

(Non è soddisfatto il requisito di attesa limitata.)

TestAndSet non soddisfa, in generale, il criterio di attesa limitata.

TestAndSet va integrata con un supporto (algoritmo) per permettere l'attesa limitata.

SEMAFORI

Motivazioni legate ai semafori

(Perché servono)

Le soluzioni al problema della sezione critica viste in precedenza sono vincolate al particolare tipo di problema.

Esiste un meccanismo più generale?

Meccanismo semplice per decidere se utilizzare o meno una risorsa condivisa tramite una sezione critica.

Mascherare le particolarità della applicazione (niente riferimenti a struttura dati interne quali, ad esempio, il vettore `flag[]`).

Il meccanismo esiste e prende il nome di semaforo.

Il nome non è casuale: emula un vero e proprio semaforo stradale.

Definizione

(Che cosa sono)

Semaforo: variabile intera S , che conta il numero di slot di uso della risorsa condivisa.

Ideato da Edsger Dijkstra nel 1965.

La variabile S viene acceduta solo tramite tre funzioni.

init(S): inizializza S ad un valore > 0 .

wait(S): sezione di accesso (originariamente chiamata $P()$, dall'olandese *proberen*, ossia "testare").

signal(S): sezione di uscita (originariamente chiamata $V()$, dall'olandese *verhogen*, ossia "incrementare").

Locking/Unlocking con i semafori

(Incredibilmente semplice)

Il protocollo di sincronizzazione è il seguente.

```
init(S) ;  
do {  
    wait(S) ;  
    <sezione critica>;  
    signal(S) ;  
} while (TRUE) ;
```


Implementazione di wait() e signal()

(Aggiornano il contatore)

Lo scheletro implementativo delle funzioni **wait()** e **signal()** è il seguente.

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

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

Semafori contatore e binari

(Binario → mutua esclusione; contatore → sincronizzazione)

Dal punto di vista degli slot di risorsa messi a disposizione, si possono distinguere due categorie di semafori: semafori contatore e semafori binari.

Semafori contatore.

S è inizializzata ad un valore > 1 .

Rappresenta una risorsa che può gestire fino ad $S > 1$ elementi concorrentemente.

Semafori binari.

S è inizializzata ad 1 (caso comune).

La risorsa è presente in una istanza; o la si usa, oppure non la si usa (mutua esclusione).

Viene anche detto mutex (MUTual Exclusion).

Attesa attiva

(La CPU continua a controllare il cambio di valore di S)

L'implementazione ora vista è ad **attesa attiva**, ossia l'attesa implementata nella `wait()` è di tipo “attivo” (busy waiting).

Un ciclo while che esegue fino a quando non se ne vanifica la condizione.

Dal punto di vista funzionale, somiglia in maniera sinistra ai meccanismi di locking con istruzioni hardware.

Un semaforo attivo (o un lock implementato con istruzioni hardware) prende il nome di **spinlock**.

La CPU continua a ciclare (spin) sul while, bloccando (lock) l'accesso.

Esiste un modello di semaforo “passivo”?

Semafori attivi e passivi

(Attesa passiva → il processo è sospeso)

Nell'implementazione ad **attesa passiva**, l'attesa implementata nella **`wait()`** è di tipo “passivo”.
il processo viene messo in attesa (sleep) che si verifichi l'evento che consente l'accesso alla sezione critica.

Quando tale evento si verifica, il processo è risvegliato ed inserito in coda di pronto.

Quando si usa lo spinlock?

(Attese brevi, tipicamente per strutture dati in memoria)

Ideale per attese brevi, dell'ordine dei ns (accesso in mutua esclusione a strutture dati in memoria).

Bloccare e ripristinare il processo introdurrebbe un ritardo di alcuni secondi (1000 volte più lento).

Disastroso per attese lunghe.

L'attesa "brucia" il processore, surriscaldandolo e impedendo ad altri processi di avanzare (tragico per la latenza).

Quando si usa il semaforo passivo?

(Attese lunghe, tipicamente per operazioni su dispositivi)

Ideale per attese lunghe, dai ms in su.

Si blocca un processo che impallerebbe, altrimenti, la macchina.

Disastroso per attese brevi.

La sospensione ed il ripristino del processo introducono un ritardo inaccettabile.

Semaforo passivo: definizione

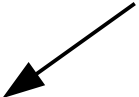
(Una variabile più una coda di processi)

Nel caso di attesa passiva, il SO implementa il semaforo con una struttura dati contenente:

- la singola variabile intera S , che conta il numero di slot associati alla risorsa.

- una lista di processi che attendono lo sblocco della risorsa.

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



Puntatore alla lista dei processi in attesa di entrare nella sezione critica.

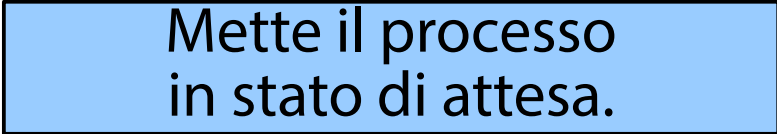
Semaforo passivo: wait()

(Accoda un processo se la risorsa è occupata)

La funzione **wait(S)** va estesa con un meccanismo di accodamento dei processi in caso di blocco.

```
void wait(semaforo S) {  
    S.valore--;  
    if (S.valore < 0) {  
        <accoda processo a S.p>;  
        block();  
    }  
}
```

Mette il processo
in stato di attesa.

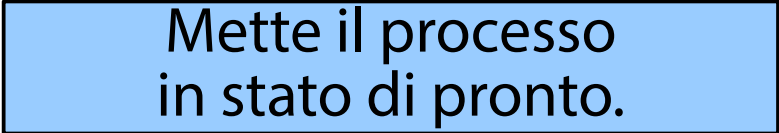


Semaforo passivo: signal()

(Sveglia un processo se la risorsa è libera)

La funzione **signal(S)** va estesa con dei meccanismi per il prelievo di un processo dalla coda di attesa del semaforo e per l'inserimento del processo nella coda di pronto.

```
void signal(semaforo S) {  
    S.valore++;  
    if (S.valore <= 0) {  
        <estrai un processo P da S.p>;  
        wakeup(P);  
    }  
}
```



Mette il processo
in stato di pronto.

Dettagli implementativi

(Da non trascurare)

Le funzioni interne **block()** e **wakeup()** devono essere fornite dal kernel.

Il criterio di estrazione usato nella **wakeup()** è, generalmente, FIFO, ma può non essere il solo.

L'esecuzione di **wait()** e **signal()** è atomica.

wait() e **signal()** sono considerate come vere e proprie sezioni critiche.

Criteri di estrazione LIFO e FIFO

(Di solito si adotta il criterio FIFO)

In generale, si evita il criterio LIFO (stack) nella gestione della coda dei processi in attesa della sezione critica.

Il criterio LIFO favorisce gli ultimi processi e non i primi.

→ Attesa indefinita dei processi più vecchi (starvation).

Si adotta il criterio FIFO (coda).

I difetti dei semafori

(Il programmatore deve preoccuparsi di come piazzarli)

I semafori sono, sostanzialmente, variabili globali condivise.

Formalmente, non c'è alcuna relazione fra semaforo e la struttura dati protetta.

Formalmente, un semaforo può essere definito dappertutto nel programma (anche a notevole distanza dalla struttura dati).

Formalmente, un semaforo può essere acceduto da punti molto distanti rispetto alla definizione della struttura dati.

Nella pratica, il programmatore fa tutto il possibile per piazzare il semaforo vicino alla struttura dati da proteggere.

I difetti dei semafori

(Il programmatore deve preoccuparsi di come piazzarli)

Il programmatore è personalmente responsabile del piazzamento opportuno di **wait()** e **signal()**.

Se la sequenza **wait()**, sezione critica, **signal()** non è rispettata, ne succedono di tutti i colori.

Errore: stallo

(Analizzato in dettaglio nella lezione successiva)

A seguito di un errore di uso, può capitare che: un insieme di processi attenda il verificarsi di un evento.

l'evento in questione può essere causato solo da uno dei processi in attesa.

Tale situazione prende il nome di **stallo** (**deadlock**).

Errore: stallo

(Deadlock ABBA)

P_0

P_1

Errore tipico che porta allo stallo:
non viene preservato l'ordine
di richiesta/rilascio delle sezioni critiche.

Errore: stallo

(Deadlock ABBA)

P_0

P_1

`wait(S) ;`

Errore tipico che porta allo stallo:
non viene preservato l'ordine
di richiesta/rilascio delle sezioni critiche.

Errore: stallo

(Deadlock ABBA)

P_0

`wait(S) ;`

P_1

`wait(Q) ;`

Errore tipico che porta allo stallo:
non viene preservato l'ordine
di richiesta/rilascio delle sezioni critiche.

Errore: stallo

(Deadlock ABBA)

P_0

```
wait(S) ;  
wait(Q) ;
```

P_1

```
wait(Q) ;
```

Errore tipico che porta allo stallo:
non viene preservato l'ordine
di richiesta/rilascio delle sezioni critiche.

Errore: stallo

(Deadlock ABBA)

P_0

`wait(S) ;`

`wait(Q) ;`

P_1

`wait(Q) ;`

`wait(S) ;`

Errore tipico che porta allo stallo:
non viene preservato l'ordine
di richiesta/rilascio delle sezioni critiche.

Errore: stallo

(Deadlock ABBA)

P_0

```
wait(S) ;  
wait(Q) ;
```

P_1

```
wait(Q) ;  
wait(S) ;
```

Risultato: P_0 e P_1 si aspettano a vicenda, in eterno.
DEADLOCK.

Errore: stallo

(Deadlock ABBA)

P_0

```
wait(S) ;  
wait(Q) ;  
...  
signal(S) ;  
signal(Q) ;
```

P_1

```
wait(Q) ;  
wait(S) ;  
...  
signal(Q) ;  
signal(S) ;
```

Inserire le rispettive `signal()` alla fine non serve assolutamente a nulla.: P_0 e P_1 non le raggiungeranno mai.

Errore: accesso multiplo

(Si viola la mutua esclusione)

P_0

P_1

P_2

Errore tipico che porta all'accesso multiplo:
si invertono `signal()` e `wait()`.

Errore: accesso multiplo

(Si viola la mutua esclusione)

P_0

P_1

P_2

```
wait(S) ;  
<sez.  cr.>
```

P_0 acquisisce il semaforo S ed accede alla risorsa in mutua esclusione.

Errore: accesso multiplo

(Si viola la mutua esclusione)

P_0

P_1

P_2

```
wait(S) ;    signal(S) ;  
<sez. cr.>  <sez. cr.>
```

P_1 rilascia il semaforo S invece di acquisirlo.
Di conseguenza, entra in sezione critica.

Errore: accesso multiplo

(Si viola la mutua esclusione)

P_0

P_1

P_2

<code>wait(S) ;</code>	<code>signal(S) ;</code>	<code>wait(S) ;</code>
<code><sez. cr.></code>	<code><sez. cr.></code>	<code><sez. cr.></code>

La `signal()` di P_1 potrebbe risvegliare un processo P_2 precedentemente in attesa della stessa risorsa.

Errore: accesso multiplo

(Si viola la mutua esclusione)

P_0

P_1

P_2

<code>wait(S) ;</code>	<code>signal(S) ;</code>	<code>wait(S) ;</code>
<code><sez. cr.></code>	<code><sez. cr.></code>	<code><sez. cr.></code>

Risultato: ben tre processi sono riusciti ad entrare in sezione critica!

Errore: accesso multiplo

(Si viola la mutua esclusione)

P_0

P_1

P_2

```
wait(S) ;    signal(S) ;  
<sez. cr.>  <sez. cr.>
```

P_1 rilascia il semaforo S invece di acquisirlo.
Di conseguenza, entra in sezione critica.

Errore: stallo

(Deadlock doppio lock)

P_0

P_1

Errore tipico che porta allo stallo:
si sostituisce una `signal()` con una `wait()`.

Errore: stallo

(Deadlock doppio lock)

P_0

P_1

```
wait(S) ;  
<sez.cr.>
```

P_0 acquisisce il semaforo S ed accede alla risorsa in mutua esclusione.

Errore: stallo

(Deadlock doppio lock)

P_0

```
wait(S) ;  
<sez.cr.>
```

P_1

```
wait(S) ;
```

P_1 prova ad acquisire il semaforo S
e si blocca.

Errore: stallo

(Deadlock doppio lock)

P_0

```
wait(S) ;  
<sez.cr.>  
wait(S) ;
```

P_1

```
wait(S) ;
```

P_0 invoca la `wait(S)` al posto della `signal(S)`.

Errore: stallo

(Deadlock doppio lock)

P_0

```
wait(S) ;  
<sez.cr.>  
wait(S) ;
```

P_1

```
wait(S) ;
```

Risultato: P_0 e P_1 aspettano lo sblocco,
in eterno. DEADLOCK.