Group Mutual Exclusion (GME) Algorithms

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

By Carrie Chu     May, 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


Algorithm review (1)

- **Local-spin algorithm:**
  - Use an exclusive lock to protect access to other shared variables
  - The lock is implemented by a semaphore.

```java
Semaphore s_lock = new Semaphore(1);
s_lock.acquire();
// access shared variables
s_lock.release();
```
Algorithm review (2)

A process in local-spin algorithm does:

**Try Section**

- Acquire lock \( s \_\text{lock} \)
- Check if can go to CS
- Release lock \( s \_\text{lock} \)
- Busy wait if can not enter CS

**Exit Section**

- Acquire lock \( s \_\text{lock} \)
  - If it’s the last process left the CS and there are still processes waiting to enter CS, establish a new session
- Release lock \( s \_\text{lock} \)

**CS**
Algorithm review (3)

- Space-efficient FCFS algorithm
  - It doesn’t use lock, semaphore, compare-and-swap, compare-and-set atomic mechanisms
    - the code is sequential with some busy waits.
  - Shared variables are owned by each process, each of which has a single writer (its owner) and multiple readers.
    - Shared variables are implemented as private attributes in a process object, with only public read methods.
  - It satisfies property FCFS.
  - Modular composition of two parts: FCFS+ME
Verifications

- Local-spin algorithm
  - group ME property
  - use of lock is essential to ensure group ME

- Space-efficient FCFS algorithm
  - group ME property
  - some codes are essential to avoid deadlock
  - data race
Local-spin algorithm verification
- group ME property (1)

Available shared variables:
  s_session: int // current session established in CS
  m_need: int    // session of the thread

In CS:
  assert s_session == m_need;
sleep(5);
  assert s_session == m_need;
Local-spin algorithm verification
- group ME property (2)

iterations=1; threads =3; sessions = 2

<table>
<thead>
<tr>
<th>Search</th>
<th>Time</th>
<th>Memory</th>
<th>States</th>
<th>Result</th>
</tr>
</thead>
</table>
| DFS    | 5:20:22| 1648M  | 43905194    | Completed:
|        |        |        |             | No errors detected          |

- Local-spin algorithm satisfies group ME property
Local-spin algorithm verification

- lock

Comment out lock acquire() and release()

iterations=1; threads =3; sessions = 2

<table>
<thead>
<tr>
<th>Search</th>
<th>Time</th>
<th>Memory</th>
<th>States</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>0:11:34</td>
<td>22M</td>
<td>220</td>
<td>NOT completed: out of memory No errors detected</td>
</tr>
<tr>
<td>BFS</td>
<td>0:00:19</td>
<td>220M</td>
<td>44510</td>
<td>Assertion error</td>
</tr>
</tbody>
</table>

Lock is essential to ensure group ME
Space-efficient FCFS algorithm verification
- group ME property (1)

Available shared variables:

m_need: int  // session of the thread

Added shared variables (used only in CS):

s_session: int  // current session established in CS
s_num: int  // number of threads in CS
s_lock: new Semaphore (1)
Space-efficient FCFS algorithm verification - group ME property (2)

In CS:

```c
s_lock.acquire();
s_num++;
if (s_num == 1)
    s_session = m_need;
s_lock.release();
assert s_session == m_need;
sleep(5);
assert s_session == m_need;
s_lock.acquire();
s_num--;
s_lock.release();
```
Space-efficient FCFS algorithm verification
- group ME property (3)

iterations=1; threads =3; sessions = 2

<table>
<thead>
<tr>
<th>Search</th>
<th>Time</th>
<th>Memory</th>
<th>States</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>8:17:17</td>
<td>2030M</td>
<td>65681631</td>
<td>NOT completed: out of memory No errors detected</td>
</tr>
<tr>
<td>BFS</td>
<td>0:16:18</td>
<td>2408M</td>
<td>1253171</td>
<td>NOT completed: out of memory No errors detected</td>
</tr>
</tbody>
</table>

- For DFS, JPF created more states to verify FCFS algorithm (local-spin algorithm: 43905194 states).
- All the results we obtained don’t show this algorithm violates group ME property.
Space-efficient FCFS algorithm verification - deadlock (1)

- FCFS part review
  - FCFS property: $i$ would block on $j$ if $j$ completes the doorway before $i$.
  - How does it ensure FCFS property?
    - $\textit{turn}$ variable vs. its local copy $\textit{turn\_snap}$
    1. Doorway starts - $i$ reads all other processes’ $\textit{turn}$ and make a local copy of them $\textit{turn\_snap}$
    2. $i$ possibly increments its $\textit{turn}$ – doorway ends
    3. $i$ checks $\textit{turn}[j] \neq \textit{turn\_snap}[j]$
Space-efficient FCFS algorithm verification - deadlock (2)

- A possible deadlock could occur when:
  1. A fast process $j$, requesting same session as a slow process $i$, enters CS repeatedly, each time over-passing $i$ in the doorway and increments its $turn$ variable;
  2. $i$ falls asleep after exiting the doorway;
  3. The over-passing happens enough times, $turn[j]$ wraps back to the value $i$ read.
  4. The $i$ wakes up and $j$ then requests a different session.
Space-efficient FCFS algorithm verification - deadlock (3)

Though the deadlock is easily produced by java, jpf seems can’t detect such deadlocks directly.
Space-efficient FCFS algorithm verification - data race

- turn variable clearly has race: FCFS property is ensured by checking the order of turn read and increment.
- JPF can detect the race:

```
gov.nasa.jpf.tools.PreciseRaceDetector
race for: "int FcfsGME.m_turn"
Thread-0 at FcfsGME.fcfs(FcfsGME.java:69)
  "(FcfsGME.java:69)" : putfield
Thread-1 at FcfsGME.getTurn(FcfsGME.java:24)
  "(FcfsGME.java:24)" : getfield
```

```
Questions?