OS161-SYNCHRO

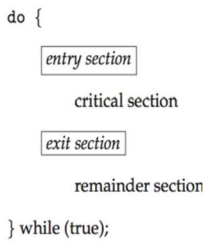
Programmazione concorrente e sincronizzazione, accesso cordinato alle risorse condivise.

Problemi:

- Race condition
  - Risultati che dipendono dal modo casuale in cui si sono effettuate le operazioni
- Deadlocs
  - A aspetta B
  - B aspetta C
  - C aspetta A
- Resource starvation
  - Quelli aggressivi vanno avanti ma c'è qualcuno che continua a richiedere ma non ha accesso alle risorse

CRITICAL SECTION:

Pezzo di codice critico in cui bisogna far attenzione *perché ci potrebbe essere il rischio che ci sia qualcun altro su quella sezione o qualcuno che stia lavorando ad un'altra sezione di codice che ha però influenza sulla mia*.



Soluzioni:

- **Single core**
  - Quando si è in esecuzione, si può stare tranquilli perché nessun'altro è in esecuzione
  - Non pre-empty → non puoi essere interrotto
  - Preempty → può stoppare in esecuzione
- **Multicore**
  - Quando si è in esecuzione c'è un alto rischio che ci sia qualcuno che stia lavorando sulla tua sezione critica

Nota: processo in kernel-mode se sta eseguendo una system call.

Soluzioni di Peterson:

- Per garantire **mutua esclusione**:
  - Se c'è sezione crtitica e c'è un flag che indica a chi tocca, potremmo fare in modo di settare il flag in modo tale che indichi a chi tocca
    - Ci deve essere un modo che setti il flag e ci deve essere qualcosa che mi indichi occupato
      - **Flag** → vorrei usare sezione critica
      - **Turn** → chi sta lavorando

```
while (true){
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j)
                ;
        /* critical section */
        flag[i] = false;
        /* remainder section */
}
```

→ mi prenoto
→ do la precedenza all'altro
→ finchè l'altro è prenotato e tocca a lui, aspetto (se l'altro non era prenotato, procedo io)

Ha un difetto, si lavora a basso livello → usiamo variabili qualunque per fare sincronizzazione. Inoltre, potrebbero esserci race conditions.

Meglio usare i lock:

Implementazioni:

- **Acquire** lock
- **Release** lock

Ci sono **due possibilità**:

- Keep trying
  - Continui a chiedere
- Give up the processor
  - Lasci perdere e vuoi essere svegliato

Hardware fondamentale in aiuto:

- **Memory barriers**: cercano di lavorare all'ordinamento delle operazioni in memoria e fare in modo che le istruzioni di scrittura su un processore, siano subito disponibili agli altri
- **Istruzioni hw**
  - **Test-and-set** → operazione che legge e scrive insieme senza far passare nulla in mezzo
  - **Compare-and-swap** → variante del test-and-set che si basa su comparazione tra due variabili e swap
- **Variabili atomiche**

Serve qualcuno che garantisca che tra quando leggo libero e provo ad occupare, nessuno occupi.

- La cosa importante è la sequenza ocn cui si scrive e si legge in memoria

Solution using test_and_set()

- Shared boolean variable `lock`, initialized to `false`
- Solution:

```
do {
       while (test_and_set(&lock))
        ; /* do nothing */

            /* critical section */

    lock = false;
            /* remainder section */
 } while (true);
```

0 = lock non occupato

Iplementazione busy-wait → continui a testare il processore.

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}

release() {
    available = true;
}
```

These two functions must be implemented atomically. Both test-and-set and compare-and-swap can be used to implement these functions.

Mutex locks → lock usato per la mutua esclusione

Spin-locks: in os161 esistono già:

- Ok per operazioni rapide → puoi fare busying waiting se sai che durerà poco
- Implementati con test-and-set

```
class TASspinlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

 void lock() {
  while (state.getAndSet(true)) {}
 }

 void unlock() {
  state.set(false);
 }}
```

Nota: esiste implementazione **test(Test-and-set)** : leggi senza effettuare operazione atomica, se leggi libero allora prova la test-and-set

Semafori:

- Può essere visto come estensione del lock
  - **Wait**(S) → decrementa s
    - Finchè S≤0 → non disponibile, aspetto
  - **Signal** → incrementa S

➔ Operazione effettuata con compare-and-swap

Anche per il semaforo, ci sono implementazioni senza busy waiting