

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-protocollo  
}
```

Ipotesi:

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

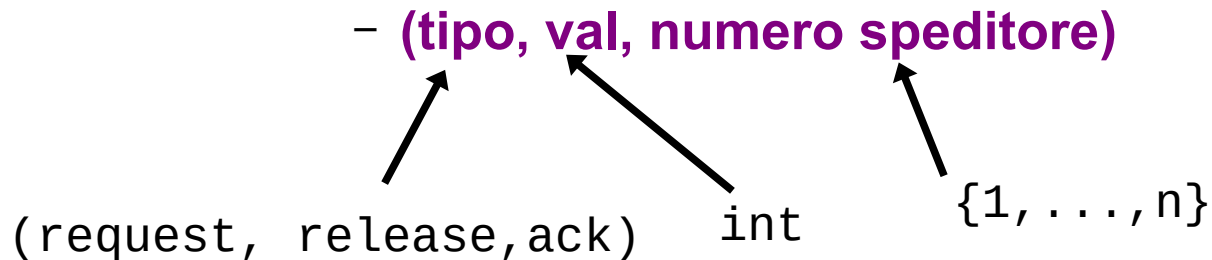
**Comunicazione
asincrona**

Altri ipotesi

- Processi sono su macchine diverse e comunicano con messaggi
- Ogni messaggio mandato è ricevuto un giorno (comunicazione affidabile)
- Se P manda 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:



- Emissione: $\text{Send}(\text{msg}) \rightarrow j$: Manda il messaggio msg a P_j
- Per gestire la ricezione, 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 data locale a tutti gli altri processi e conserva una copia 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 copia 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;]
```

```
}
```



**Biocchi
atomici
localmente**

- $\text{date}(\text{typ}, c, i) = (c, i)$
- $(c, i) << (d, j)$ sse $c < d$ o $(c = d \text{ e } i < j)$
- $\text{date}(\text{null}) << (c, i)$ for all $c \geq 0$ e $1 \leq i \leq n$

Algoritmo di Lamport (1978)

--Process P_i // i in $\{1, \dots, n\}$

Ricezione messaggi

```
case (request, t, j):  
  [ if ( $t > c$ ) {  $c = t$  }  
     $c = c + 1$ ;  
     $F[j] = (\text{request}, t, j)$ ;  
    Send(ack,  $c, i \rightarrow j$ ); ]
```

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

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



**Biocchi
atomici
localmente**

Algoritmo di Lamport (1978)

- Due osservazione:
 - 1) i valori degli orologi locali crescono
 - 2) nel suo pre-protocollo, il processo P_i 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 P_i1 e P_i2 sono insieme in SC
- Abbiamo $F^{i1}[i1]=(request,t1, i1)$ e $F^{i2}[i2]=(request,t2,i2)$
- Supponiamo $(t1,i1)<<(t2,i2)$
- Cosa può essere in $F^{i2}[i1]$? i.e. quale l'ultimo messaggio ricevuto da P_i2 venendo da P_i1 ?
 - Come P_i2 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 $F^{i1}[i1] \neq (request, t1, i1) \Rightarrow$ **Impossibile**
 - 2) $msg=ack$ se $Pi2$ ha conservato questo messaggio vuol dire che prima di riceverlo $F^{i2}[i1] \neq (req, ..., i1)$, ma quindi $Pi1$ era uscito dalla SC richiesta a $t1$, quindi $F^{i1}[i1] \neq (request, t1, i1) \Rightarrow$ **Impossibile**
 - 3) $msg=release$ 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 P_i il processo per il quale la data di request è minima
- Ma allora P_i è bloccato per un P_j che è in SNC (altrimenti la data di request di P_j sarebbe più piccola, mentre P_i è quello con la data più piccola)
- Ma quindi come P_j non è in SC, prima o poi P_i riceverà l'ack di P_j 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 contatore c ne salva il messaggio
- Poi P1 riceve ack e release di P3 e c vale allora 3, poi $F^1[1]=(\text{request},12,1)$
- P2 manda (request,11,2) a P1 e riceve (request,10,1) di P1
- Abbiamo in P1 : $F^1[1]=(\text{request},12,1)$ e $F^1[2]=(\text{request},11,2)$ con $(11,2) << (12,1)$, quindi P1 è bloccato
- Abbiamo in P2 : $F^2[1]=(\text{request},10,1)$ e $F^2[2]=(\text{request},11,2)$ 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 P_i sia in stato di starvation e che sia con quello con la data di request più piccola
- Quindi è bloccato per colpa di un P_j 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 P_j riceve request di P_i , risponde con un messaggio ok se:
 - non vuole andare in SC
 - oppure la sua richiesta è meno prioritaria di quella di P_i
 - Se P_j non mando subito ok a P_i , 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 P{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 P_{i1} e P_{i2} sono insieme in SC
- Hanno scelto una priorità $pri1$ e $pri2$
- Ciascuno ha mandato un messaggio ok al altro
- Due casi:
 - 1) P_{i2} manda ok prima di scegliere $pri2$, ma allora $pri2 > pri1$ e P_{i1} non manda OK \Rightarrow contraddizione
 - 2) P_{i2} sceglie prima $pri2$ prima di mandare ok, ma allora abbiamo
 - $(pri1, i1) << (pri2, i2)$ e $i1$ non manda ok \Rightarrow contraddizione
 - oppure $(pri2, i2) << (pri1, i1)$ e $i2$ non manda ok \Rightarrow 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 P_i è il processo in starvation con la più piccola priorità p_i
- Sia P_k un processo che non ha mandato ok a P_i
- Quindi quando P_k ha ricevuto (req, p_i, i) avevamo $(p_k, k) << (p_i, i)$ altrimenti P_k avrebbe mandato ok
- Nel frattempo la priorità di k ha cambiato e P_k ha mandato ok a P_i (perché in P_k 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