Mutual Exclusion

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 10

Mutual exclusion

- Concurrent update of a shared variable can lead to data inconsistenccy
  - Race condition
- Control behaviour of threads to regulate concurrent updates
  - Critical sections — sections of code where shared variables are updated
  - Mutual exclusion — at most one thread at a time can be in a critical section

Mutual exclusion for two processes

- First attempt

```
Thread 1
...
while (turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 2;
...
```

```
Thread 2
...
while (turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 1;
...
```

Mutual exclusion for two processes

- First attempt

```
Thread 1
...
while (turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 2;
...
```

```
Thread 2
...
while (turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 1;
...
```

- Shared variable `turn` — no assumption about initial value, atomic update

Mutual exclusion for two processes

- First attempt

```
Thread 1
...
while (turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 2;
...
```

```
Thread 2
...
while (turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 1;
...
```

- Shared variable `turn` — no assumption about initial value, atomic update
- Mutually exclusive access is guaranteed ...

Mutual exclusion for two processes

- First attempt

```
Thread 1                          Thread 2
...                               ...
while (turn != 1){                while (turn != 2){
  // "Busy" wait                    // "Busy" wait
}                                 }
// Enter critical section         // Enter critical section
   ...                               ...
// Leave critical section         // Leave critical section
turn = 2;                         turn = 1;
...                               ...
```

- Shared variable `turn` — no assumption about initial value, atomic update
- Mutually exclusive access is guaranteed ...
- ... but one thread is locked out permanently if other thread shuts down
  - Starvation!

Mutual exclusion for two processes . . .

- Second attempt

```
Thread 1
...
request_1 = true;
while (request_2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2
...
request_2 = true;
while (request_1)
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

Mutual exclusion for two processes . . .

- Second attempt

```
Thread 1
...
request_1 = true;
while (request_2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2
...
request_2 = true;
while (request_1)
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

- Mutually exclusive access is guaranteed . . .

Mutual exclusion for two processes . . .

- Second attempt

```
Thread 1
...
request_1 = true;
while (request_2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2
...
request_2 = true;
while (request_1)
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

- Mutually exclusive access is guaranteed . . .
- . . . but if both threads try simultaneously, they block each other
  - Deadlock!

Peterson's algorithm

```
Thread 1
...
request_1 = true;
turn = 2;
while (request_2 &&
       turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2
...
request_2 = true;
turn = 1;
while (request_1 &&
       turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

- Combines the previous two approaches

Peterson's algorithm

```
Thread 1
...
request_1 = true;
turn = 2;
while (request_2 &&
       turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2
...
request_2 = true;
turn = 1;
while (request_1 &&
       turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

- Combines the previous two approaches
- If both try simultaneously, `turn` decides who goes through

Peterson's algorithm

```
Thread 1
...
request_1 = true;
turn = 2;
while (request_2 &&
       turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2
...
request_2 = true;
turn = 1;
while (request_1 &&
       turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

- Combines the previous two approaches
- If both try simultaneously, `turn` decides who goes through
- If only one is alive, `request` for that process is stuck at false and `turn` is irrelevant

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial
- For n process mutual exclusion other solutions exist

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial
- For n process mutual exclusion other solutions exist
- Lamport's Bakery Algorithm
  - Each new process picks up a token (increments a counter) that is larger than all waiting processes
  - Lowest token number gets served next
  - Still need to break ties — token counter is not atomic

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial
- For n process mutual exclusion other solutions exist
- Lamport's Bakery Algorithm
  - Each new process picks up a token (increments a counter) that is larger than all waiting processes
  - Lowest token number gets served next
  - Still need to break ties — token counter is not atomic
- Need specific clever solutions for different situations

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial
- For n process mutual exclusion other solutions exist
- Lamport's Bakery Algorithm
  - Each new process picks up a token (increments a counter) that is larger than all waiting processes
  - Lowest token number gets served next
  - Still need to break ties — token counter is not atomic
- Need specific clever solutions for different situations
- Need to argue correctness in each case

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial
- For n process mutual exclusion other solutions exist
- Lamport's Bakery Algorithm
  - Each new process picks up a token (increments a counter) that is larger than all waiting processes
  - Lowest token number gets served next
  - Still need to break ties — token counter is not atomic
- Need specific clever solutions for different situations
- Need to argue correctness in each case
- Instead, provide higher level support in programming language for synchronization

Summary

- We can construct protocols that guarantee mutual exclusion to critical sections
  - Watch out for starvation and deadlock
- These protocols cleverly use regular variables
  - No assumptions about initial values, atomicity of updates
- Difficult to generalize such protocols to arbitrary situations
- Look to programming language for features that control synchronization