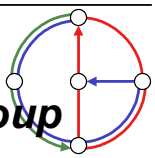
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group



HS 2009 Prof. Dr. Roger Wattenhofer, Thomas Locher, Remo Meier, Benjamin Sigg

Distributed Systems
Solution 5

Assigned: December 4, 2009
Discussion: December 11, 2009

1 Spin Locks

A read-write lock is a lock that allows either multiple processes to read some resource, or one process to write some resource.

a) Write a simple read-write lock using only spinning, one shared integer and the CAS operation. Do not use local variables (it is ok to have variable within a method, but not outside).

b) What is the problem with your lock?

Hint: what happens if a lot of processes access the lock repeatedly?

We now build a queue lock using only spinning, one local integer per process and the CAS operation.

c) To prepare for this task, answer the following questions:

i) Head and tail of the queue have to be stored in the shared integer. What is the "head" and the "tail", and how can they be stored in one integer?

Hint: could the head be a process id? Or is there a much easier solution?

ii) How could a process add itself to the queue?

Hint: you need the local integer of the process for this operation.

iii) When has a process acquired the lock?

iv) How does a process release the lock?

d) Write down the lock using pseudo-code. Do not forget to initialize all variables.

Solution

a) We use the shared integer `state` to indicate the state of the lock. The lock is free if `state` is 0. The lock is in write mode if `state` is -1. And it is in read-mode if `state` is n, with $n > 0$.

```
// the shared integer
int state = 0;

// acquire the lock for a read operation
read_lock(){
  while( true ){
    int value = state.read();
    if( value >= 0 ){
      if( state.CAS( value, value+1 ) == value ){
        // lock acquired
        return;
      }
    }
  }
}

// release the lock
read_unlock(){
  while( state.CAS( state, state-1 ) != state );
}


// acquire the lock for a write operation
write_lock(){
  while( true ){
    int value = state;
    if( state == 0 ){
      if( state.CAS( 0, -1 ) == 0 ){
        // lock acquired
        return;
      }
    }
  }
}

// release the lock
write_unlock(){
  // no need to test, no other process can call this at
  // the same time.
  state.CAS( -1, 0 );
}
```

b) Starvation is a problem. Example: if many processes constantly acquire and release the

read-lock, then the `state` variable always remains bigger than 0. If one process wants to acquire the write-lock, it will never get the chance.

c) The basic idea behind this lock is a ticketing service as can be found in swiss post offices.

i) The tail is the ticket which can be drawn by the next process. The head denotes the ticket which can acquire the lock. If we assume an integer consists of 32 bits, then we can use the first 16 bits for the head, and the last 16 bits for the tail.

ii) The process reads the value of the tail, and then increments the tail. This should of course happen in a secure way, i.e. no two processes have the same ticket.

iii) When its ticket equals the head.

iv) The process increments the head by one.

d)
```
// the shared integer containing head|tail
shared int queue = 0;

// the ticket of this process
int local = 0;

// acquire the lock
lock(){
  // 1. add this process to the queue
  local = add();
  // 2. wait until the lock is acquired
  while( head() != local() );
}

// add this process to the queue
int add(){
  while( true ){
    int value = queue.read();
    if( queue.CAS( value, value+1 ) == value ){
      return value & 0xFF;
    }
  }
}

// returns the current head of the queue
int head(){
  int value = state.read();
  return (value >>> 16) & 0xFF;
}

// releases the lock
unlock(){
  while( true ){
    int value = queue.read();
    int head = (value >>> 16) & 0xFF
```

```
      int tail = value & 0xFF
      int next = (head+1) << 16 | tail;
      if( queue.CAS( value, next ) == value ){
        return;
      }
    }
  }
```

2 Bus and Caches

See slides 8/17, 8/20 and 8/29. We simulate some processes trying to acquire a lock. For this task use a dice or a coin to generate a sequence of random numbers $s_1 ... s_m$ in the range of 1 to 4. In the i'th round the process p_{s_i} can execute its next step. One step is long enough to access the state-variable exactly once.

a) Four processes perform Test&Set-locking. For a sequence of 10 to 20 rounds, write down what states the caches are in, and what data has to be moved around on the bus. In each round one randomly chosen process can execute one step. If a process acquires the lock, it releases the lock in its next step.

b) Repeat a) for Test&Test&Set-locking.

c) Explain why TTAS is faster than TAS using the results from a) and b).

Solution

A little Java application creating these tables can be downloaded from our web-page. Please read the comments within the source file to work with the application. Of course there are about 4^{10} different solutions, and only one of them can be presented here.

a) Example table for Test&Set. The random sequence was 0, 2, 2, 1, 1, 1, 0, 2, 1, 2.

Round		p_0	p_1	p_2	p_3
-1		invalid (0)	invalid (0)	invalid (0)	invalid (0)
0	p_0	Bus: load-request, memory-to-cache, invalidate			
		dirty (1)	invalid (0)	invalid (0)	invalid (0)
1	p_2	Bus: load-request, cache-to-memory, memory-to-cache, invalidate			
		invalid (1)	invalid (0)	dirty (1)	invalid (0)
2	p_2	Bus: -			
		invalid (1)	invalid (0)	dirty (1)	invalid (0)
3	p_1	Bus: load-request, cache-to-memory, memory-to-cache, invalidate			
		invalid (1)	dirty (1)	invalid (1)	invalid (0)
4	p_1	Bus: -			
		invalid (1)	dirty (1)	invalid (1)	invalid (0)
5	p_1	Bus: -			
		invalid (1)	dirty (1)	invalid (1)	invalid (0)
6	p_0	Bus: invalidate			
		dirty (0)	invalid (1)	invalid (1)	invalid (0)
7	p_2	Bus: load-request, cache-to-memory, memory-to-cache, invalidate			
		invalid (0)	invalid (1)	dirty (1)	invalid (0)
8	p_1	Bus: load-request, cache-to-memory, memory-to-cache, invalidate			
		invalid (0)	dirty (1)	invalid (1)	invalid (0)
9	p_2	Bus: invalidate			
		invalid (0)	invalid (1)	dirty (0)	invalid (0)

b) Example table for Test&Test&Set.

Round		p_0	p_1	p_2	p_3
-1		invalid (0)	invalid (0)	invalid (0)	invalid (0)
0	p_0	Bus: load-request			
		valid (0)	invalid (0)	invalid (0)	invalid (0)
1	p_2	Bus: load-request			
		valid (0)	invalid (0)	valid (0)	invalid (0)
2	p_2	Bus: invalidate			
		invalid (0)	invalid (0)	dirty (1)	invalid (0)
3	p_1	Bus: load-request, cache-to-memory, memory-to-cache			
		invalid (0)	valid (1)	valid (1)	invalid (0)
4	p_1	Bus: -			
		invalid (0)	valid (1)	valid (1)	invalid (0)
5	p_1	Bus: -			
		invalid (0)	valid (1)	valid (1)	invalid (0)
6	p_0	Bus: load-request, memory-to-cache, invalidate			
		dirty (1)	invalid (1)	invalid (1)	invalid (0)
7	p_2	Bus: invalidate			
		invalid (1)	invalid (1)	dirty (0)	invalid (0)
8	p_1	Bus: load-request, cache-to-memory, memory-to-cache			
		invalid (1)	valid (0)	valid (0)	invalid (0)
9	p_2	Bus:			
		invalid (1)	valid (0)	valid (0)	invalid (0)

c) If we just count how often the bus was used in a) and b), we see that TAS yields more load than TTAS. In TAS every process writes to the shared variable in each round, leading to many invalidate-messages. On the other hand in TTAS processes often read and only write if there is a real chance of acquiring the lock. A read does not affect the other caches.

   Like every resource, the bus has its limits. The bus can broadcast only one message at a time. If, like in TAS, the bus is heavily used, then the one message telling everyone that the lock was released gets a delay.

3 DHT (Optional)

As we have discussed in the lecture, most DHTs are built on the same idea: a binary search tree. Each leaf of the tree is represented by a peer, nodes (including the root) do not really exist. A peer knows only a small subset of all the other existing peers. But if a peer knows the address of another peer it can always contact the other peer. The network is unreliable, messages can be lost, altered or arrive out of order.

Introduce new messages, protocols, restrictions or other ideas to protected a DHT from various Byzantine attacks. If you need to make assumptions, write them down. Each answer should be about 5-10 sentences.

a) Wrong lookup: A search for a key roughly requires $O(\log n)$ steps. In each step one new peer is included in the search. One way to search is to send a message through the DHT. The message contains the searched key and the address of the peer that started the search. The message is forwarded from one peer to the next such that it always gets a bit nearer to its (yet unknown) destination. Once the destination is reached, the final peer answers.

   A Byzantine peer sends the message either in the wrong direction, or to a non-existing peer.

b) Incorrect routing updates: Each peer maintains a routing table containing the addresses of about $\log n$ other peers. The peers send update messages to each other in order to keep their routing tables up to date.

   A Byzantine peer sends false updates, e.g, it tries to place dead links in a routing table.

c) Partitioning: If a peer wants to join a DHT it has to make contact with a peer that is already part of the DHT. The new peer can then ask the old peer about other nodes of the DHT and insert itself at an appropriate place.

   A Byzantine peer builds up his own private DHT by sending wrong messages to joining peers. It gives joining peers only the addresses of peers in his own isolated net.

Solution

There are no “correct” and “wrong” answers for this task. Just answers that are better than others. The answers given here are just examples of potential solutions.

a)
- The peer starting the search sends many messages in different directions, hoping they take a different path.
- Each peer receiving the message sends an acknowledgement and the address of the next peer back to the one peer that started the search. The starting peer notices if the message gets lost or takes the wrong path. It then reacts, for example by restarting the search. The drawback of this solution are the additional connections that have to be made.
- Assuming we have authentication. Every process receiving the message sends back an acknowledgement to the previous peer. The previous peer then forwards the message to its predecessor. If a peer does not receive two times an acknowledgement (from the correct peers), then its message might have been lost. In this case the peer sends the message again, but in another direction.

b)
- When receiving an update for the routing table, the peer first contacts the new entries to ensure they really exist.
- A peer only accepts an update if received from enough other peers, for example if received $f + 1$ times.
- A peer keeps its old table for some time. Should the new table prove to be bad (e.g. messages are often delayed), then the peer switches back to the old table.

c)
- Some trusted servers are used for joining. But this would mean that the DHT is no longer fully decentralized.
- The joining peer contacts $3f + 1$ servers. These servers run a consensus-protocol. The result of the consensus is the place and the routing table of the joining peer.
- The joining peer contacts $f + 1$ servers. If they do not agree, the peer randomly chooses another set of $f + 1$ servers. This works only if f is small compared to the total number of servers.