COL380

Introduction to
Parallel & Distributed Programming

Agenda

- Synchronizing for Sorted List
- Parallel and Distributed Mutual exclusion
- Programming Models

- Atomic reads and writes can be implemented from nonatomic reads and writes without some pre-built facility for “mutual exclusion”
  - ➡ Eliminates circular argument

Linearizable Registers: Read ‘most recent’ write
Determined by linearization point

Can be built from:

Single Reader, Single Writer Safe Bit —
Overlapping reader sees ‘any’ value
Non-overlapping reader sees most recent write

Peterson’s Mutex Algorithm

Initially: want = {false, false}

Thread 0

1 want[0] = true
2 turn = 1

3 4

while (want[1] && turn == 1);
{// critical section
}

want[0] = false

Thread 1

a want[1] = true
b turn = 0

c d

while (want[0] && turn == 0);
{ // critical section
}

want[1] = false

Subodh Kumar

Peterson's Mutex Algorithm

Initially: want = {false, false}

Thread 0

1 want[0] = true
2 turn = 1
3 4
while (want[1] && turn == 1);
want[1]:false
Or turn:0
{// critical section
}
want[0] = false

Thread 1

a want[1] = true
b turn = 0
c d
while (want[0] && turn == 0);
want[0]:false
Or turn:1
{ // critical section
}
want[1] = false

Subodh Kumar

Peterson's Mutex Algorithm

Initially: want = {false, false}

Thread 0

1 want[0] = true
2 turn = 1
3 4
while (want[1] && turn == 1);
{// critical section
}
want[0] = false

want[1]:false
Or turn:0

Thread 1

a want[1] = true
b turn = 0
c d
while (want[0] && turn == 0);
{ // critical section
}
want[1] = false

want[0]:false
Or turn:1

Suppose: b → 2

Subodh Kumar

Peterson's Mutex Algorithm

Initially: want = {false, false}

Thread 0

1 want[0] = true

2 turn = 1

3 4

while (want[1] && turn == 1);

want[1]:false
Or turn:0

{// critical section
}

want[0] = false

Thread 1

a want[1] = true

b turn = 0

c d

while (want[0] && turn == 0);

want[0]:false
Or turn:1

{ // critical section
}

want[1] = false

Suppose: b → 2

Subodh Kumar

Peterson's Mutex Algorithm

Initially: want = {false, false}

Thread 0

1 want[0] = true
2 turn = 1
3 4
while (want[1] && turn == 1);
want[1]:false
Or turn:0
{// critical section
}
want[0] = false

Thread 1

a want[1] = true
b turn = 0
c d
while (want[0] && turn == 0);
want[0]:false
Or turn:1
{ // critical section
}
want[1] = false

Suppose: b → 2 3 → a

Subodh Kumar

Peterson's Mutex Algorithm

Initially: want = {false, false}

Thread 0

1 want[0] = true
2 turn = 1
3 4
while (want[1] && turn == 1);
{// critical section
}
want[0] = false

want[1]:false
Or turn:0

Thread 1

a want[1] = true
b turn = 0
c d
while (want[0] && turn == 0);
{ // critical section
}
want[1] = false

want[0]:false
Or turn:1

Suppose: b → 2 → 3 → a

Subodh Kumar

Peterson's Mutex Algorithm

Initially: want = {false, false}

Thread 0

```
1 want[0] = true
2 turn = 1
while (want[1] && turn == 1);
{// critical section
}
want[0] = false
```

want[1]:false
Or turn:0

Thread 1

```
a want[1] = true
b turn = 0
while (want[0] && turn == 0);
{ // critical section
}
want[1] = false
```

want[0]:false
Or turn:1

Suppose: b → 2 → 3 → a

Subodh Kumar

Peterson's Mutex Algorithm

Initially: want = {false, false}

Thread 0	**Thread 1**
1 want[0] = true	a want[1] = true
2 turn = 1	b turn = 0
3 4	c d
while (want[1] && turn == 1);	while (want[0] && turn == 0);
want[1]:false Or turn:0	want[0]:false Or turn:1
{// critical section }	{ // critical section }
want[0] = false	want[1] = false

Suppose: b → 2 → 3 → a

Subodh Kumar

Mutex w/Registers

Bakery

```
— Not Critical Section —
want [ID] = 1;
token[ID] = 1 + max(token)
want[ID] = 0;
for other != ID {
      while(want([other] == 1);
      while(token[other] > 0 && (token[other]#other) < (token[ID]#ID);
}
— Critical Section —
token[ID] = 0
```

- Mutual exclusion does not require hardware synchronization
- Peterson and Bakery use minimal number of registers

Bakery

```
— Not Critical Section —
1: want [ID] = 1;
2: token[ID] = 1 + max(token)
3: want[ID] = 0;
4: for other != ID {
5:       while(want([other] == 1);
6:       while(token[other] > 0 && (token[other]#other) < (token[ID]#ID);
7: }
— Critical Section —
8: token[ID] = 0
```

Mutex w/Registers

- Mutual exclusion does not require hardware synchronization
- Peterson and Bakery use minimal number of registers
  - ➡ Still too many? (and ever increasing counter values)

Bakery

```
— Not Critical Section —
want [ID] = 1;
token[ID] = 1 + max(token)
want[ID] = 0;
for other != ID {
      while(want([other] == 1);
      while(token[other] > 0 && (token[other]#other) < (token[ID]#ID);
}
— Critical Section —
token[ID] = 0
```

Non-shared Logical Clock

- Each entity maintains a counter
    - ➡ increments every *step*, at its own pace
- Interaction between entities is through messages
    - ➡ Data + counter
- On message receipt:
    - ➡ If recipient counter < received count
        - Increase local counter to received count
        - Receive is also a '*step*,' so increment by one

[Lamport's Timestamp algorithm]

Subodh Kumar

Request Critical Section:

Broadcast **R** = <request, time(entity)>

Add **R** to local-queue(entity)

Enter Critical section (**R**)

R has the lowest timestamp in local-queue. AND.

Have received some **m** from every other entity with **m**.Time > **R**.time

---

Exit Critical section (**R**):

Remove **R** from local-queue

Broadcast <release> message to all

Distributed Mutex

Request Critical Section:

Broadcast **R** = <request, time(entity)>

Add **R** to local-queue(entity)

Enter Critical section (**R**)

R has the lowest timestamp in local-queue. AND.

Have received some **m** from every other entity with **m**.Time > **R**.time

---

Exit Critical section (**R**):

Remove **R** from local-queue

Broadcast <release> message to all

Receive **R**

update(time(entity))

if(type == request)

Add **R** to local-queue

Reply <ack, time(entity)>

if(type == release)

Remove **R** from local-queue

Subodh Kumar

Review

- Shared memory distributed synchronization
- Message passing distributed synchronization