Course 02158

Monitor Testing

Hans Henrik Løvengreen

DTU Compute

Testing Concurrent Programs

Cons

- *Testing can show the presence of errors but never their absence*

  — E.W. Dijkstra
- Execution of **concurrent** programs is *non-deterministic*:
  - Only a fraction of all execution paths can be covered
  - Subtle errors, like *race conditions*, hard to "hit"
  - Errors found cannot be reproduced

Pros

- *Beware of bugs in [my program]; I have only proved it correct, not tried it*

  — Donald Knuth
- Programs must be *functionally validated* anyhow
- *Mundane bugs* like typos, ± 1, etc. likely to be found
- Process *interaction* is (or should be) concentrated in a few components

Testing Monitors

- [Brinch-Hansen 78] *Reproducible Testing of Monitors*
- Around 2010 applied to Java programs (Harvey, Hoffman, Long, Strooper)

Idea

- Monitors operations $\sim$ atomic chunks of sequential code
- Monitors may be structurally tested like other objects

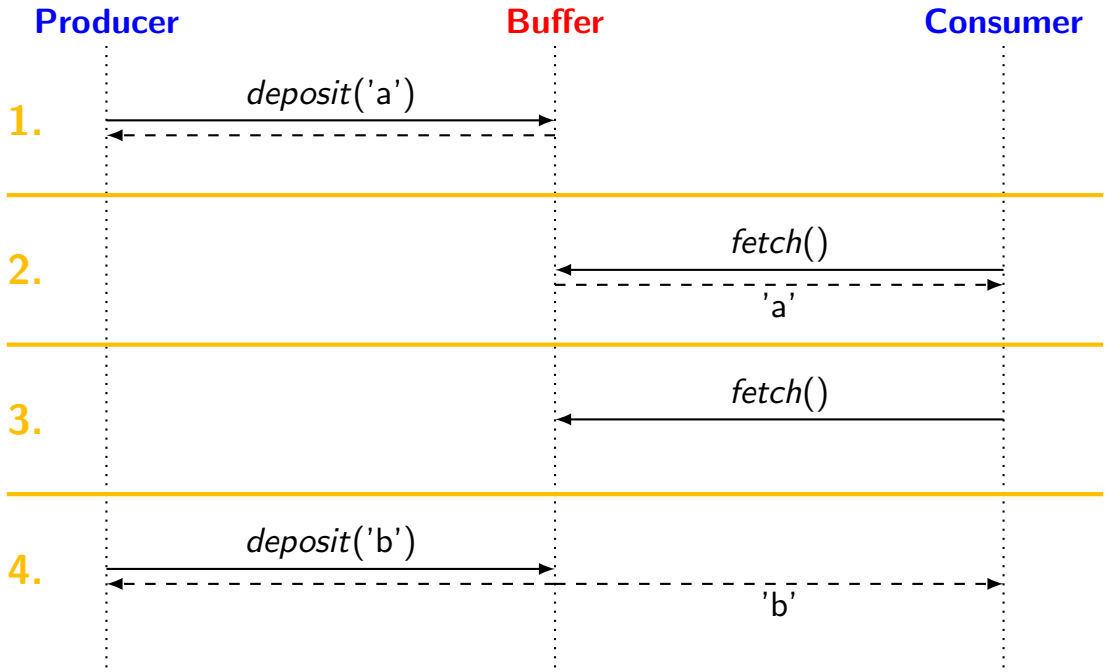
Difficulties

- Monitor operations may *block* — calls for concurrent calls
- Ordering of operation calls is determined by scheduler
- Queue orderings may not be determined
- In Java, there is only a single condition queue

Monitor Test Method

1. Determine *test conditions* to cover all branches and iterations
2. Construct *test sequences/scenarios* that will cover all conditions
3. Implement test sequences in a *test program*
4. Determine a "sufficiently long" *operation separation time*
5. *Run* test program in a *test environment* and *compare* results

Example: Buffer Scenario



Test Sequence Implementation

- **process** *Producer*$_1$

```
Timer.await(1);
Buf.deposit('a');
assert(Timer.time = 1);

Timer.await(4);
Buf.deposit('b');
assert(Timer.time = 4);

print("Producer 1 finished")
```

process *Consumer*$_1$

```
var c : char;

Timer.await(2);
c := Buf.fetch();
assert(c = 'a' ∧ Timer.time = 2);

Timer.await(3);
c := Buf.fetch();
assert(c = 'b' ∧ Timer.time = 4);

print("Consumer 1 finished")
```

Test Environment

- **monitor** *Timer*

```
var clock : int := 0;
    check : condition;

procedure await(when : int)
  while clock < when do wait(check);

function time() : int
  return clock;

procedure tick()
  clock := clock + 1;
  signal_all(check);

end
```

```
process ClockWork
  loop
    delay "sufficient time"
    Timer.tick();
  end loop
```