Peterson's Mutual Exclusion Algorithm

Mark Greenstreet

CpSc 418 – Nov. 14, 2013

Lecture Outline

This is a draft version of the slides.

- Mutual Exclusion
- Peterson's algorithm
- Proving Peterson's algorithm correct
- Mutual Exclusion in the real world.

Mutual Exclusion: Usage

- A mutual exclusion algorithm provides two operations:
  - `lock(threadId)`: the thread specified by `threadId` acquires the lock.
  - `unlock(threadId)`: the thread specified by `threadId` releases the lock.
- Usage:
  - Initially, no thread has the lock.
  - If a thread does not have the lock, it may call `lock(threadId)`.
  - When `lock(threadId)` returns, the thread specified by `threadId` has the lock.
  - If a thread has the lock it must eventually call `unlock(threadId)`.
  - When `unlock(threadId)` returns, the thread specified by `threadId` no longer has the lock.
  - It is an error:
    - To call `lock(threadId)` if the thread already has the lock.
    - To call `unlock(threadId)` if the thread does not have the lock.

Mutual Exclusion: Guarantees

- A correct mutual exclusion algorithm guarantees:
  - Mutual exclusion: at most one thread has the lock at any time.
  - No Deadlock: if one or more threads have requested the lock, some thread will eventually receive the lock.
  - No Starvation: if a thread requests the lock, it will eventually acquire it.
- A few notes:
  - Even if the mutual-exclusion algorithm is deadlock free, a program may deadlock, e.g. cycles of locks.
  - Freedom from starvation is nice, but there are practical algorithms that don't guarantee it on the basis that starvation is highly unlikely and not worth adding complexity to the implementation.
  - Some algorithms provide other features or guarantees:
    - E.g. "first-come, first-served".
    - Offer a "nacking" lock – instead of blocking, the lock function just returns `true` to indicate the lock was granted, and `false` to indicate that some other thread has/had the lock.

Peterson's Mutual Exclusion Algorithm

```
% shared variables:
bool flag[2] = {false, false};
int victim = 0;

lock(myId) {
  int otherId = 1 - myId; % know your neighbour
  flag[myId] = true;      % express intent to lock
  victim = myId;          % you go first, please
  while(flag[otherId] && (victim == myId)); % spin
}

unlock(myId) {
  flag[myId] = false;
}
```

The Peterson Principle

- When a thread tries to acquire the lock, it gives priority to the other thread before spinning.
- If both threads try to acquire the lock at roughly the same time, then the last one to set `victim` defers to the other thread.
- Consider a few executions:
  - Thread 0 acquires the lock without contention; then thread 1 requests the lock; then thread 0 releases the lock.
  - Thread 0 sets its flag; thread 1 sets its flag; thread 0 proceeds to spin; thread 1 proceeds to spin. Who gets the lock?
  - Think of your own example.

Proving Peterson Correct: Thread "states"

- Each thread cycles through the following four states in the order below:
  - Idle: Both threads are initially idle. Furthermore, they return to the idle state at line 14 of `unlock`. The idle state include the non-critical section code of the thread. The idle state continues to line 7 of `lock`.
  - Entering: Line 8 of `lock`.
  - Spinning: Line 9 of `lock`.
  - Critical: Starting at line 10 of `lock`, the critical section for the thread, and continuing to line 13 of `unlock`.
- I'll write `state(threadId)` to indicate the current "state" of the given thread.
- Note: when we say that a thread is at line L, that means that execution has reached line L, but no actions for line L have been performed.

Proving Peterson Correct: Mutual Exclusion – why?

- Why does Peterson's algorithm guarantee mutual exclusion?
  - What does `flag[id]` tell us?
    - Hint: think about the relation between `flag[id]`, and `state(id)`.
  - What is the role of `victim`?
    - What if one thread is spinning and the other is in its critical region?
    - What if both threads are spinning?

Proving Peterson Correct: Mutual Exclusion – the proof

- Write the previous observations as an invariant:
- Show that each operation of the algorithm preserves the invariant.
- Show that the invariant guarantees mutual exclusion

Proving Peterson Correct: Deadlock Freedom – why

- Both threads can proceed to the `Spin` state without being blocked by the other.
- Need to show that if one or both threads are spinning, then eventually, some thread enters its critical region.
- Why must some thread eventually be in its critical region?
  1. Assume thread 0 is spinning.
  2. Think about what happens for the various states that thread 1 could be in.

Peterson is Deadlock Free – the proof

Assume thread 0 is spinning:

- case thread 1 is idle:
- case thread 1 is entering:
- case thread 1 is spinning:
- case thread 1 is critical:

Proving Peterson Correct: No Starvation – why

- If thread 0 is not idle, what is the longest sequence of state transitions by threads 0 and 1 before thread 0 enters its critical region?
- Construct a function based on the states of threads 0 and 1 and the value of `victim` that gives the maximum number of state transitions remaining until thread 0 will enter its critical region.
- Why must each step be taken?
- Note 1: I would never ask you to prove starvation freedom on anything for credit in this class.
- Note 2: This is why tools like PReach are great: they automate all of these proofs!

Mutual exclusion with more than two threads

- Peterson's algorithm generalizes to any number of threads.
- The algorithm is called a "filter lock".
- One flag variable per thread,
- and an array of $N - 1$ victim variables.

Mutual exclusion with more than two threads

- Why so many variables:
  - The Bakery algorithm also uses N shared variables for N-way mutual exclusion.
  - Can show that $N - 1$ variables are required to guarantee mutual-exclusion if the only atomic operations are individual reads and writes.
  - This is why real processors have "compare-and-set" (or similar) operations.

Some Performance Experiments (I hope)

The rest of the course

- Nov. 19: Mesh sorting, and distributed Erlang
- Nov. 21: GPUs
- Nov. 26: Map Reduce
- Nov. 28: The future, or my research, or course review, or . . .

Review