# Laboratorio di Reti e Sistemi Distribuiti

## 14: Task Synchronization

Roberto Marino, PhD[1]
roberto.marino@unime.it

[1]Dipartimento di Matematica, Informatica, Fisica e Scienze della Terra
Future Computing Research Laboratory
Università di Messina

Last Update: 9th April 2025

# Introduzione

## Necessità di protezione

In un sistema a **memoria condivisa** bisogna sempre assicurare che le risorse siano protette da **accessi concorrenti**. Questo avviene perchè più thread di esecuzione possono accedere e manipolare i dati nello stesso istante e/o trovare i dati in uno stato **inconsistente**.

Prima dell'avvento del Simmetrical Multi Processing (SMP) questo obiettivo era di facile soluzione. Con l'avvento dell'SMP (kernel 2.0) e dello scheduler **preemptive** (kernel 2.6, diritto di prelazione, a.k.a. diritto di preferenza) le cose si sono fatte molto più complesse.

# Regioni Critiche e Race Conditions

**Def. Regione Critica**

Sequenza di istruzioni (thread di esecuzione) che accede e manipola dati **condivisi**

**Def. Atomicità**

Per prevenire accessi concorrenti il programmatore deve essere certo che l'accesso alla regione critica avvenga in modo atomico, cioè come se fosse una unica, indivisibile, istruzione eseguita senza interruzioni.

**Def. Race Condition**

Se non si garantisce atomicità due flussi di esecuzione possono trovarsi simultaneamente nella stessa regione critica e si trovano in una condizione di competizione chiama race condition

**Def. Sincronizzazione**

Prevenire una race condition significa "sincronizzare" le sequenze di istruzioni

# Esempio

Consideriamo una semplice risorsa condivisa, una variabile intera, ed una semplice regione critica, l'operazione di incremento su di essa

```
1    int i; /* risorsa condivisa */
2    i++;   /* regione critica */
```

A livello macchina questa viene tradotta in:

1. Prenti il valore corrente di i dalla memoria e copialo in un registro (**get**)
2. Aggiungi uno al valore (**increment**)
3. Copia il nuovo valore di i in memoria (**write back**)

# Esempio

Supponiamo di avere due thread che eseguono entrambi la stessa regione critica di codice e che la variabile sia inizializzata a 7.
Il risultato desiderato è il seguente:

Thread 1:
get i (7)
increment i ($7 \rightarrow 8$)
write back i (8)
–
–
–

Thread 2:
–
–
–
get i (8)
increment i ($8 \rightarrow 9$)
write back i (9)

# Critical Region Interleaving

In realtà, senza alcun meccanismo di sincronizzazione, otteniamo qualcosa del genere (inconsistenza):

Thread 1:
get i (7)
increment i ($7 \rightarrow 8$)
–
write back i (8)
–
–

Thread 2:
get i (7)
–
increment i ($7 \rightarrow 8$)
–
write back i (8)

# Atomic Operation

Fortunatamente tutti i processori moderni posseggono delle istruzioni di incremento atomiche cioè che permettono di leggere, incrementare e riscrivere in memoria in un'unico, indivisibile, passo di esecuzione (no interleaving).

Thread 1:
increment i ($7 \rightarrow 8$)
–

Thread 2:
–
increment i ($8 \rightarrow 9$)

oppure:

Thread 1:
–
increment i ($7 \rightarrow 8$)

Thread 2:
increment i ($8 \rightarrow 9$)
–

# Locking

Supponiamo ora di avere non una variabile ma una **struttura dati condivisa**. Come fare in modo di agire atomicamente su di essa evitando stati inconsistenti della memoria? Bisogna pensare a dei **meccanismi software** che permettano l'accesso atomico (indivisibile) alle strutture dati (qualunque esse siano).

## Locking

Immaginiamo una stanza dietro ad una porta come una sezione critica (ad esempio un bagno). La serratura funziona da **lock**. Due persone vogliono entrare nella stanza. La prima che arriva si chiude dietro la porta (**acquisisce il lock**) a chiave mentre la seconda aspetta. Appena finito agisce nuovamente sulla serratura, aprendola (**rilascia il lock**). Chi stava aspettando dietro la porta può ora entrare.

# Lock Example

Thread 1:
T1: try to lock the queue
T2: succeeded: acquired lock
T3: access queue...
T4: unlock the queue
T5: ...
T6: ...
T7: ...

Thread 2:
T1: try to lock the queue
T2: failed: waiting...
T3: waiting...
T4: waiting...
T5: succeeded: acquired lock
T6: access queue...
T7: unlock the queue

**Implementazione a basso livello**

I lock e tutti i meccanismi di sincronizzazione si basano su istruzioni atomiche specifiche del processore (test-and-set: settano un intero ad uno solo se zero. Zero significa unlocked)

# Cosa causa concorrenza?

- Se due thread accedono alla stessa regione critica dopo un evento di preemption, su una macchina monoprocessore siamo in presenza di **pseudo-concorrenza**.
- Se due thread accedono alla stessa regione critica su una macchina SMP, quindi potenzialmente nello stesso identico istante di tempo, siamo in presenza di **concorrenza reale**.

## Cosa causa concorrenza?

1. Interrupts: un interrupt può occorrere in ogni momento.
2. Kernel Preemption: un task può essere interrotto dallo scheduler
3. SMP: due o più processori/core possono eseguire regioni critiche nello stesso identico istante

Il nostro obiettivo è realizzare codice **interrupt-safe**, **SMP-safe** e **Preemption-safe**.

# Uso improprio dei lock

La programmazione concorrente è difficile!

**Deadlock**

Uno o più threads vogliono accedere ad uno o più risorse ma ognuna delle risorse è in stato di lock (Esempio dell'incrocio e delle quattro automobili)

**Starvation**

Un thread non riesce mai ad accedere alla sezione critica alla quale vuole accedere, e muore di fame! (starvation)

# SpinLock

A livello kernel esistono due tipologie principali di lock:

## SpinLock

Il thread che resta fuori dalla sezione critica controlla in busy-loops (spins) finchè non riesce ad acquisire il lock. Velocissimo ma consuma CPU. Permette un unico accesso alla volta.

## Semafori/Mutex

Il thread che resta fuori dalla sezione critica viene messo in SLEEP. Un semaforo permette N accessi simultanei. Se N=1 siamo in presenza di un mutex (mutua esclusione). Più lento ma non consuma CPU.