PCAD
Programmazione Concorrente
Algoritmi Distribuiti

Arnaud Sangnier
arnaud.sangnier@unige.it

Mutua esclusione per più processi

Problema della sezione critica

The Filter Lock

Processi P[0]...P[N-1]

int level[N]={0,...,0}; //variabili condivise

int victim[N]=?

```
Process P[id]
while(true){
p1: SNC
p2:for (int j=1; j<N; j++){
p3:   level[id]=j
p4:   victim[j]=id
p5:   while( ∃k≠id.level[k]≥ j && victim[j]==id){}
      }
p6:SC
p7:level[id]=0;}
```

Un processo può andare avanti se non è la vittima o se non c'è nessuno ad un livello superiore

Problema della sezione critica

Filter Lock - Proprietà

Definizione:

- Un processo ha passato il livello k se ha passato p5 per j=k (o se k=0)
- $num_{\geq k}$: numero di processi chi ha passato il livello k

```
Process P[id]
while(true){
p1: SNC
p2:for (int j=1; j<N; j++){
p3:   level[id]=j
p4:   victim[j]=id
p5:   while( ∃k≠id.level[k]≥ j && victim[j]==id){}
      }
p6:SC
p7:level[id]=0;}
```

Problema della sezione critica

Filter Lock - Proprietà

Proprietà:
Abbiamo $num_{\geq j} \leq n\text{-}j$ for all $0 \leq j \leq n\text{-}1$.

Prova (per induzione) :

- $j=0$: si abbiamo $num_{\geq 0} \leq n$
- $j>0$:
  - per ipotesi di induzione, sappiamo che al masssimo n-j+1 processi hanno passato il livello j-1
  - Supponiamo che tutti hanno passato il livello j
  - Sia M, l'ultimo ad avere fatto victim[j]=M
  - Tutti gli altri M', hanno fatto prima
    - level[M']=j
    - victim[j]=M'
  - **Non è posssibile che M abbia passato il while!!**

```
Process P[id]
while(true){
p1: SNC
p2:for (int j=1; j<N; j++){
p3:   level[id]=j
p4:   victim[j]=id
p5:   while( ∃k≠id.level[k]≥ j && victim[j]==id){}
    }
p6:SC
p7:level[id]=0;}
```

Problema della sezione critica

Filter Lock - Proprietà

Proprietà:
Abbiamo $num_{\geq j} \leq n-j$ for all $0 \leq j \leq n-1$.

Corollario:
$num_{\geq(n-1)} \leq 1$

Teorema:
L'algoritmo di Filter Lock verifica la mutua esclusione.

Problema della sezione critica

Filter Lock - Proprietà

Teorema:
L'algoritmo di Filter Lock 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

Proprietà:
Ogni processo al livello $j \geq 1$ arriverà al livello $j+1$ o in SC se $j=N-1$.

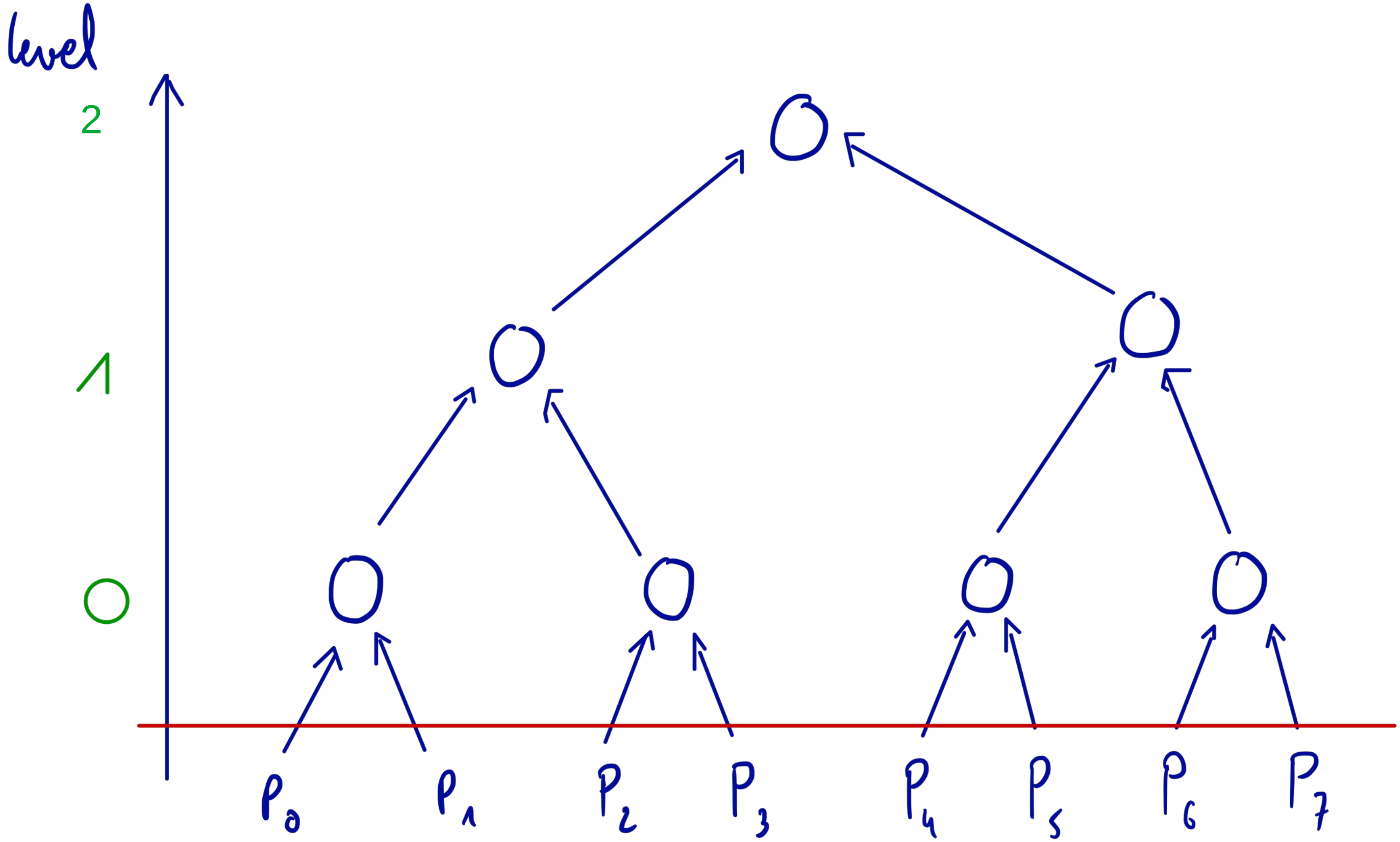
Prova (per induzione rovesciata):

- j=N-1 -> OK, come Peterson, abbiamo al massimo due processi
- j < N-1: Sia M al livello j bloccato nel while
  - Per ipotesi, gli processi dei livelli superiori arrivano in CS
  - Nessun processo del livello j-1 può arrivare al livello j (altrimenti victim cambia e M è sbloccato). Quindi victim[j] non cambia.
  - M è dunque l'unico ad essere bloccato al livello j e dopo il 'successo' degli altri verrà per forza sbloccato.

Il torneo

- Abbiamo $n=2^k$ processi : P0,P1,...Pn
- Facciamo un torneo con eliminazione in log(n)=k giri
- Dopo il primo giro, sono ‘eliminati’ n/2 processi
  - Ogni competizione fra due processi è gestito da un algoritmo di mutua esclusione per due processi
- Dopo il secondo giro, sono ‘eliminati’ n/4 processi
- etc
- I giri sono numerati da 0 a log(n)-1

Il torneo



Il torneo

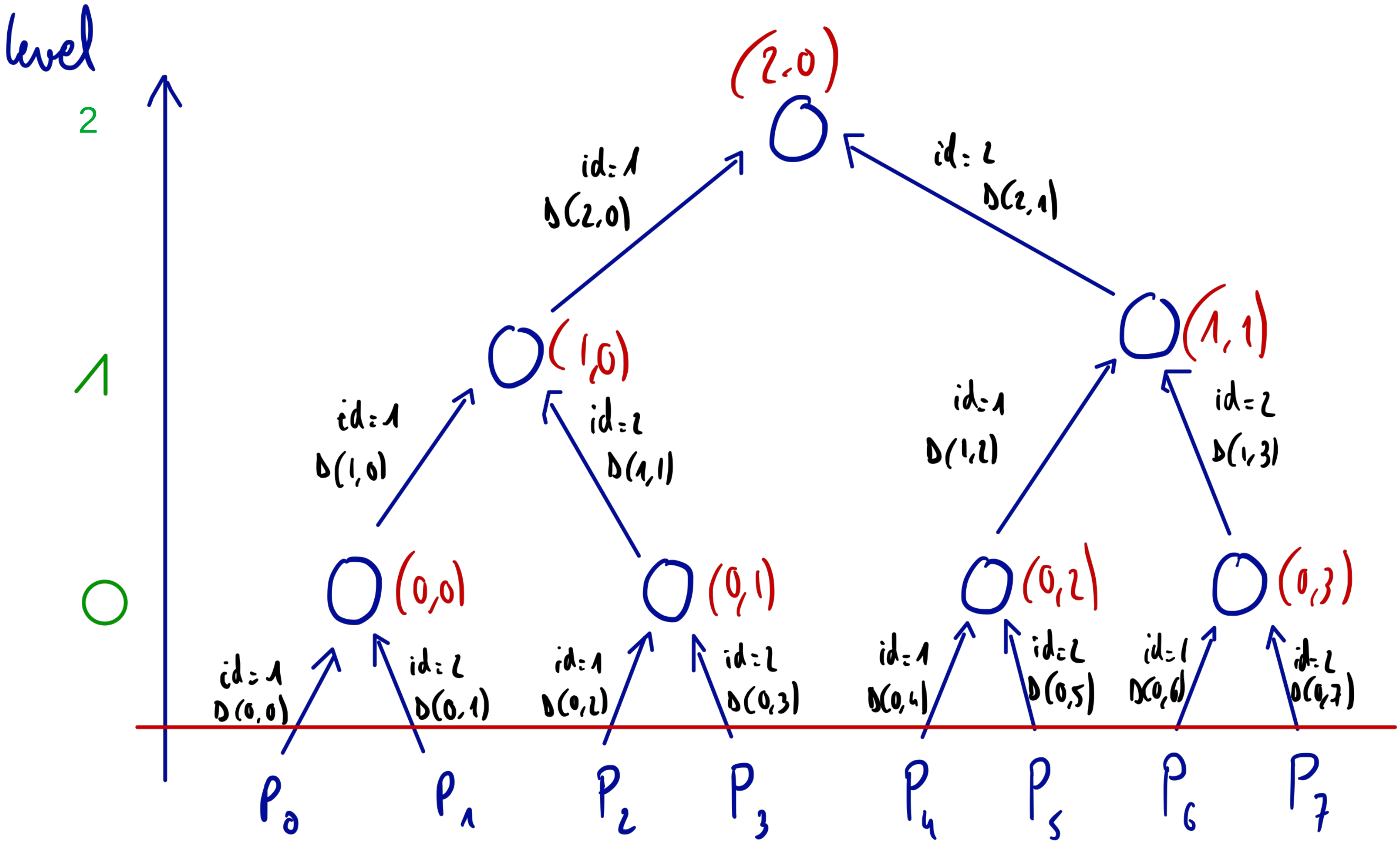
- Ogni ‘partita’ è identificata da un giro/livello l e da un numero k con :
  - le ϵ {0,...,log(n)-1} e k ϵ {0,...,$2^{\log(n)-1-le}$-1}
- Per usare l’algoritmo di Peterson, dobbiamo avere l’equivalente delle variabile condivise:
  - int turn(le,k) : chi è prioritaria per la partita
  - bool D(le,2k) il processo di **sinistra** vuole accedere alla sezione criticata
  - bool D(le,2k+1) il processo di **destra** vuole accedere alla sezione criticata
- Ogni processo deve anche sapere se è il processo di sinistra o di destra nella partita che sta giocando
- Per ogni processo, abbiamo quindi :
  - le : il giro corrente
  - k : il numero della partita nel giro l
  - id ϵ {1,2} che dice si il processo è a sinistra (1) o a destra (2)

Problema della sezione critica

il torneo

```
Process P[i]
while(true){
p1: SNC
p2: k=i
p3: for(int le=0; le<log(n); le++){
p4:    id=k%2+1 //id vale 1 o 2
p5:    k=k/2
p6:    D(le,2k+(id-1))=true
p7:     turn(le,k)=3-id
p8:     while(turn(le,k)!=id &&D(le,2k+2-id)==true){}
      }
p9: SC
p10: for(int le=log(n)-1; le>=0; le--){
p11:     D(le,i/2^le)=false}
```

Il torneo



level
2
1
0
(2,0)
id=1
D(2,0)
id=2
D(2,1)
(1,0)
(1,1)
id=1
D(1,0)
id=2
D(1,1)
id=1
D(1,2)
id=2
D(1,3)
(0,0)
(0,1)
(0,2)
(0,3)
id=1
D(0,0)
id=2
D(0,1)
id=1
D(0,2)
id=2
D(0,3)
id=1
D(0,4)
id=2
D(0,5)
id=1
D(0,6)
id=2
D(0,7)
P_0
P_1
P_2
P_3
P_4
P_5
P_6
P_7

Problema della sezione critica

Algoritmo del torneo - Proprietà

Teorema:
L'algoritmo del torneo verifica la mutua esclusione.

Prova:
- Se due processi sono insieme in SC,sono passati tutti due dalla stessa partita ad un momento
- Sia (le,k) quella partita.
- Come i due processi vengono da due rami diversi hanno due id diversi
- Ma se uno è andato oltre il while e l'altro l'ha raggiunto senza eseguire il post-protocollo, vuole dire che l'algoritmo di Peterson è falso....

Problema della sezione critica

Algoritmo del torneo - Proprietà

Teorema:
L’algoritmo del torneo Lock 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:

- Se un processo è bloccato alla partita (le,k), ad un momento sarà prioritario, una volta che l’altro processo partecipante alla partita avrà eseguito il suo post-protocollo

Storia

- Algoritmo del fornaio (*bakery algorithm*): algoritmo di mutua esclusione per $N \geq 2$ processi
- Inventato da Leslie Lamport in 1974
- Usa delle variabili **proprie** (multiple readers/single writer)
  - ogni processo ha le sue variabili che possono essere lette ma non modificate dagli altri processi
- È l’algoritmo usato nell’amministrazione, o dalle banche:
  - 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]...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])+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 atomic

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 t2
- Dobbiamo fare vedere che a questo istante, nessun altro processo P[k] (con k≠i) è in SC
- Scegliamo un k≠i e vediamo dove è P[k] in t2
- Usiamo due altri istanti:
  - t0: P[i] passa il while(choosing[k]){}
  - t1: 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])+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 t0 ?
  - **in p3,p4 ?** no perché à t0, 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])+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 t0 ?
  - **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 t1
      - P[k] passera 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] dovra aspettare che nb[i] sia calcolato e verrà bloccato in p8

P[k] non può essere in SC in t2

```
Process P[id]
while(true){
p1: SNC
p2: choosing[id]=true;
p3: nb[id]=Max(nb[0],...,nb[n-1])+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 tal 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])+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])+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])+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
```

In realtà, bisogna ad iterare questo passo

Problema della sezione critica

Algoritmo del fornaio - Proprietà

Teorema:
Nel l'algoritmo del fornaio i valori nel nb possono crescere al 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])+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 variabili 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){}
p3: **SC**
p4: x.reset()}

- Mutua esclusione ✅
- Assenza di deadlock ✅
- Assenza di starvation ❌
- Attesa limitata ❌