

PCAD

Programmazione Concorrente

Algoritmi Distribuiti

Arnaud Sangnier
arnaud.sangnier@unige.it

SEMAFORI

Semafori

- Introdotti da Dijkstra nel 1965
- Oggetti di alto livello per modellare delle risorse

Idea generale

- Abbiamo un numero (conosciuto di risorse)
- I processi che ne hanno bisogno, possono chiederle
- Se non ce ne sono più, i richiedenti sono **mesi in attesa**
- Ci sono diversi modi per fare aspettare i processi (secondo il tipo di semafori)
- Un semaforo può essere intero o boolean (binario)

Tipi di semafori

Faremo la distinzione fra tre tipi di semafori

- 1) I **semafori deboli** (weak): i processi in attesa sono bloccati, e quando la risorsa è disponibile, **UNO** processo in attesa è svegliato
- 2) I **semafori forti** (strong): i processi in attesa sono bloccati e inseriti in una struttura FIFO, e quando la risorsa è disponibile, **IL** processo in attesa da più tempo è svegliato
- 3) I **semafori busy-wait**: i processi rimangono attivi.... e testano in continuo se la risorsa è disponibile

Semafori deboli/forti

Un semaforo S è composto da due campi:

- 1) **S.V**: un intero (o boolean) che rappresenta il valore di S
- 2) **S.L**: la liste dei processi in attesa

Per i semafori forti:

S.L è una struttura FIFO (anche solo giusta (fair), per la quale si può assumere un limite sul numero di volte in cui un processo in attesa può essere 'superato' da altri)

Per i semafori deboli:

Non ci sono ipotesi di giustizia su S.L. Ma sappiamo che sarà sempre un **processo bloccato** che avrà la risorsa

Operazione sui semafori


Ci sono due operazione **atomiche** su un semaforo S:

- 1) S.wait() -> per chiedere una risorsa
- 2) S. signal() -> per rilasciare una risorsa

L'operazione wait

```
S.wait(){  
  if (S.V>0){S.V--}  
  else{  
    S.L=S.L U {P}  
    P.stato=bloccato  
  }  
}
```

Si assume che wait
è chiamata da il
processo P



**NB1: gli stati possibili di un processo sono inattivo,
pronto, attivo, finito, bloccato**

L'operazione signal

```
S.signal(){  
  if (S.L=Ø){S.V++}  
  else{  
    P'=S.L.estrarre()  
    P'.stato=pronto  
  }  
}
```

NB1: gli stati possibili di un processo sono inattivo, pronto, attivo, finito, bloccato

NB2: per i semafori binari, il valore è bloccato ad 1, fare V++ se vale 1 non fa nulla

Per i semafori busy/wait

```
S.wait(){  
  while(S.V==0){}  
  S.V--  
}
```

```
S.signal(){  
  S.V++  
}
```

**Warning : abbiamo detto che wait et signal sono atomiche,
ma in questo caso dobbiamo assumere che durante il
while, il processo può essere interrotto**

Proprietà dei semafori

Lemma:

Sia k il valore iniziale di $S.V$ all'inizializzazione. Abbiamo sempre (se si usa solo `wait` e `signal` per accedere al semaforo):

1) $S.V \geq 0$

2) $S.V = k + \#S.signal() - \#S.wait()$

$\#S.signal()$ è il numero di chiamate a `signal()` fatte fino ad ora

$\#S.wait()$ è il numero di chiamate a `wait()` fatte fino ad ora

Prova:

- Si dimostra che 1) e 2) sono degli invarianti

Problema della sezione critica

Con i semafori

S: semaforo binario con $S.V=1$

```
Process P1
while(true){
p1: SNC
p2: S.wait();
p3: SC
p4: S.signal();}
```

```
Process P2
while(true){
q1: SNC
q2: S.wait();
q3: SC
q4: S.signal();}
```

- Mutua esclusione ✓
- Assenza di deadlock ✓
- Assenza di starvation ✓
- Attesa limitata ✓

Queste proprietà sono vere che il semaforo sia debole o forte

Problema della sezione critica

Con i semafori e N processi

S: semaforo binario con S.V=1

```
Process P[1]/P[2]/.../P[N]
while(true){
  p1: SNC
  p2: S.wait();
  p3: SC
  p4: S.signal();}
```

- Mutua esclusione ✓
- Assenza di deadlock ✓
- Assenza di starvation ✗/✓
- Attesa limitata ✗/✓

Queste proprietà dipendono se il
semaforo è debole o forte

Esempi di uso dei semafori

Il problema di precedenza

Una task T deve essere realizzata prima di T'

- S: semaforo binario inizializzato a 0
- T finisce facendo S.signal()
- T' inizia facendo S.wait()

Il problema di produttori/consumatori

Comunicazione asincrona:

- Risorse sono prodotte e messe in un buffer
- Sono consumate da altri processi (se disponibile)

Il buffer può essere infinito o finito

Produttori/Consumatori

Buffer infinito

F: struttura FIFO = \emptyset // variabile condivise
NonVuota : semaforo con S.V=0

```
Process Produttore:  
while(true){  
  p1: d=product();  
  p2: F.add(d);  
  p3: NonVuota.signal();}
```

```
Process Consumatore  
while(true){  
  q1: NonVuota.wait();  
  q2: d=F.get();  
  q3: consume(d);}
```

Invariante : $|F| = \text{NonVuota.V}$

=> Il consumatore non puo prendere una risorsa non disponibile

Produttori/Consumatori

Buffer finito (lunghezza N)

F: struttura FIFO = \emptyset // variabile condivise

NonVuota : semaforo con S.V=0

NonPiena : semaforo con S.V=N

Posti occupati

Posti liberi

Process Produttore:

```
while(true){  
  p1: d=product();  
  p2: NonPiena.wait();  
  p3: F.add(d);  
  p4: NonVuota.signal();}
```

Process Consumatore

```
while(true){  
  q1: NonVuota.wait();  
  q2: d=F.get();  
  q3: NonPiena.signal();  
  q4: consume(d);}
```

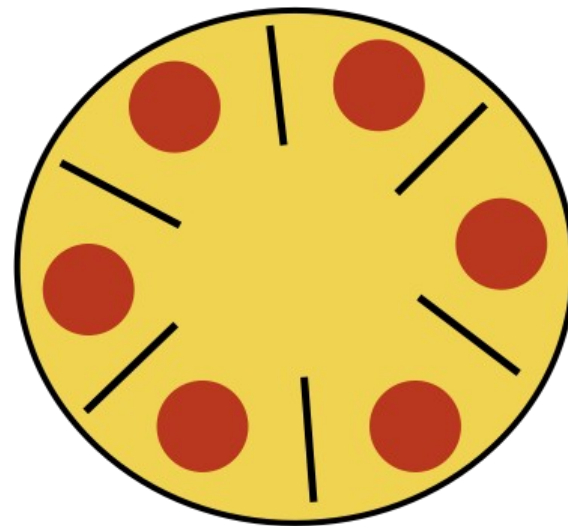
Invariante : $N = \text{NonVuota.V} + \text{NonPiena.V}$

I filosofi

- N filosofi
- N bacchette
- Per mangiare, un filosofo deve avere la sua bacchetta di destra e di sinistra

Schema generale :

```
Process Filosofo:  
while(true){  
  p1: pensa;  
  p2: pre-protocollo  
  p3: mangia  
  p4: post-protocollo}
```

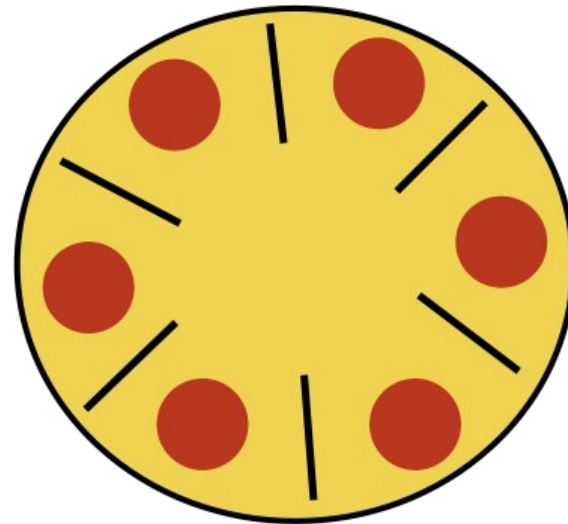


**Mangia finisce sempre, pensa
non necessariamente!**

I filosofi

Proprietà attese

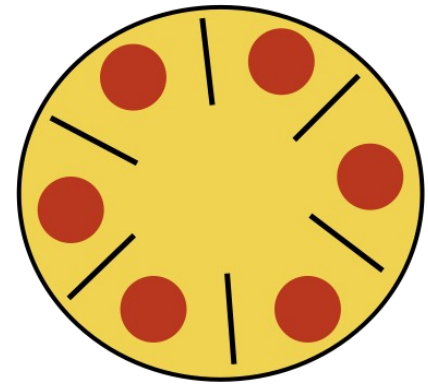
- Un filosofo può mangiare solo con le bacchette alla sua destra e sinistra
- Un bacchetta è presa dal al massimo un filosofo
- Assenza di deadlock
- Assenza di starvation



I filosofi

Valore di V per gli N semafori

Semaforo bacchette[N]={1,...1}; // variabile condivise

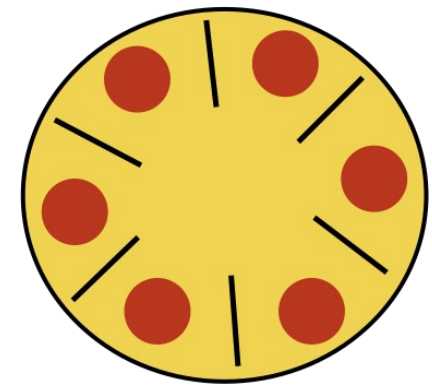


```
Process Filosofo id: //id ∈ {0,...,N-1}
while(true){
  pensa
  bacchette[i].wait()
  bacchette[(i+1)%N].wait()
  mangia
  bacchette[i].signal()
  bacchette[(i+1)%N].signal()
}
```

Proprietà attese

- Un filosofo può mangiare solo
- con le bacchette alla sua destra e sinistra ✓
- Un bacchetta è presa dal
- al massimo un filosofo ✓
- Assenza di deadlock ✗
- Assenza di starvation ✗

I filosofi



Soluzione possibile: limitare il numero di filosofi che possono fare il pre-protocollo insieme

```
Semaforo bacchette[N]={1,...,1}; // variabile condivise  
Semaforo ticket=M; //M<N
```

```
Process Filosofo id: //id ∈ {0,...,N-1}  
while(true){  
  pensa  
  ticket.wait()  
  bacchette[i].wait()  
  bacchette[(i+1)%N].wait()  
  mangia  
  bacchette[i].signal()  
  bacchette[(i+1)%N].signal()  
  ticket.signal()  
}
```

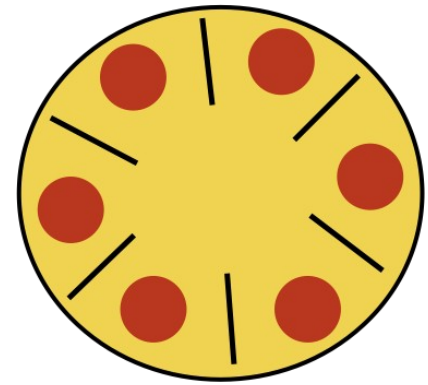
Proprietà attese

- Un filosofo può mangiare solo
- con le bacchette alla sua destra e sinistra ✓
- Un bacchetta è presa dal
- al massimo un filosofo ✓
- Assenza di deadlock ✓
- Assenza di starvation ✓

Vero se ticket è strong o se ticket è weak e $M=N-1$



I filosofi



Soluzione simetrica

Semaforo bacchette[N]={1,...,1}; // variabile condivise

```
Process Filosofo id: //id ∈ {1,...,N-1}
while(true){
  pensa
  bacchette[i].wait()
  bacchette[(i+1)%N].wait()
  mangia
  bacchette[i].signal()
  bacchette[(i+1)%N].signal()
}
```

```
Process Filosofo 0:
while(true){
  pensa
  bacchette[1].wait()
  bacchette[0].wait()
  mangia
  bacchette[1].signal()
  bacchette[0].signal()
}
```

Semafori in C

- `include <semaphore.h>;`

- Dichiarazione:

```
sem_t sem;
```

- Inizializzazione:

```
int sem_init(sem_t *sem, int pshared, unsigned int  
value); // pshared to 0 for semaphore shared between  
thread, value is the initial value
```

- Operazione:

```
int sem_wait(sem_t *sem)  
int sem_post(sem_t *sem) // signal
```