

PCAD
Programmazione Concorrente
Algoritmi Distribuiti

Arnaud Sangnier
arnaud.sangnier@unige.it

MUTUA ESCLUSIONE ANCORA

Problema della sezione critica

Algoritmo di Dekker

```
int turn=1; //variabili condivise  
bool D1=false;  
bool D2=false;
```

```
Process P1  
while(true){  
p1: SNC  
p2: D1=true;  
p3: while(D2==true){  
p4:  if(turn==2){  
p5:   D1=false;  
p6:   while(turn!=1){}  
p7:   D1=true;}}  
p8: SC  
p9: turn=2;  
p10:D1=false;}
```

```
Process P2  
while(true){  
q1: SNC  
q2: D2=true;  
q3: while(D1==true){  
q4:  if(turn==1){  
q5:   D2=false;  
q6:   while(turn!=2){}  
q7:   D2=true;}}  
q8: SC  
q9: turn=1;  
q10:D2=false;}
```

Quale proprietà verifica l'algoritmo ? Come verificarle?

Problema della sezione critica

Algoritmo di Dekker - Proprietà

- Un **invariante** è una proprietà che è vera in tutti gli stati
- Un **invariante induttivo** è una proprietà che è vera in uno stato e per **tutti** gli stati (anche quelli non raggiungibili), se è vera allora è vera in ogni stato successivo

Proprietà:

L'algoritmo di Dekker verifica gli invariante (induttivi) successivi:

- $\text{turn} == 1$ or $\text{turn} == 2$
- $(p3 \text{ or } p4 \text{ or } p5 \text{ or } p8 \text{ or } p9 \text{ or } p10) \iff D1 == \text{true}$
- $(q3 \text{ or } q4 \text{ or } q5 \text{ or } q8 \text{ or } q9 \text{ or } q10) \iff D2 == \text{true}$

Prova:

- il valore iniziale di turn è 1 e le uniche istruzioni che modificano turn sono o $\text{turn} = 1$ o $\text{turn} = 2$
- D1 è modificato solo da P1 e D2 solo da P2

Problema della sezione critica

Algoritmo di Dekker - Proprietà

Teorema:

L'algoritmo di Dekker verifica la mutua esclusione.

Prova:

- Supponiamo che P1 arriva in SC (quindi in p8)
- Allora $D2 == \text{false}$ (usciamo del while)
- Grazie al invariante
 $(q3 \text{ or } q4 \text{ or } q5 \text{ or } q8 \text{ or } q9) \iff D2 == \text{true}$
P2 non può essere anche lui in q8

```
Process P1
while(true){
p1: SNC
p2: D1=true;
p3: while(D2==true){
p4:  if(turn==2){
p5:   D1=false;
p6:   while(turn!=1){}
p7:   D1=true;}}
p8: SC
p9: turn=2;
p10:D1=false;}
```

Problema della sezione critica

Algoritmo di Dekker - Proprietà

Teorema:

L'algoritmo di Dekker verifica l'assenza di deadlock.

Prova:

- Supponiamo che c'è un deadlock.
- P1 e P2 sono entrambi 'bloccati' nel pre-protocollo
- Supponiamo che $turn == 1$ (NB: il pre-protocollo non cambia il valore di $turn$)
- Allora P1 finirà per essere in loop fra p3 e p4
- E P2 finirà per essere in loop su q6 (perché $turn == 1$)
- Quindi finiremmo per avere $D2 = false$ grazie a l'invariante ($q3 \text{ or } q4 \text{ or } q5 \text{ or } q8 \text{ or } q9$) $\Leftrightarrow D2 == true$
- **Ma allora P1 non può essere in loop fra p3 e p4**

```
Process P1
while(true){
p1: SNC
p2: D1=true;
p3: while(D2==true){
p4:  if(turn==2){
p5:   D1=false;
p6:   while(turn!=1){}
p7:   D1=true;}}
p8: SC
p9: turn=2;
p10:D1=false;}
```

Problema della sezione critica

Algoritmo di Dekker - Proprietà

Teorema:

L'algoritmo di Dekker 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

int turn=1; //variabile condivisa

bool D1=false;

bool D2=false;

```
Process P1
while(true){
p1: SNC
p2: D1=true;
p3: while(D2==true){
p4:  if(turn==2){
p5:   D1=false;
p6:   while(turn!=1){}
p7:   D1=true;}}
p8: SC
p9: turn=2;
p10:D1=false;}
```

```
Process P2
while(true){
q1: SNC
q2: D2=true;
q3: while(D1==true){
q4:  if(turn==1){
q5:   D2=false;
q6:   while(turn!=2){}
q7:   D2=true;}}
q8: SC
q9: turn=1;
q10:D2=false;}
```

Problema della sezione critica

Algoritmo di Dekker - Prova dell'assenza di starvation

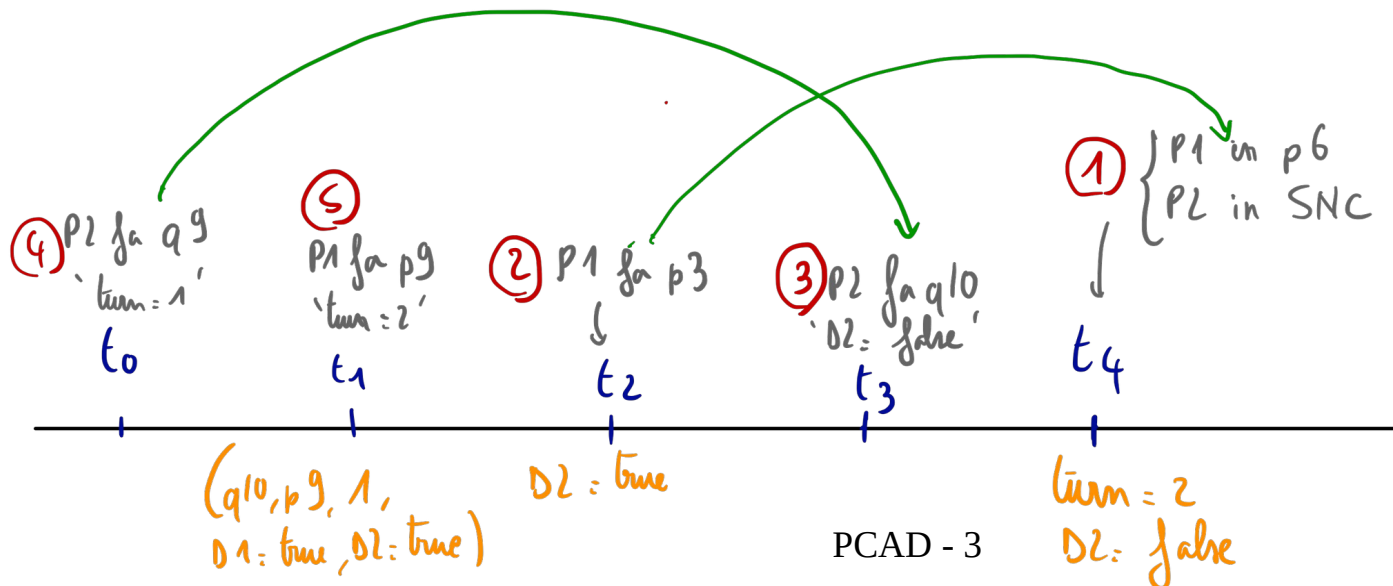
- Supponiamo che P1 è in starvation. Dunque P1 è bloccato nel suo pre-protocollo. Dove è P2 ?
 - Non può essere bloccato nel pre-protocollo (assenza di deadlock)
 - Se non è bloccato nella sua SNC e richiede la SC **infinitamente spesso**, allora dopo avere accesso a SC:
 - mette turn a 1 dando la priorità a P1
 - turn non cambierà più, quindi P1 non può essere bloccato in p6
 - allora P1 è bloccato fra P3 e P4 e P2 finirà bloccato in q6
 - perché $turn == 1$ e $D1 == true$
 - ma allora abbiamo un deadlock -> **contradizione**

```
Process P1
while(true){
p1: SNC
p2: D1=true;
p3: while(D2==true){
p4:  if(turn==2){
p5:   D1=false;
p6:   while(turn!=1){}
p7:   D1=true;}}
p8: SC
p9: turn=2;
p10:D1=false;}
```

Problema della sezione critica

Algoritmo di Dekker - Prova dell'assenza di starvation

- Se P2 è bloccato nella sua SNC
 - **t4**: tempo in cui P2 è bloccato in SNC e P1 è bloccato nel pre-protocollo
 - **D2==false** per tutto il futuro, quindi P1 è bloccato in p6 e **turn==2**
 - **t2**: ultimo momento in cui P1 ha fatto p3 ($t_2 < t_4$)
 - come P1 è in p6 in t4, in t2 abbiamo **D2==true**
 - fra t2 e t4, D2 è cambiato, dunque in **t3**, P2 ha fatto q10 ($t_2 < t_3 < t_4$)
 - prima di t3, al tempo **t0**, P2 ha fatto q9 (mettendo **turn=1**) ($t_0 < t_3$) (e P2 non fa nulla fra t0 e t3)
 - dopo t0, in t1, P1 ha fatto p9 e abbiamo $t_1 < t_2$
 - in t1, abbiamo lo stato
 - **(p9, q10, turn=1, D1=true, D2=true)**



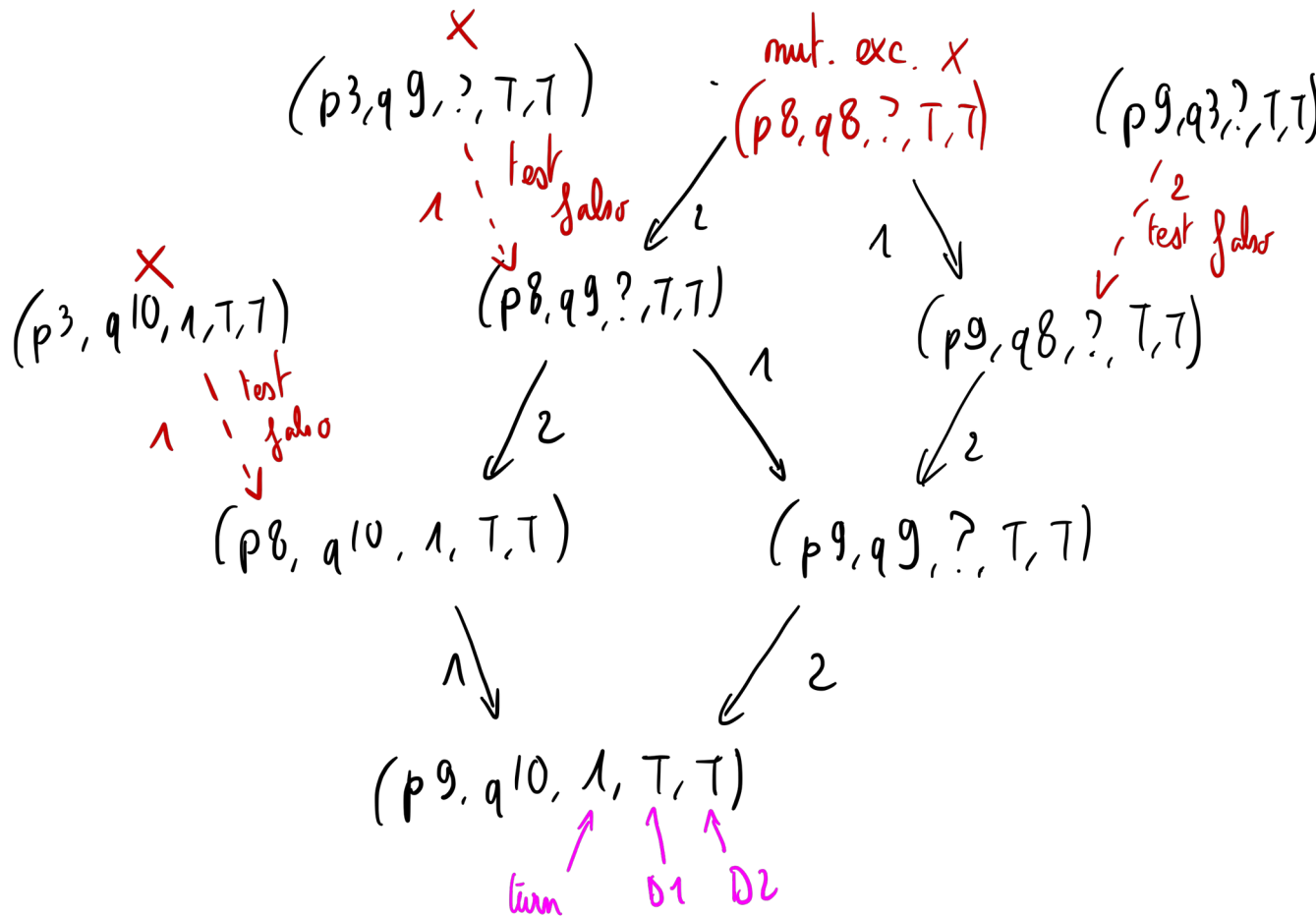
```

Process P1
while(true){
  p1: SNC
  p2: D1=true;
  p3: while(D2==true){
  p4:  if(turn==2){
  p5:   D1=false;
  p6:   while(turn!=1){}
  p7:   D1=true;}}
  p8: SC
  p9: turn=2;
  p10: D1=false;}
    
```


Problema della sezione critica

Algoritmo di Dekker - Prova dell'assenza di starvation

- Lo stato (**p9,q10,turn=1,D1=true, D2=true**) non è raggiungibile (prova con computazione 'in dietro')



```

Process P1
while(true){
  p1: SNC
  p2: D1=true;
  p3: while(D2==true){
  p4:  if(turn==2){
  p5:   D1=false;
  p6:   while(turn!=1){}
  p7:   D1=true;}}
  p8: SC
  p9: turn=2;
  p10: D1=false;}
    
```

Altri algoritmi di mutua esclusione

Volume 12, number 3

INFORMATION PROCESSING LETTERS

13 June 1981

MYTHS ABOUT THE MUTUAL EXCLUSION PROBLEM

G.L. PETERSON

Department of Computer Science, University of Rochester, Rochester, NY 14627, U.S.A.

Received 13 January 1981; revised version received 30 March 1981

Parallelism, mutual exclusion

Recently in these pages appeared a report by Doran and Thomas [2] which gave partially simplified versions of Dekker-like solutions to the two process mutual exclusion problem with busy-waiting. This report presents a truly simple solution to the problem and attempts in a small way to dispel some myths that seem to have arisen concerning the problem.

Briefly, the *mutual exclusion problem* for two processes is to find sections of code (*trying protocol*, *exit protocol*) for each of two asynchronous processes to use when trying to enter and upon exiting their designated critical sections. The protocols must preserve mutual exclusion and not have deadlock or lockout. *Mutual exclusion* means that both processes can never be in their critical sections at the same time. No *deadlock* or *lockout* means that no process waits forever inside a protocol. More formal definitions can be found in [5] and elsewhere.

The original solution due to Dekker is discussed at length by Dijkstra in [1]. Of the many reformulations given since, perhaps the best appears in [3]. (Unfortunately the authors believe their correct solution is incorrect.) The solutions of Doran and Thomas are slight improvements which eliminate the 'loop inside a loop' structure of the previously published solutions.

The protocols of P_1 and P_2 are given in Fig. 1. Q_1 and Q_2 are initially *false* and $TURN$ may start as either 1 or 2. (The busy wait loop '*wait until Boolean*' is just another way of saying "*repeat/* empty statement/* until Boolean*". The Boolean formula is *not* evaluated atomically.)

As can be seen, the algorithm has a very simple structure. This results in an easy proof of correctness. First, neither process can be locked out. Consider P_1 , it has only one wait loop, and assume it can be forced to remain there forever. After a finite amount of time, P_2 will be doing one of three general things: not trying to enter, waiting in its protocol, or repeatedly cycling through its protocols. In the first case, P_1 notes that Q_2 is *false* and proceeds. The second case is impossible due to $TURN$ being either 1 or 2; and one of the processes will proceed. In the third case P_2 will quickly set $TURN$ to 2 and never change it back to 1, allowing P_1 to proceed.

If mutual exclusion were not preserved and both processes could somehow end up in their critical sections at the same time, then we have $Q_1 = Q_2 = \text{true}$. Their tests in their wait loops just prior to entering their critical sections at this point could not have been at approximately the same time as $TURN$ would have

Storia

- Algoritmo di Dekker: 1964
- Algoritmo di Peterson: 1981
 - Più facile di dimostrare la sua correttezza
 - Più facile da adattare per $N > 2$ processi

Problema della sezione critica

Algoritmo di Peterson

```
int turn=1; //variabili condivise  
bool D1=false;  
bool D2=false;
```

```
Process P1  
while(true){  
p1: SNC  
p2: D1=true;  
p3: turn=2;  
p4: while(turn!=1 &&D2==true){}  
p5: SC  
p6:D1=false;}
```

```
Process P2  
while(true){  
q1: SNC  
q2: D2=true;  
q3: turn=1;  
q4: while(turn!=2 &&D1==true){}  
q5: SC  
q6:D2=false;}
```

Problema della sezione critica

Algoritmo di Peterson - Proprietà

Teorema:

L'algoritmo di Peterson verifica la mutua esclusione.

Prova:

- Supponiamo che P1 arriva in SC dopo P2
- Allora $D2=D1=true$ (P1 e P2 sono in p4 e p5)
 - e $turn=1$ (perché P1 è uscito dal while)
- Come P2 ha passato il while ?
 - **con $D1=false$?**
 - dunque nel fra tempo P1 ha fatto $D1=true$ e $turn=2$
 - ma allora P1 non può passare il while
 - **con $turn=2$?**
 - ma allora $turn$ è sempre uguale a 2 finché P2 è in SC e P1 non può passare il while

```
Process P1
while(true){
p1: SNC
p2: D1=true;
p3: turn=2;
p4: while(turn!=1 &&D2==true){}
p5: SC
p6:D1=false;}
```

Problema della sezione critica

Algoritmo di Peterson - Proprietà

Teorema:

L'algoritmo di Peterson verifica l'assenza di deadlock.

Prova:

- Se c'è deadlock, P1 è bloccato in p4 e P2 in q4
- Abbiamo
- $(\text{turn} \neq 1 \ \&\& \ D2 == \text{true}) \ \&\& \ (\text{turn} \neq 2 \ \&\& \ D1 == \text{true})$
 - IMPOSSIBILE perché turn prende come valore 1 o 2!!!

```
Process P1
while(true){
p1: SNC
p2: D1=true;
p3: turn=2;
p4: while(turn!=1 &&D2==true){}
p5: SC
p6:D1=false;}
```

Problema della sezione critica

Algoritmo di Peterson - Proprietà

Teorema:

L'algoritmo di Peterson 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:

- Supponiamo starvation di P1, allora
 - D1=true per sempre
 - e D2=true e turn=2 infinitamente spesso
(ogni volta che lo scheduler da la mano a P1)
 - D2=true => P2 è in q3,q4,q5,q6
 - Se è in q3, mette turn a 1 => P1 proseguirà ✗
 - Se è in q4 e turn=1,=> P1 proseguirà ✗
 - Se è in q4 e turn=2, andrebbe in q5 e ...
 - Se è in q5, andrebbe in q6, e ...
 - Se è in q6, andrebbe in q1, poi in q2 e in q3!

```
Process P1
while(true){
  p1: SNC
  p2: D1=true;
  p3: turn=2;
  p4: while(turn!=1 &&D2==true){}
  p5: SC
  p6:D1=false;}
```

Passare da 2 a $N > 2$ processi

- Diversi algoritmi per passare da 2 a N processi, vedremmo:
 - the filter lock
 - ci sono $N-1$ sale d'attesa che bloccano al meno un processo
 - bisogna ad attraversare queste sale
 - il torneo
 - ad ogni turno, il numero massimo di processi che va avanti è diviso per 2

Idea dietro the Filter Lock

- Algoritmo per $N > 2$ processi
- Ci sono $N-1$ sale d'attese da attraversare
- Per ogni sala (**level**):
 - Tra i processi che provano a passare, **al meno uno riesce**
 - Se ci sono più processi che provano, **al meno uno viene bloccato**

Problema della sezione critica

Algoritmo di Peterson

```
int turn=1; //variabile condivisa  
bool D1=false;  
bool D2=false;
```

```
Process P1  
while(true){  
p1: SNC  
p2: D1=true;  
p3: turn=2;  
p4: while(turn!=1 &&D2==true){}  
p5: SC  
p6:D1=false;}
```

```
Process P2  
while(true){  
q1: SNC  
q2: D2=true;  
q3: turn=1;  
q4: while(turn!=2 &&D1==true){}  
q5: SC  
q6:D2=false;}
```

Chi ha turn è prioritario!!!

Problema della sezione critica

Algoritmo di Peterson - Cambiamento punto di vista

```
int victim=1; //variabile condivisa  
bool D1=false;  
bool D2=false;
```

```
Process P1  
while(true){  
p1: SNC  
p2: D1=true;  
p3: victim=1;  
p4: while(victim==1 &&D2==true){}  
p5: SC  
p6:D1=false;}
```

```
Process P2  
while(true){  
q1: SNC  
q2: D2=true;  
q3: victim=2;  
q4: while(victim==2 &&D1==true){}  
q5: SC  
q6:D2=false;}
```

Self mutilation, la vittima viene bloccata!!!

Problema della sezione critica


The Filter Lock

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

$\text{int level}[N] = \{0, \dots, 0\};$ //variabili condivise

$\text{int victim}[N] = ?$

```
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(  $\exists k \neq \text{id}. \text{level}[k] \geq j$  && 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 $k=0$)
- $\text{num}_{\geq k}$: numero di processi che 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(  $\exists k \neq \text{id} . \text{level}[k] \geq j \ \&\& \ \text{victim}[j] == \text{id}$  ){
      }
p6: SC
p7: level[id]=0;}
```

Problema della sezione critica

Filter Lock - Proprietà

Proprietà:

Abbiamo $\text{num}_{\geq j} \leq n-j$ for all $0 \leq j \leq n-1$.

Prova (per induzione) :

- **j=0:** si abbiamo $\text{num}_{\geq 0} \leq n$
- **j>0:**
 - per ipotesi di induzione, sappiamo che al massimo $n-j+1$ processi hanno passato il livello $j-1$
 - Supponiamo che tutti hanno passato il livello j
 - Sia M , l'ultimo ad avere fatto $\text{victim}[j]=M$
 - Tutti gli altri M' , hanno fatto prima
 - $\text{level}[M']=j$
 - $\text{victim}[j]=M'$
 - **Non è possibile 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(  $\exists k \neq \text{id}. \text{level}[k] \geq j \ \&\& \ \text{victim}[j] == \text{id}$  ){
      }
  p6: SC
  p7: level[id]=0;}
```

Problema della sezione critica

Filter Lock - Proprietà

Proprietà:

Abbiamo $\text{num}_{\geq j} \leq n-j$ for all $0 \leq j \leq n-1$.

Corollario:

$\text{num}_{\geq (n-1)} \leq 1$

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 $j=N-1$.

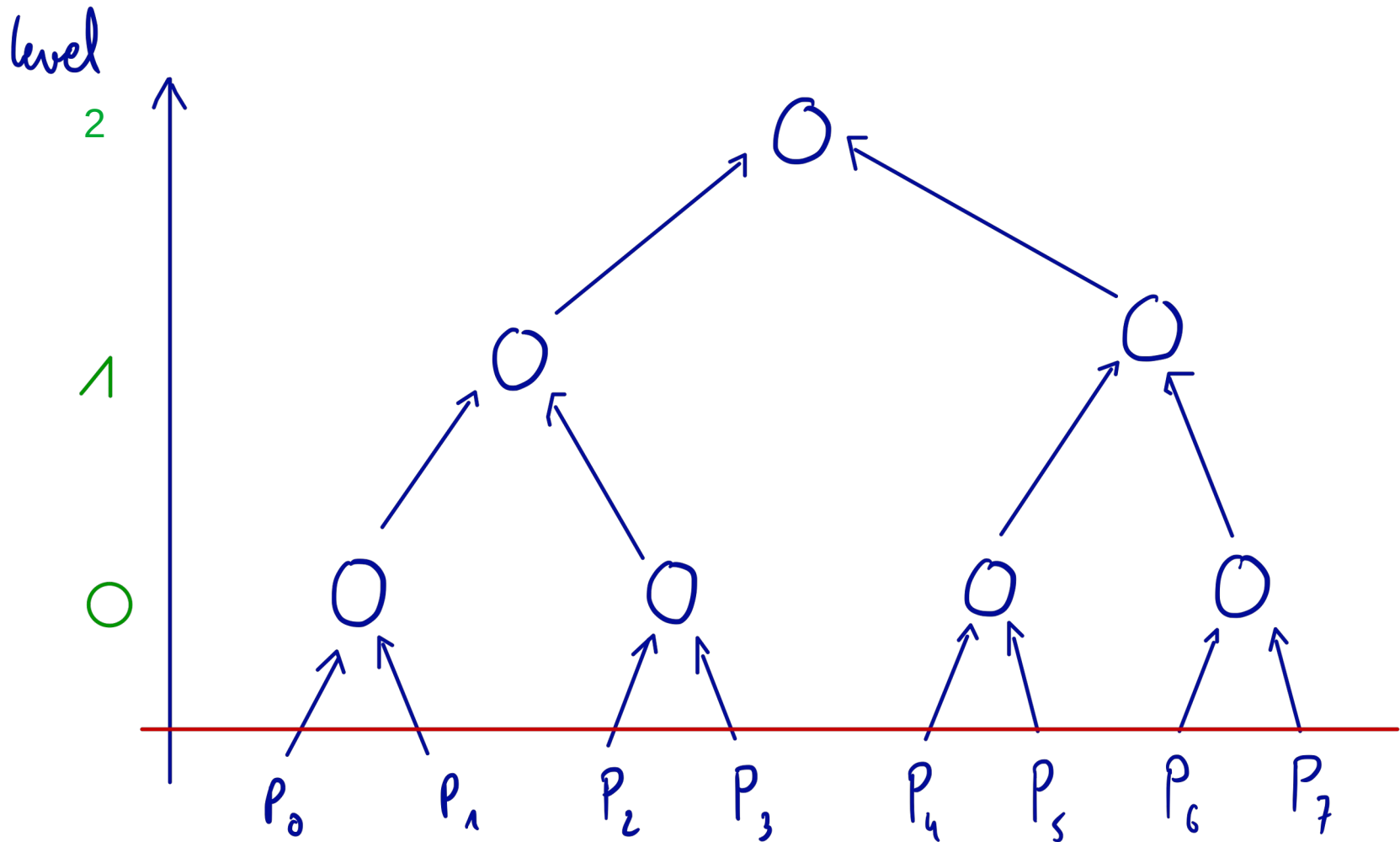
Prova (per induzione rovesciata):

- $j=N-1 \rightarrow$ 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 : P_0, P_1, \dots, P_n
- 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

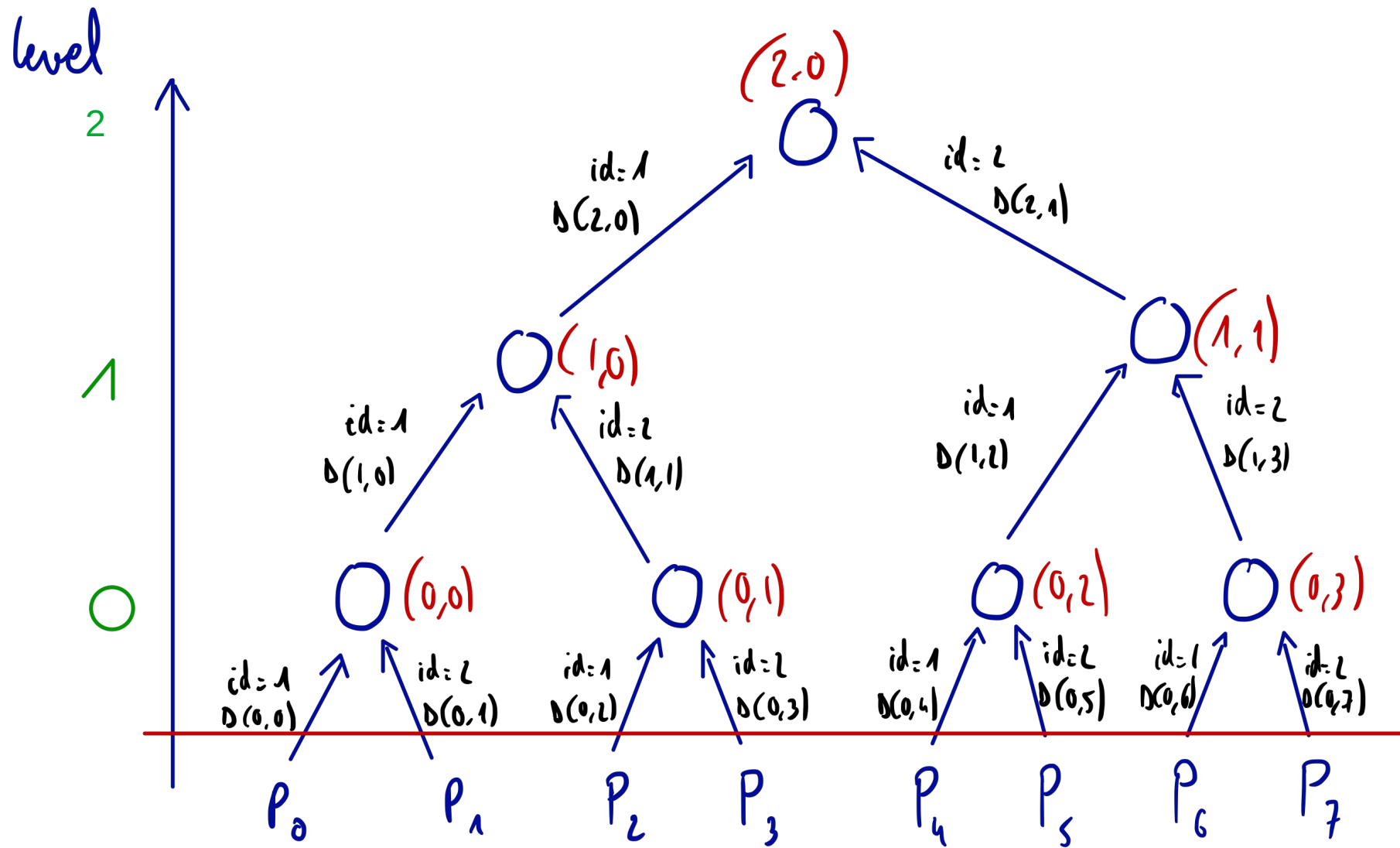
- Ogni 'partita' è identificata da un giro/livello l e da un numero k con :
 - $le \in \{0, \dots, \log(n)-1\}$ e $k \in \{0, \dots, 2^{\log(n)-1-l}-1\}$
- Per usare l'algoritmo di Peterson, dobbiamo avere l'equivalente delle variabile condivise:
 - $\text{int turn}(le, k)$: chi è prioritaria per la partita
 - $\text{bool D}(le, 2k)$ il processo di **sinistra** vuole accedere alla sezione criticata
 - $\text{bool D}(le, 2k+1)$ 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
 - $\text{id} \in \{1, 2\}$ che dice se 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=id
p3: for(int le=0; le<log(n); le++){
p4:   id=k%2+1
p5:   k=k/2
p6:   D(le,2k+(id-1))=true
p7:   turn(le,k)=3-id
p8:   while(turn(le,k)!=id &&D(2,2k+2-id)==true){}
      }
p9: SC
p10: for(int le=log(n)-1; le>=0; le--){
p11:   D(le,i/2le)=false}
```

Il torneo



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 (l, 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 (e_i, k) , ad un momento sarà prioritario, una volta che l'altro processo partecipante alla partita avrà eseguito il suo post-protocollo