

# **PCAD**

## **Programmazione Concorrente**

### **Algoritmi Distribuiti**

**Arnaud Sangnier**  
arnaud.sangnier@unige.it

**MUTUA ESCLUSIONE ANCORA**

# Storia

- Algoritmo del fornaio (*bakery algorithm*): algoritmo di mutua esclusione per  $N \geq 2$  processi
- Inventato da Leslie Lamport in 1974
- Usa delle variabile **proprie** (multiple readers/single writer)
  - ogni processo ha le sue variabile che possono essere lette ma non modificate dagli altri processsi
- È l'algoritmo usato nell amministrazione, o dai banchi:
  - un nuovo 'cliente' che vuole avere accesso alle risorse prende un numero superiore ai numeri dei clienti già in attesa
  - il cliente prioritario è quello che ha il numero più piccolo

# Problema della sezione critica

## Algoritmo del fornaio

Processi  $P[0] \dots P[N-1]$

`bool choosing[N]={ false,...,false}; //variabile condivise`

`int nb[N]={ 0,...,0};`

```
Process P[id]
while(true){
p1: SNC
p2: choosing[id]=true;
p3: nb[id]=Max(nb[0],...,nb[n])+1;
p4: choosing[id]=false;
p5: for (int j=0; j<N; j++){
p6:   if(j!=id){
p7:     while(choosing[j]){ }
p8:     while(nb[j]!=0 && (nb[j],j) << (nb[id],id)){ } }
p9: SC
p10: nb[id]=0
```

**NB: il calcolo del  
Max non è atomico**



**Nuova relazione totale di ordinamento:**

$(nb[j],j) << (nb[id],id)$  sse  $((nb[j]<nb[id])$  o  
 $(nb[j]==nb[id]$  e  $(j<id)$ )

# Problema della sezione critica

## Algoritmo del fornaio - Proprietà

### Teorema:

L'algoritmo del fornaio verifica la mutua esclusione.

### Prova:

- Scegliamo un processo  $P[i]$  che arriva in SC ad un istante  $t_2$
- Dobbiamo fare vedere che a questo istante, nessun altro processo  $P[k]$  (con  $k \neq i$ ) è in SC
- Scegliamo un  $k \neq i$  e vediamo dove è  $P[k]$  in  $t_2$
- Usiamo due altri istanti:
  - $t_0$ :  $P[i]$  passa il  
`while(choosing[k]){}`
  - $t_1$ :  $P[i]$  passa il  
`while(nb[k] != 0 && (nb[k],k) << (nb[i],i)){}`

```
Process P[id]
while(true){
p1: SNC
p2: choosing[id]=true;
p3: nb[id]=Max(nb[0],...,nb[n])+1;
p4: choosing[id]=false;
p5: for (int j=0; j<N; j++){
p6:   if(j!=id){
p7:     while(choosing[j]){}
p8:     while(nb[j] != 0 && (nb[j],j) << (nb[id],id)){} }
p9: SC
p10: nb[id]=0
```

# Problema della sezione critica

## Algoritmo del fornaio - Proprietà

- Dove può essere  $P[k]$  a  $t_0$  ?
  - in **p3,p4** ? no perché a  $t_0$ ,  $P[i]$  passa `while(choosing[k]){} quindi`  
`choosing[k]=false`
  - in **p1,p2** ? quindi quando  $P[k]$  eseguirà il suo preprotocollo, non potrà essere in SC prima che  $P[i]$  ne esca

```
Process P[id]
while(true){
p1: SNC
p2: choosing[id]=true;
p3: nb[id]=Max(nb[0],...,nb[n])+1;
p4: choosing[id]=false;
p5: for (int j=0; j<N; j++){
p6:   if(j!=id){
p7:     while(choosing[j]){}
p8:     while(nb[j]!=0 && (nb[j,j] << (nb[id],id))){} }
p9: SC
p10: nb[id]=0
```

# Problema della sezione critica

## Algoritmo del fornaio - Proprietà

- Dove può essere  $P[k]$  a  $t_0$  ?
  - in **p5,p6,p7,p8,p9,p10** ? ma allora  $nb[k]$  è fisso e abbiamo due casi

1)  $(nb[k] < k) < (nb[i], i)$

- allora  $nb[k]$  deve essere messo a 0 prima di  $t_1$
- $P[k]$  passerà quindi in p1,p2 (cf prima)

2)  $(nb[i] < i) < (nb[k], k)$

- il calcolo di  $nb[i]$  inizia prima della fine di quello di  $nb[k]$
- con `while(choosing[i]){} P[k]` dovrà aspettare che  $nb[i]$  sia calcolato e verrà bloccato in p8

**$P[k]$  non può essere in SC in  $t_2$**

```
Process P[id]
while(true){
p1: SNC
p2: choosing[id]=true;
p3: nb[id]=Max(nb[0],...,nb[n])+1;
p4: choosing[id]=false;
p5: for (int j=0; j<N; j++){
p6:   if(j!=id){
p7:     while(choosing[j]){}
p8:     while(nb[j]!=0 && (nb[j],j) < (nb[id],id)){} }
p9: sc
p10: nb[id]=0
```

# Problema della sezione critica

## Algoritmo del fornaio - Proprietà

### Teorema:

L'algoritmo del fornaio verifica l'assenza di deadlock

### Prova:

- Se tanti processi si bloccano, si ritrovano tutti bloccati in p8
- Ma come l'ordine  $<<$  è totale (ogni processo ha un numero diverso), c'è per forza un  $i$  tale quale  $(nb[i], i)$  è l'elemento più piccolo e lui passerà  
=> quindi **non ci sarà un deadlock**

```
Process P[id]
while(true){
p1: SNC
p2: choosing[id]=true;
p3: nb[id]=Max(nb[0],...,nb[n])+1;
p4: choosing[id]=false;
p5: for (int j=0; j<N; j++){
p6:   if(j!=id){
p7:     while(choosing[j]){ }
p8:     while(nb[j]!=0 && (nb[j],j) << (nb[id],id)){ } }
p9: SC
p10: nb[id]=0
```

# Problema della sezione critica

## Algoritmo del fornaio - Proprietà

### Teorema:

L'algoritmo del fornaio verifica l'assenza di starvation sotto gli ipotesi:

- equità fra i processi (se un processo può eseguire una istruzione, la eseguirà un giorno)
- la SC termina sempre e finisce con l'esecuzione del post-protocollo

### Prova:

- Assumiamo che  $P[i]$  rimane bloccato nel suo pre-protocollo
- È per forza bloccato in p8 per un  $j$ , i.e.  
 $nb[j] > 0 \ \&\& \ (nb[j], j) << (nb[i], i)$
- $P[j]$  finirà in SC (non c'è deadlock) e dopo:
  - $P[j]$  rimarrà in SC, e  $nb[j] = 0$
  - $P[j]$  rifarà il suo pre-protocollo e il calcolo di  $nb[j]$  prenderà in conto il valore di  $nb[i]$  e  $P[i]$  passerà

```
Process P[id]
while(true){
p1: SNC
p2: choosing[id]=true;
p3: nb[id]=Max(nb[0],...,nb[n])+1;
p4: choosing[id]=false;
p5: for (int j=0; j<N; j++){
p6:   if(j!=id){
p7:     while(choosing[j]){ }
p8:     while(nb[j]!=0 && (nb[j],j) << (nb[id],id)){ } }
p9:   SC
p10: nb[id]=0
```



# Problema della sezione critica

## Algoritmo del fornaio - Proprietà

### Teorema:

L'algoritmo del fornaio verifica l'assenza di starvation sotto gli ipotesi:

- equità fra i processi (se un processo può eseguire una istruzione, la eseguirà un giorno)
- la SC termina sempre e finisce con l'esecuzione del post-protocollo

### Prova:

- Assumiamo che  $P[i]$  rimane bloccato nel suo pre-protocollo
- È per forza bloccato in p8 per un  $j$ , i.e.  
 $nb[j]>0 \ \&\& \ (nb[j],j)<<(nb[i],i)$
- **$P[j]$  finirà in SC (non c'è deadlock)** e dopo:
  - $P[j]$  rimarrà in SC, e  $nb[j]=0$
  - $P[j]$  rifarà il suo pre-protocollo e il calcolo di  $nb[j]$  prenderà in conto il valore di  $nb[i]$  e  $P[i]$  passerà

In realtà, bisogna ad iterare questo passo

```
Process P[id]
while(true){
  p1: SNC
  p2: choosing[id]=true;
  p3: nb[id]=Max(nb[0],...,nb[n])+1;
  p4: choosing[id]=false;
  p5: for (int j=0; j<N; j++){
  p6:   if(j!=id){
  p7:     while(choosing[j]){ }
  p8:     while(nb[j]!=0 && (nb[j],j) << (nb[id],id)){ } }
  p9: SC
  p10: nb[id]=0
```

# Problema della sezione critica

## Algoritmo del fornaio - Proprietà

### Teorema:

Nel l'algoritmo del fornaio i valori nel nb possono crescere all'infinito (anche con solo due processi)!

```
Process P[id]
while(true){
p1: SNC
p2: choosing[id]=true;
p3: nb[id]=Max(nb[0],...,nb[n])+1;
p4: choosing[id]=false;
p5: for (int j=0; j<N; j++){
p6:   if(j!=id){
p7:     while(choosing[j]){ }
p8:     while(nb[j]!=0 &&& (nb[j],j) << (nb[id],id)){ } }
p9: SC
p10: nb[id]=0
```

**Altri metodi basati su 'tecnologie' più avanzate che variabile condivise**

# Test-and-set

- Un test-and-set è una variabile boolean con due operazione atomiche:
  - 1) **test-and-set**: legge il valore della variabile, la mette a true, e ritorna il valore letto
  - 2) **reset**: mette false nella variabile

Algoritmo per mutua esclusione

**test-and-set x=false; //variabile condivisa**

```
Process P
while(true){
p1: SNC
p2: while(x.test-and-set()==true){ }
p5: SC
p6: x.reset() }
```

# Test-and-set

- Un test-and-set è una variabile boolean con due operazione atomiche:
  - 1) **test-and-set**: legge il valore della variabile, la mette a true, e ritorna il valore letto
  - 2) **reset**: mette false nella variabile

Algoritmo per mutua esclusione

**test-and-set x=false; //variabile condivisa**

```
Process P
while(true){
p1: SNC
p2: while(x.test-and-set()==true){}
p5: SC
p6: x.reset()}
```

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

# 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

**NB2:** 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**