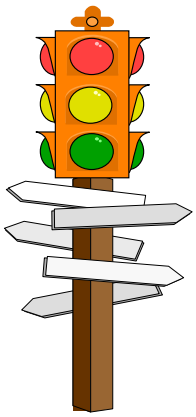
# SISTEMI OPERATIVI

04.a

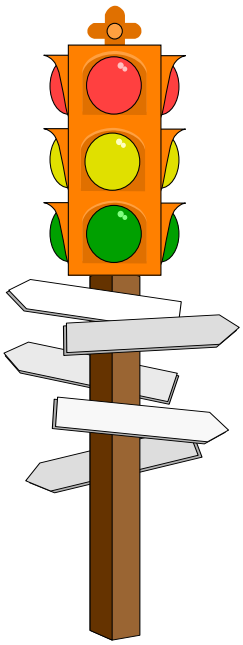


Semafori

## Semafori

1

- Approfondimento sulla mutua esclusione
- Manipolazione di risorse mutex
- Strategie di soluzione
- Lock e Unlock
- Primitive semaforiche



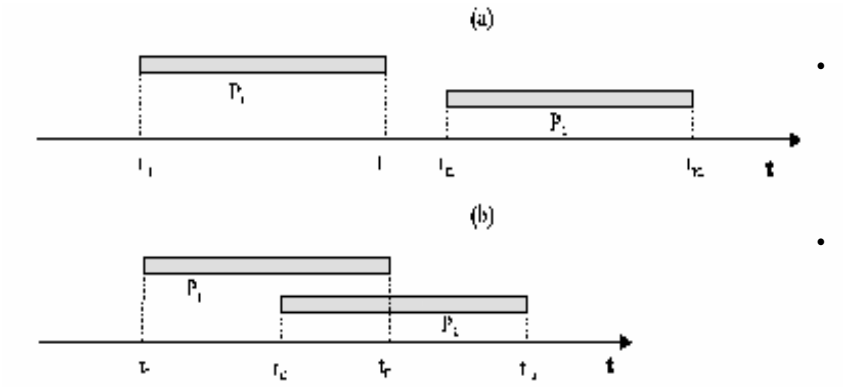
27

# Approfondimento sulla mutua esclusione



- **Mutua esclusione**: la sincronizzazione consente a più processi di operare su dati condivisi, controllando l'interferenza
  Il problema nasce quando le risorse sono non condivisibili, cioè tali da non poter essere utilizzate da più di un processo alla volta

  È sufficiente che le operazioni sulle risorse comuni non si sovrappongano nel tempo !!




- Uso della risorsa mutex corretto da parte di P1 e P2
- Uso della risorsa mutex errato da parte di P1 e P2

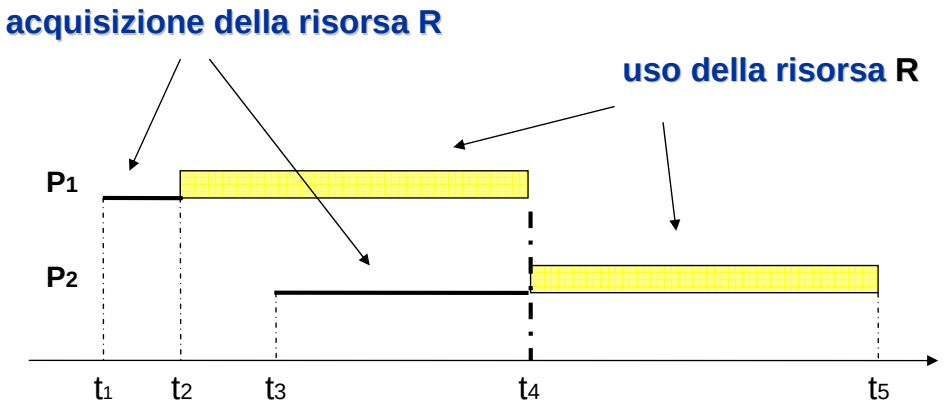


# Utilizzo di risorse mutex



$t_1$-$t_2$= tempo di attesa del processo P1 per accedere alla risorsa R

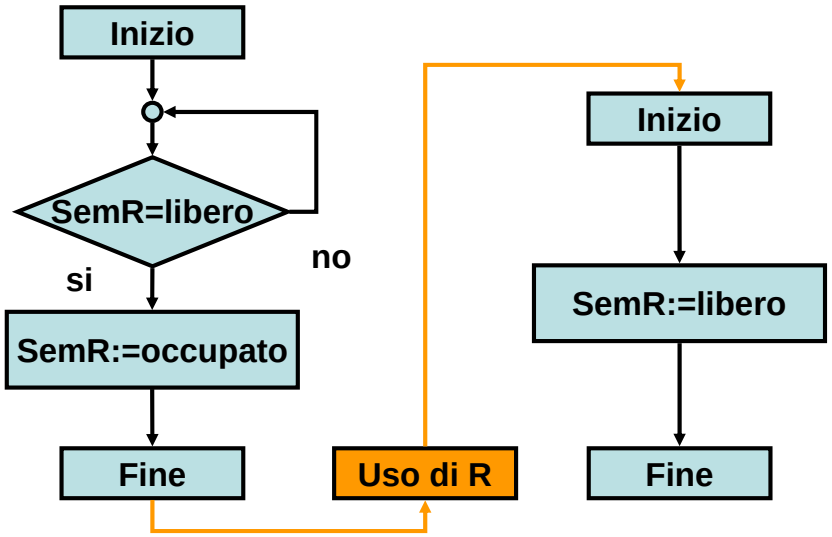
$t_3$-$t_4$= tempo di attesa del processo P2 per accedere alla risorsa R



# Acquisizione e rilascio di risorse mutex



- Controllo con **semaforo**: variabile **SemR** booleana




Sequenza di acquisizione (WAIT)

Sequenza di rilascio (SIGNAL)



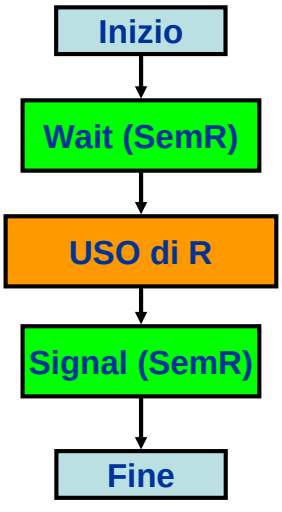


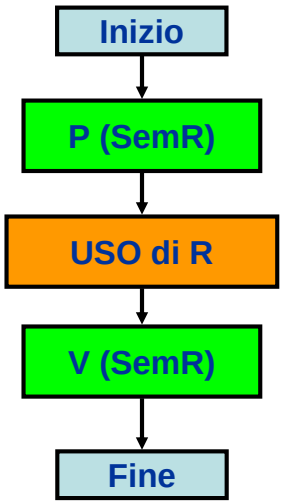
# Operazioni primitive su un semaforo



- wait (SemR)
- signal (SemR)

**Terminologia di Dijkstra**
- **P** (SemR) – Proberen
- **V** (SemR) – Verhogen

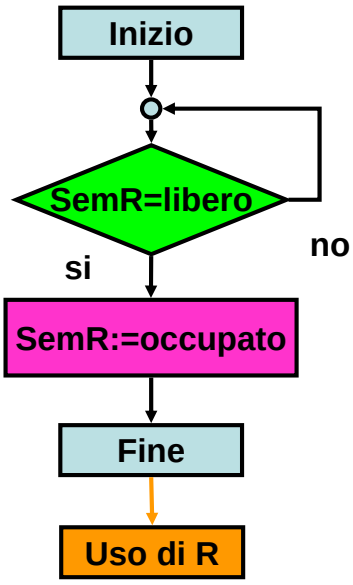


# Acquisizione di risorse mutex



**Sequenza di acquisizione**
**WAIT - accesso alla risorsa**




**Problema:**
**due operazioni per acquisire la risorsa:**

1. controllo dello stato (SemR = ?)
2. occupazione (SemR := occupato)



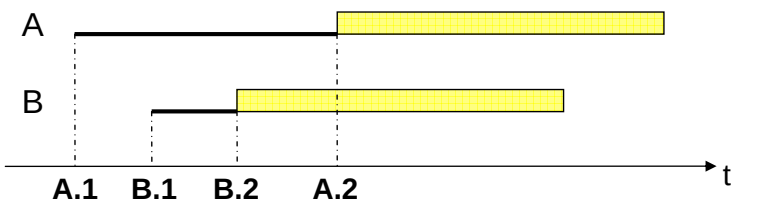

# Acquisizione di risorse mutex [2]



- Problema: due operazioni per acquisire la risorsa:
  1. controllo dello stato (SemR == ?)
  2. occupazione (SemR := occupato)

  Queste operazioni sono temporalmente separate: se i processi A e B cercano di acquisire la risorsa, si potrebbe verificare la sequenza:
- A.1 A controlla lo stato della risorsa e la trova libera
  ma, prima che A la occupi, …
- B.1 B controlla lo stato della risorsa e la trova libera
- B.2 B, avendo trovato la risorsa libera, la occupa
  ma, anche …
- A.2 A, avendo trovato la risorsa libera, la occupa

Uso errato della risorsa R



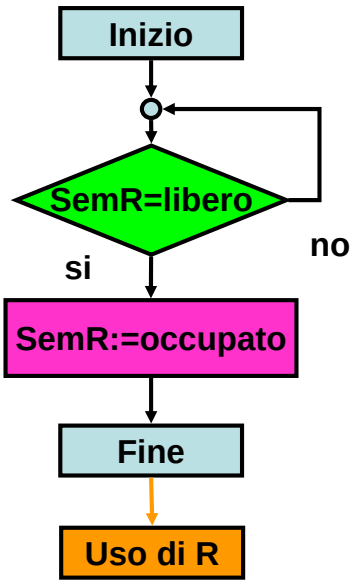
# Acquisizione di risorse mutex [3]



- Le due operazioni per acquisire la risorsa:
  1. controllo dello stato (SemR == ?)
  2. occupazione (SemR := occupato)

  vanno eseguite in modo indivisibile

Sequenza di acquisizione (WAIT)




La sequenza di istruzioni va eseguita
in mutua esclusione




# Acquisizione di risorse mutex [4]



- Per assicurare la mutua esclusione nell' uso della risorsa R è necessario assicurare che:
  - l'accesso ad R (l'uso di SemR) avvenga in mutua esclusione
- Si ripropone, a livello più basso, lo stesso problema:
- Per assicurare la mutua esclusione nell'uso di SemR è necessario assicurare che:
  - l'accesso a SemR avvenga in mutua esclusione
- E così via, all'infinito …

# Strategie di soluzione



- Conviene fermarsi al primo o secondo passo e rendere indivisibile (mutuamente esclusivo)
  - o direttamente l'uso della risorsa R
  - oppure l'accesso ad R (l'uso di SemR)

  cioè la sequenza di istruzioni che:
  1. controlla lo stato della risorsa (SemR = = ?)
  2. la occupa (SemR = occupato)

- Nota:
  l'operazione su SemR di solito dura molto meno dell'uso di R



# Strategie di soluzione [2]



- Definita Sezione Critica la sequenza di istruzioni con la quale un processo opera su una risorsa comune, il problema si risolve con:

| **Soluzione A** | oppure | **Soluzione B** |
|---|---|---|
| Uso indivisibile della risorsa | | Mutua esclusione delle sezioni critiche di accesso alla risorsa |

- Operazioni su variabili di stato della risorsa ***sono*** Sezioni critiche
- L'operazione di rilascio opera sulla stessa variabile di stato ("SemR = occupato") dell'operazione di accesso.

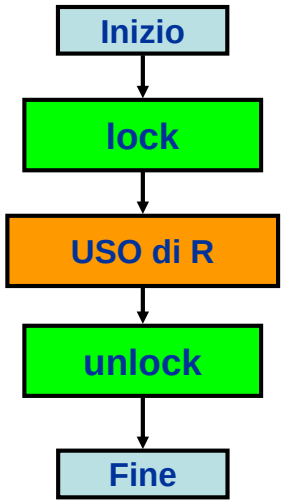
# Soluzione A - (Lock e Unlock)



- lock ( )
- unlock ( )

- Istruzioni (coppia di parentesi) che rendono indivisibile la sequenza di operazioni.

- Lock svolge le funzioni di wait unlock quelle di signal

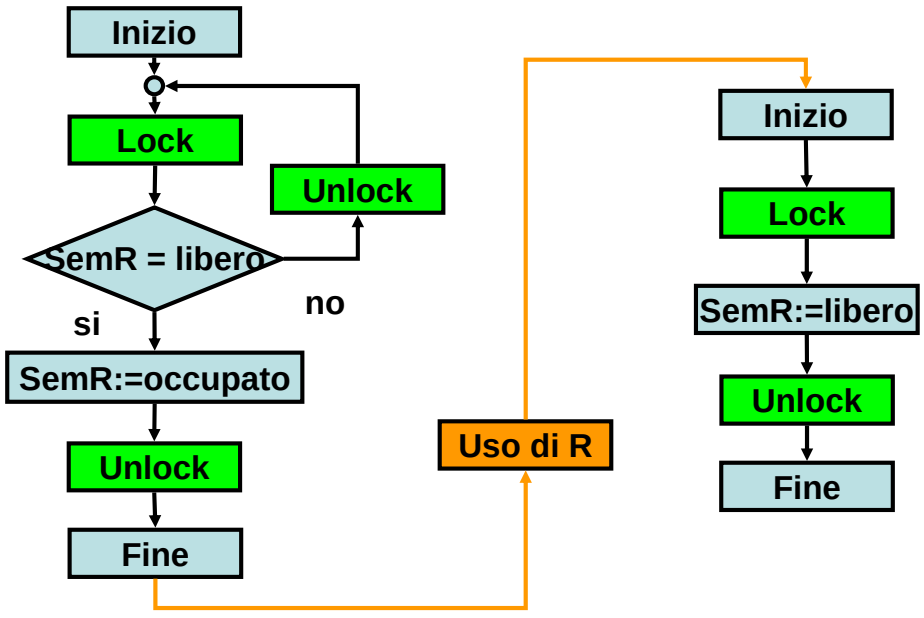





# Soluzione B - (Wait e Signal)



- Acquisizione (WAIT) Rilascio (SIGNAL)



# Lock e Unlock in sistemi Monoprocessore



- **Lock( )** ⟹ **disabilita le interruzioni**

ARM:
```
MRS r0, CPSR
ORR r0, r0, #0x80 ; bit 7 <- 1
                  ; 0x40 per FIQ, bit 6 <- 1
MSR CPSR_c, r0
```

- **Unlock( )** ⟹ **riabilita le interruzioni**

ARM:
```
MRS r0, CPSR
BIC r0, r0, #0x80
MSR CPSR_c, r0
```

- Va bene per la soluzione B (breve), non per la A (lunga)



# Lock e Unlock in sistemi Multiprocessore



**I processori devono avere una istruzione che:**

1. legge un dato da memoria,
2. lo esamina (bit di condizione),
3. lo riscrive in memoria.

IN MODO INDIVISIBILE

(CON UN UNICO CICLO DI MEMORIA)

# Lock e Unlock in sistemi Multiprocessore [2]



- Ad esempio: l'istruzione TEST AND SET (TAS)

68000:
```
TAS.B S.L ; S->Z e N, S=1 (ciclo READ-MODIFY-WRITE)
```

- L'istruzione ARM SWAP (SWP)

ARM:
```
SWP{<cond>} Rd, Rm, [Rn] ;[Rn]->tmp, Rm->[Rn], tmp->Rd
; se Rd==Rm avviene lo scambio tra Rd e [Rn]
; operazione atomica
```

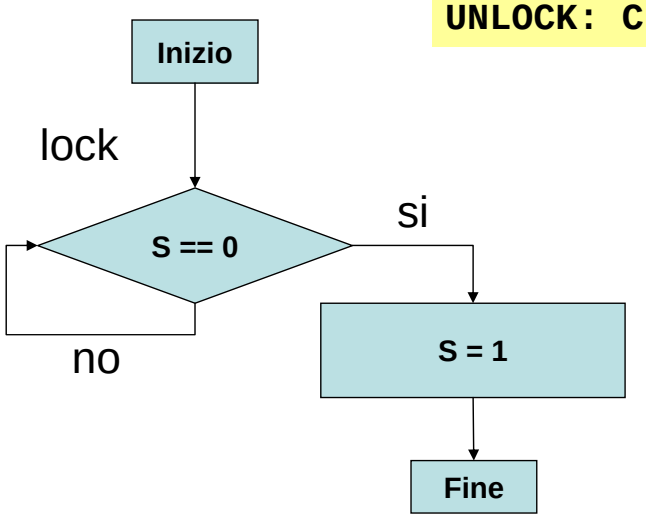


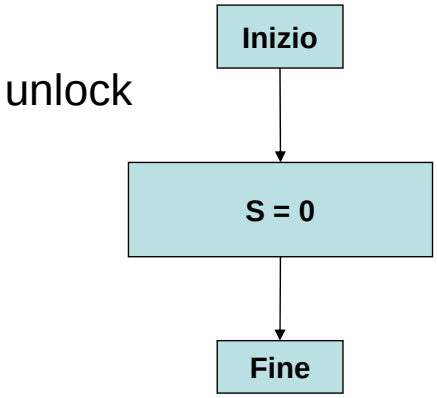
# Lock e Unlock - realizzazione con TAS



- Usando l'istruzione TAS si può costruire l'operazione di lock (istruz. M68000)

```
  LOCK: TAS.B S.L ;S->Z e N, S=1
        BNE LOCK
UNLOCK: CLR.B S
```

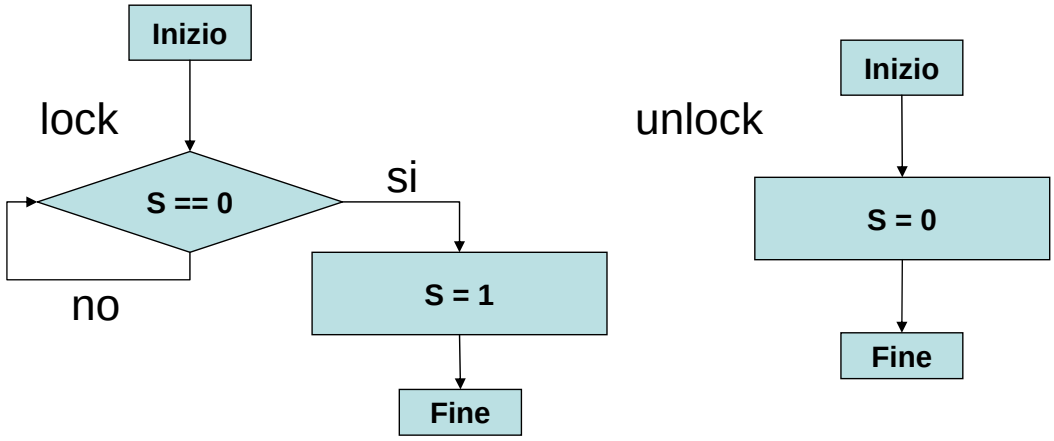


# Lock e Unlock - con TAS



- Esaminando i grafi, ci si accorge che:
  - la variabile di lock S svolge la funzione di un semaforo
  - l'operazione lock svolge la funzione della wait
  - l'operazione unlock svolge la funzione della signal





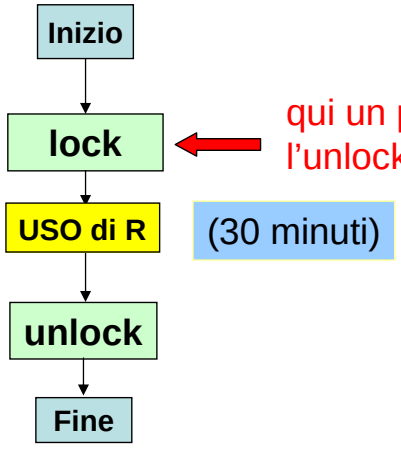


# Soluzione A - (Lock e Unlock)



- **lock ( )**
- **unlock ( )**

Istruzioni (coppia di parentesi) che rendono indivisibile la sequenza di operazioni interne.

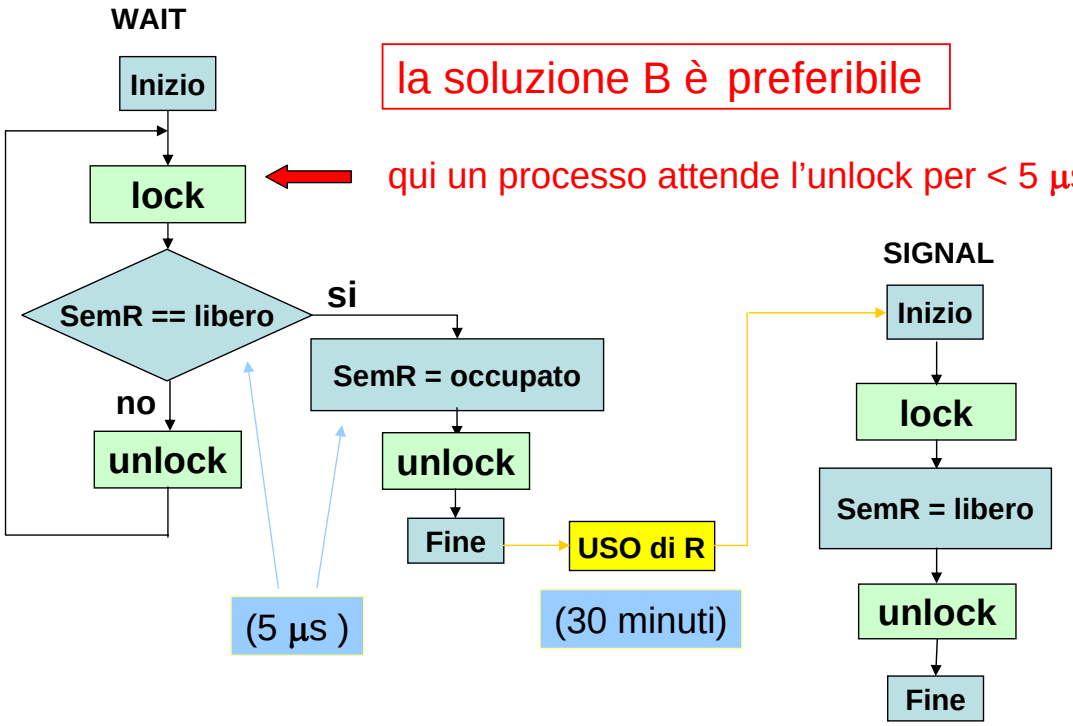



qui un processo potrebbe attendere l'unlock per 30 minuti!

(30 minuti)

# Soluzione B - (Wait e Signal)






la soluzione B è preferibile

qui un processo attende l'unlock per < 5 μs !



# Realizzazione di Lock e Unlock



in sistemi MONO-processore:

- Lock ⇒ disabilitazione delle interruzioni
- Unlock ⇒ riabilitazione delle interruzioni

in sistemi MULTI-processore:

- Lock ⇒ READ-MODIFY-WRITE (TAS, SWP)
- Unlock ⇒ reset della variabile

# PROBLEMA nella realizzazione di wait:



```
public void wait() {
        lock();
        for (;;) {
            if (semr){ //libero=true
                semr = false;
                unlock();
                break;
            }
            unlock();
            lock();
        }
    }
```

BUSY WAITING !

Anche lock (realizzata con TAS):

```
LOCK: TAS.B S.L ;S->Z e N, S=1
      BNE LOCK
```

BUSY WAITING !

```
public void lock() {
          while (tas) {}
    }
```

BUSY WAITING !

Ma l'attesa con lock è breve (5 µs)



# Wait - forma busy waiting (spinlock)

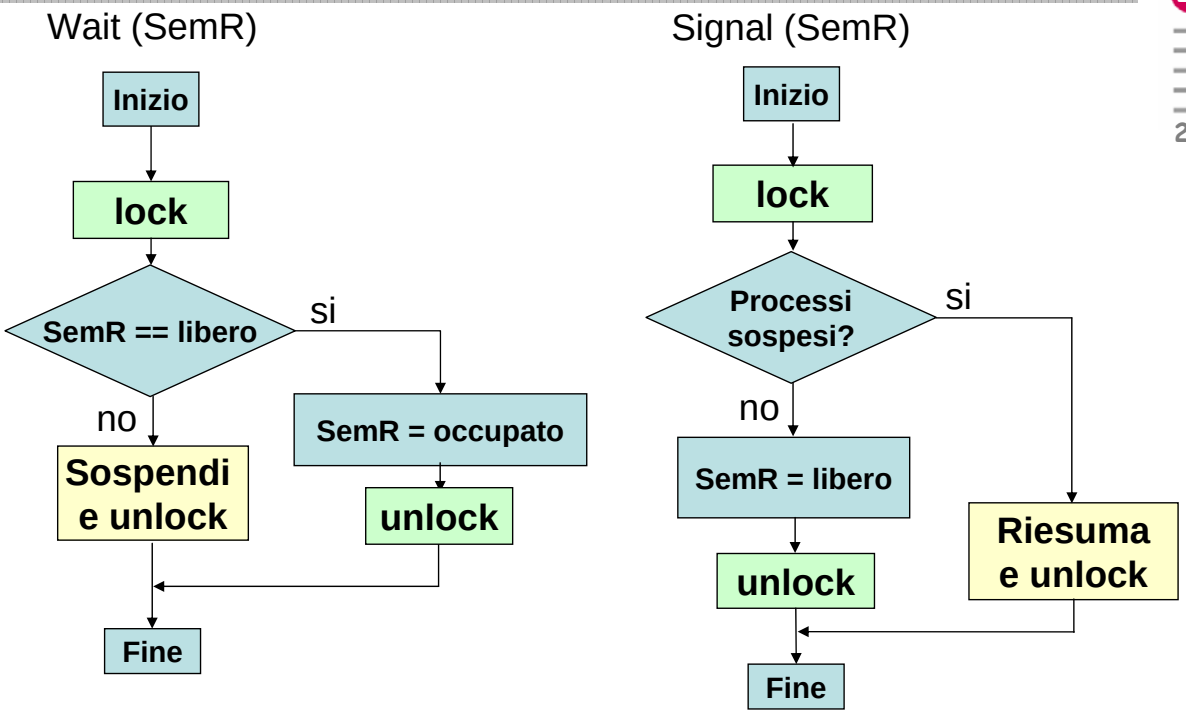


- Svantaggio:
  - non consente di usare in modo efficiente il processore
- Vantaggio:
  - non richiede commutazione di contesto
  - non appena la variabile di lock o il semaforo lo consentono, i processi ripartono subito, col minimo ritardo.
- Potrebbe essere conveniente:
  - in sistemi multi-processore,
  - quando i tempi di attesa si prevedono brevi.

Più conveniente: semaforo con sospensione dei processi.

# Semaforo con sospensione dei processi






Al semaforo va associata una CODA DI PROCESSI SOSPESI



# Semaforo (binario) in Java



```
class Semaphore {

    public Semaphore(boolean b);
            / / b valore iniziale

    public synchronized void p();
            / / operazione p indivisibile

    public synchronized void v();
            / / operazione v indivisibile

    public synchronized long p(long timeout);
            / / operazione p indivisibile con timeout ms

    public int value();
            / / valore del semaforo (0 o 1)

    public int queue();
            / / num. thread in attesa sulla coda del sem.

    public synchronized Thread waitingThread(int pos);
            / / thread in attesa in posizione pos

    public String toString();
            / / stringa descrittiva
}
```

# Mutua esclusione in Java



```
public void metodoConSezCritica(..., Semaphore s)
{
    . . .
    s.p();
    <codice della sezione critica>
    s.v();
    . . .
}

. . .
public void metodoRicorsivo (..., MutexSem s)
{
    . . .
    s.p();
    <codice della sezione critica>
    . . .
    metodoRicorsivo(..., s);
          / / il MutexSem non provoca deadlock
    . . .
    s.v();
    . . .
}
```



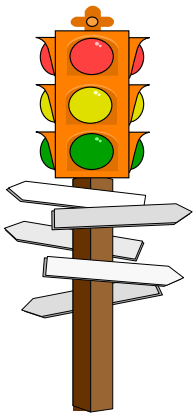
# Esercizi



1. Far in modo che un thread T1 invii un signal su un semaforo ogni volta che si preme RETURN sulla tastiera; sul semaforo si mette in attesa un thread T2 che, quando riceve il signal, visualizza il valore di una variabile di conteggio che si incrementa.
2. Creare un numero arbitrario di thread (numero passato come parametro sulla linea di comando) che, in mutua esclusione, visualizzano su stdout: il proprio nome, l'ora corrente, un contatore che indica se e' il primo che visualizza, oppure il secondo ecc. (cioè un contatore condiviso tra i vari thread e incrementato da ciascuno).
3. Adottare il modello del produttore/consumatore per far sì che un thread T1 legga una riga da stdin, e la invii ad un thread T2 attraverso una variabile stringa comune, e che T2 la visualizzi su stdout assieme alla sua lunghezza (numero di caratteri).

# Fine

04.a



## Semafori