

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
#define MAXPAROLA 30
#define MAXRIGA 80
```

```
int main(int argc, char *argv[])
{
    int freq[MAXPAROLA]; /* vettore di contatori
delle frequenze delle lunghezze delle parole */
    char riga[MAXRIGA];
    int i, inizio, lunghezza;
    FILE *f;
```

```
for(i=0; i<MAXPAROLA; i++)
    freq[i]=0;
```

```
if(argc != 2)
```

```
{
    fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
    exit(1);
}
```

```
f = fopen(argv[1], "r");
if(f==NULL)
```

```
{
    fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
    exit(1);
}
```

```
while( fgets( riga, MAXRIGA, f ) != NULL )
```



# Sincronizzazione

## Soluzioni hardware

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

# Informazioni sul Copyright

This work is licensed under the license



## Attribution-NonCommercial-NoDerivatives 4.0 International

This license requires that reusers give credit to the creator. It allows reusers to copy and distribute the material in any medium or format in unadapted form and for noncommercial purposes only.

① **BY:** Credit must be given to you, the creator.

② **NC:** Only noncommercial use of your work is permitted.

Noncommercial means not primarily intended for or directed towards commercial advantage or monetary compensation.

③ **ND:** No derivatives or adaptations of your work are permitted.

To view a copy of the license, visit:

<https://creativecommons.org/licenses/by-nc-nd/4.0/?ref=chooser-v1>

## Classificazione

- ❖ Le soluzioni hardware al problema della SC possono essere distinte in base al fatto che il SO
  - **Non** permetta il diritto di **prelazione**
  - **Permetta** il diritto di **prelazione**
- ❖ Definizione
  - **Prelazione (preemption)**
    - Garantisce al SO di poter interrompere un task in esecuzione e di assegnare la CPU a un altro task (solitamente con priorità più alta)

## Sistemi senza diritto di prelazione

- ❖ Un sistema **senza** diritto di **prelazione** è tale per cui
  - I task in esecuzione **non** possono essere interrotti
  - Il controllo verrà rilasciato al kernel solo quando il task lo lascerà volontariamente

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

- ❖ Nei sistemi **mono**-processore
  - In assenza del diritto di prelazione **non esiste** il problema della SC in quanto solo un task impegna l'unica CPU in un ogni momento e tale task non può essere interrotto

## Sistemi senza diritto di prelazione

- ❖ Questa situazione però si presenta raramente
  - I sistemi sono spesso multi-processore (muti-core)
    - Anche senza prelazione il parallelismo è effettivo utilizzando processori o core distinti
  - Eliminare il diritto di prelazione è insicuro
    - I task possono operare in maniera scorretta e rimane in esecuzione per sempre
    - Inoltre, il SO risulta avere tempi di risposta eccessivi
  - I sistemi senza diritto di prelazione **non** sono adatti ai SO moderni
    - Il diritto di prelazione è fondamentale per garantire che si possano gestire più processi contemporaneamente in modo efficiente e reattivo

## Sistemi con diritto di prelazione

- ❖ In un sistema con diritto di **prelazione**
  - Un task in esecuzione può essere interrotto
  - 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

- ❖ Le tecniche di soluzione del problema della SC si distinguono a seconda il sistema sia mono o multi-processore (core)



## 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

L'interrupt deve poter essere abilitato/disabilitato dal processo stesso

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

- In generale, disabilitare gli interrupt ha però gli stessi svantaggi dei sistemi senza prelazione

## Sistemi con diritto di prelazione

❖ Nel sistemi **multi**-processore **con** diritto di prelazione disabilitare l'interrupt

➤ **Può**

- Ridurre i conflitti, in quanto i task possono completare il loro lavoro senza essere interrotti, riducendo il rischio di inconsistenze
- Semplificare la sincronizzazione, in quanto non è necessario implementare complessi meccanismi di locking o sincronizzazione per gestire la contesa tra processi

➤ **Non** può risolvere completamente il problema della sezione critica



## Meccanismi di lock - unlock

- ❖ Una strategia alternativa è mimare le soluzioni software, utilizzando
  - Lucchetti di protezione, i.e., **"lock"**
    - È possibile 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", ovvero **non** può essere interrotta
      - Quindi, ad esempio, 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  
}
```

Prologo:  
**Test and Set**

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

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

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

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

Se  
lock==TRUE  
attendo

## 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 task 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 stata introdotta da Burns [1978]

# Mutua esclusione senza starvation

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

$P_i$

```
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  
}
```

Lock globale, unico,  
inizializzato a FALSE

# Mutua esclusione senza starvation

 $P_i$ 

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

**SC**

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

sezione non critica

}

# 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 verifica 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 solo 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)