

# Sistemi Operativi

Laurea in Ingegneria Informatica

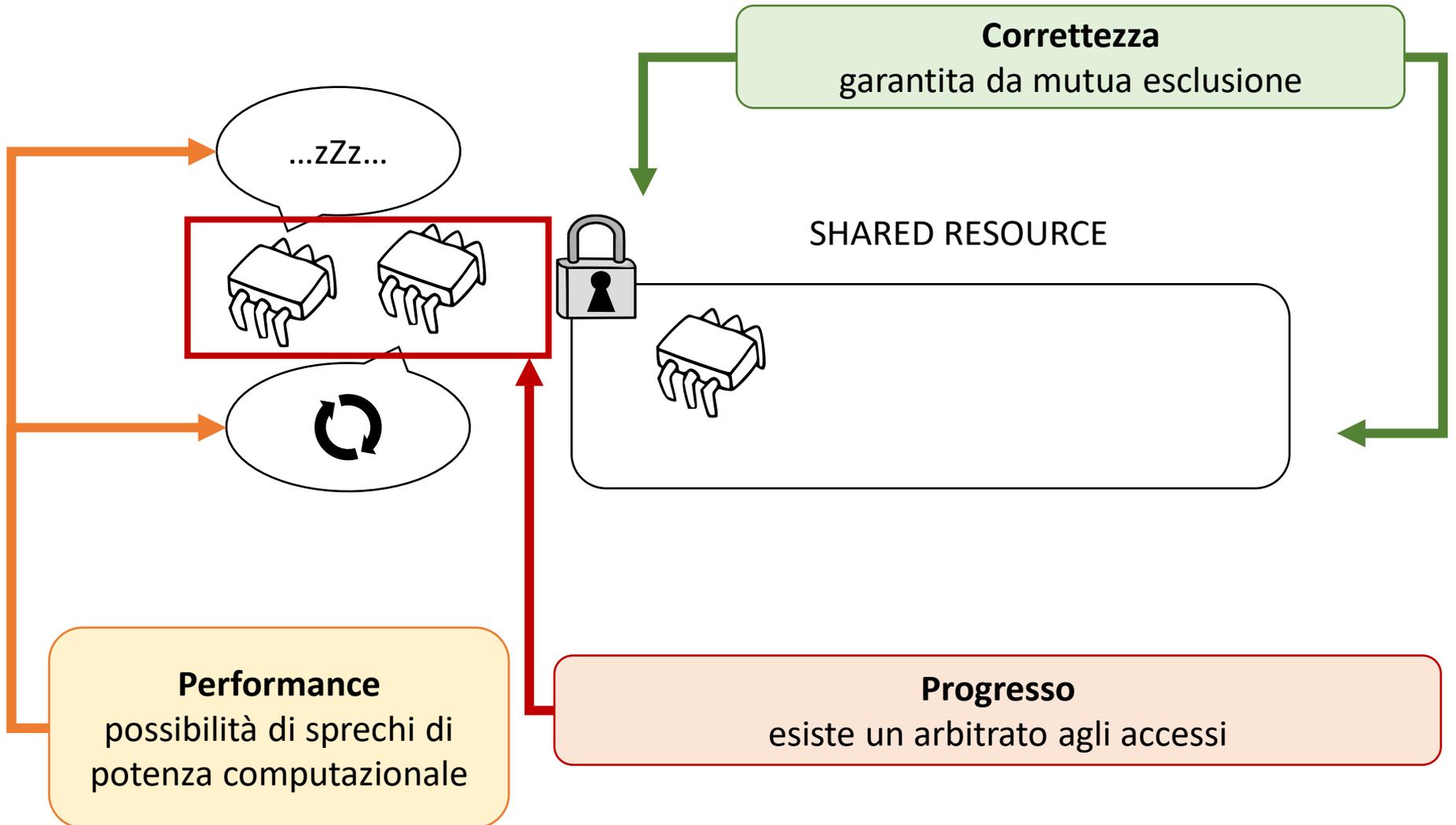
Università Roma Tre

Docente: Romolo Marotta

## Sincronizzazione

1. Mutua esclusione
2. Soluzioni software
3. Soluzioni hardware
4. Strumenti POSIX per la sincronizzazione

# Sincronizzazione



# Sincronizzazione

- Sezione critica
  - Porzione di codice che manipola variabili condivise
  - Richiesta mutua esclusione
    - Al più un processo può eseguire la propria sezione critica
- Protocolli per arbitrare l'accesso alla sezione critica
  - Eseguiti prima e dopo la sezione critica
  - Partecipano solo i processi che intendono eseguire la sezione critica
  - La decisione deve essere presa in un tempo finito

Entry protocol

***Critical section***

Exit protocol

# Tentativo 1 (per 2 processi)

- Variabili condivise:
  - `int turn; /* init 0 */`
  - `int num_proc = 2;`
- Variabili per processo  $P_i$ :
  - `int myturn = i`

```
while(turn != myturn)/* noop */;
```

*Critical section*

```
turn = (myturn+1)%num_proc
```

Processo P0

```
while(turn != myturn)/* noop */;
```

*Critical section*

```
turn = (myturn+1)%num_proc
```

```
while(turn != myturn)/* noop */;
```

Processo P1

**Progresso**

Il processo P0 deve attendere che P1 acceda alla sezione critica prima di potervi accedere nuovamente

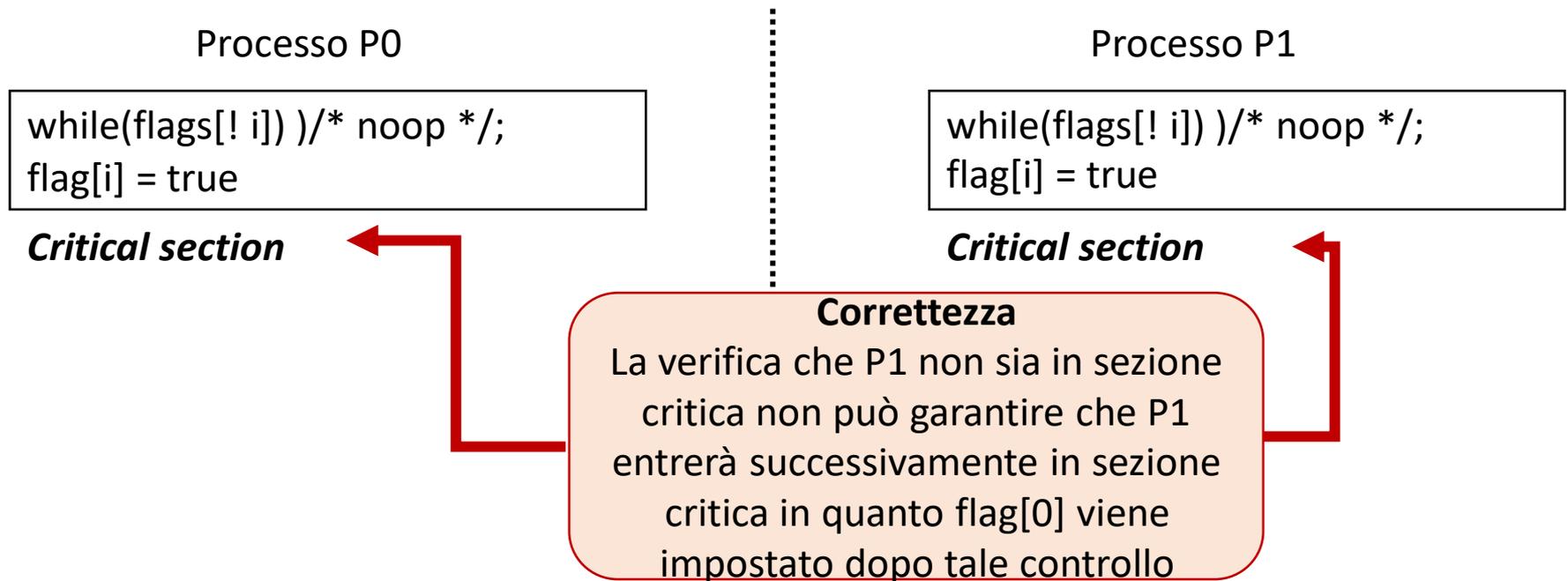
# Tentativo 2 (per 2 processi)

- Variabili condivise:
  - `int flags[2]; /* init false */`
- Variabili per processo  $P_i$ :
  - **nessuna**

```
while(flags[! i]) /* noop */;  
flag[i] = true
```

***Critical section***

```
flag[i] = false
```



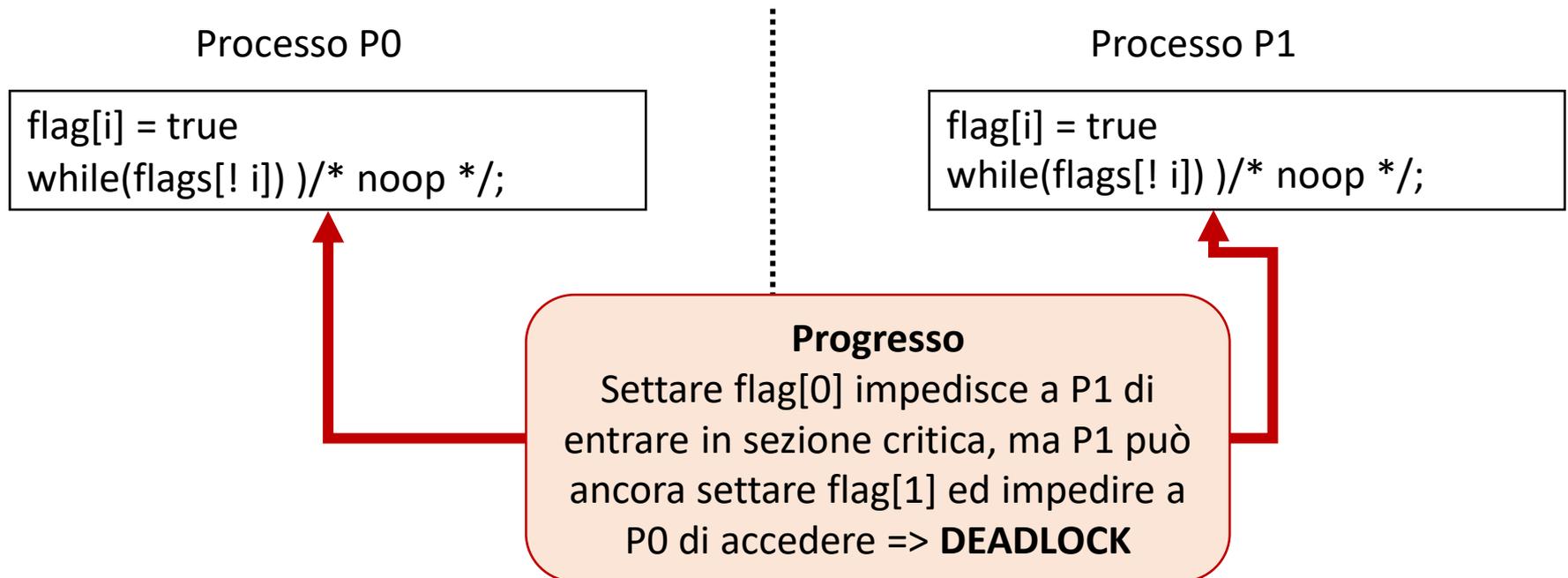
# Tentativo 3 (per 2 processi)

- Variabili condivise:
  - `int flags[2]; /* init false */`
- Variabili per processo  $P_i$ :
  - **nessuna**

```
flag[i] = true  
while(flags[! i]) /* noop */;
```

*Critical section*

```
flag[i] = false
```



# Tentativo 4 (per 2 processi)

- Variabili condivise:
  - `int flags[2]; /* init false */`
- Variabili per processo  $P_i$ :
  - **nessuna**

```
flag[i] = true
while(flags[! i]) ){
    flag[i] = false;
    randomDelay();
    flag[i] = true;
}
```

**Critical section**

```
flag[i] = false
```

Processo P0

```
flag[i] = true
flags[! i] == true // while cond.
flags[i] = false
// ..
// ..
flags[i] = true
flags[! i] == true // while cond.
```

**Progresso**  
Scenario simile al tentativo 3, ma più improbabile. Inoltre, i processi hanno la potenziale opportunità di procedere, ma sono «sfortunati» => **LIVELOCK**

Processo P1

```
flag[i] = true
flags[! i] == true // while cond.
flags[i] = false
// ..
flags[i] = true
flags[! i] == true // while cond.
```

# Tentativo 5 (per 2 processi)

- Variabili condivise:
  - `int flags[2]; /* init false */`
- Variabili per processo  $P_i$ :
  - **nessuna**

Processo P0

**Progresso**  
La decisione è presa  
deterministicamente in  
tempo finito a scapito di P0

```
flag[i] = true
flags[! i] == true // 1st while cond.
flag[i] = false
while(flags[! i]); // 2nd while
```

```
flags[i] = true
flags[! i] == true // 1st while cond.
flags[i] = false
```

```
flag[i] = true
while(flags[! i]) {
  if(i == 0){
    flag[i] = false;
    while(flags[! i] );
    flag[i] = true;
  }
}
```

**Critical section**

```
flag[i] = false
```

Processo P1

```
flag[i] = true
while(flags[! i] == true) // 1st while
```

**Critical section**

```
flag[i] = false
```

```
flag[i] = true
while(flags[! i] == true) // 1st while
```

# Tentativo 6 (per 2 proc.) aka Peterson's algorithm

- Variabili condivise:
  - **int** flags[2]; /\* init false \*/
  - **int** turn;
- Variabili per processo P<sub>i</sub>:
  - **nessuna**

Processo P0

```
flag[i] = true  
  
turn = !i // 1  
while(flags[! i] && turn != i);
```

***Critical section***

```
flag[i] = true  
turn = ! i;  
while(flags[! i] && turn != i);
```

***Critical section***

```
flag[i] = false
```

Processo P1

```
flag[i] = true  
turn = !i // 0  
  
while(flags[! i] && turn != i);
```

***Critical section***

```
flag[i] = false
```

```
flag[i] = true  
turn = !i // 0  
while(flags[! i] && turn != i);
```

# Tentativo 6 (per 2 proc.) aka Peterson's algorithm

- Variabili condivise:
  - `int flags[2]; /* init false */`
  - `int turn;`
- Variabili per processo  $P_i$ :
  - **nessuna**

Processo P0

```
flag[i] = true
```

***Critical section***

```
flag[i] = true  
turn = ! i;  
while(flags[! i] && turn != i);
```

***Critical section***

```
flag[i] = false
```

Processo P1

```
flag[i] = true  
turn = !i // 0
```

```
while(flags[! i] && turn != i);
```

***Critical section***

```
flag[i] = false
```

```
flag[i] = true  
turn = !i // 0  
while(flags[! i] && turn != i);
```

# Bakery algorithm [Lampport1974]

- Variabili condivise:
  - **boolean** choosing[N];
  - **int** number[N];
- Variabili per processo  $P_i$ :
  - **nessuna**

```
choosing[i] = true
number[i] = max(number[])+1;
choosing[i] = false
for(j = 0 to n-1){
  while(choosing[j]);
  while number[j] != 0 and
    (number[j,j] < (number[i],i));
}
```

*Critical section*

```
number[i] = 0
```

- I processi acquisiscono un numero
- I processi accedono alla sezione critica in accordo al numero assegnatogli
- Processi non interessati alla sezione critica non vengono considerati
- L'acquisizione è non atomica
  - Più processi possono ottenere il medesimo numero
    - A parità di numero, accede alla sezione critica il processo con identificativo minore
  - Tempo di acquisizione non istantaneo
    - Prima di entrare in sezione critica si attende che un processo completi l'operazione di acquisizione di un ticket

# Approcci hardware

- Le soluzioni proposte utilizzano solo operazioni di scrittura e lettura
- L'architettura può disporre di alcune operazioni per attuare operazioni più complesse su una data cella di memoria

```
val Read-Modify-Write(val *ptr, val (*op) (val) ) {  
    val = *ptr;  
    new_val = op(val);  
    *ptr = new_val;  
    return val;  
}
```

- Istruzioni RMW possono essere eseguite atomicamente
  - disabilitando le interruzioni (su architetture monoprocesore)
  - supporto hardware all'atomicità (tipicamente offerto in architetture multiprocesore)

# Istruzioni Read-Modify-Write

Esempi su x86:

- BTS

```
val bit-test-and-set(int *ptr, int pos){
    atomic{
        int val = *ptr;
        int new_val = val | (1 << pos);
        *ptr = new_val;
        return (val & (1<<pos)) != 0;
    }
}
```

- XCGH

```
val exchange(int *ptr, int val){
    atomic{
        int res = *ptr;
        *ptr = val;
        return res;
    }
}
```

# Istruzioni Read-Modify-Write

Esempi su x86:

- XADD

```
val fetch_and_add(int *ptr, int val){
    atomic{
        int res = *ptr;
        int new_val = res + val;
        *ptr = new_val;
        return res;
    }
}
```

- CMPXCHG

```
val compare_and_swap(int *ptr, int old_val, int new_val){
    atomic{
        int res = *ptr;
        if(res == old_val) *ptr = new_val;
        return res;
    }
}
```

# Test-and-set lock

- Test-and-set lock è la forma più semplice di lock
- Thread che vogliono accedere alla sezione critica tentano di settare una variabile con una RMW

```
int lock = 0;
```

```
void acquire(int *lock){  
    while(XCHG(lock, 1));  
}  
  
void release(int *lock){  
    *lock = 0;  
}
```

# Mini-benchmark

- Un array di interi
- Ogni thread inverte un array

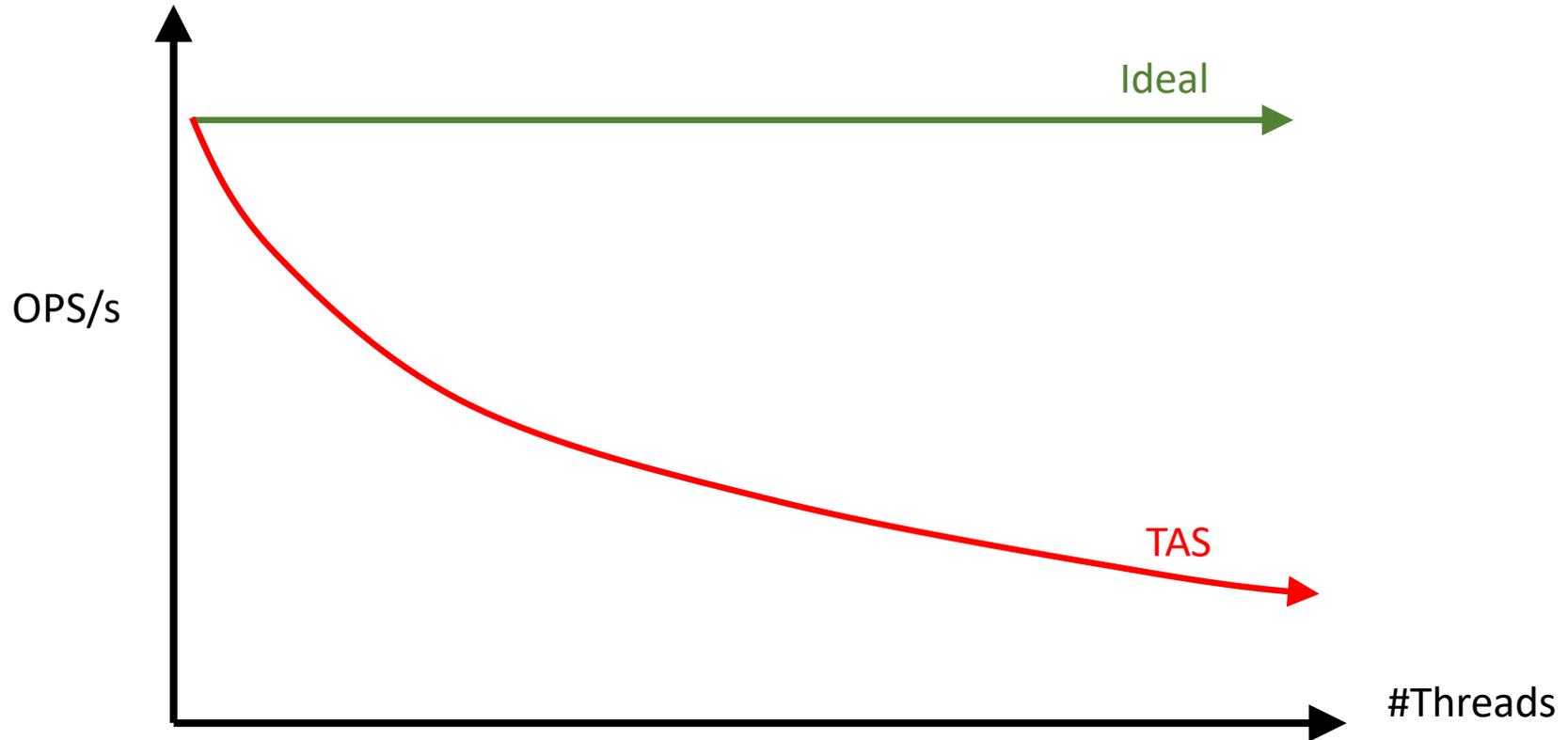


- L'operazione è eseguita all'interno di una sezione critica

```
while(!stop){  
    acquire(&lock);  
    reverse_array();  
    release(&lock);  
}
```

- Metrica di performance:
  - Throughput = #Flips per second

# Risultato



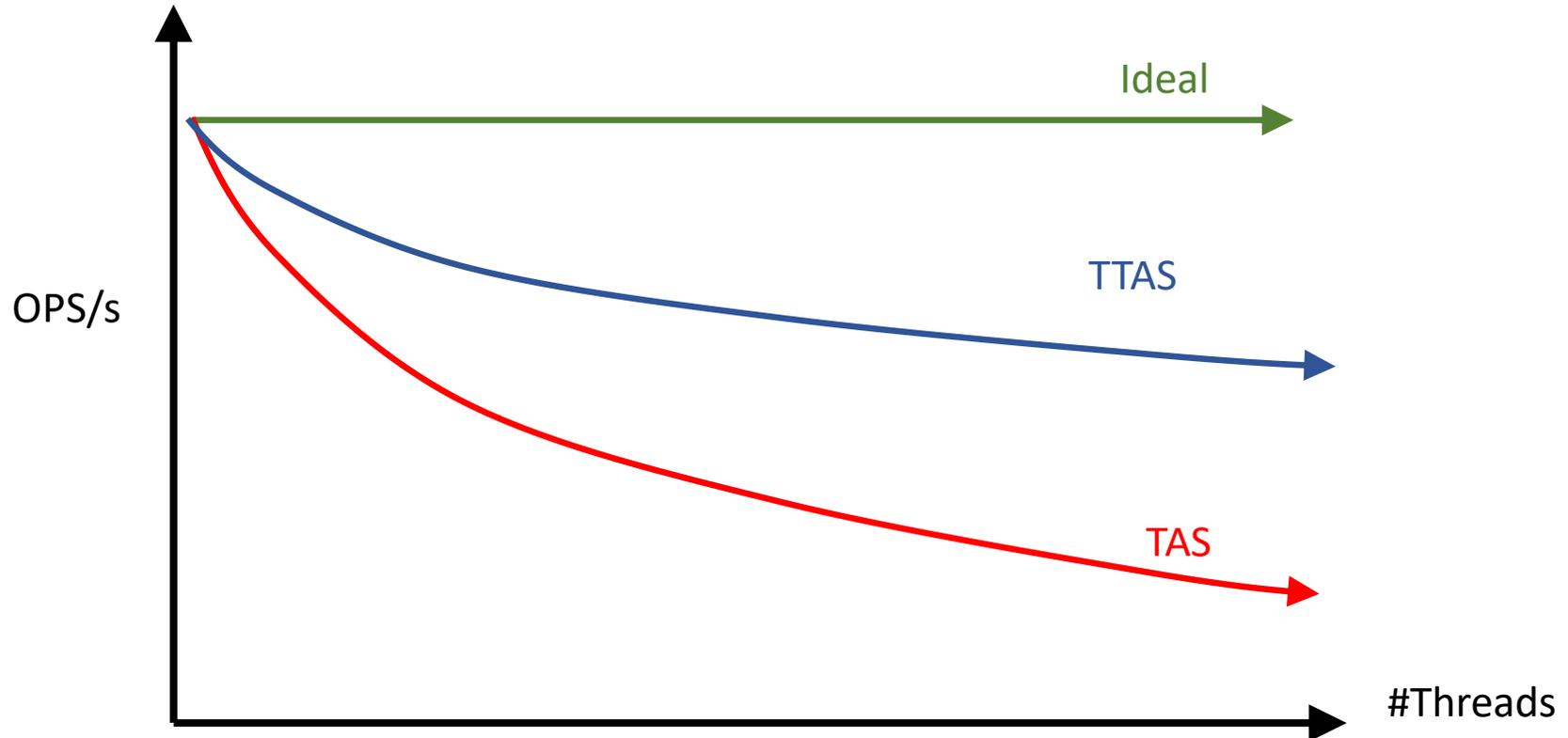
# Test-and-Test-and-set lock

- Come test-and-set, ma i processi aspettano facendo solo operazioni di lettura
- Istruzioni RMW sono utilizzate solo quando il lock è rilasciato

```
int lock = 0;
```

```
void acquire(int *lock){  
    while(XCHG(lock, 1))  
        while(*lock);  
}  
  
void release(int *lock){  
    *lock = 0;  
}
```

# Risultato



# POSIX spin lock

- `pthread_spinlock_t`
- `pthread_spin_lock_init(`  
    `pthread_spinlock_t *lock, int pshared)`
  - `PTHREAD_PROCESS_PRIVATE`
  - `PTHREAD_PROCESS_SHARED`
- `pthread_spin_lock(pthread_spinlock_t *lock)`
- `pthread_spin_trylock(pthread_spinlock_t *lock)`
- `pthread_spin_unlock(pthread_spinlock_t *lock)`

# POSIX MUTEX (MUTual EXclusion)

- Gli spin locks spremano cicli di clock per implementare attese attive
- In scenari di oversubscribing ( $\#threads > \#cores$ ) può essere conveniente togliere i thread dalla ready-to-run queue
- Serve supporti dal Sistema Operativo
  - In Linux basati sulla syscall FUTEX
- `pthread_mutex_t mutex;`
- `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)`
- `pthread_mutex_lock(pthread_mutex_t *mutex)`
- `pthread_mutex_trylock(pthread_mutex_t *mutex)`
- `pthread_mutex_unlock(pthread_mutex_t *mutex)`

# Semafori

- Struttura dati a cui è associato:
  - Un intero **S** positivo
  - Operazione di **wait**
  - Operazione di **signal**
- Wait:
  - Tenta il decremento di una unità di S
  - Se  $S == 0$  prima del decremento, il thread rimane in attesa
- Signal:
  - Incrementa S di una unità
  - Se un qualche thread è in attesa, questo viene sbloccato
- I Mutex possono essere visti come semafori binari
  - 1 semaforo libero
  - 0 semaforo occupato

# POSIX semaphore

- `sem_t`
- `int` `sem_init(sem_t *sem, int pshared, unsigned value);`
- `int` `sem_destroy(sem_t *);`
- `int` `sem_wait(sem_t *sem);`
- `int` `sem_trywait(sem_t *sem);`
- `int` `sem_timedwait(sem_t *restrict, const struct timespec *restrict);`
- `int` `sem_post(sem_t *sem);`
- `int` `sem_getvalue(sem_t *restrict, int *restrict);`