

PCAD

Programmazione Concorrente

Algoritmi Distribuiti

Arnaud Sangnier
arnaud.sangnier@unige.it

INTERLEAVING e MUTUA ESCLUSIONE

Esecuzione concorrente

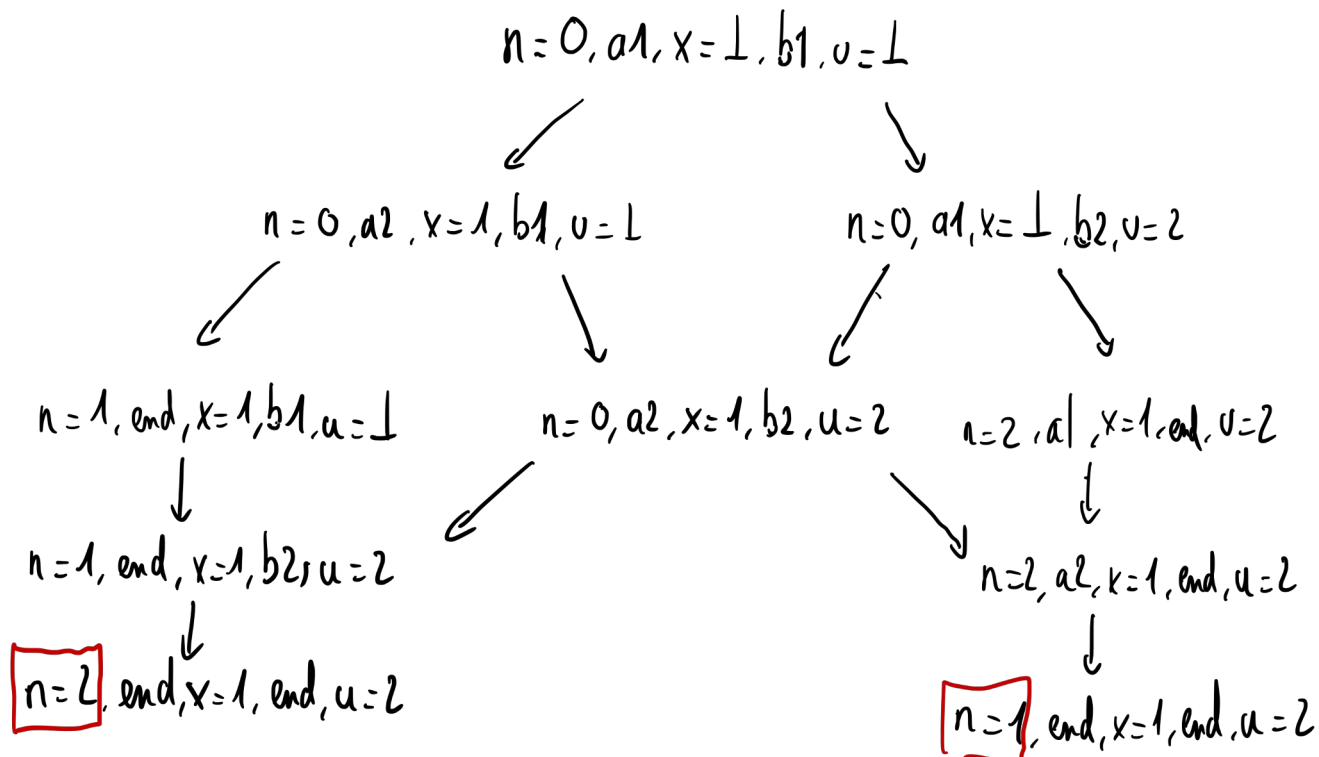
int n=0; //variabile condivisa

Process P1

```
int x;  
a1: x=1;  
a2: n=x;
```

Process P2

```
int u;  
b1: u=2;  
b2: n=u;
```



Istruzione atomiche

- Nozione **molto importante**
- La correzione dei programmi dipende degli ipotesi di atomicità
- **Una istruzione atomica è eseguita completamente senza essere interrotta**
- Non sono sempre facile a definire
- In questo corso, supponiamo che **ogni riga negli programmi sarà atomica** (a parte eccezioni)
- Esempi
 - $x=5$
 - $x=x+1$ diverso di $\begin{array}{l} \text{tmp}=x; \\ x=\text{tmp}+1; \end{array}$
 - $\text{If } (x==2) \{$
 $\quad x=3$
 $\quad \}$ spesso non atomica tranne test-and-est

Esecuzione concorrente

`int n=0; //variabile condivisa`

Process P1
`a1:n=n+1`

Process P2
`b1: n=n+1`

Se `n=n+1` è atomica

Senno

`int n=0; //variabile condivisa`

Process P1
`int tmp;`
`a1:tmp=n;`
`a2:n=tmp+1;`

Process P2
`int tmp2;`
`b1:tm2=n;`
`b2: n=tmp2+1`

Possiamo ottenere risultati molto diversi !!!

Ipotesi di atomicità

- In realtà, gli ipotesi fatti sono poco realiste (ma permettono di ragionare, e di avere modelli a cui riferirsi)
- **Difficoltà:** gli ipotesi possono cambiare secondo il compilatore, il processore, etc

Una regola:

Evitare istruzioni manipolando più di una occorrenza di variabile condivise fra processi

- al posto di fare $n=n+m+5$ dove n e m sono condivise meglio fare:

`tmp=n;`

`tmp=tmp+m;`

`n=tmp+5;`

`tmp:` **variabile locale al processo**

Ipotesi di atomicità

- Diversi casi:
 - **registri atomici** -> operazione di lettura e di scrittura atomiche
 - **test-and-set** -> si può testare E modificare una variabile senza essere interrotto
 - **swap** -> si può scambiare il valore di un registro locale e di uno condiviso in un modo atomico
 - ...

Semantica di intreccio (interleaving)

Definizione:

Uno **stato** di un programma concorrente P è un tuple con:

- un puntatore d'istruzione per ciascun processo
- un valore per ciascuna delle variabile (locale e condivise)

- Per esempio $(n=0, \text{end}, x=1, b1, u=2)$

Definizione:

Per due stati $s1$ e $s2$, se scrive **$s1 \rightarrow s2$** quando si può passare da $s1$ a $s2$ usando una degli istruzioni puntata in $s1$

- Per esempio $(n=0, a1, x=1, b1, u=2) \rightarrow (n=2, \text{end}, x=1, \text{end}, u=2)$

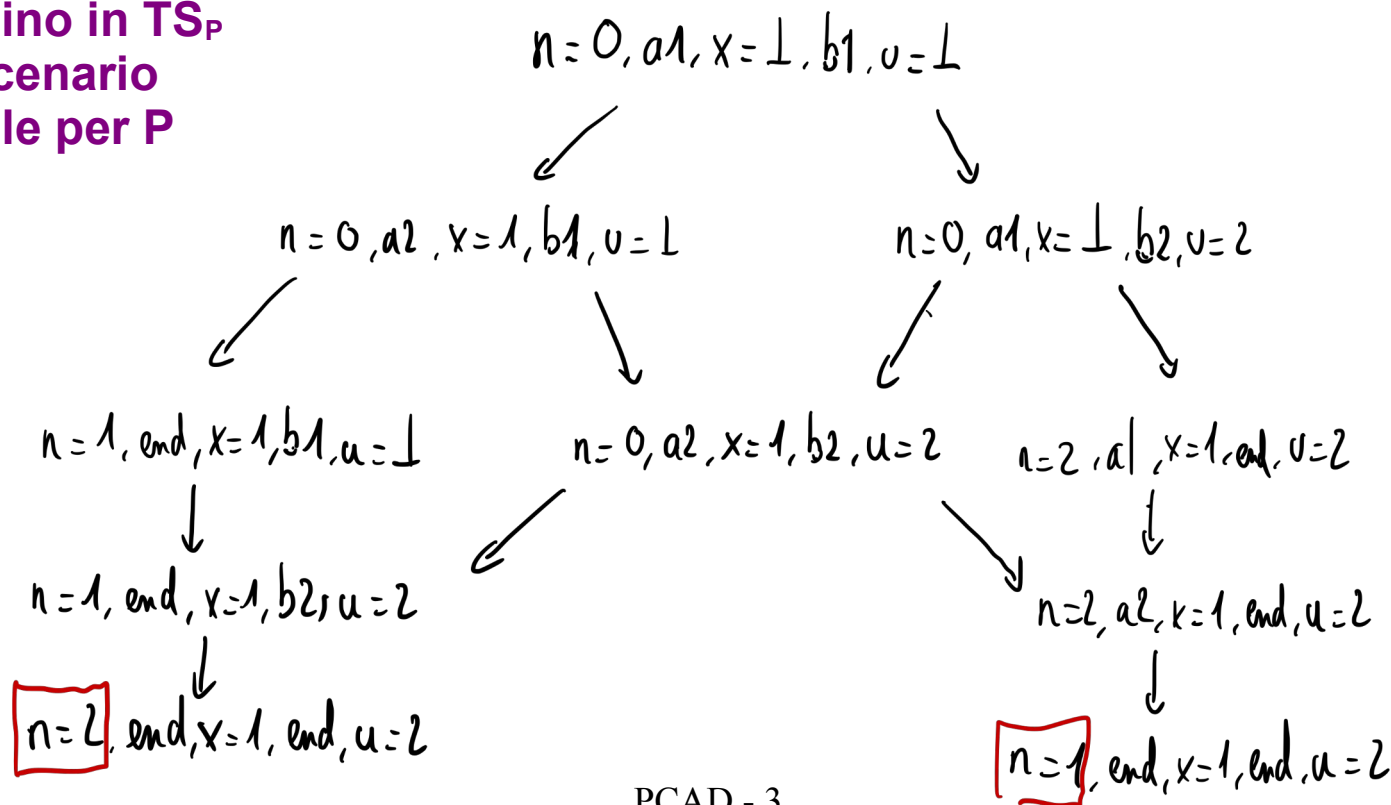
Semantica di intreccio (interleaving)

Definizione:

Il **diagramma degli stati** di P è il sistema di transizione (i.e. il grafo) $TS_P = (S, \rightarrow, s_0)$ dove:

- S è l'insieme degli stati di P
- \rightarrow è la relazione di transizione inclusa in $S \times S$
- s_0 è lo stato iniziale di P

Un cammino in TS_P
è un scenario
possibile per P



Semantica di intreccio (interleaving)

- I Cammini in TS_P corrispondono a tutti gli intrecci possibili per P.
- Facciamo quindi l'ipotesi che tutti questi intrecci sono possibili
- È una astrazione (in pratica è molto poco probabile di avere tutti gli intrecci)
- Avvolte guarderemmo solo i scenari 'fair' (giusti?), dove ogni processo che può fare una istruzione avrà un giorno la possibilità di farla
- Proveremmo la correzione degli algoritmi per **TUTTI gli intrecci** (o avvolte per tutti gli intrecci fair) -> verifichiamo più di quello che possiamo osservare in pratica, ragionamento robusto

Semantica di intreccio (interleaving)

- Una semantica ragionevole:
 - Per i sistemi con un processore condivisi fra più processi
 - C'è una successione d'istruzione di un processo, poi di un altro, etc
 - Per i sistemi con più processori
 - Ogni processo è legato ad un processore
 - La semantica d'intreccio non corrisponde esattamente alla realtà (ci sono delle azioni parallele), ma è corretta se non ci sono conflitti sulle risorse (evitare accesso simultaneo alla stessa data)
 - Per i sistemi distribuiti
 - Diversi computer, niente variabile condivise, dei canali di comunicazione
 - Molto diversa dai due primi casi, ma la semantica rimane interessante se uno ci aggiunge l'invio e la ricezione di messaggi

Un problema centrale

Il problema di mutua esclusione

permette di garantire che ad un certo momento, un processo è solo ad accedere a delle risorse comune

Problema della sezione critica

Il problema:

$N \geq 2$ processi eseguono in modo continue due sequenze d'istruzione

- Una sezione non critica (**SNC**)
- Una sezione critica (**SC**)

in tal modo che la mutua esclusione sia garantita in SC: **ad ogni istante, c'è al massimo un processo in sezione critica**

- Per risolvere questa programma, useremmo delle variabile condivise (**che non verranno usate in SC e SNC**) dentro
 - un pre-protocollo prima di accedere a SC
 - un post-protocollo dopo l'accesso a SC

Problema della sezione critica

Struttura dei programmi

```
Process P1

while(true){
  p1: SNC
  p2...: pre-protocollo
  pi: SC
  pj...: post-protocollo
}
```

Ipotesi:

- la SC finisce sempre
- la SNC può non finire
- i due processi rimangono sempre attivi

Problema della sezione critica

Esempio

`int turn=1; //variabile condivisa`

```
Process P1
while(true){
  p1: SNC
  p2: while(turn!=1){}
  p3: SC
  p4: turn=2;
}
```

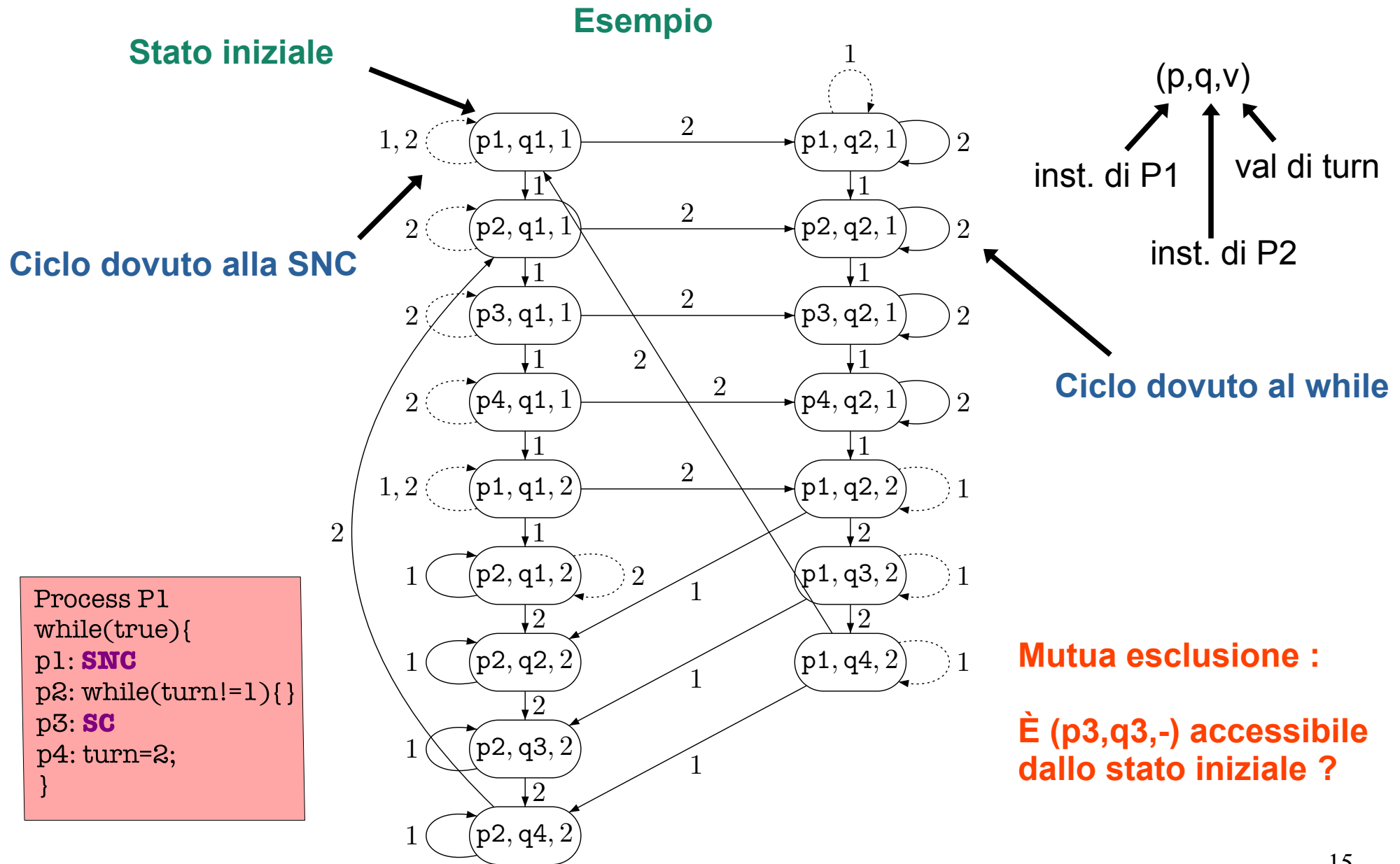
```
Process P2
while(true){
  q1: SNC
  q2: while(turn!=2){}
  q3: SC
  q4: turn=1;
}
```

Quali scenari ?

Uno **stato** di un programma concorrente P è un tuple con:

- un puntatore d'istruzione per ciascun processo
- un valore per ciascuna delle variabile (~~locale~~ e condivise)

Problema della sezione critica



Problema della sezione critica

- Proprietà da verificare:
 - **Mutua esclusione:**
 - Al più, un processo in SC ad ogni momento
 - **Assenza di deadlock:**
 - Se più processi provano allo stesso momento di accedere alla SC, non devono essere bloccati nel pre-protocollo ed al meno un processo deve arrivarci
 - **Assenza di 'starvation'** (carestia?):
 - Se un processo desidera accedere alla sua SC, un giorno ci riuscirà
 - **Attesa limitata:**
 - Se un processo aspetta per accedere alla sua SC, si può limitare via una **costante** il numero di volte in cui dovrà concedere il posto ad un altro processo

Attesa limitata => Assenza di starvation => Assenza di deadlock





Problema della sezione critica

Esempio

`int turn=1; //variabile condivisa`

```
Process P1
while(true){
p1: SNC
p2: while(turn!=1){}
p3: SC
p4: turn=2;
}
```

```
Process P2
while(true){
q1: SNC
q2: while(turn!=2){}
q3: SC
q4: turn=1;
}
```

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

Problema della sezione critica

Algoritmo di Dekker

```
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;}
```

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:

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

Prova:

- il valore iniziale di $turn$ è 1 e le uniche istruzioni che modificano $turn$ sono o $turn = 1$ o $turn = 2$
- $D1$ 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 $\text{turn} == 1$ (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 $\text{turn} == 1$)
- Quindi finiremmo per avere $\text{turn} = 2$ grazie a l'invariante ($q3 \text{ or } q4 \text{ or } q5 \text{ or } q8 \text{ or } q9$) $\Leftrightarrow D2 == \text{true}$
- **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 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;}
```

Problema della sezione critica

Algoritmo di Dekker - Prova dell'assenza di starvation

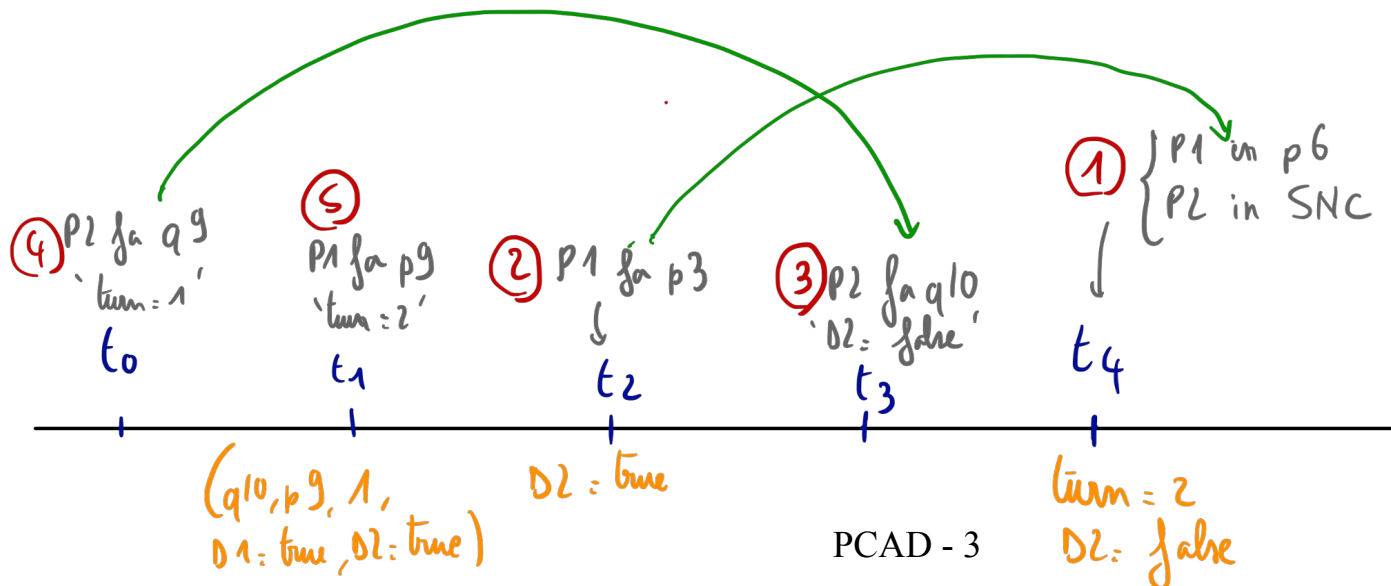
- Supponiamo che P1 è in starvation. Dunque P1 è bloccato nel suo pre-protocollo. Dove è P2 ?
 - Non può essere bloccato nel pre-protocollo (assenza di deadlock)
 - Se non è bloccato nella sua SNC e richiede la SC **infinitamente spesso**, allora dopo avere accesso a SC:
 - mette turn a 1 dando la priorità 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 un 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**)



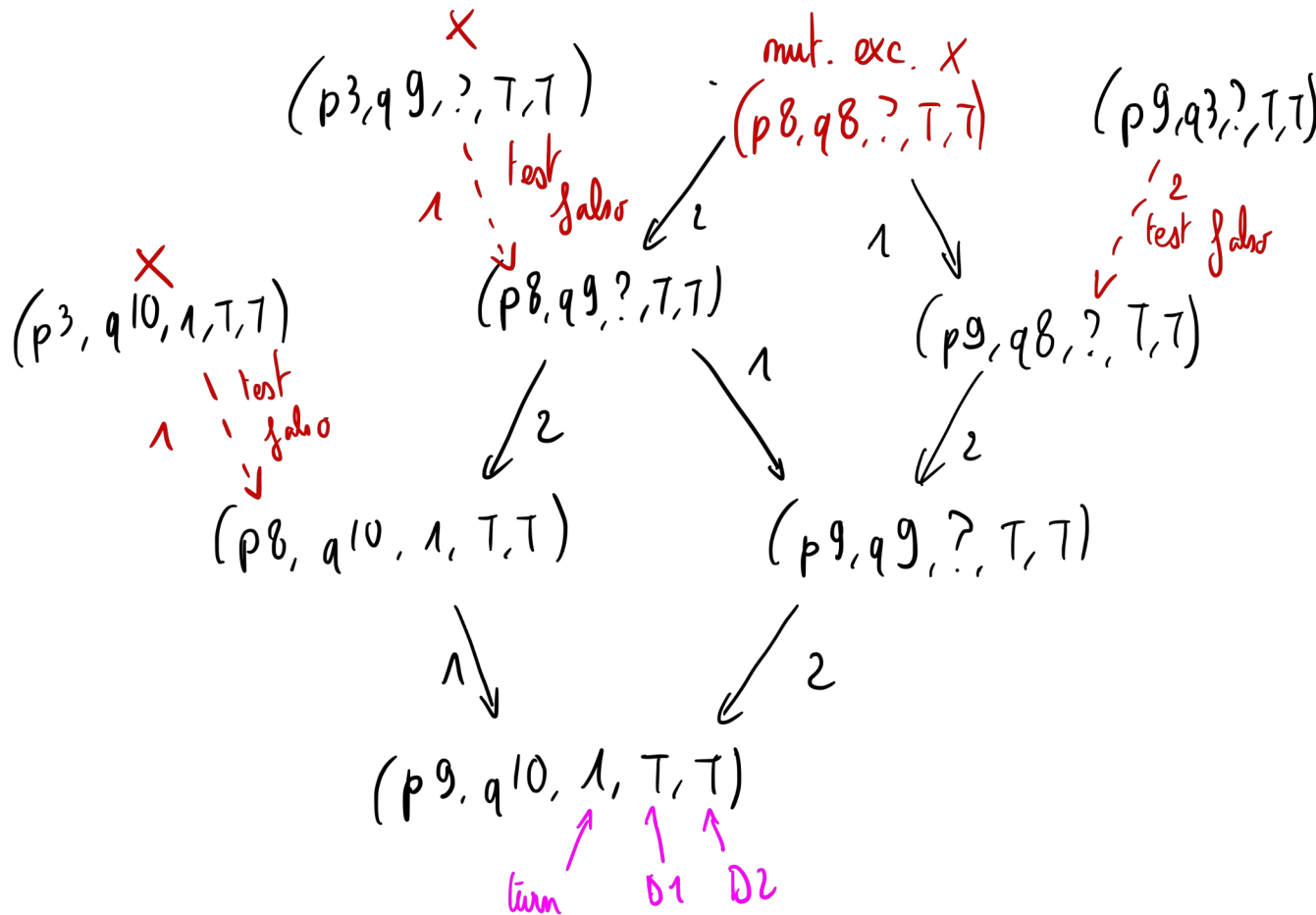
```

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;}

```