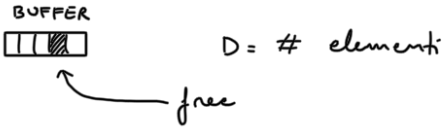
Processi Concorrenti Asincroni

sezione critica utilizza anche una variabile condivisa

PRODUTTORE

```
① buffer[free] = dato ;
② free = (free+1) % D;
```



Il buffer è condiviso, immaginiamo che venga condiviso tra due processi produttori P_1 e P_2. Senza modifiche al codice è possibile che celle libere vengano saltate e celle con dati vengano sovrascritte in caso di context switch

BUONE PROPRIETÀ delle soluzioni al problema della sezione critica

1. MUTUA ESCLUSIONE
2. PROGRESSO
3. ATTESA LIMITATA

SOLUZIONI < SOFTWARE / HARDWARE

SOLUZIONI SW : (per due processi)

```
1.      < sez. non critica >
sez. di ingresso || while ( turno != i) do NOP;   ← busy waiting (attesa attiva)
                    < sez. critica >
sez. di uscita || turno = j ;
```

Dove `turno` è una variabile condivisa inizializzata in maniera arbitraria. Questa soluzione prevede una STRETTA alternanza dei processi, viene violata la proprietà del PROGRESSO nel caso uno dei due processi termini due turni prima dell'altro

2.

sez. di ingresso
```
flag[i] = true;
while (flag[j]) do NOP;
```
<sezione critica>

sez. di uscita
```
flag[i] = false;
```

Dove `flag[i]` e `flag[j]` variabili condivise. Se entrambe le flag vengono assegnate a true, per via di context switch, l'esecuzione si blocca

3. ALGORITMO DI PETERSON

sez. di ingresso
```
flag[i] = true;
turno = j;
while (flag[j] && turno = j) do NOP;
```
<sezione critica>

sez. di uscita
```
flag[i] = false;
```

Garantisce tutte e tre le proprietà per due processi

SOLUZIONI HW:

1. Disabilitare gli interrupt durante la sezione di ingresso

Variabile condivisa LOCK

Vengono implementate le istruzioni Test & Set e Swap che vengono eseguite a interrupt disabilitati

Test & Set

```
TestAndSet ( boolean *var)
{
    boolean valore = *var;
    *var = true;
    return valore;
}
```

il codice viene eseguito in maniera atomica

Il processo quindi esegue:

```
while (TestAndSet (&lock)) do NOP;
< sezione critica >
lock = false;
```

Swap

```
Swap ( boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Il processo esegue:

```
chiuso = true;
while (chiuso) Swap (&lock, &chiuso);
< sezione critica >
lock = false;
```

SEMAFORI

inventati da Dijkstra

- P (PROBEREN = VERIFICARE) (down / wait)
- V (VERHOGEN = INCREMENTARE) (up / signal)

Implementazione: S variabile intera condivisa con 0 come valore bloccante

eseguiti in maniera atomica

```
P(s) {
    | while (s ≤ 0) NOP;
    | s--;
    }

V(s) {
    | s++;
    }
```

```
P(s) {
  s->valore--;
  if(s->valore<0)
    block;
  }
```

In questo modo |s| è il numero di processi in attesa

```
V(s) {
  s->valore++;
  if(s->valore<0)
    wakeup(s);
  }
```

BUFFER MUTEX 1

MUTEX deve essere inizializzato a 1 nel codice

Processo :

```
P(MUTEX)
<sezione critica>
V(MUTEX)
```

SEMAFORI
- MUTUA ESCLUSIONE inizializzato a 1
- CONTATORI inizializzato a n con n numero di risorse libere

Posso inizializzare un semaforo a 0 per sincronizzare l'esecuzione di due processi paralleli. Il processo che aspetta chiama P(s) e aspetta che gli venga dato il via libera dall'altro processo tramite un V(s)

ATTESA DELLO 0

Serve a sincronizzare un certo numero di processi inizializzando il semaforo al numero n di processi. Ogni processo fa P(s) fino a quando non si arriva a 0 e quello che sta facendo l'attesa dello 0 si risveglia

errori tipici dell'uso di semafori:

- P(s) → s 0
  P(s) → s 0 DEADLOCK
  <sezione critica>
  V(s)

- V(s) → s 2
  <sezione critica> sezioni critiche eseguite in parallelo
  P(s)

- Risorsa 1 S_1
  Risorsa 2 S_2

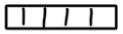
$P(S_1)$	$P(S_2)$
$P(S_2)$	$P(S_1)$
<sezione critica>	<sezione critica>
$V(S_1)$	$V(S_1)$
$V(S_2)$	$V(S_2)$

DEADLOCK che può emergere quando avviene context switch tra due processi con P(s) in ordine diverso, entrambi i programmi si trovano l'altra risorsa occupata

RISOLUZIONE DI PROBLEMI COI SEMAFORI

① PRODUTTORI - CONSUMATORI
② LETTORI - SCRITTORI
③ 5 FILOSOFI

1) BUFFER LIMITATO



P_j produttore sospeso se buffer pieno — deve inserire dati nelle celle
C_j consumatore sospeso se buffer vuoto — estrae dati dalle celle

1x SEMAFORO MUTEX
2X SEMAFORO CONTATORE — LIBERE / OCCUPATE

per il produttore sono risorse le celle libere, per il consumatore lo sono le celle occupate

```
SEM_MUTEX = 1;
SEM_LIBERE = N_CELLE;
SEM_OCCUPATE = 0;
```

PRODUTTORE

```
forever {
    < produci elemento >
    P(LIBERE)
    P(MUTEX)
    < sezione critica >
    V(MUTEX)
    V(OCCUPATE)
}
```

CONSUMATORE

```
forever {
    P(MUTEX)
    < sezione critica : estrai elemento >
    V(MUTEX)
```

2)
- non ci sono limiti di spazio
- i lettori non consumano

MUTUA ESCLUSIONE solo agli scrittori
I lettori non modificano i dati

```
SEM_MUTEX = 1;
SEM_SCRITTURA = 1;
int numLettori = 0;
```
variabile condivisa

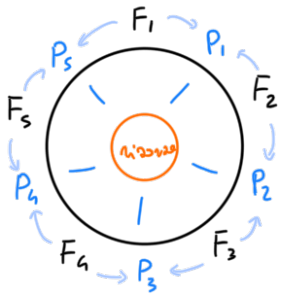
SCRITTORE

```
forever {
    P(SCRITTURA)
    <sezione critica>
    V(SCRITTURA)
}
```

LETTORE

```
forever {
    P(MUTEX)                    // sez. ingresso
    numLettori++;
    if(numLettori==1)
        P(SCRITTURA)
    V(MUTEX)
    <legge>
    P(MUTEX)                    // sez. uscita
    numLettori--;
    if(numLettori==0)
        V(SCRITTURA)
    V(MUTEX)
}
```

3)



ogni filosofo ha bisogno di 2 posate

Fi filosofi che hanno bisogno di 2 posate per mangiare ma abbiamo solo 5 posate

```
Fi  forever {
        <pensa>      non ha bisogno di risorse
        <mangia>     richiede risorse
    }
```

Possibilità di
- DEADLOCK — tutti i filosofi (processi) fermi
- STARVATION
- LIVELOCK — filosofi in loop continuo, deadlock ma con processo non bloccato

ASSOCIARE STATO ALLE POSATE: SPORCA / PULITA

- Se il filosofo ha una posata sporca e un processo fa richiesta glielo passa. Dopo aver utilizzato la risorsa la posata diventa pulita