Mutual Exclusion in Shared Memory

Arvind Krishnamurthy
Fall 2003

Shared Memory Model

- Two alternatives for concurrent systems to communicate:
  - Message passing
  - Shared memory
- Many issues to consider in choosing one of the two forms of communication:
  - Underlying hardware
  - Need for asynchronous interactions
  - Need for isolation (or fault-tolerance)

Attiya-Welch model

- Variables have a type:
  - determines what operations can be performed
  - Determines what values can be stored
- No channels
- Configuration: processor states and state of shared memory
- Only event type is a computation step
  - Processor's old state specifies what shared variable will be accessed and what operation will be performed
  - When operation is done:
    - Variable's value changes
    - Processor enters new state
- Admissible execution: every processor takes an infinite number of steps

Canonical Issue: Mutual Exclusion

- Assume that each processor is executing:
  entry (synchronize to enter critical section code)
  critical section code
  exit
  remaining non-critical code
- Mutual exclusion: at most one processor is executing critical section at any point
  - Assume: processor cannot be in critical section for ever
- Properties to enforce:
  - No deadlock: if a processor is in its entry section, then later some processor is in its critical section
  - No lockout: if a processor is in its entry section, then later the *same* processor is in its critical section
  - Bounded waiting: no lockout + while processor is waiting, others enter the critical section only a bounded number of times

Mutual Exclusion

- Main complexity measure of interest for shared memory mutual exclusion algorithms:
  - Number of shared variables
  - Size of each shared variable
- Influenced by:
  - How powerful is the type of the shared variables
  - How strong is the liveness condition to be satisfied
- We will consider two types of shared variables:
  - Weaker type: only reads and writes can be performed
  - Stronger types: allows for atomic read-modify-write

Mutual Exclusion Using Test-and-Set

- A test-and-set variable holds two values: 0 or 1
- Supports the following operations:
  - test&set(V):
```
temp = V;
V = 1;
return temp;
```
  - reset(V):
```
V = 0;
```
- Mutual exclusion using one test&set variable:
  - Entry: while (test&set(V) == 1);
  - Exit: reset(V);
- Guarantees mutual exclusion, no deadlock. Lockout possible

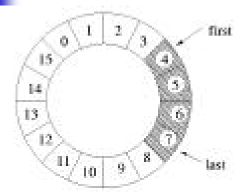
Mutual Exclusion using read-modify-write

- Read-modify-write variables: more powerful that test&set
- Supports the following general operation:
  - RMW(V, f):

```
temp = V;
V = f(V);
return temp;
```

- Mutual exclusion using one RMW variable:
  - Conceptually, the list of waiting processors is stored in a circular queue of length "n"
  - Each waiting processor remembers in its local state:
    - Its location in the queue
    - Does not need a shared variable for this purpose

Mutual exclusion using RMW (contd.)



- Shared variable: "first"
- RMW variable: "last"

- Entering critical section:

```
f(V) = increment V modulo n;
Entry:   temp = RMW(V, f);  while (first != temp);
```

- Exit critical section:

```
first++;
```

Analysis

- Satisfies properties:
  - Provides mutual exclusion
  - n-Bounded wait
- Space complexity:
  - Two shared variables: each O(log n) bits wide
  - Different values these two variables can take: n^2
- Lower bound result:
  - If you want k-bounded waiting, then there must be at least "n" states of shared memory

Mutual Exclusion using Read/Write Variables

- Suppose that the shared variables are of read/write type:
  - Processors can atomically read or write each variable but not both
- Bakery algorithm for mutual exclusion:
  - Provides no-lockout property
  - Uses 2n shared variables
- Variables:
  - choosing[i]: initially 0, written by p_i, read by others
  - number[i]: initially 0, written by p_i, read by others
  - No concurrent writes by two processors to the same variable

Bakery Algorithm

```
entry:
        choosing[i] = 1;
        number[i] = max(number[0], … , number[n-1]) + 1;
        choosing[i] = 0;
        for j=0 to n-1 (except i) do:
                wait until choosing[j] == 0;
                wait until number[j] == 0 or
                        (number[j], j)  >  (number[i], i);

exit:
        number[i] = 0;
```

- choosing[i]: processor i is choosing a number
- number[i] = 0 implies that processor i is in remainder code
- number[i] != 0 implies that processor i is either in critical section or at the entry point

Analysis of Bakery Algorithm

- Useful to thing of entry code to consist of:
  - Compute maximum + 1
  - Write my number
  - Wait till my number is lowest
- How does one prove that this provides mutual exclusion?
  - Will it still work if "choosing" is eliminated?
- Algorithm provides n-bounded waiting
- What drawbacks does this algorithm have?

Summary so far

- Special variables:
  - One test-and-set variable sufficient for mutual exclusion
    - But does not provide “no lockout” property
  - One read-modify-write variable + one shared variable sufficient to provide “no lockout + bounded-wait”
- Read-write variables:
  - Bakery algorithm: uses “n” read-write variables
    - Provides mutual-exclusion and bounded-wait
    - Counters could grow arbitrarily
- Today: read-write variables where values do not grow arbitrarily
  - Version 1: asymmetric two-processor code
  - Version 2: symmetric two-processor code
  - Version 3: symmetric n-processor code

Mutual exclusion for two processors

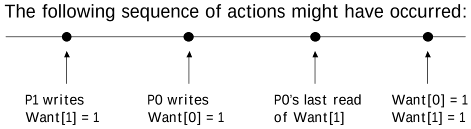
- Give priority to one processor:
  - First processor checks whether second processor has the lock
  - Second processor checks whether first processor has the lock or wants the lock

```
Processor 0

entry:


   want[0] = 1;


   while (want[1]);

exit:
   want[0] = 0;
```

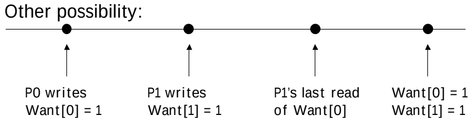
```
Processor 1

entry:
   want[1] = 0;
   while (want[0]);
   want[1] = 1;

   if (want[0]) goto entry;


exit:
   want[1] = 0;
```

Mutual Exclusion

- Suppose in contradiction that p0 and p1 are simultaneously critical
  - Which implies that want[0] = want[1] = 1
- The following sequence of actions might have occurred:



- Other possibility:



Mutual exclusion w/o bias

- Uses three binary shared variables:
  - Want[0] written by p0 and read by p1, initially 0
  - Want[1] written by p1 and read by p0, initially 0
  - Priority: written and read by both, initially 0

```
Processor 0

entry:
   want[0] = 0;
   while (want[1] && Priority == 1);
   want[0] = 1;
   if (Priority == 1)
      if (want[1]) goto entry;
   else
      while (want[1]);

exit:
   Priority = 1;
   want[0] = 0;
```

```
Processor 1

entry:
   want[1] = 0;
   while (want[0] && Priority == 0);
   want[1] = 1;
   if (Priority == 0)
      if (want[0]) goto entry;
   else
      while (want[0]);

exit:
   Priority = 0;
   want[1] = 0;
```

Correctness

- If one is performing “Enter” while the other does not have the lock and does not want the lock
  - Trivially falls through
- If both processors “Enter” at the same time:
  - They see the same value of Priority
  - Correctness follows from the biased version
- If one is performing “Enter” while the other has the lock:
  - Assume processor 1 has the lock
  - Priority could be either 0 or 1
    - If Priority is 0, simply wait in the last while statement
    - If Priority is 1, wait in the first while loop and then fall into the second while loop, and eventually get the lock

Generalizing Mutual Exclusion

- How do you generalize to more than 2 processors?

Other Mutual Exclusion Results

- Lower bound on number of read-write variables required to provide mutual exclusion: O(n)

- Fast mutual exclusion algorithm:
  - Reads O(1) variables if no contention
  - If contention, defaults to a traditional algorithm that reads O(n) variables