INF346: Shared-memory computing

Correctness of algorithms:
safety and liveness

INF346, 2014

How to treat a (computing) system formally

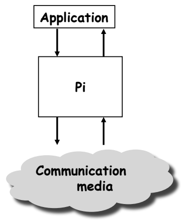
- Define models (tractability, realism)
- Devise abstractions for the system design (convenience, efficiency)
- Devise algorithms and determine complexity bounds

Basic abstractions

- *Process* abstraction – an entity performing independent computation
- Communication
  - ✓Message-passing: *channel* abstraction
  - ✓Shared memory: *objects*

Processes

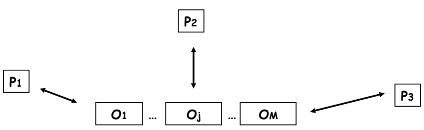
- Automaton P_i (i=1,...,N):
  - ✓States
  - ✓Inputs
  - ✓Outputs
  - ✓Sequential specification



Algorithm = {P_1,…,P_N}
- Deterministic algorithms
- Randomized algorithms

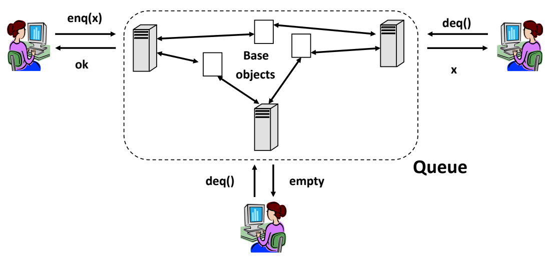
Shared memory

- Processes communicate by applying operations on and receiving responses from *shared objects*
- A shared object instantiates a state machine
  - ✓States
  - ✓Operations/Responses
  - ✓Sequential specification
- Examples: read-write registers, TAS,CAS,LL/SC,…



Implementing an object

Using *base* objects, create an illusion that an object O is available



Correctness

What does it mean for an implementation to be correct?

- Safety ≈ nothing bad ever happens
  - ✓Can be violated in a finite execution, e.g., by producing a wrong output or sending an incorrect message
  - ✓What the implementation is allowed to output
- Liveness ≈ something good eventually happens
  - ✓Can only be violated in an *infinite* execution, e.g., by never producing an expected output
  - ✓Under which condition the implementation outputs

In our context

Processes access an (implemented) abstraction (e.g., bounded buffer, a queue, a mutual exclusion) by invoking operations

- An operation is implemented using a sequence of accesses to base objects
  - E.g.: a bounded-buffer using reads, writes, TAS, etc.
- A process that never fails (stops taking steps) in the middle of its operation is called correct
  - We typically assume that a correct process invokes infinitely many operations, so a process is correct if it takes infinitely many steps

Runs

A system run is a sequence of events

✓E.g., actions that processes may take

Σ – event alphabet

✓ E.g., all possible actions

$\Sigma^{*\cup\{\infty\}}$ is the set all finite and infinite runs

A property P is a subset of $\Sigma^{*\cup\{\infty\}}$

An implementation satisfies P if every its run is in P

Safety properties

P is a safety property if:

- P is prefix-closed: if σ is in P, then each prefix of σ is in P
- P is limit-closed: for each infinite sequence of traces $\sigma_0, \sigma_1, \sigma_2, \ldots$, such that each σ_i is a prefix of σ_{i+1} and each σ_i is in P, the limit trace σ is in P

(Enough to prove safety for all finite traces of an algorithm)

Liveness properties

P is a liveness property if every finite σ in Σ* has an extension in P

(Enough to prove liveness for all infinite runs)

A liveness property is dense: intersects with extensions of every finite trace

Safety? Liveness?

- Processes propose values and decide on values:

$\Sigma = \cup_{i,v}\{propose_i(v), decide_i(v)\} \cup \{\text{base-object accesses}\}$

✓Every decided value was previously proposed
✓No two processes decide differently
✓Every correct (taking infinitely many steps) process eventually decides
✓No two correct processes decide differently

Quiz: safety

1. Let S be a safety property. Show that if all finite runs of an implementation I are safe (belong to S) that all runs of I in are safe
2. Show that every unsafe run σ has an unsafe finite prefix σ’ : every extension of σ’ is unsafe
3. Show that every property is a mixture of a safety property and a liveness property

How to distinguish safety and liveness: rules of thumb

Let P be a property (set of runs)

- If every run that violates P is infinite
  - ✓P is liveness
- If every run that violates P has a finite prefix that violates P
  - ✓P is safety
- Otherwise, P is a mixture of safety and liveness

Example: implementing a concurrent queue

What *is* a concurrent FIFO queue?

✓FIFO means strict temporal order
✓Concurrent means ambiguous temporal order

When we use a lock…

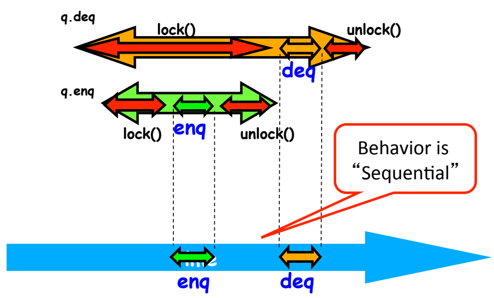
```
shared
      items[];
      tail, head := 0

deq()

  lock.lock();
    if (tail = head)
       x := empty;
    else
       x := items[head];
       head++;
  lock.unlock();
  return x;
```


Intuitively…

```
deq()
  lock.lock();
    if (tail = head)
       x := empty;
    else
       x := items[head];
       head++;
  lock.unlock();
  return x;
```

All modifications of queue are done in mutual exclusion

We describe the concurrent via the sequential



Linearizability (atomicity): A Safety Property

- Each complete operation should
  - ✓"take effect"
  - ✓Instantaneously
  - ✓Between invocation and response events
- A concurrent execution is correct if its "sequential equivalent" is correct

(To be defined formally later)

Why not using one lock?

- Simple – automatic transformation of the sequential code
- Correct – linearizability for free
- In the best case, starvation-free: if the lock is "fair" and every process cooperates, every process makes progress
- Not robust to failures/asynchrony
  - ✓ Cache misses, page faults, swap outs
- Fine-grained locking?
  - ✓ Complicated/prone to deadlocks

Liveness properties

- *Deadlock-free:*
  - ✓If every process cooperates (takes enough steps), some process makes progress
- *Starvation-free:*
  - ✓If every process cooperates, every process makes progress
- *Lock-free (*sometimes called *non-blocking):*
  - ✓Some active process makes progress
- *Wait-free:*
  - ✓Every active process makes progress
- *Obstruction-free:*
  - ✓Every process makes progress if it executes in isolation

Periodic table of liveness properties

[© 2013 Herlihy&Shavit]

	independent non-blocking	**dependent non-blocking**	**dependent blocking**
every process makes progress	wait-freedom	obstruction-freedom	starvation-freedom
some process makes progress	lock-freedom	?	deadlock-freedom

What are the relations (weaker/stronger) between these progress properties?