Distributed Algorithms
Lecture 11

Lecture by Dr. Gadi Taubenfeld
Typeset by Steven Karas

2017-06-21
Last edited 17:56:46 2017-06-21

Disclaimer These lecture notes are based on the lecture for the course Distributed Algorithms, taught by Dr. Gadi Taubenfeld at IDC Herzliyah in the spring semester of 2017. Sections may be based on the lecture slides and accompanying book written by Dr. Gadi Taubenfeld.

Agenda

- Barrier synchronization

1 Barriers

Definition A barrier is a synchronization approach for multiple processes in which execution only continues after all the processes reach the barrier point.

Design goals

- Reusability (this is necessary for correctness)
- Symmetric-ness (same amount of work for all processes)
- No need for initialization
- Shared space complexity
- Contention complexity
- Time complexity

1.1 Fetch-and-increment Register

Provides an atomic operation that increments the register and returns the previous value.
Some variants impose a modulo as part of the operation.

1.2 Await macro

Spin on a condition.

1.3 Simple barrier with an atomic counter

Using a shared fetch and increment and an atomic register, and local copies of each:

For process i in 0...n

```
1 local.go = go
2 local.counter = fetch-and-increment(counter)
3 if local.counter + 1 = n:
4   counter = 0
5   go = 1 - 0
6 else:
7   await(local.go != go)
```

1.3.1 An incorrect variant

This is incorrect because a starved process can get stuck at the barrier and lead to deadlock in a second round through the same barrier.

For process i in 0...n

```
1 local.go = go
2 local.counter = fetch-and-increment(counter)
3 if local.counter + 1 = n:
4   counter = 0
5   go = 1 - 0
6 else:
7   await(local.go != go)
```

1.3.2 Reducing shared memory contention

There is high contention on the go bit.

We can reduce this by replacing the single bit with a set of n bits, where each process p_i spins only on $go[i]$.

In distributed shared memory, this can reduce the remote memory accesses to n.

1.4 Barrier without initialization

If we are able to initialize variables, we must already have synchronization. This approach assumes no prior synchronization.

For process i in 0...n

```
1 local.go = go
2 local.counter = counter
3 counter := counter + 1 (mod n)
4 until local.go != go:
5   if counter = local.counter:
6     go = 1 - local.go
```

Some minor modifications can make this correct for

1.5 Test-and-set bit

Provides two atomic operations:

test-and-set sets the bit to 1, and returns the previous value.

reset sets the bit to 0.

test-and-test-and-set bit Provides an atomic read operation.

1.6 Test-and-set barrier

With three shared variables: leader (test-and-set), countflag (test-and-test-and-set), and go (atomic-register).

Steven: The algorithm in the slides is not correct for n=1

For process i in 0...n

```
1  local.go = go
2  if test-and-set(leader) = 0:
3    local.counter = 0
4    until local.counter = n - 1:
5      await(countflag = 1)
6      local.counter += 1
7      reset(countflag)
8    reset(leader)
9    go = 1 - go
10 else:
11   await(test-and-set(countflag) = 0)
12   await(local.go != go)
```

1.6.1 Variant without initialization

The basic approach here is to preelect the leader and have it count each process twice (until the counter is $2n-2$). This ensures that at most $n-1$ processes have been counted twice, and one has been counted either once or twice.

For process i in 0...n

```
1  local.go = go
2  if i = 1:
3    local.counter = 0
4    until local.counter = 2n - 2:
5      await(countflag = 1)
6      local.counter += 1
7      reset(countflag)
8    reset(leader)
9    go = 1 - go
10 else:
11   await(test-and-set(countflag) = 0)
12   await(test-and-set(countflag) = 0)
13   await(local.go != go)
```

Note this algorithm is asymmetric, which we consider to be undesireable.

1.7 Tree-based barriers

Organize the processes into a binary tree where each node is owned by a process.

Each node waits for its children, and then reports to its parent. Once the root gets reports from both its children, it sends a message back down the tree.

Another approach is to run a tournament between nodes until a root is chosen, and then continue.

For process i in 0...n

```
1  if i = 1:
2    await(arrive[2] = 1); arrive[2] = 0
3    await(arrive[3] = 1); arrive[3] = 0
4    go[2] = 1; go[3] = 1;
5  else if i ≤ (n-1)/2:
6    await(arrive[2i] = 1); arrive[2i] = 0
7    await(arrive[2i+1] = 1); arrive[2i+1] = 0
8    arrive[i] = 1
9    await(go[i] = 1); go[i] = 0
10   go[2i] = 1; go[2i+1] = 1
11 else:
12   arrive[i] = 1
13   await(go[i] = 1); go[i] = 0
```

1.8 Read-modify-write Registers

A register r with n bits. Supports an atomic operation read-modify-write, which does the following atomically:

```
1  read-modify-write(r: register, f: function):
2    orig_r = r
3    r = f(r)
4    return orig_r
```

1.9 See-saw barrier

Presented by M. J. Fischer, S. Moran, S. Rudich, and G. Taubenfeld in 1996

3 bit algorithm, but we need 2 RMW bits.

The idea is that each process has 2 tokens, which can either be absorbed or emitted at a time.

The seesaw has one side up and one side down. Each process that enters the algorithm needs to enter the seesaw on either the left or right side.

The token bit has two states: no-token-present and token-present.

The see-saw bit has two states: left-side-down and right-side-down.

Each round, a process runs one of five rules. Once a process has gathered at least $2n$ tokens, it sets the go bit. A process that has gotten off the seesaw awaits the go bit to flip.

1. if the process has never been on; save the go bit locally, get on the up side, and swing the seesaw.
2. if the process is on the down side, has tokens, and there is no token; emit a token to the shared state, if no tokens left, get off and swing the seesaw.
3. if the process is on the up side and there is a shared token; take the token.
4. if the process has at least $2n$ tokens, it gets off the seesaw and flips the go bit.
5. if the go bit has been flipped, all processes have arrived, and continues past the barrier.

The shared memory is accessed at least $O(n)$ times, and at worst $O(n^2)$.

2 Next week

Semaphore barriers.

Basically, a way to snapshot the entire system and get a consistent state.