Group Mutual Exclusion (GME) Algorithms

-- Implementation of the local-spin GME and the space-efficient FCFS GME

By Carrie Chu   April, 2009

Problem review

- A process requests a "session".
- Processes requesting the same session can be in CS simultaneously.
- Processes requesting different sessions can not.
- A group mutual exclusion process:

repeat

    NCS: sleep(5)

    Try section

    CS: sleep(5)

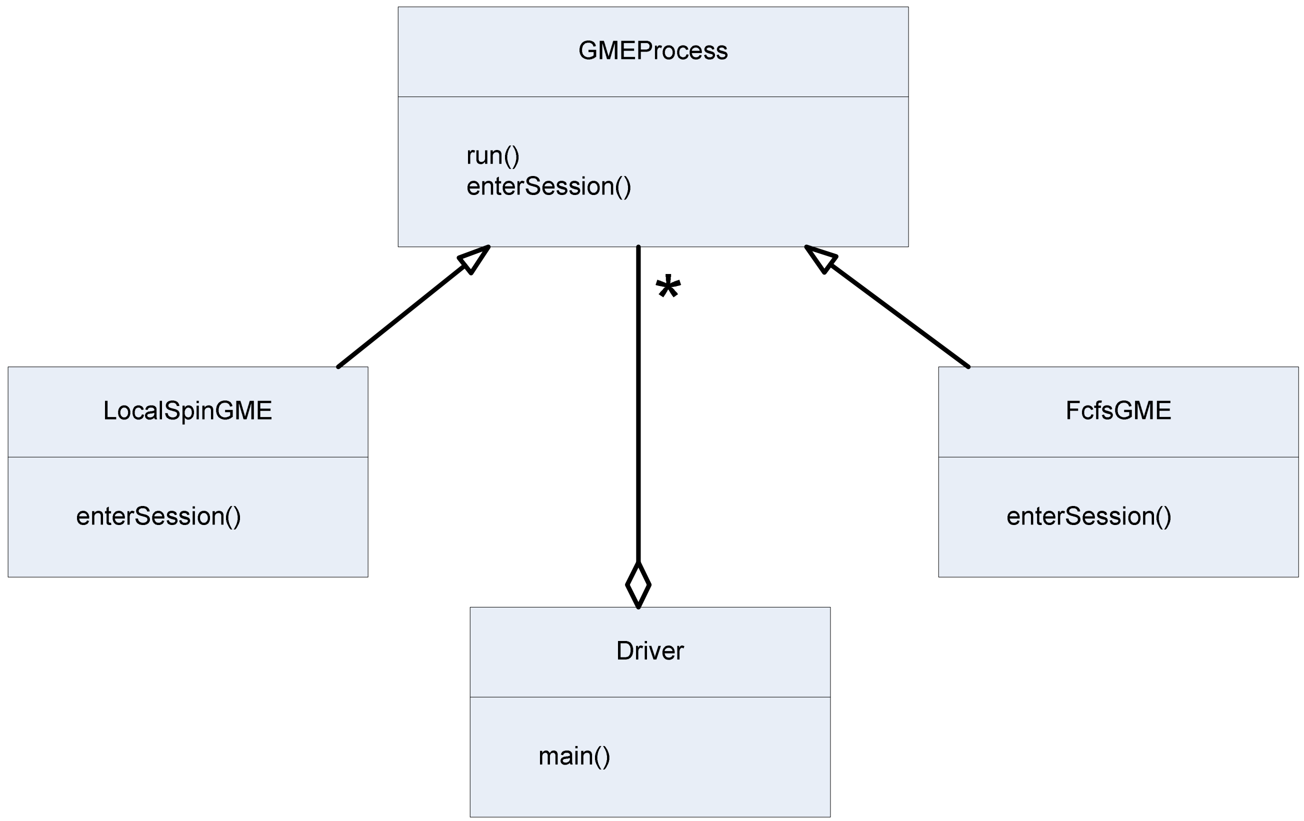
    Exit section

forever

Two GME algorithms

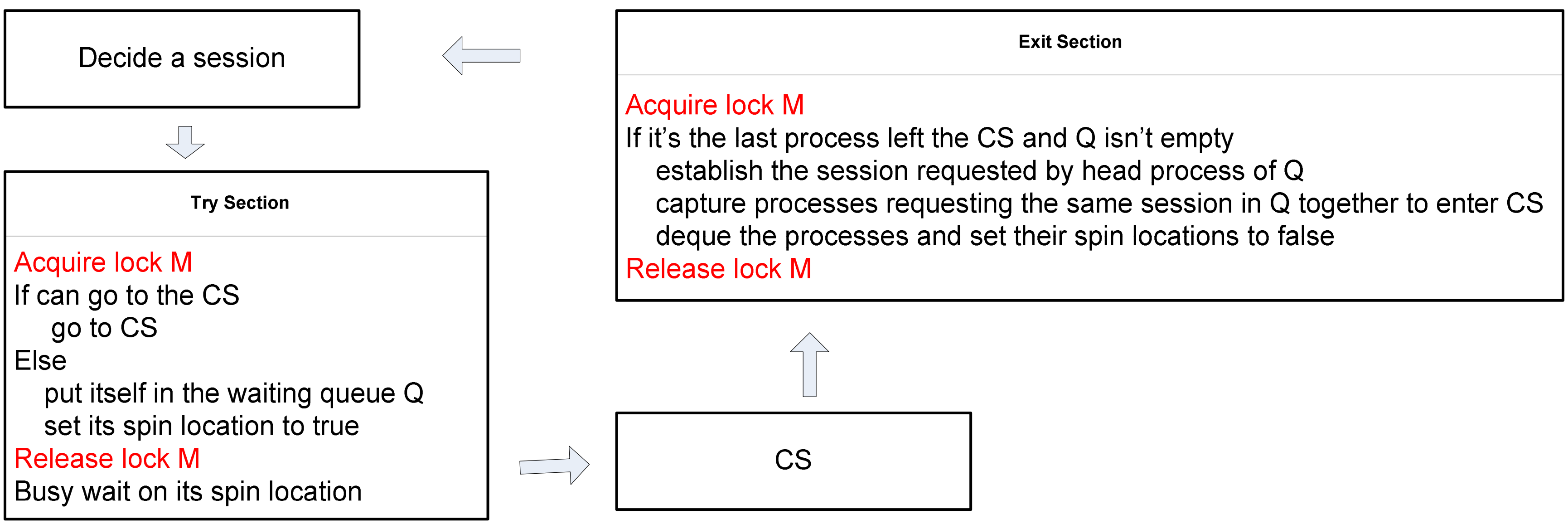
- Patrick Keane and Mark Moir. A simple local-spin group mutual exclusion algorithm. In *Proceedings of the 18th annual ACM Symposium on Principles of Distributed Computing*, pages 23-32, Atlanta, Georgia, United States, 1999. ACM.
- Srdjan Petrovic. Space-efficient FCFS group mutual exclusion. *Information Processing Letters*, 95(2): 343-350, July 2005.

Program structure



Algorithm 1: local-spin GME(1)

Each process does:



Algorithm 1: local-spin GME(3)

```
public class LocalSpinGME extends GMEProcess {
    private static final Semaphore s_lock = new Semaphore(1);
    private static final ArrayList<Thread> s_queue = new ArrayList<Thread>();
    private boolean m_wait;

    protected void enterSession() {
        // Try section
        s_lock.acquire();
        …
        s_lock.release();

        while(m_wait) {
            sleep(1)
        }
        …
    }
}
```

Algorithm 2: space-efficient FCFS GME(1)

- Shared variables are owned by each process, each of which has a single writer (its owner) and multiple readers.
- It doesn’t use lock, semaphore, compare-and-swap, compare-and-set atomic mechanisms.
- Think about “bakery algorithm”.
- It satisfies property FCFS.
- Modular composition of two parts: FCFS+ME

Algorithm 2: space-efficient FCFS GME(2)

Each process does:

Decide a session

Try Section

FCFS: the order to enter CS depends on the order in which processes complete the doorway

ME: high-numbered processes let low-numbered processes proceed first

CS

Exit Section

ME
FCFS

- The code is sequential with busy wait loops.

Algorithm 2: space-efficient FCFS GME(3)

```
public class FcfsGME extends GMEProcess {
   private int m_turn;
   private boolean m_compting;

   protected void enterSession() {
       fcfs();
      mutualExclusion();
      …
   }

   private void fcfs() {
      …
      while(…) {
         sleep(1)
      }
      …
   }
}
```

Test (1)

- Two ways
  - Create threads with fixed session numbers.
  - Create threads with randomly assigned session numbers.
- The test tuned the number of threads, sessions and iterations to produce different cases.

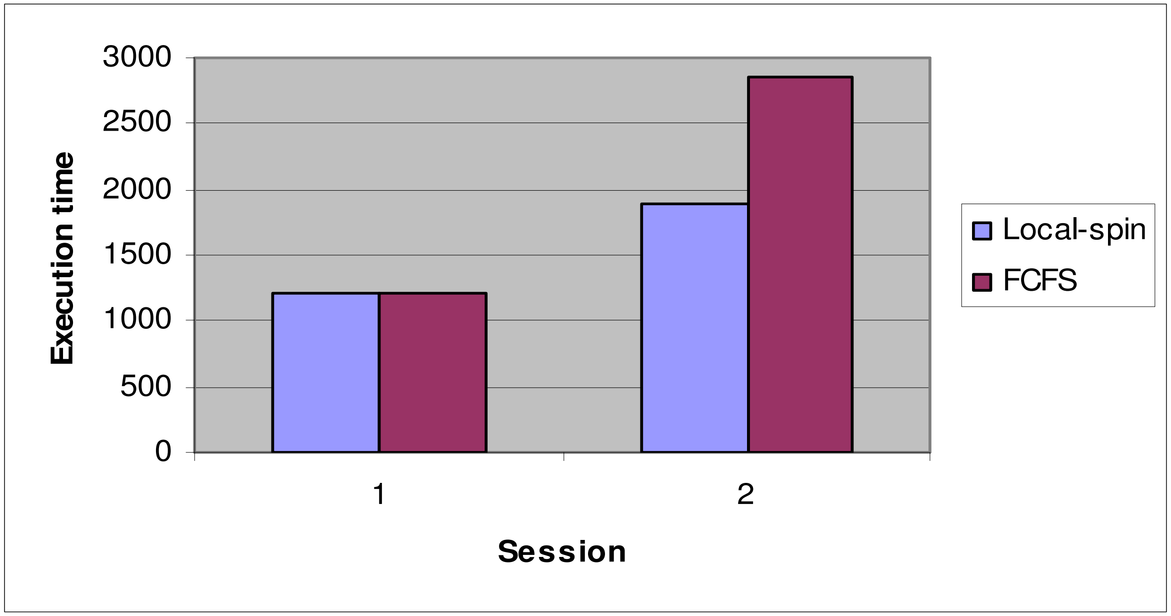
Test(2)

Process	i	j	k	l	m	n
Session	s1	s1	s2	s2	s1	s2

- The test is able to produce the expected results for both algorithms.
- The test didn’t find cases that violate ME.

Performance comparison (1)

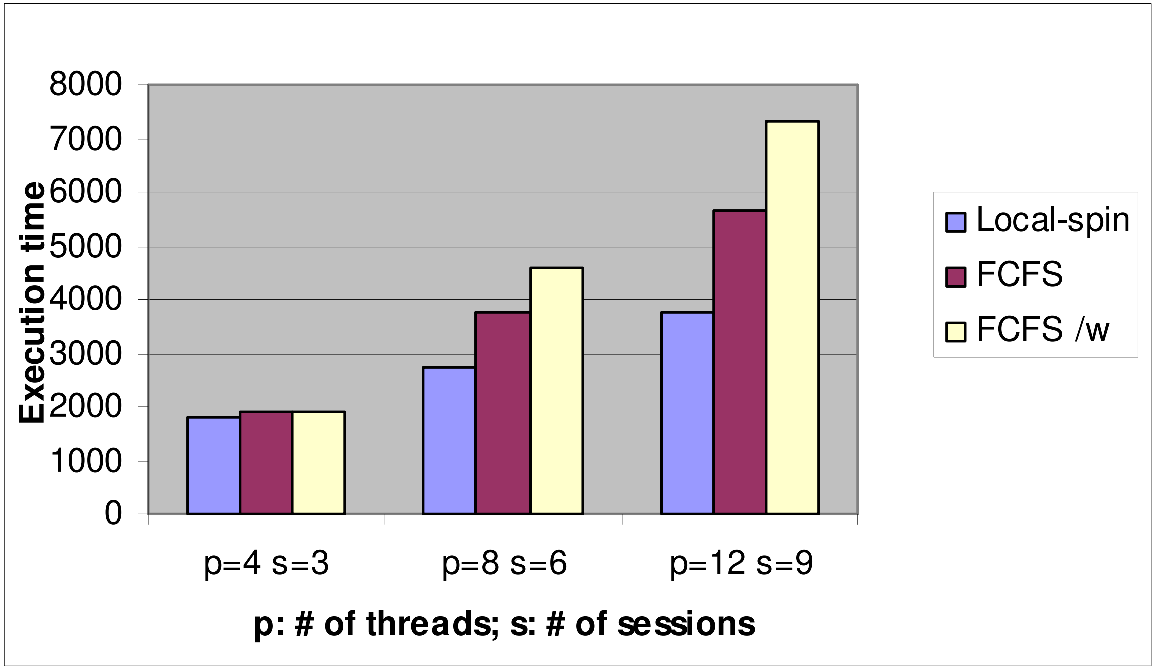
8 processes, 100 iterations on navy:



- When # of session =1, execution time is almost the same. Lock doesn’t create much overhead.
- When # of session =2, FCFS has more session switch costs.

Performance comparison (2)

100 iterations on navy:



- Local spin algorithm takes less time than FCFS algorithm, even comparing with FCFS algorithm without FCFS code.

Looking ahead

- Further verify ME property for both algorithms
- Verify FCFS property for the space-efficient algorithm
- Verify deadlock solution for the space-efficient algorithm

Questions?

```
shared variables
    M: lock;  Session, Num: integer;  Q: queue of 0..N-1;
    Wait: array [0..N-1] of boolean;  Need: array [0..N-1] of integer
local variables
    t, v: integer;
initially
    Num = 0 ∧ Session = 1 ∧ Q = ∅
```

0: t=

```
1:   Wait[p] := false;
2:   Need[p] := t;
3:   Acquire(M);
4:   if Session = t ∧ Q = ∅ then
5:       Num := Num+1
6:   else if Session ≠ t ∧ Num = 0 then
7:       Session := t;
8:       Num := 1
     else
9:       Wait[p] := true;
10:      Enqueue(Q, p)
     fi;
11:  Release(M);
12:  while Wait[p] do od;
```

Try section

<<Attend session t>>

```
13:  Acquire(M);
14:  Num := Num-1;
15:  if Q ≠ ∅ ∧ Num = 0 then
16:      Session:= Need[Head(Q)];
17:      for each v ∈ Q do
18:          if Need[v] = Session then
19:              Delete(Q, v);
20:              Num := Num+1;
21:              Wait[v] := false
     fi od fi;
22:  Release(M)
```

23: go to 0

Exit section

A simple local-spin group mutual exclusion algorithm. Code is shown for process p.

Shared variables for each $i \in \{1, 2, \ldots, N\}$

```
  session_i : integer
  turn_i : {0, 1, ..., 11}
  competing_i : boolean
Local variables
  turn_snap: array [1 ... N] of {0, 1, ..., 11}

repeat
    Remainder Section
    session_i = mysession
    for j = 1 to N do turn_snap[j] = turn_j
    if conflict(mysession)
        turn_i = (turn_i + 1) mod 12
    for j = 1 to N do
        wait until (session_j ∈ {0, mysession})
                   ∨ (turn_snap[j] ≠ turn_j)
 L: competing_i = true
    for j = 1 to i − 1 do
        if competing_j ∧ (session_j ∉ {0, mysession})
            competing_i = false
            wait until (¬competing_j)
                       ∨ (session_j ∈ {0, mysession})
            go to L
    for j = i + 1 to N do
        wait until (¬competing_j)
                   ∨ (session_j ∈ {0, mysession})
    CS
    competing_i = false
    session_i = 0
forever
```

Lines 2–7 → **FCFS**; lines L–15 → **ME**; lines 2–15 → **Try section**

Line 17 → **ME**; line 18 → **FCFS**; lines 17–18 → **Exit section**

Space efficient FCFS algorithm – code for process i