PCAD
Programmazione Concorrente
Algoritmi Distribuiti

Arnaud Sangnier
arnaud.sangnier@unige.it

Algoritmi via message passing

Problema della sezione critica

```
Process P

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

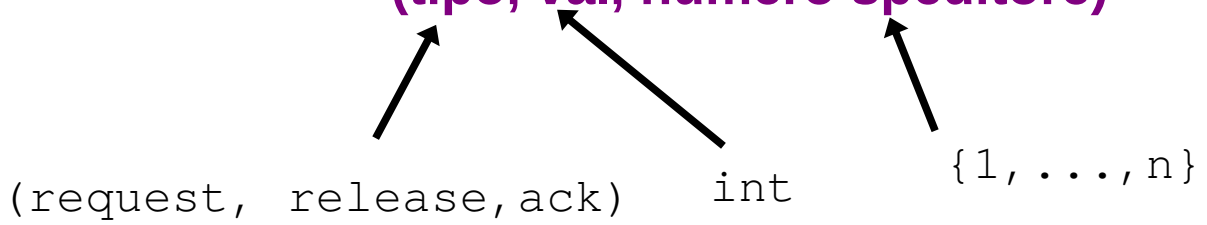
Ipotesi:
- la SC finisce sempre
- la SNC può non finire
- i processi rimangono sempre attivi

Comunicazione asincrona

Altri ipotesi
- Processi sono su macchine diverse e comunicano con messaggi
- Ogni messaggio mandato è ricevuto un giorno (comunicazione affidabile)
- Se P mando un messaggio M1 a Q poi un messaggio M2, Q riceve prima M1 poi M2

Comunicazione via messaggi

- I messaggi hanno un formato specificato, ad es:
  - **(tipo, val, numero speditore)**



- Emissione: `Send(msg) -> j` : Manda il messaggio msg a Pj
- Per gestire le recezione, supponiamo che su ogni macchina ci sono due processi che condividono la memoria
  1) il processo che esegue il protocollo
  2) il processo che gestisce la ricezione dei messaggi

Problema della sezione critica

Process P

main

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

||

ricezione messaggi

```
case(msg1):
      codice
case(msg2):
      codice
....
```

Altri ipotesi

- La parte ricezione messaggi funziona sempre
- Nella parte locale, autorizziamo l'uso di blocchi [...] per garantire l'atomicità al livello locale

Algoritmo di Lamport (1978)

- Ogni processo usa una variabile intera come orologio locale (aumenta di 1 ogni volta che un messaggio è mandato/ricevuto)
- **Pre-protocollo:**
  - Pi manda un messaggio `request` con la sua date locale a tutti gli altri processi e conserva una coppia di questo messaggio
  - Aspetta di essere il più prioritario fra gli altri candidati alla SC
  - (più prioritario = la data associata al suo messaggio `request` è la più vecchia)
- **Post-protocollo:**
  - Pi manda un messaggio `release` con la sua data locale a tutti gli altri processi e conserva una coppia di questo messaggio
- **Ricezione:**
  - Ogni processo manda un `ack` per ogni messaggio ricevuto

Algoritmo di Lamport (1978)

```
--Process Pi  // i in {1,..,n}

main
Message F[1..N]:={null,...,null}
int c=0
while(true){
  p1: SNC
  p2:  [ for all j!=i  {Send(request,c,i) -> j;}
  p3:     F[i]=(request,c,i);
  p4:     c=c+1;]
  p5: while( exists j. j!=i  date(F[j]) << date(F[i]) ) {}
  p6: SC
  p7: [ for all j!=i  {Send(release,c,i) -> j;}
  p8:     F[i]=(release,c,i);
  p9:     c=c+1;]
}
```

Blocchi atomici localmente

- date(typ,c,i)= (c,i)
- (c,i) << (d,j) sse c<d o (c=d e i < j)
- date(null) << (c,i) for all c ≥ 0 e 1 ≤ i ≤ n

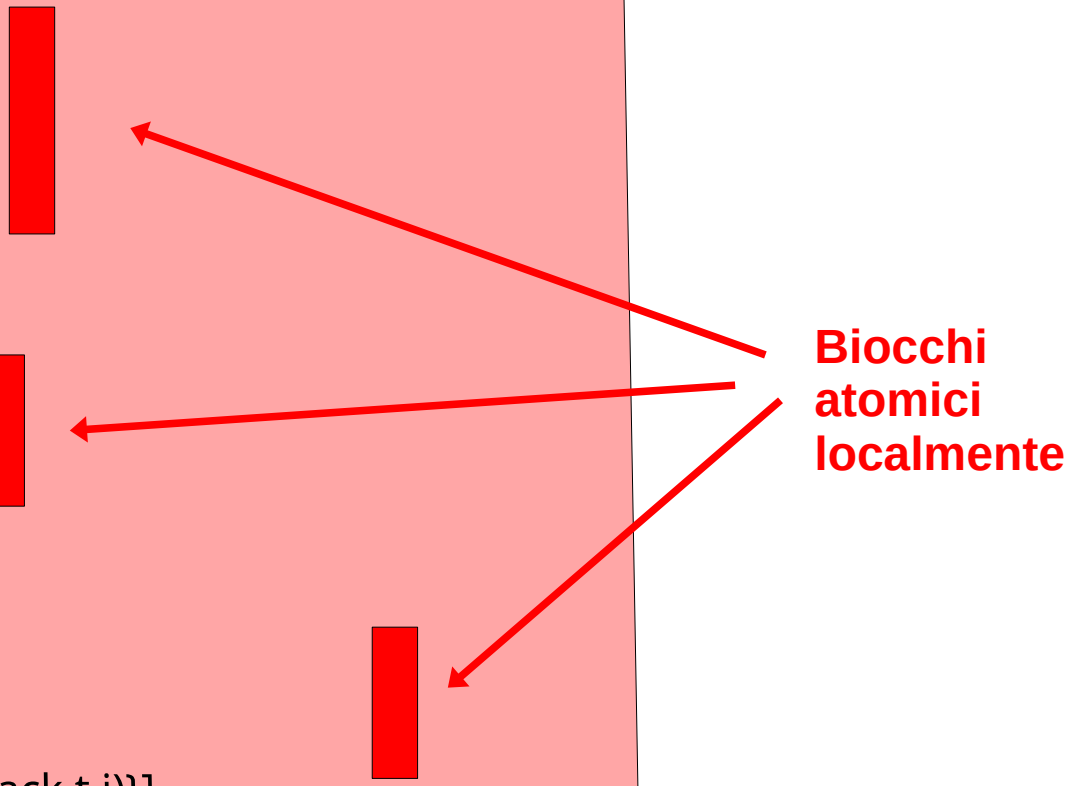
Algoritmo di Lamport (1978)

```
--Process Pi  // i in {1,..,n}

Ricezione messaggi
case (request, t, j):
  [ if (t>c) { c=t}
     c=c+1;
     F[j]= (request, t, j);
     Send(ack,c,i)->j;]

case (release,t,j):
  [ if (t>c) { c=t}
     c=c+1;
     F[j]= (release, t, j);]

case (ack,t,j):
[ if (t>c) { c=t}
     c=c+1;
     if(type(F[j]!=request){F[j]=(ack,t,j)}]
```



Algoritmo di Lamport (1978)

- Due osservazioni:

  1) i valori degli orologi locali crescono

  2) nel suo pre-protocollo, il processo Pi manda lo stesso messaggio `(request,c,i)` a tutti gli altri processi e lo conserva in F[i]

Algoritmo di Lamport (1978)

Teorema:
L'algoritmo di Lamport verifica la mutua esclusione.

Prova:

- Supponiamo Pi1 e Pi2 sono insieme in SC
- Abbiamo $F^{i1}[i1]$=(request,t1, i1) e $F^{i2}[i2]$=(request,t2,i2)
- Supponiamo (t1,i1)<<(t2,i2)
- Cosa puo essere in $F^{i2}[i1]$? i.e. quale l'ultimo messaggio ricevuto da Pi2 venando da Pi1 ?
  - Come Pi2 è in SC, è per forza della forma: (msg,t1',i1) con **t1' > t1**

Algoritmo di Lamport (1978)

Prova:

- $F^{i1}[i1]$=(request,t1, i1) e $F^{i2}[i2]$=(request,t2,i2) e $F^{i2}[i1]$=(msg,t1',i1) con **t1' > t1**
- Analisi di casi per msg:

1) msg=request allora Pi1 dopo avere mandato (request,t1, i1) ha mandato (request,t1', i1), quindi è entrato in SC è uscito mandando release ed ha rimandato (request,t1', i1), ma allora **$F^{i1}[i1]$≠(request,t1, i1) => Impossibile**
2) msg=ack se Pi2 ha conservato questo messagio vuol dire che prima di riceverlo $F^{i2}[i1]$≠(req,...,i1), ma quindi Pi1 era uscito dalla SC richiesta a t1, quindi **$F^{i1}[i1]$≠(request,t1, i1) => Impossibile**
3) msg=relsease stesso ragionamento che 1)

Algoritmo di Lamport (1978)

Teorema:
L'algoritmo di Lamport verifica l'assenza di deadlock

Prova:

- Supponiamo che dei processi sono bloccati e gli altri sono in SNC
- Sono bloccati sul while e prendiamo Pi il processo per il quale la data di request è minima
- Ma allora Pi è bloccato per un Pj che è in SNC (altrimenti la data di request di Pj sarebbe più piccola, mentre Pi è quello con la data più piccola)
- Ma quindi come Pj non è in SC, prima o poi Pi riceverà l'ack di Pj e potrà continuare (perché in F[j] da P[i] non ci può essere request)

Algoritmo di Lamport (1978)

Perché è necessario avere blocchi atomici?

- Se P1 manda (request,10,1) a P1 ma non mette a giorno il contattore c ne salva il messagio
- Poi P1 riceve ack e release di P3 e c vale allora 3, poi $F^1[1]$=(request,12,1)
- P2 manda (request,11,2) a P1 e riceve (request,10,1) di P1
- Abbiamo in P1 : $F^1[1]$=(request,12,1) e $F^1[2]$=(request,11,2)) con (11,2) << (12,1), quindi P1 è bloccato
- Abbiamo in P2 : $F^2[1]$=(request,10,1) e $F^2[2]$=(request,11,2)) con (10,1) << (11,2), quindi P2 è bloccato
- **=> DEADLOCK**

Algoritmo di Lamport (1978)

Teorema:
L'algoritmo di Lamport verifica l'assenza di starvation

Prova:

- Supponiamo che Pi sia in stato di starvation e che sia con quello con la data di request più piccola
- Quindi è bloccato per colpa di un Pj che non ha ancora mandato il suo ack ma finirà per riceverlo e lo salverà

Algoritmo di Lamport (1978)

Quanti messaggi per accedere alla SC ?

- Un processo manda (N-1) request
- Un processo aspetta (N-1) ack
- Un processo manda (N-1) release

=>3*(N-1) messaggi

Algoritmo di Ricart-Agrawala (1981)

- Simile a l'algoritmo di Lamport
- **Idea: limitare il numero di messaggi**
  - Quando Pj riceve request di Pi, risponde con un messaggio ok se:
    - non vuole andare in SC
    - oppure la sua richiesta è meno prioritaria di quella di Pi
  - Se Pj non manda subito ok a Pi, lo farà dopo avere lasciato la sua SC
  - La priorità **pr** è calcolata per ogni processo prendendo il max delle priorità ricevute nelle request degli altri.
  - Più **pr** è piccolo, più il processo è prioritario

Algoritmo di Ricart-Agrawala (1981)

```
--Process Pi  // i in {1,..,n}
main
boolean Waiting[1..N]:={false,...false}
boolean reqCS=false;
int pr=0;
int maxpr=0; // piu grande numero ricevuto
int nba=0; // numero di risposte attese
while(true){
  p1: SNC
  p2:  [reqCS=true;
  p3:     nba=n-1;
  p4:     pr=maxpr+1;]
  p5:  for all j!=i  {Send(request,pr,i) -> j;}
  p6: while(nba!=0){}
  p7:  SC
  p8:  [reqCS=false;
  p9:     for all j in{1..n}{
  p10:        if (Waiting[j]){
  p11:           Waiting[j]=false;
  p12:         Send(ok,i) -> j;}}]
}
```

Algoritmo di Ricart-Agrawala (1981)

```
--Process Pi  // i in {1,..,n}

Ricezione messaggi
case (request,k, j):
  [ if (k>maxpr) { maxpr=k;}
     if (!reqCS or (k,j) << (pr,i)){
          Send(ok,i) -> j;}
    else{
       Waiting[j]=true;
    } ]

 case (ok,j):
   [ nba=nba-1;]
```

Algoritmo di Ricart-Agrawala (1981)

Teorema:
L'algoritmo di Ricart-Agrawala verifica la mutua esclusione.

Prova:

- Supponiamo Pi1 e Pi2 sono insieme in SC
- Hanno scelto una priorità pri1 e pri2
- Ciascuno ha mandato un messaggio ok al altro
- Due casi:

1) Pi2 manda ok prima di scegliere pri2, ma allora pri2>pri1 e Pi1 non manda OK => contraddizione
2) Pi2 sceglie prima pri2 prima di mandare ok, ma allora abbiamo
   - (pri1,i1)<<(pri2,i2) e i1 non manda ok => contraddizione
   - oppure (pri2,i2)<<(pri1,i1) e i2 non manda ok => contraddizione

Algoritmo di Ricart-Agrawala (1981)

Teorema:
L'algoritmo di Ricart-Agrawala verifica l'assenza di starvation
(e quindi l'assenza di deadlock)

Prova:

- Supponiamo che Pi è il processo in starvation co la più piccola priorità pri
- Sia Pk un processo che non ha mandato ok a Pi
- Quindi quando Pk ha ricevuto (req,pri,i) avevamo (prk,k)<<(pri,i) altrimenti Pk avvrebbe mandato ok
- Nel fra tempo la priorità di k ha cambiato e Pk ha mandato ok a Pi (perché in Pk avevamo Waiting[i]=true)

Algoritmo di Ricart-Agrawala (1981)

Quanti messaggi per accedere alla SC ?

- Un processo manda (N-1) request
- Un processo aspetta (N-1) ok

=>2*(N-1) messaggi