Sincronizzazione

Soluzioni hardware

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Classificazione

- Le soluzioni hardware al problema della SC possono essere classificate come segue
  - Soluzioni per sistemi che **non** permettono il diritto di **prelazione**
  - Soluzioni per sistemi che **permettono** il diritto di **prelazione**
    - Tramite la gestione delle **interruzioni**
    - Mediante "estensioni" delle soluzioni software
      - Utilizzo di "**lock**"
      - Utilizzo di istruzioni "**atomiche**"

La prelazione / non prelazione è anche funzione della presenza di sistemi multi-processore / core

Sistemi senza diritto di prelazione

- Un sistema **senza** diritto di **prelazione** è tale per cui
  - I P (o T) in esecuzione nela CPU **non** possono essere interrotti
  - Il controllo verrà rilasciato al kernel solo quando il P (o T) lo lascerà volontariamente

La CPU **non** può essere sottratta a un P (o T) in esecuzione

Sistemi senza diritto di prelazione

- Nei sistemi **mono**-processore e **senza** diritto di prelazione
  - **Non esiste** il problema della SC in quanto solo un P (o T) impegna l'unica CPU in un certo momento e tale P (o T) non può essere interrotto
- Questa situazione però si presenta raramente
  - I sistemi sono spesso multi-processore (o core)
    - Anche senza prelazione il parallelismo è effettivo utilizzando processori o core distinti
  - Eliminare il diritto di prelazione è insicuro
    - I kernel hanno tempi di risposta eccessivi
    - **Non** sono adatti alla programmazione "real-time"

Sistemi con diritto di prelazione

- In un sistema con diritto di **prelazione**
  - Un processo in esecuzione in modalità kernel può essere interrotto
  - Di fatto l'arrivo di un **interrupt** sposta il controllo del flusso su un altro processo
  - Il processo originario verrà terminato in seguito

La CPU **può** essere sottratta a un P (o T) in esecuzione

Sistemi con diritto di prelazione

- Nel sistemi **mono**-processore **con** diritto di prelazione
  - È possibile risolvere il problema della SC mediante controllo dell'interrupt
    - Disabilitare interrupt nella sezione di ingresso
    - Abilitare interrupt nella sezione di uscita

```
while (TRUE) {
   disabilita interrupt
   SC
   abilita l'interrupt
   sezione non critica
}
```

Si osservi che l'interrupt deve poter essere abilitato/disabilitato dal processo stesso

Sistemi con diritto di prelazione

- In generale, disabilitare gli interrupt ha però diversi svantaggi
  - La procedura è intrinsecamente insicura
    - Che cosa succede se a un processo utente si fornisce il diritto di disabilitare l'interrupt e tale processo funziona in modo scorretto?
    - Occorrerebbe quindi fornire tale opportunità ai soli processi kernel (super-user)
  - Nei sistemi multi-processore (o core) occorre disabilitare l'interrupt su tutti i processori
    - Occorre trasmettere la richiesta di disattivazione
    - Occorrono tempi lunghi di processamento
    - La gestione del sistema diventa **non** real-time

Meccanismi di lock - unlock

- Una strategia alternativa è mimare le soluzioni software, utilizzando
  - Lucchetti di protezione, i.e., **"lock"**
    - Si potrebbe utilizzare un lock per ciascuna SC
    - Il valore del lock permette l'accesso oppure lo vieta, proteggendo così la risorsa da accessi multipli
  - Istruzioni indivisibili, i.e., "**atomiche**", per manipolare tali lock
    - Un'istruzione atomica viene eseguita in un unico "memory cycle"
    - Non può essere interrotta
    - Permette verifica e modifica contestuale di una variabile globale

Meccanismi di lock - unlock

- Esistono due principali istruzioni atomiche di lock
  - **Test-And-Set**
    - Setta e restituisce una variabile di lock globale
    - Agisce in maniera **atomica,** ovvero in **un solo ciclo indivisibile**
  - **Swap**
    - Scambia (swap) due variabili, di cui una di lock globale
    - Agisce in maniera **atomica,** ovvero in **un solo ciclo indivisibile**

Test-And-Set: Implementazione

Riceve per riferimento la variabile di lock globale (attuale).
Il lock è di tipo char o int (basta 1 bit/byte) inizializzato a FALSE

```
char TestAndSet (char *lock) {
  char val;
  val = *lock;
  *lock = TRUE;
  return val;
}
```

In ogni caso, assegna TRUE al (nuovo) lock (effettua il lock della risorsa)

In ogni caso, restituisce il valore della variabile ricevuta (il vecchio lock)

Test-And-Set: Utilizzo

```
char lock = FALSE;
```

Variabile di lock globale

```
char TestAndSet (char *lock) {
  char val;
  val = *lock;
  *lock = TRUE;    // Set new lock
  return val;      // Return old lock
}
```

Se alla chiamata lock==TRUE la SC è occupata e si attende

Prologo: **Test and Set**

```
while (TRUE) {
   while (TestAndSet (&lock));     // lock
   SC
   lock = FALSE;                   // unlock
   sezione non critica
}
```

Se alla chiamata lock==FALSE si pone lock=TRUE e si occupa la SC

Test-And-Set: Svantaggi

```
char lock = FALSE;
```

La procedura TestAndSet non deve essere interrompibile

```
char TestAndSet (char *lock) {
  char val;
  val = *lock;
  *lock = TRUE;    // Set new lock
  return val;      // Return old lock
}
```

Busy-waiting su spin-lock: consuma cicli mentre attente

```
while (TRUE) {
   while (TestAndSet (&lock));      // lock
   SC
   lock = FALSE;                    // unlock
   sezione non critica
}
```

Swap: Implementazione

Riceve per riferimento la variabile di lock globale e un lock locale
Il lock è di tipo char o int (basta 1 bit/byte) inizializzato a FALSE

```
void swap (char *v1, char *v2) {
  char tmp;

  tmp = *v1;
  *v1 = *v2;
  *v2 = tmp;
  return;
}
```

Effettua uno scambio

Swap: Utilizzo

```
void swap (char *v1, char *v2) {
  char tmp;

  tmp = *v1;
  *v1 = *v2;
  *v2 = tmp;
  return;
}
```

```
char lock = FALSE;
```

Variabile di lock globale

Mettendo key=TRUE prenoto la SC

Se lock==FALSE la SC è libera, setto key a FALSE e lock a TRUE e accedo

Se lock==TRUE attendo

```
while (TRUE) {
  key = TRUE;
  while (key==TRUE)
    swap (&lock, &key); // Lock
  SC
  lock = FALSE;          // Unlock
  sezione non critica
}
```

Swap: Svantaggi

```
void swap (char *v1, char *v2) {
  char tmp;

  tmp = *v1;
  *v1 = *v2;
  *v2 = tmp;
  return;
}
```

```
char lock = FALSE;
```

La procedura swap non deve essere interrompibile

Busy-waiting su spin-lock: consuma cicli mentre attente

```
while (TRUE) {
  key = TRUE;
  while (key==TRUE)
    swap (&lock, &key); // Lock
  SC
  lock = FALSE;          // Unlock
  sezione non critica
}
```

Mutua esclusione senza starvation

- Le tecniche precedenti
  - Assicurano la mutua esclusione
  - Assicurano il progresso, evitando il deadlock
  - **Non** assicurano l'attesa definita di un processo, ovvero **non** garantiscono la non starvation
  - Sono simmetriche

I P/T lenti non entrano mai nella SC perchè i veloci la occupano ripetutamente

- Per soddisfare tutti e quattro i criteri occorre estendere le soluzioni precedenti
  - La soluzione successiva utilizza TestAndSet e deriva da quella originale
  - È dovuta a Burns [1978]

Mutua esclusione senza starvation

P_i

Un vettore di attesa, i.e., prenotazione, con un elemento per ogni P_i/T_i inizializzato a FALSE

Lock globale, unico, inizializzato a FALSE

```
while (TRUE) {
  waiting[i] = TRUE;
  while (waiting[i] && TestAndSet (&lock));
  waiting[i] = FALSE;
  SC
  j = (i+1) % N;
  while ((j!=i) && (waiting[j]==FALSE))
    j = (j+1) % N;
  if (j==i)
    lock = FALSE;
  else
    waiting[j] = FALSE;
  sezione non critica
}
```

Mutua esclusione senza starvation

P_i

```
while (TRUE) {
  waiting[i] = TRUE;
  while (waiting[i] && TestAndSet (&lock));
  waiting[i] = FALSE;
  SC

  sezione non critica
}
```

Entra in SC se
desidera entrare (NON è in waiting)
OR
la SC è libera (lock=FALSE → return TRUE)

Mutua esclusione senza starvation

P_i

```
while (TRUE) {

  SC
  j = (i+1) % N;
  while ((j!=i) && (waiting[j]==FALSE))
    j = (j+1) % N;
  if (j==i)
    lock = FALSE;
  else
    waiting[j] = FALSE;
  sezione non critica
}
```

I P/T in coda entrano in SC perché ricevono il testimone dal precedente

Si verirfica se qualcuno è in waiting

Se nessuno è in waiting si mette il lock a FALSE

Altrimenti si "libera" il primo P_i/T_i in attesa

Conclusioni

- Vantaggi delle soluzioni hardware
  - Utilizzabili in ambienti multi-processore
  - Facilmente estendibili a N processi
  - Relativamente semplici da utilizzare dal punto di vista software, cioè dal punto di vista utente
  - Simmetriche

Conclusioni

- Svantaggi delle soluzioni hardware
  - Non facili da implementare a livello hardware
    - Operazioni atomiche su variabili globali (qualsiasi)

Argomento corso di architettura dei calcolatori

  - Starvation
    - La selezione dei processi in busy-waiting per la SC è arbitraria e gestita dai processi stessi non dal SO
  - Busy waiting su spin-lock
    - Spreco di risorse ovvero di cicli di CPU nell'attesa
      - In pratica il busy-waiting è utilizzabile sono quando le attese sono molto brevi

Conclusioni

- Inversione di priorità (**priority inversion**)
  - Si considerino 3 processi in esecuzione
    - A con priorità alta, M con priorità media e B con priorità bassa
  - Il processo B occupa una risorsa
  - Il processo A entra in attesa della stessa risorsa
  - Il processo M entra in esecuzione
  - Dato che la CPU viene dedicata a M e non a B, B non rilascia la SC e previene A dall'entrarci
  - In altre parole la risorsa viene occupata dal processo a minore priorità molto più a lungo del necessario
  - Una possibile cura a questo problema consiste nell'utilizzare il **protocollo di ereditarietà della priorità**
    - Un processo in possesso di un lock eredita automaticamente la priorità del processo con priorità maggiore in attesa dello stesso lock

Problema generale nello scheduling (esempio con 3 processi di priorità Bassa, Media, Alta)